UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA

DEPARTAMENTO DE ELECTRÓNICA, TELEMÁTICA Y AUTOMÁTICA

TESIS DOCTORAL

LENGUAJE DE ESPECIFICACIÓN Y PARTICIONADO HARDWARE/SOFTWARE PARA CODISEÑO DE SISTEMAS

JUAN FRANCISCO PÉREZ CASTELLANO

Las Palmas de Gran Canaria, Mayo de 1999

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

68/1998-99

UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
UNIDAD DE TERCER CICLO Y POSTGRADO

Reunido el día de la fecha, el Tribunal nombrado por el Excmo. Sr. Rector Magfco. de esta Universidad, el/a aspirante expuso esta TESIS DOCTORAL.
Terminada la lectura y contestadas por el/a Doctorando/a las objeciones formuladas por los señores miembros del Tribunal, éste calificó dicho trabajo con la nota de SOBRESALIENTE CUM LAUDE (5 VOTOS)

Las Palmas de Gran Canaria, a 14 de julio de 1999.

El/a Presidente/a: Dr. D. Carlos Delgado Kloos,

El/a Secretario/a: Dr. D. Luis Gómez Déniz,

El/a Vocal: Dr.D. Santiago Candela Solá,

El/a Vocal: Dr.D. Jorge García Vidal,

El/a Vocal: Dr.D. Luis Sánchez Fernández,

El Doctorando: D. Juan Francisco Pérez Castellano,

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

BIBLIOTECA UNIVERSITARIA
LAS PALMAS DE G. CANARIA
N.º Documento 190.507
N.º Copia 570.328

UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA

Tesis Doctoral

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

Lenguaje de Especificación y Particionado Hardware/Software para Codiseño de Sistemas

AUTOR: Juan Francisco Pérez Castellano

DIRECTOR: Álvaro Suárez Sarmiento

FECHA: Mayo - 1999

UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA

Tesis Doctoral

Lenguaje de Especificación y Particionado Hardware/Software para Codiseño de Sistemas

AUTOR: Juan Francisco Pérez Castellano

DIRECTOR: Álvaro Suárez Sarmiento

FECHA: Mayo - 1999

Presidente **Secretario**

Vocal **Vocal** **Vocal**

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

A mis padres.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

Agradecimientos

Gracias a Álvaro, por impulsar y alentar este trabajo decididamente. También quiero agradecer a David, Juan Carlos y Onassis su interés y esfuerzo en los trabajos relacionados con esta tesis. Gracias a Luis por los buenos ratos que hemos pasado. Gracias a María, por todo eso que ella sabe.

Por último, agradecer a todos los que de alguna forma han contribuido al desarrollo de este trabajo.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

Índice

Capítulo 1 Introducción *1*

1.1. Antecedentes 3

1.2. Estado del arte 5

1.3. Motivaciones 6

1.4. Objetivos 7

1.5. Estructura de la memoria 8

Capítulo 2 Codiseño Hardware/Software *13*

2.1. Introducción 15

2.2. Metodología de codiseño hardware/software 15

2.3. Diseño de hardware y software 19

2.3.1. Síntesis de hardware 19

2.3.2. Síntesis de software 21

2.4. Algunos entornos de codiseño 22

2.5. Entorno de codiseño GACSYS 29

Capítulo 3 Lenguajes de Especificación y Particionado Hw/Sw *35*

3.1. Introducción 37

3.2. Características básicas para la especificación de sistemas 37

3.2.1. Jerarquía de comportamiento y estructural 37

3.2.2. Concurrencia, comunicación y sincronización 38

3.2.3. Temporización 39

3.2.4. Manejo de excepciones 39

3.2.5. No determinismo 40

3.3. Lenguajes de especificación para codiseño 40

3.3.1. Lenguajes basados en máquinas de estados 41

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

3.3.2. Lenguajes basados en múltiples procesos secuenciales 42
3.4. Particionado hardware/software 46
3.5. Algoritmos de planificación 48
3.5.1. Algoritmos de planificación ASAP y ALAP 49
3.5.2. Planificación basada en listas (*list-based*) 51
3.5.3. Planificación basada en el *simulated annealing* 56

Capítulo 4 Lenguaje VSS ***61***

4.1. Introducción 63
4.2. Jerarquía de procesos 64
4.3. Composición secuencial y paralela de procesos: *seq_fork / par_fork* 64
4.4. Sincronización de procesos: *join* 66
4.5. Activación y desactivación de procesos: *start, kill e initialize* 66
4.6. Comunicación entre procesos 68
4.7. Excepciones 71
4.8. Chequeo de excepciones: *check* 73
4.9. Variables compartidas 74
4.10. Temporización 75
4.11. Traducción de las sentencias de la jerarquía de procesos 77
4.11.1. Propagación de la señal *kill* 80
4.11.2. Sentencia *par_fork* 82
4.11.3. Sentencia *seq_fork* 82
4.11.4. Sentencias *kill, initialize, start* y *join* 84
4.12. Traducción de las sentencias de comunicación 86
4.12.1. Comunicación síncrona en modo simple 86
4.12.2. Comunicación asíncrona en modo simple 88
4.12.3. Comunicación síncrona en modo *broadcast* 90
4.12.4. Comunicación asíncrona en modo *broadcast* 92
4.12.5. Comunicación síncrona en modo *incast* 94
4.12.6. Comunicación asíncrona en modo *incast* 94

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

4.12.7. Sentencias *resetbuffer, buffer_empty* y *buffer_full* 94
4.13. Traducción de las sentencias de manejo de excepciones 96
4.13.1. Sentencia *exception* 96
4.13.2. Sentencia *check* 98
4.13.3. Sentencias *vssenable* y *vssdisable* 101
4.14. Traducción de las sentencias de manejo de variables compartidas 101
4.14.1. Sentencia *lock* 102
4.14.2. Sentencias *get* y *put* 103
4.15. Traducción de las sentencias de temporización 106
4.15.1. Sentencia *delay* 107
4.15.2. Sentencias *timeout* y *reset_timer* 107
4.16. Representación intermedia 110
4.16.1. Semántica del grafo ASCIS 110
4.16.2. Grafo ASCIS extendido 112
4.16.3. Ejemplo 113

Capítulo 5 Estimaciones ***117***

5.1. Introducción 119
5.2. Información de entrada 120
5.2.1. Ficheros de contadores 122
5.2.2. Fichero de unidades funcionales 123
5.3. Información de salida 125
5.4. Evaluación de las estimaciones 126
5.4.1. Estimaciones hardware 128
5.4.2. Estimaciones software 138
5.5. Algoritmos de planificación 139
5.5.1. Planificación basada en listas (*list-based*) 140
5.5.2. Planificación basada en el *simulated annealing* 141
5.6. Selección del algoritmo de planificación 147
5.6.1. Diseño de los experimentos 147

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

5.6.2. Primera fase: selección de la vecindad 148
5.6.3. Segunda fase: selección del algoritmo 153

Capítulo 6 Particionado Hardware/Software *161*

6.1. Introducción 163
6.2. Información de entrada y salida 165
6.3. Segmentación a nivel de procesos 166
6.4. Diseño de la función de coste 168
6.5. Algoritmos de particionado 173
6.5.1. Métodos A y A/N 173
6.5.2. Métodos B y B/N 177
6.5.3. Métodos C y C/N 181

Capítulo 7 Resultados experimentales *185*

7.1. Introducción 187
7.2. Descripción del ejemplo 187
7.3. Librerías utilizadas 188
7.4. Estimaciones 190
7.5. Particionado hardware/software 192
7.5.1. Espacio de diseño 193
7.5.2. Segmentación a nivel de procesos 193
7.5.3. Comparación de los métodos 194
7.5.4. Calidad de los resultados 194

Capítulo 8 Conclusiones y trabajo futuro *209*

8.1. Conclusiones 211
8.2. Trabajo futuro 213

Bibliografía *217*

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

Apéndice A *Sintaxis del lenguaje VSS*

Apéndice B *Resultados de los algoritmos de planificación: simulated annealing*

Apéndice C *Resultados de los algoritmos de planificación: list-based*

Apéndice D *Especificación VSS: Ejemplo*

Apéndice E *Especificación VHDL: Ejemplo*

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

Capítulo 1

Introducción

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

1.1. Antecedentes

Los sistemas electrónicos e informáticos están presentes en muchas de nuestras actividades diarias y tienen un fuerte impacto social y económico. Estos sistemas se utilizan en campos de aplicación muy variados tales como el hogar, los medios de transporte, la computación personal, las comunicaciones, la salud, la ingeniería, la gestión empresarial, etc. En la Tabla 1.1 se muestran algunas aplicaciones.

Un conjunto importante de estos sistemas son empotrados (*embedded systems*) [Cal93]. Se trata de sistemas de propósito específico que están integrados (empotrados) en sistemas mayores. En general, la mayoría de los sistemas empotrados se implementan de forma mixta, es decir, una parte del sistema se implementa en software mediante un procesador programable, y otra parte se implementa mediante hardware específico. El hardware ofrece mayor velocidad pero tiene un coste superior y una menor flexibilidad que el software.

La metodología que se ha venido utilizando tradicionalmente para el diseño de sistemas empotrados es la siguiente [Kum96]: en primer lugar, el sistema es especificado de manera informal. En segundo lugar, se deciden las partes del sistema que se implementarán en hardware o software. Este proceso de particionado hardware/ software es realizado manualmente por los diseñadores en base a su experiencia previa. Posteriormente, se desarrollan por separado el hardware y software utilizando las

Campos de aplicación	Sistemas
Hogar	Procesado de señales (audio, video, etc.), sistemas de control para electrodomésticos, etc.
Automóviles	Sistemas de control para la inyección, suspensiones, frenado, etc.
Computación personal	PCs, computadores portátiles, etc.
Comunicaciones (telefonía, videoconferencia, etc.)	Protocolos, compresión de imágenes, etc.
Medicina	Procesado de imágenes, bases de datos genéticas, etc.
Ingeniería	Sistemas de control, herramientas CAD, etc.
Gestión empresarial	Bases de datos, redes de área local, etc.

Tabla 1.1. Diversos campos de aplicación de los sistemas electrónicos e informáticos.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

metodologías de diseño hardware y software, respectivamente. Finalmente, se realiza la integración del hardware y software desarrollados.

La tecnología de fabricación de circuitos integrados permite el desarrollo de sistemas empotrados cada vez más complejos. La metodología tradicional de diseño de estos sistemas no es adecuada para sistemas complejos, ya que el aumento de la complejidad genera los siguientes problemas:

- El espacio de diseño crece de forma considerable. Un diseñador, por mucha experiencia que acumule, solamente puede realizar una exploración manual de un conjunto pequeño de compromisos entre hardware y software. Por tanto, es muy poco probable que el particionado hardware/software realizado manualmente por un diseñador sea óptimo.
- Se incrementa la aparición de errores al realizarse la integración final de hardware y software. La causa principal de este problema es que el proceso de diseño se realiza a partir de una especificación informal del sistema. Este problema genera ciclos de rediseño que aumentan el coste y el tiempo de diseño de los sistemas. Además, puede generar una pérdida de fiabilidad en los mismos.

Estos problemas son muy importantes, ya que la velocidad de los avances tecnológicos reduce cada vez más el tiempo de vida de los diseños. Actualmente, para que un diseño sea competitivo son necesarias las siguientes condiciones: a) el tiempo de diseño debe ser pequeño, b) el diseño debe estar optimizado, es decir, debe cumplir sus objetivos con el mínimo coste, y c) el diseño debe tener un alto grado de fiabilidad.

En los últimos años se han desarrollado las denominadas metodologías de codiseño hardware/software [Ern93][Kum93][Gaj94][Gup95][Ism95][Bal97], que aportan algunas soluciones para resolver los problemas anteriores:

- La exploración del espacio de diseño y la optimización del particionado hardware/software pueden ser mejorados mediante estrategias automáticas o semi-automáticas de particionado. Estas estrategias necesitan estimaciones previas de algunos parámetros relativos a la implementación del sistema, tales como tiempo de ejecución, área y potencia.
- Los errores detectados en la fase de integración del hardware y software pueden disminuirse diseñando los sistemas a partir de lenguajes de especificación que permiten la simulación y verificación conjunta del hardware y software de los diseños.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

En definitiva, los lenguajes de especificación de sistemas y la automatización del particionado hardware/software son dos aspectos importantes que abordan las metodologías de codiseño. A continuación se presentan los trabajos más relevantes en ambas áreas.

1.2. Estado del arte

En este apartado se presentan los trabajos más importantes que se han publicado sobre lenguajes de especificación de sistemas y sobre particionado hardware/software. En el capítulo 3 de esta memoria se presentan estos trabajos de forma más extensa.

Los lenguajes de especificación más utilizados se pueden clasificar en dos clases en función del paradigma que utilizan: a) lenguajes basados en máquinas de estados, y b) lenguajes basados en múltiples procesos secuenciales.

Los lenguajes basados en máquinas de estados son adecuados para las aplicaciones de control (*control dominated*). Los lenguajes más importantes son los siguientes: a) *StateCharts* [Har87][Dru89], utilizado en el entorno de codiseño *Statemate* [Har88][Har90]. b) *SpecCharts* [Nar91][Vah91][Nar92b], utilizado en el entorno de codiseño SPECSYN [Gaj94][Gaj95]. c) SDL (*Specification and Description Language*) [Tur93], utilizado en el entorno de codiseño COSMOS [Ism95]. d) CFSM (*Codesign Finite State Machines*) [Chi93a][Chi93b], utilizado en el entorno POLIS [Bal97].

Los lenguajes basados en múltiples procesos secuenciales son más adecuados para las aplicaciones dominadas por los datos (*data dominated*), aunque también se pueden utilizar para aplicaciones de control. Los lenguajes más importantes son los siguientes: a) *HardwareC* [Ku88][Ku90] está basado en el lenguaje C y se utiliza en el entorno de codiseño VULCAN [Gup95]. b) C^x [Ben93] también está basado en el lenguaje C y se utiliza en el entorno de codiseño COSYMA [Ern93]. c) OCCAM [Pou87], utilizado como lenguaje de especificación en trabajos como [Bar94][The97]. d) VHDL [Lip93][Vhd93], que se utiliza en trabajos tales como [Sri92][Ern93][Ele94][The94] [Eck97][Car98][Icl99]. e) LOTOS [Lot89][Tur93], utilizado en trabajos tales como [Klo95][Car96][Wol97][San98].

La complejidad de los sistemas empotrados está alcanzando un nivel tan elevado que el mantenimiento y la reutilización de los diseños es cada vez más difícil. Recientemente se han publicado algunos trabajos que proponen lenguajes orientados a objetos para abordar estos problemas. Entre los más relevantes podemos citar los siguientes: *Objective VHDL* [Rad98][eis99], *SUAVE* (*SAVANT and University of Adelaide VHDL Extensions*) [Ash98][Ash99] y Java [Kuh98].

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

Los trabajos más importantes que se han publicado sobre particionado hardware/ software pueden clasificarse atendiendo a múltiples aspectos. En este apartado se presentan algunas clasificaciones atendiendo a los siguientes puntos: estrategia, algoritmos, y parámetros considerados.

En primer lugar, los trabajos pueden ser clasificados en dos grupos según la estrategia de particionado que utilizan: a) particionado manual/interactivo [Lag91][Ism94], y b) particionado automático. Estos últimos pueden ser clasificados en tres grupos según la orientación del particionado: a) métodos de orientación software, que inicialmente suponen que todo el sistema se implementa en software, y posteriormente realizan una migración hacia el hardware de algunas partes del sistema mientras no se verifiquen los requerimientos del diseñador [Ern93][Har96]. b) Métodos de orientación hardware, opuestos a los anteriores [Gup93][Kal93]. c) Métodos mixtos, en los que no se presupone una distribución orientada hacia el software ni hacia el hardware [Ele94][Knu96][Vah96].

En segundo lugar, los trabajos pueden clasificarse en dos grupos atendiendo a los algoritmos de optimización: a) métodos que utilizan algoritmos que proporcionan soluciones óptimas, tales como ILP (*Integer Linear Programming*) [Ben96][Nie97], *branch & bound* [DAm94], y programación dinámica [Knu96]. b) Métodos que utilizan algoritmos heurísticos basados en técnicas tales como *clustering* [Lag91][Vah96] [Wol96], migración de grupos [Vah96][Agr97][Vah97], *simulated annealing* [Ern93][Ele94][Vah96][Hen97], o bien heurísticas *ad-hoc* [Kal94][Kal97].

Finalmente, los trabajos pueden clasificarse en dos grupos según los parámetros considerados para optimizar el particionado: a) trabajos que sólo consideran el tiempo de ejecución [Ern93][Ele94][The94][Ben96], y b) trabajos que tienen en cuenta el tiempo de ejecución y el área del diseño [Gup93][DAm94][Kal94][Vah96] [Wol96].

1.3. Motivaciones

En este apartado se presentan las principales motivaciones que nos llevaron al planteamiento de nuestros objetivos.

En primer lugar, VHDL [Vhd93] es uno de los lenguajes de especificación de sistemas más utilizado. Las razones principales son las siguientes: a) soporta la mayoría de las características para la especificación de sistemas, tales como jerarquía, concurrencia, comunicación, sincronización, temporización y manejo de excepciones, b) soporta el diseño *top-down*, c) soporta las diversas tareas de codiseño, y d) es un

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

lenguaje estándar para el que se han desarrollado muchas herramientas de diseño. De hecho, VHDL se utiliza como lenguaje de especificación en trabajos de codiseño [Sri92][Ern93][Ele94][The94][Eck97]. Además, se utiliza como lenguaje para la síntesis del hardware en la mayoría de los sistemas de codiseño.

Sin embargo, VHDL tiene algunos problemas: a) la jerarquía de comportamiento es limitada, b) el manejo de excepciones, la jerarquía de comportamiento, la temporización y las comunicaciones necesita especificaciones complejas, extensas y tediosas, y c) no soporta variables compartidas. Estos problemas aumentan la complejidad de las especificaciones, y por tanto aumenta el tiempo de diseño. Recientemente, algunos trabajos [Car98] identifican algunos de estos problemas y proponen el lenguaje VHDL+ [Icl99] para su solución. Sin embargo, VHDL+ no soporta variables compartidas ni especificaciones sencillas para el tratamiento de excepciones.

En segundo lugar, los trabajos de particionado hardware/software de los entornos de codiseño actuales presentan algunas carencias: en primer lugar, dado un grafo DAG (*Directed Acyclic Graph*) de procesos que se quiere particionar, los principales trabajos publicados no soportan la ejecución segmentada del mismo [Kal94][Ben96][Wol96]. Otros trabajos, como [Car96], mencionan la segmentación como trabajo futuro, o bien proponen la segmentación a nivel interno de un proceso (*software pipelining*) [Bak97]. En segundo lugar, la potencia consumida por el diseño no es considerada entre los parámetros utilizados para optimizar el particionado. Sin embargo, se trata de un parámetro cada vez más importante debido a la expansión de sistemas portátiles y a la densidad de integración de los circuitos integrados. Estas dos carencias limitan el espacio de diseño, y por tanto la posibilidad de obtener particiones más optimizadas.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

1.4. Objetivos

El objetivo principal de este trabajo es solucionar los problemas mencionados en el apartado anterior, mejorando así dos aspectos importantes de las metodologías de codiseño hardware/software: los lenguajes de especificación y el particionado hardware/software. En particular, en este trabajo se proponen los siguientes objetivos:

- Diseñar un lenguaje de especificaciones basado en VHDL que resuelva los problemas mencionados anteriormente. Se propone que el lenguaje se denomine VSS (*VHDL-based System Specification*). Además, se propone el desarrollo de un compilador que traduzca a VHDL una especificación VSS. La consecución de este objetivo contribuiría principalmente a obtener especificaciones más sencillas y disminuir el tiempo de especificación.

- Diseñar e implementar un método automático de particionado hardware/ software que soporte la ejecución segmentada de grafos de procesos y que considere la potencia consumida por el diseño. La consecución de este objetivo contribuiría a explorar un mayor espacio de diseño, es decir, el diseñador podría realizar un mayor número de compromisos hardware/ software y podría obtener particiones de mayor calidad.

Para un desarrollo integral y pragmático de estos objetivos, se propone el desarrollo de un entorno de codiseño denominado GACSYS (*GAC's Codesign System*). Las tareas principales para alcanzar nuestros objetivos son las siguientes:

- Diseñar el lenguaje de especificación VSS. Este lenguaje debe: a) extender la jerarquía de comportamiento de VHDL, b) soportar especificaciones sencillas para el manejo de excepciones, la jerarquía de procesos, la temporización y las comunicaciones, y c) soportar variables compartidas.
- Desarrollar un compilador que genere código VHDL a partir de VSS.
- Generar una representación intermedia que soporte el lenguaje VHDL.
- Diseñar e implementar el método de particionado hardware/software. Por un lado, se necesitan las estimaciones de diversos parámetros relativos a la implementación de un sistema, tales como tiempo de ejecución, área y potencia. Por otro lado, el particionado necesita el diseño y desarrollo de algoritmos de optimización que soporten la segmentación de los grafos de procesos.

1.5. Estructura de la memoria

El trabajo que hemos llevado a cabo se presenta en los ocho capítulos de la presente memoria. La estructura de cada capítulo es la siguiente: en el capítulo 1, se ha presentado los antecedentes y motivaciones que llevaron a plantear los objetivos de este trabajo.

En el capítulo 2, se presenta los principios básicos de las metodologías de codiseño hardware/software. Primeramente, se argumenta la necesidad de nuevas metodologías de diseño de sistemas empotrados. Por un lado, se comenta la metodología tradicional de diseño, los problemas de la misma para diseñar los complejos sistemas actuales, y las soluciones que aportan las nuevas metodologías de codiseño. Posteriormente, se presenta las características más relevantes de los entornos de codiseño, y el entorno de codiseño sobre el que hemos desarrollado los objetivos de este trabajo.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

En el capítulo 3, se presenta el trabajo previo que hemos realizado sobre lenguajes de especificación de sistemas y particionado hardware/software. De esta forma se justifica con mayor profundidad los objetivos planteados en este trabajo. En primer lugar, se presenta las características básicas de los lenguajes de especificación de sistemas. En segundo lugar, se presenta las características de los lenguajes más utilizados y se justifica la necesidad de ampliar el lenguaje VHDL. En tercer lugar, se presenta los trabajos más importantes sobre particionado hardware/software y se identifica algunas carencias que justifican nuestra propuesta. Finalmente, se presenta una introducción a los algoritmos básicos de planificación que se utilizan en nuestro método de particionado.

En el capítulo 4, primeramente se explica el lenguaje VSS (*VHDL-based system specification*). VSS se basa en un amplio subconjunto de VHDL al que hemos añadido un conjunto de sentencias para facilitar a los diseñadores la especificación de sistemas. Posteriormente, se explica el esquema de traducción que hemos utilizado para traducir a VHDL una especificación VSS. Finalmente, se describe la representación intermedia que hemos diseñado para el código VHDL.

En el capítulo 5, se explica las estimaciones: tiempo de ejecución, área y potencia para cada proceso especificado en la descripción del sistema. En primer lugar, se describe la información de entrada y salida. En segundo lugar, se explica el proceso seguido para la evaluación de las estimaciones. Posteriormente, se explica los algoritmos de planificación que hemos comparado para seleccionar el más adecuado. Finalmente, se presenta los resultados y conclusiones de los experimentos realizados para seleccionar el algoritmo de planificación.

En el capítulo 6, se explica el método de particionado novedoso. En primer lugar, se describe la información de entrada y salida. En segundo lugar, se explica la segmentación a nivel de procesos. En tercer lugar, se describe el diseño de la función de coste que hemos utilizado en los algoritmos de optimización. Finalmente, se explica los algoritmos de particionado novedosos.

En el capítulo 7, se presenta una serie de resultados experimentales que demuestran la consecución de los objetivos planteados en el trabajo. En primer lugar, se describe el ejemplo que se ha utilizado para los experimentos. En segundo lugar, se indican las librerías utilizadas. En tercer lugar, se comentan las estimaciones obtenidas. Finalmente, se comentan los resultados más interesantes.

En el capítulo 8, se presenta las conclusiones de este trabajo y se comenta algunas líneas de trabajo que consideramos que sería interesante abordar.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

Finalmente, se presenta la bibliografía y los apéndices que son referenciados durante la memoria.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

Capítulo 2

Codiseño Hardware/Software

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

2.1. Introducción

En este capítulo se presentan los principios básicos de las metodologías de codiseño hardware/software. Primeramente, se presenta la metodología tradicional de diseño, los problemas de la misma, y las soluciones que aportan las nuevas metodologías de codiseño hardware/software. En segundo lugar, se explican los procesos de síntesis de hardware y software. Posteriormente, se presentan los entornos de codiseño más importantes. Finalmente, se presenta el entorno de codiseño que hemos desarrollado para alcanzar los objetivos de este trabajo.

2.2. Metodología de codiseño hardware/software

La implementación de los sistemas empotrados no siempre se realiza completamente en hardware o software. En muchos casos, estos sistemas son implementados de forma mixta, es decir, una parte en hardware y otra en software. El hardware ofrece mayor velocidad pero tiene un coste superior y una flexibilidad menor que el software. En general, un objetivo importante en el diseño de estos sistemas es optimizar la relación coste/*performance*. Para ello es necesario encontrar el compromiso adecuado entre hardware y software. Este compromiso puede entenderse desde los siguientes puntos de vista:

- Un diseño cuya implementación software no satisface los requerimientos de *performance* exigidos, debe implementar en hardware algunas partes del mismo para acelerar su ejecución.
- Un diseño cuya implementación hardware satisface con creces los requerimientos de *performance* exigidos, debe implementar en software algunas partes del mismo para disminuir el coste del diseño.

En la Figura 2.1 se muestra la metodología de diseño que se ha venido utilizando para sistemas empotrados [Cal93][Kum96]. En primer lugar, se realiza una descripción informal del sistema. El objetivo de la misma es describir los objetivos y requerimientos del sistema. Esta descripción no está basada en lenguajes formales. En segundo lugar, se deciden las partes del sistema que se implementarán en hardware y software. Este proceso de particionado hardware/software se realiza manualmente en base a la experiencia previa de los diseñadores. Seguidamente, las particiones hardware y software se desarrollan mediante las metodologías de diseño de hardware y software, respectivamente. Finalmente, se lleva a cabo la integración del hardware y software desarrollados.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

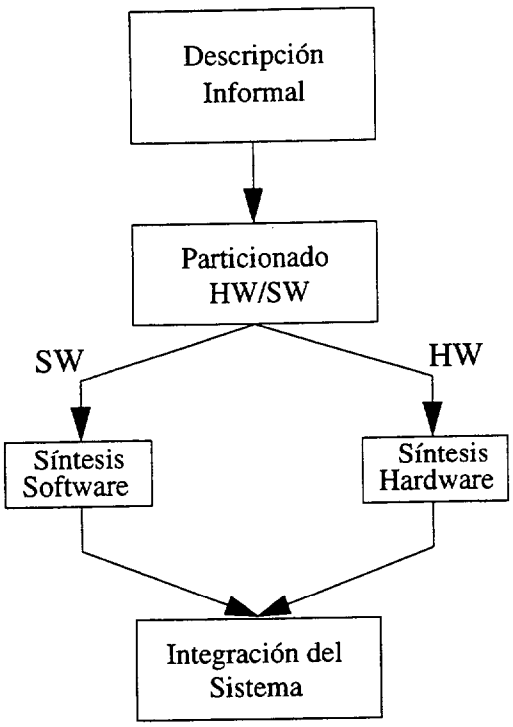


Figura 2.1. Metodología tradicional de diseño de sistemas empotrados.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

Los avances tecnológicos permiten el desarrollo de sistemas cada vez más complejos. El aumento de la complejidad hace que la metodología tradicional de diseño de sistemas empotrados no sea adecuada principalmente debido a tres problemas importantes:

- El espacio de diseño crece considerablemente. Por tanto, la exploración manual de los diversos compromisos entre hardware y software es muy limitada, por lo que es muy poco probable que el particionado hardware/ software sea óptimo.
- Se produce un aumento del número de errores en la fase de integración del sistema. Esto es debido a que no se realiza una especificación formal de todo el sistema y a la escasa interacción que existe durante el desarrollo del hardware y software.
- Las tareas de mantenimiento y rediseño son muy complejas porque el diseño completo no está formalmente descrito en un mismo lenguaje de especificación, ya que existe una especificación para el diseño del hardware y otra para el diseño del software.

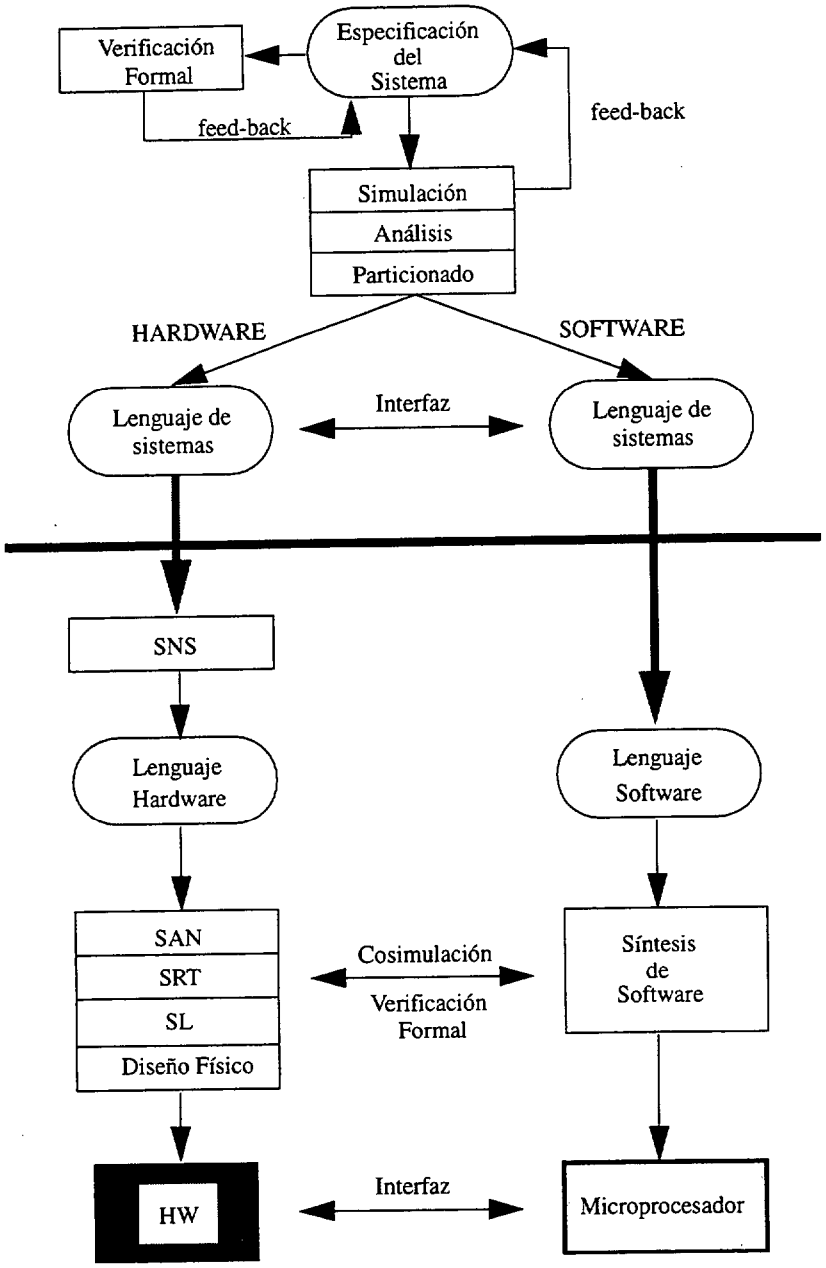
Estos problemas no se manifiestan de manera importante cuando se diseñan sistemas de baja complejidad, pero son especialmente relevantes en el caso de sistemas complejos. Las consecuencias más importantes de estos problemas son las siguientes: a) los diseños no están optimizados, b) el tiempo de diseño crece considerablemente, y c) la fiabilidad de los sistemas puede disminuir. Estos problemas aumentan el coste de los diseños y retrasa su puesta en el mercado.

Las metodologías de codiseño hardware/software tratan de resolver los problemas anteriores. Existen diversas metodologías de codiseño [Ern93][Kum93][Gaj94][Gup95] [Ism95][Bal97], aunque todas se basan principalmente en las siguientes etapas (Figura 2.2):

- En primer lugar, el sistema completo es descrito en un lenguaje de especificación de sistemas simulable y verificable.
- En segundo lugar, la especificación es simulada para comprobar que el sistema funciona correctamente. Además, la especificación es verificada formalmente para comprobar algunas propiedades, por ejemplo, verificar que no se producen interbloqueos (*deadlock*).
- Seguidamente, la especificación es analizada para obtener estimaciones de los valores de diversos parámetros relativos a la implementación hardware y software del sistema, tales como tiempo de ejecución, área, etc.
- El particionado hardware/software de la especificación se realiza de forma automática o semi-automática mediante algoritmos que optimizan una función objetivo dependiente de los parámetros estimados.
- Finalmente, las particiones hardware y software son implementadas mediante los métodos de diseño de hardware y software, respectivamente. Por un lado, la partición hardware es traducida a un lenguaje de alto nivel de descripción de hardware y posteriormente es sintetizada. Por otro lado, la partición software es traducida a un lenguaje de alto nivel para diseño de software. Ambos desarrollos se realizan al mismo tiempo. A medida que ambas especificaciones disminuyen su nivel de abstracción, se realizan procesos de cosimulación y verificación formal.

Claramente se observa que las metodologías de codiseño se centran principalmente en tres áreas: a) lenguajes de especificación de sistemas, b) particionado hardware/ software, y c) cosimulación y verificación formal.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004



SNS Síntesis a Nivel de Sistema
SAN Síntesis de Alto Nivel
SRT Síntesis RT
SL Síntesis lógica

Figura 2.2. Metodología de codiseño hardware/software.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

2.3. Diseño de hardware y software

Una vez se han explicado los fundamentos de las metodologías de codiseño hardware/ software, es preciso explicar algunas ideas básicas sobre la síntesis de hardware y software.

2.3.1. Síntesis de hardware

Para implementar un sistema en hardware pueden utilizarse diferentes metodologías de diseño. Desde el punto de vista de la jerarquía de diseño, las metodologías pueden dividirse en las siguientes clases:

- *Metodologías bottom-up*, en las que se implementan los sistemas a partir de la interconexión de una serie de componentes previamente diseñados. Para el diseño de estos componentes se utiliza la misma metodología.
- *Metodologías top-down*, en las que se parte desde especificaciones muy abstractas de los sistemas, y mediante procesos de síntesis se van obteniendo especificaciones más concretas y próximas a la implementación final [Mic92].

La ventaja principal de las metodologías *top-down* es que el tiempo de diseño de los sistemas es muy inferior respecto a las metodologías *bottom-up*. Como quiera que el tiempo de diseño de un sistema es un factor crítico para que el mismo sea competitivo en el mercado, las metodologías *top-down* tienen cada vez mayor aceptación. Además, para diseños complejos, las herramientas CAD pueden explorar un mayor espectro del espacio de diseño que un diseñador.

El diseño de hardware puede explicarse atendiendo al diagrama en Y de Gajski (Figura 2.3), en el que se representan los siguientes aspectos:

- *Dominios de representación de un diseño*: funcional, estructural y físico.
- *Niveles de abstracción utilizados en cada dominio*: nivel de sistema, nivel algorítmico, nivel de transferencia entre registros (nivel RT), nivel lógico, y nivel de circuito.

En el dominio funcional se describe la función que realiza un sistema. Un sistema se describe como una *caja negra* con un conjunto de funciones que determinan el comportamiento de las salidas en función de las entradas. En el dominio funcional también se especifican las restricciones (área, retardos, etc.) que deben tenerse en cuenta durante el proceso de síntesis. El dominio funcional también es conocido como dominio del comportamiento. La especificación del comportamiento se refina a medida que disminuye el nivel de abstracción.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

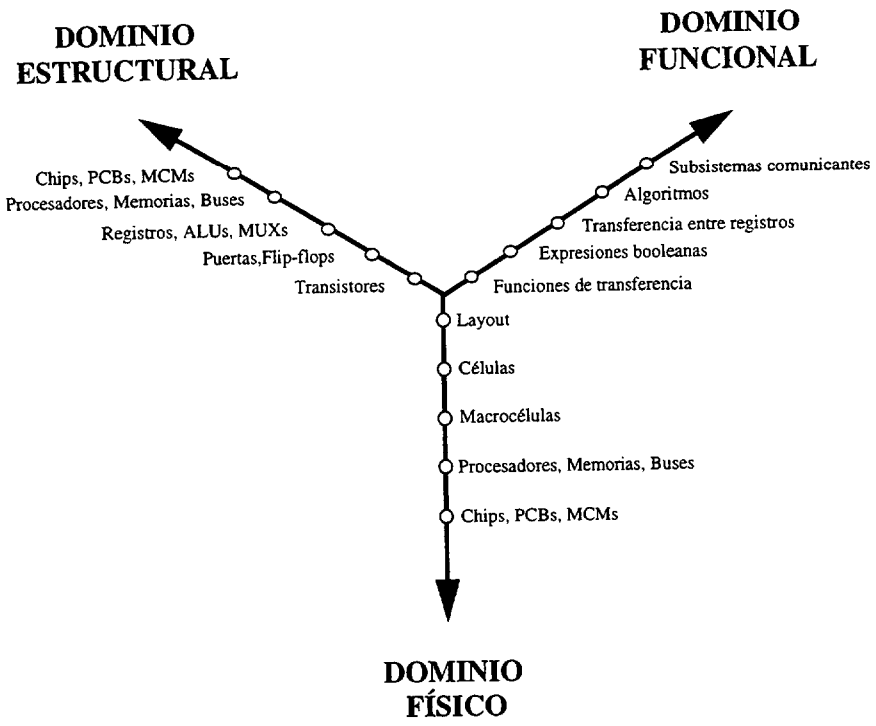


Figura 2.3. Diagrama en Y de Gajski.

En el dominio estructural se describe cómo realiza el sistema sus funciones mediante una serie de componentes interconectados lógicamente, independientemente de la ubicación física final de los componentes y las interconexiones. Los niveles de abstracción en el dominio estructural se identifican en función de los elementos básicos que se utilizan: *a) nivel de sistema*: chips, PCBs, MCMs, *b) nivel algorítmico*: procesadores, memorias y redes de interconexión, *c) nivel RT*: ALUs, multiplicadores, registros, máquinas de estados, etc, *d) nivel lógico*: puertas lógicas y flip-flops, *y e) nivel de circuito*: transistores, resistencias, condensadores, etc.

En el dominio físico se ignora la funcionalidad del sistema y únicamente se representa la ubicación física de los componentes y de las interconexiones de la representación estructural. Los niveles de abstracción en el dominio físico también se identifican en función de los elementos básicos que se utilizan: *a) nivel de sistema*: chips, PCBs, MCMs, *b) nivel algorítmico*: procesadores, memorias y redes de interconexión, *c) nivel RT*: módulos, macrocélulas, *d) nivel lógico*: células, y *e) nivel de circuito*: layout, metales, contactos, etc.

Un proceso de síntesis puede definirse como aquel que transforma, total o parcialmente, una especificación funcional de un sistema en una especificación

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

estructural. Por lo tanto, según el nivel de abstracción podemos definir distintos tipos de síntesis [Mic92]: *síntesis a nivel de sistema, síntesis de alto nivel, síntesis RT, síntesis lógica* y *síntesis física.*

Una de las metodologías de diseño *top-down* basada en procesos de síntesis que es utilizada con mucha frecuencia en los entornos industriales consiste en la aplicación sucesiva de los siguientes pasos: 1) síntesis de alto nivel, 2) síntesis RT, 3) síntesis lógica, y 4) diseño físico (*placement* y *routing*).

La síntesis a nivel de sistema es actualmente objeto de investigación, por lo que no está totalmente madura. Por lo tanto, todavía no es utilizada de forma amplia en los entornos industriales de diseño.

2.3.2. Síntesis de software

La partición software de un sistema se implementa en un microprocesador programable. Por tanto, la síntesis del software consiste en generar el conjunto de instrucciones máquina que implementan dicha partición. Precisamente, una de las ventajas principales de la partición software respecto de la partición hardware es su flexibilidad para ser reprogramada.

En función del tipo de microprocesador utilizado, existen dos tendencias diferentes para la síntesis del software:

- síntesis basada en microprocesadores de propósito general.
- síntesis basada en ASIPs (*Application Specific Instruction-Set Processors*).

La síntesis para microprocesadores de propósito general se basa en las técnicas tradicionales de generación de código [Aho86].

La síntesis de software basada en ASIPS (*Application Specific Instruction-Set Processors*) [Cli94] se basa en el diseño de un procesador específico para ejecutar la partición software. El conjunto de instrucciones del procesador se obtiene a partir de la especificación software. De esta forma el procesador es diseñado para soportar aquellas operaciones que realmente necesita ejecutar dicha partición.

Los métodos de selección del conjunto de instrucciones de un ASIP utilizan el grafo de flujo de datos que representa a la partición software. A partir del grafo, los métodos de selección de instrucciones proceden a la identificación de patrones. Los patrones son pequeños subgrafos extraídos del grafo de flujo de datos. Uno de los aspectos importantes para la selección de los patrones es la frecuencia de los mismos en el grafo. Una vez seleccionado el conjunto de patrones del ASIP, se procede al recubrimiento del

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

grafo para obtener las instrucciones que implementan la partición software. Finalmente, es importante comentar que un ASIP no se desarrolla exclusivamente para un sistema, sino para un conjunto de sistemas que utilizan instrucciones similares y por tanto pueden utilizar el mismo ASIP.

2.4. Algunos entornos de codiseño

En este apartado se comentan las principales características de algunos de los entornos de codiseño más importantes que han sido desarrollados hasta el momento. Los aspectos principales mediante los cuales se puede caracterizar a un entorno de codiseño son los siguientes: tipo de aplicaciones para el que ha sido diseñado, lenguaje de especificaciones que utiliza, modelo de arquitectura que se utiliza para realizar el particionado hardware/software, tipo de estimaciones que se utilizan, método y algoritmos de particionado, cosimulación, y verificación formal.

COSMOS

El entorno de codiseño COSMOS [Ism95] ha sido desarrollado por el Grupo de Síntesis a Nivel de Sistema del Instituto Nacional Politécnico de Grenoble de Francia (INPG). Este entorno de codiseño está orientado a sistemas complejos basados en múltiples módulos que se comunican y que deben verificar restricciones de tiempo real. Para especificar este tipo de sistemas, COSMOS utiliza el lenguaje de especificación SDL [Tur93]. Este lenguaje utiliza un paradigma basado en el modelo de máquinas de estados extendidas. Por tanto, se trata de un paradigma adecuado para la especificación de aplicaciones dominadas por el control (*control dominated*). La extensión de las máquinas de estados permite la especificación de jerarquía y paralelismo en las mismas. COSMOS traduce la especificación SDL de un sistema a una representación intermedia denominada SOLAR [Jer92][Jer95].

COSMOS utiliza un modelo abstracto de arquitectura multiprocesadora. Este modelo se compone de tres tipos de procesadores abstractos:

- *Procesadores hardware*: se componen de módulos hardware tales como ASIC's y FPGA's que deben ser diseñados de forma específica para implementar algunas partes de la especificación del sistema.
- *Procesadores software*: se componen básicamente de un microprocesador programable y una memoria RAM. Estos procesadores implementan algunas partes del sistema mediante la ejecución de un conjunto de instrucciones.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

- *Procesadores de comunicaciones*: estos procesadores pueden tener los componentes de un procesador hardware o software. El objetivo de los mismos es desacoplar las comunicaciones y la computación del sistema. Este desacoplo se produce desde la especificación inicial del sistema. Cada uno de estos procesadores es un servidor de comunicaciones para los procesadores hardware y software. Estos servidores funcionan mediante llamadas remotas a procedimiento (RPC) [And91]. Esta independencia de las comunicaciones permite que el diseñador pueda cambiar y reutilizar fácilmente los protocolos de comunicación. De hecho, este modelo de comunicaciones es una de las principales aportaciones de COSMOS [Obr92][Obr93].

El modelo de arquitectura definido en COSMOS puede implementarse en un chip, una tarjeta, o un multicomputador.

Por lo que al particionado hardware/software se refiere, en primer lugar, COSMOS no dispone de una herramienta de estimaciones. Esto se debe principalmente a dos motivos: por un lado, la obtención de estimaciones de parámetros tales como el tiempo de ejecución para sistemas en tiempo real es muy compleja. De hecho existen pocos trabajos al respecto [Her94]. Por otro lado, el modelo de arquitectura utilizado es complejo y tampoco facilita el proceso de estimaciones. Al no disponer de una herramienta de estimaciones, el particionado hardware/sotfware no se realiza automáticamente. El propio diseñador selecciona una serie de transformaciones de la especificación inicial y decide las partes de la misma que serán implementadas en hardware y software. Las transformaciones son realizadas de forma automática por una herramienta denominada PARTIF [Ism94].

Por lo que a cosimulación respecta, una vez que el diseñador ha decidido las partes del sistema que serán implementadas en hardware y software, la especificación SOLAR del sistema es traducida de forma automática a los lenguajes VHDL (hardware) [Vhd93] y C (software). La cosimulación de las especificaciones VHDL y C permite realizar un primer prototipo virtual del sistema. Posteriormente, para sintetizar el diseño deben utilizarse herramientas de síntesis de hardware y compiladores de C.

Finalmente, COSMOS no realiza una verificación formal de las especificaciones SDL. Aunque existen herramientas para verificar especificaciones SDL, normalmente están limitadas a descripciones funcionales que no contemplan las restricciones de tiempo real.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

SPECSYN

Este entorno de codiseño ha sido desarrollado en la Universidad de California (Irvine). SPECSYN [Gaj94] está orientado al diseño de sistemas empotrados (*embedded systems*). La especificación de sistemas está basada en un modelo denominado PSM (*Program-State Machine*) [Gaj93] desarrollado por el propio grupo de investigación. Se trata de un modelo mixto basado en máquinas de estados extendidas (soportan jerarquía y concurrencia) en las que los estados no jerárquicos pueden especificar comportamiento mediante sentencias similares a las de los lenguajes de programación.

Para especificar los sistemas con el modelo PSM, SPECSYN utiliza un lenguaje denominado *SpecCharts* [Nar90][Nar91][Vah91][Nar92a][Nar92b][Gaj93], también desarrollado por el propio grupo de investigación. Este lenguaje está basado en *Statecharts* [Har87][Har88][Dru89][Har90] y en VHDL [Vhd93]. SPECSYN dispone de un compilador para traducir las especificaciones *SpecCharts* al lenguaje VHDL. SPECSYN genera una representación intermedia denominada SLIF (*Specification-Level Intermediate Format*) [Vah95] a partir del código VHDL. Esta representación está basada en un grafo acíclico donde cada nodo representa a un proceso, procedimiento o variable, y cada arco representa un acceso iniciado por el nodo origen sobre el nodo destino. Por ejemplo, un arco puede representar el acceso de un proceso o procedimiento a una variable.

SPECSYN utiliza un modelo de arquitectura basado en: a) un procesador de propósito general para implementar el software, b) uno o varios procesadores específicos (ASIC's) para implementar el hardware, y c) algunas memorias y buses para almacenar los datos y realizar las comunicaciones, respectivamente.

SPECSYN realiza el particionado hardware/software de forma automática [Vah92b][Vah96]. A partir del grafo SLIF, la herramienta de particionado asigna los procesos y procedimientos a los procesadores hardware o software, asigna las variables a las memorias, y asigna los arcos del grafo a los buses. Para realizar el particionado se utilizan diversos algoritmos heurísticos: *clustering* jerárquico [Joh67], migración de grupos [Ker70], y *simulated annealing* [Met53][Kir83][Dow95]. Los algoritmos de particionado minimizan una función de coste que se basa en las estimaciones de diversos parámetros (área y performance), y en las restricciones del diseño (número de *pines* de entrada/salida de los componentes). Las estimaciones son obtenidas mediante una herramienta que anota los resultados en el grafo SLIF [Nar92c][Vah92a][Vah95].

Finalmente, SPECSYN no dispone de herramientas específicas para la cosimulación y la verificación formal.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

COSYMA

COSYMA (*cosynthesis for embedded architectures*) [Ern93] ha sido desarrollado en la Universidad alemana de Braunschweig. Se trata de un entorno orientado al diseño de controladores empotrados (*embedded controllers*).

Argumentando que el lenguaje C es uno de los más utilizados para programar este tipo de sistemas, COSYMA utiliza un lenguaje de especificaciones propio denominado C^x que está basado en el lenguaje C. Los controladores empotrados son sistemas reactivos, es decir, deben reaccionar en tiempo real ante eventos externos asíncronos. El lenguaje C no soporta todas las características necesarias para especificar este tipo de sistemas. Por tanto, C^x es un superconjunto del lenguaje C al que se han añadido principalmente la especificación de restricciones temporales (*delay* mínimo y máximo entre una pareja de sentencias), y la especificación de tareas y comunicaciones entre las mismas.

COSYMA traduce las especificaciones C^x a una representación intermedia denominada ESG (*Extended Syntax Graph*) [Ben93]. Esta representación se compone de dos grafos: por un lado, se utiliza un grafo para conservar la estructura sintáctica de la especificación original. Por otro lado, se utiliza un grafo acíclico para representar las dependencias de datos de cada bloque básico de la especificación. Esta representación interna es simulable.

El modelo de arquitectura que se utiliza en COSYMA está basado en la arquitectura estándar de los microcontroladores, es decir, se utilizan básicamente los siguientes componentes: a) un microprocesador (*core*) programable para ejecutar el software, b) una memoria, y c) uno o varios circuitos de hardware específico (ASIC's y/o FPGA's) que se utilizan como coprocesadores.

COSYMA realiza un particionado hardware/software orientado al software [Ern93], es decir, inicialmente se supone que todo el sistema va a ser implementado en software. Posteriormente, se procede a la migración hacia el hardware de algunas partes de la especificación hasta que se satisfagan las restricciones temporales del diseño. El objetivo es minimizar el hardware específico necesario para satisfacer las mencionadas restricciones. Los motivos principales que se argumentan para seguir esta aproximación son los siguientes: en primer lugar, el software proporciona mayor flexibilidad para realizar futuras modificaciones del sistema. En segundo lugar, las herramientas de síntesis de hardware desde alto nivel no están totalmente maduras, por lo que es conveniente minimizar el hardware sintetizado para evitar la falta de eficiencia de dichas herramientas. En tercer lugar, se argumenta que el hardware de los procesadores

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

estándar es más fiable que el de un coprocesador específico, ya que pasa mayores controles de *test*.

La aproximación de COSYMA al particionado es similar a la publicada en [Ath93], donde se propone acelerar las prestaciones del software ejecutado en un procesador de propósito general (MC68010) mediante la migración hacia hardware específico de aquellas partes que consumen mayor tiempo de ejecución. La selección de estas partes se realiza de forma manual. Sin embargo, COSYMA dispone de una herramienta para realizar el particionado de forma automática. La herramienta selecciona aquellos bloques básicos de la especificación que consumen mayor tiempo de ejecución como candidatos para ser migrados hacia el hardware.

El particionado a nivel de bloques básicos permite que COSYMA soporte especificaciones con recursividad. Así mismo, aquellas funciones que utilizan estructuras de datos dinámicas no son migradas hacia el hardware.

Para identificar los bloques básicos que mayor tiempo consumen, se realiza un proceso de simulación (*profiling*) de la representación intermedia del sistema. Posteriormente, se estiman el *speedup* y la penalización del tiempo de ejecución (debida a las comunicaciones extra) que se obtienen al migrar un bloque básico hacia el hardware. Para el cálculo de dichas estimaciones se utilizan librerías de unidades funcionales así como algoritmos de planificación para los bloques básicos. En particular, se pueden utilizar el algoritmo ASAP (*As Soon As Possible*) y un algoritmo basado en listas (*list-based scheduling*) [Gaj92][Mic92].

Una vez obtenidas las estimaciones, se utiliza un algoritmo de *simulated annealing* [Met53][Kir83][Dow95] para optimizar el particionado. En general, la solución inicial (toda la especificación en software) no satisface las restricciones temporales del diseño, por lo que es incorrecta. Por ello, el proceso de *annealing* utiliza una función de coste con penalizaciones. Aún así, el proceso de *annealing* podría finalizar con soluciones incorrectas, por lo que habría que ejecutarlo nuevamente. Para evitar este problema, se utiliza una función de coste adaptativa [Her94] que puede modificarse en función de las estimaciones obtenidas una vez que el diseño se sintetiza hasta el nivel RT.

Aunque el proceso de particionado es automático, el diseñador puede especificar explícitamente que determinadas funciones no sean migradas hacia el hardware, por ejemplo por motivos de flexibilidad.

Finalmente, COSYMA permite la cosimulación del sistema a nivel RT [Wye93] y dispone de compiladores que generan código C (software) y código *HardwareC* (hardware) a partir de la representación intermedia. El código C es compilado por un

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

compilador de propósito general mientras que el código *HardwareC* es sintetizado utilizando la herramienta de síntesis de alto nivel Olympus [DeM90].

POLIS

El entorno de codiseño POLIS [Bal97] ha sido desarrollado en la Universidad de California (Berkeley) en colaboración con la División Electrónica de Magneti Marelli, uno de los principales fabricantes europeos de sistemas electrónicos para automóviles (control de inyección, ABS, suspensiones, *airbag*, etc.). Por tanto, POLIS es un entorno orientado al diseño de sistemas de control empotrados.

La especificación de un sistema se realiza utilizando un modelo formal propio denominado CFSM (*Codesign Finite State Machines*) [Chi93a][Chi93b][Bal97]. Este modelo está basado en máquinas de estados extendidas que se comunican asíncronamente mediante eventos. El comportamiento de cada máquina de estados se describe en formato textual utilizando el lenguaje síncrono ESTEREL [Ben91][Ber91]. La comunicación asíncrona entre las máquinas de estados se especifica de forma gráfica. POLIS utiliza comunicación asíncrona sin *buffers*, por lo que el diseñador es el responsable de evitar las posibles pérdidas de eventos. La especificación del sistema puede simularse utilizando la herramienta PTOLEMY [Buc93][Buc94]. POLIS genera una representación intermedia denominada SHIFT (*Software Hardware Interchange FormaT*) [Bal97]. Esta representación es una extensión del formato BLIF (*Berkeley Logic Interchange Format*) [Bra91].

POLIS utiliza un modelo de arquitectura que está basado en los sistemas que desarrolla Magneti Marelli [Bal97]. Por un lado, se utiliza uno o varios microcontroladores para la ejecución del software del sistema, y por otro lado se utilizan componentes hardware diseñados específicamente para la aplicación.

El particionado hardware/software se realiza de forma manual [Bal97]. No obstante, el diseñador dispone de una herramienta de cosimulación denominada PTOLEMY [Buc93][Buc94] que le permite analizar los resultados obtenidos para una determinada partición. La herramienta de cosimulación ejecuta código C obtenido a partir de la especificación del sistema. Este código puede ser anotado con estimaciones de parámetros relativos a la implementación que se obtienen una vez que el diseño es sintetizado. Por último, esta herramienta permite generar código VHDL, por lo que se pueden utilizar simuladores VHDL.

Una vez que se realiza la partición hardware/software, POLIS soporta la síntesis de un prototipo del sistema que se ejecuta sobre una tarjeta emuladora basada en un

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

microcontrolador y en varias FPGA's para el hardware específico. La síntesis del hardware se realiza utilizando herramientas propias de síntesis a partir del formato BLIF [Bra91]. La síntesis del software se realiza a partir de un CDFG (*Control-Data Flow Graph*) propio denominado S-GRAPH [Bal97]. Cada nodo del grafo corresponde a un bloque básico de la especificación. A partir del grafo se genera código C mediante un compilador específico. Además, POLIS genera de forma automática un pequeño sistema operativo en tiempo real (RTOS) específico para la aplicación.

Una vez que se sintetiza el sistema, se obtienen estimaciones hardware y software de parámetros tales como tiempo de ejecución, número de bytes, etc. Estas estimaciones son anotadas (*back-annotation*) en el código C que se utiliza para la cosimulación del sistema. De esta forma, el diseñador puede cosimular nuevamente el sistema y comprobar los resultados. En caso de no ser satisfactorios, el diseñador puede realizar modificaciones y realizar una nueva iteración del proceso de diseño.

Por lo que a verificación formal respecta, el modelo de especificación de POLIS permite que el diseñador utilice herramientas disponibles para especificaciones basadas en máquinas de estados. Muchas de estas herramientas están basadas en *model checking* [Alu93][Sri96].

VULCAN

VULCAN [Gup92a][Gup93][Gup94][Gup95] es un entorno de codiseño que ha sido desarrollado en la Universidad de Stanford. Este entorno está orientado al diseño de controladores empotrados.

El lenguaje de especificaciones que se utiliza en VULCAN se denomina *HardwareC* [Ku88][Ku90]. Estas especificaciones están basadas en el lenguaje de programación C, al que se han añadido algunas características necesarias para la especificación de sistemas, tales como las siguientes: a) procesos concurrentes, b) comunicación entre procesos basada en el paso de mensajes, c) restricciones temporales tales como la especificación de tiempo de ejecución mínimo y/o máximo entre cualquier pareja de sentencias. Por otra parte, cabe señalar que *HardwareC* no soporta especificaciones recursivas. VULCAN traduce una especificación descrita en HardwareC a una representación intermedia denominada SIF que está basada en un grafo de flujo de datos jerárquico [Gup95].

Análogamente a la mayoría de los entornos de codiseño, VULCAN utiliza un modelo de arquitectura genérico compuesto principalmente por los siguientes componentes [Gup92a][Gup93][Gup95]: a) un microprocesador de propósito general

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

para ejecutar el software, b) uno o varios ASIC's para implementar el hardware específico, y c) una memoria compartida para almacenar el código y los datos del sistema. Esta memoria contiene un *buffer* para implementar la interfaz entre las particiones hardware y software. Además, cabe señalar que VULCAN supone una jerarquía de memoria de un único nivel, argumentando que es muy complejo obtener estimaciones de parámetros de la implementación teniendo en cuenta una jerarquía de memoria completa. Finalmente, la arquitectura definida en VULCAN utiliza un único *bus* para conectar los componentes anteriormente citados.

VULCAN realiza un particionado hardware/software automático y orientado al hardware [Gup92a][Gup95]. El objetivo de la estrategia de particionado es minimizar el coste del sistema. Inicialmente se supone que toda la especificación va a implementarse en hardware. Posteriormente, mediante un algoritmo heurístico de optimización descendente avaricioso (*greedy*), VULCAN trata de mover operaciones de la partición hardware hacia la partición software. Esta traslación de operaciones se produce mientras se satisfagan las restricciones temporales impuestas por el diseñador. El algoritmo utilizado trata de optimizar una función de coste que se basa en una serie de parámetros estimados a partir del grafo de la representación intermedia. Estos parámetros están relacionados con el *performance* y coste de la implementación del sistema.

Una vez que se obtienen las particiones hardware y software, VULCAN soporta la síntesis de ambas particiones [Gup95]. Por un lado, para la síntesis del hardware se utiliza la herramienta de síntesis de alto nivel OLYMPUS [DeM90]. Por otro lado, para la síntesis del software se utiliza un compilador que genera código C a partir del grafo SIF. Este compilador genera un código basado en la ejecución concurrente de *threads* [Gup95]. Posteriormente, el código C es traducido al lenguaje ensamblador de la máquina DLX [Pat90]. Finalmente, el hardware sintetizado y el código ensamblador son cosimulados mediante la herramienta POSEIDON [Gup92b].

2.5. Entorno de codiseño GACSYS

Actualmente, nuestro grupo de investigación está desarrollando un entorno de codiseño denominado GACSYS (*GAC's Codesign System*) [Cas94][Cas96][Cas97a][Cas97b] [Cas97c][Cas98a][Cas98b][Cas98c]. La metodología de GACSYS se muestra en la Figura 2.4. Las diversas partes del entorno se han realizado en el ámbito de este trabajo, unas como resultado directo de los objetivos planteados, y otras como necesidades para el desarrollo de los mismos. En concreto, se han diseñado y desarrollado las siguientes partes:

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

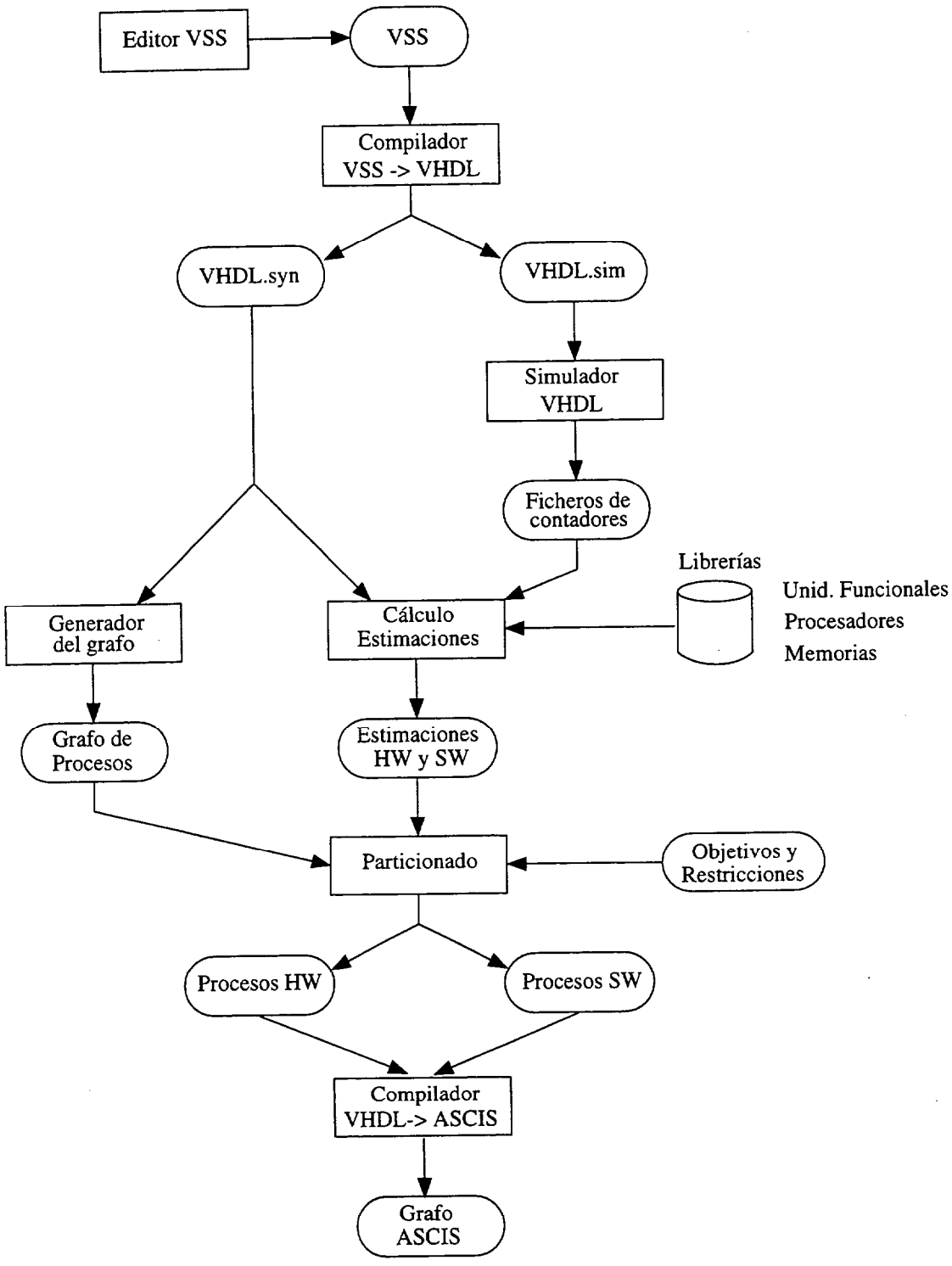


Figura 2.4. Sistema de codiseño GACSYS.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

- Un lenguaje de especificación denominado VSS (*VHDL-based System Specification*) que permite una codificación sencilla y soporta primitivas con un nivel de abstracción adecuado para especificar sistemas. Además, se ha desarrollado un editor orientado a los lenguajes VSS y VHDL, así como un compilador que genera código VHDL a partir de VSS.
- Un método automático de particionado que permite una mejor exploración del espacio de diseño. Este método soporta la segmentación de los grafos de procesos, y utiliza estimaciones de diversos parámetros relativos a la implementación de un sistema, tales como tiempo de ejecución, área y potencia.
- Una representación intermedia basada en un grafo de flujo de datos denominado ASCIS. Esta representación ha sido ampliada para soportar algunas sentencias de VHDL tales como *wait* y la asignación a señales. Además, se ha desarrollado un compilador que genera la representación intermedia a partir de código VHDL.

Los pasos que deben seguirse para realizar un diseño con nuestra metodología son los siguientes:

- Primeramente, el diseñador debe especificar el sistema utilizando el lenguaje VSS. Para ello dispone de un editor orientado a dicho lenguaje.
- La especificación debe ser traducida a VHDL para su simulación. De esta forma el diseñador puede corregir los posibles errores.
- Una vez que la especificación es correcta, el diseñador debe simular la misma una o varias veces para obtener información relativa al número de veces que se ejecuta cada bloque básico de la especificación. De esta forma se generan los denominados ficheros de contadores.
- Posteriormente, dado un conjunto de librerías de unidades funcionales, procesadores y memorias, el diseñador debe obtener las estimaciones hardware y software de un conjunto de parámetros relativos a la implementación del sistema, tales como tiempo de ejecución, área y potencia.
- Seguidamente, dado el grafo de procesos de la especificación, las estimaciones obtenidas anteriormente, y los objetivos y restricciones del diseño, el diseñador debe realizar el proceso de particionado hardware/ software.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

- Finalmente, el diseñador puede generar una representación intermedia basada en un grafo de flujo de datos denominado ASCIS. A partir de este grafo se puede sintetizar el hardware y software del diseño, bien mediante herramientas existentes o bien mediante el diseño de herramientas propias.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

Capítulo 3

Lenguajes de Especificación y Particionado Hw/Sw

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

3.1. Introducción

El primer objetivo de este trabajo es diseñar un lenguaje de especificaciones denominado VSS (*VHDL-based System Specification*) para solucionar una serie de problemas del lenguaje VHDL. El segundo objetivo es diseñar un método automático de particionado hardware/software que soporte la ejecución segmentada de grafos de procesos y que considere la potencia consumida por el diseño.

El objetivo principal de este capítulo es presentar el trabajo previo que se ha realizado sobre lenguajes de especificación de sistemas y particionado hardware/ software. De esta forma se justifican con mayor profundidad los objetivos planteados en este trabajo. Además, en este capítulo se presenta una introducción a los algoritmos básicos de planificación que se utilizan en nuestro método de particionado.

Para conseguir estos objetivos, el contenido de este capítulo es el siguiente: en el apartado 2, se presentan las características básicas de los lenguajes de especificación de sistemas. En el apartado 3, se presentan los lenguajes más utilizados en los trabajos publicados y se justifica la necesidad de ampliar el lenguaje VHDL identificando algunos problemas del mismo. En el apartado 4, se presentan los trabajos más importantes que se han publicado sobre particionado hardware/software. De esta forma se identifican algunas carencias que justifican el método de particionado propuesto en este trabajo. Finalmente, en el apartado 5, se presentan los algoritmos de planificación que se utilizan en nuestro método de particionado: ASAP, ALAP, planificación basada en listas y *simulated annealing*.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

3.2. Características básicas para la especificación de sistemas

En general, los lenguajes de especificación de sistemas deben soportar una serie de características básicas [Nar92a][Nar93] tales como: jerarquía de comportamiento, jerarquía estructural, concurrencia, comunicación, sincronización, temporización, manejo de excepciones y no-determinismo.

3.2.1. Jerarquía de comportamiento y estructural

La primera concepción que tiene un diseñador de un sistema está basada normalmente en un diagrama de bloques interconectados entre sí. Inicialmente, el diseñador define el comportamiento de cada bloque, aunque la implementación de los mismos esté aún por definir. En una segunda fase, el diseñador realiza un proceso de refinamiento en el que cada uno de los bloques se descompone en unidades cuyo comportamiento y/o

estructura se definen con un grado inferior de abstracción. Por lo tanto, los lenguajes de especificación de sistemas deben soportar:

- *Jerarquía de comportamiento*: el lenguaje permite describir el comportamiento del sistema como un conjunto de actividades jerárquicas que se realizan secuencialmente y/o concurrentemente. Además, cada actividad puede ser iniciada o interrumpida en cualquier instante (activación y desactivación jerárquica).
- *Jerarquía estructural*: el lenguaje permite describir la estructura del sistema como un conjunto de componentes interconectados entre sí. A su vez, estos componentes pueden definirse mediante la interconexión de nuevos componentes.

3.2.2. Concurrencia, comunicación y sincronización

En cualquiera de sus niveles jerárquicos, un sistema se compone de un conjunto de módulos concurrentes que se comunican y sincronizan. Por consiguiente, un lenguaje de especificación de sistemas debe soportar primitivas para especificar la concurrencia, comunicación y sincronización.

La concurrencia puede especificarse mediante primitivas de diferentes niveles de granularidad, tales como procesos, procedimientos o sentencias.

Los métodos de comunicación pueden clasificarse principalmente en dos clases: a) paso de mensajes, y b) memoria compartida. En el caso de paso de mensajes, se utilizan canales o puertos de comunicación entre los módulos concurrentes. Los canales o puertos pueden ser de diferentes clases en función de parámetros tales como: a) la sincronización (síncronos o asíncronos), b) la dirección (unidireccionales o bidireccionales), y c) el número de emisores y receptores asociados a un canal o puerto (punto a punto, *broadcast*, *incast*, *multicast*). La comunicación entre los diversos módulos se establece mediante sentencias de lectura y escritura de la información en dichos canales o puertos. En el caso de memoria compartida, se utilizan variables compartidas y mecanismos de exclusión mutua tales como semáforos, regiones críticas y monitores para evitar los conflictos de acceso a las variables entre los diversos módulos concurrentes.

En general, los módulos concurrentes de un sistema necesitan sincronizar algunas actividades. Por ejemplo, la comunicación entre dos procesos a través de un canal sin memoria necesita que los mismos sincronicen la transmisión y recepción de los datos. En general, los métodos de sincronización pueden clasificarse de la siguiente forma:

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

- *Sincronización mediante inicialización*: la inicialización de un módulo lleva consigo la inicialización de todos sus módulos secuenciales y/o concurrentes.
- *Sincronización mediante eventos*: un módulo puede permanecer inactivo a la espera de un evento que le permita iniciar o continuar su ejecución.
- *Sincronización mediante estados*: un módulo inicia o continúa su ejecución como consecuencia de que otros módulos están en un determinado estado.
- *Sincronización mediante comunicación:* los métodos de comunicación, ya sea mediante variables compartidas o mediante paso de mensajes, también pueden utilizarse para la sincronización entre módulos.

3.2.3. Temporización

La especificación de un sistema requiere en muchos casos la incorporación de información temporal. Este tipo de información temporal puede hacer referencia a:

- *restricciones temporales*, que especifican una serie de condiciones temporales que cualquier implementación del sistema debe verificar para que su funcionamiento sea correcto. Normalmente expresan el mínimo, o máximo, intervalo de tiempo que puede transcurrir entre la ejecución de dos sentencias de la especificación.
- *temporizaciones y retardos*, que especifican la activación de algún temporizador, activándose algún tipo de excepción si el mismo finaliza su cuenta sin que se haya producido algún evento determinado. Los retardos son un tipo particular de temporización que permite mantener inactivo a un módulo durante un tiempo determinado.

3.2.4. Manejo de excepciones

En general, la especificación de un sistema necesita primitivas para que los módulos del mismo puedan responder ante determinados eventos (excepciones) mediante la suspensión del flujo de control y la transferencia del mismo hacia las rutinas de tratamiento correspondientes. En general, una vez que se han ejecutado dichas rutinas, el sistema recupera el flujo de control previo a las excepciones.

El tratamiento de las excepciones es normalmente jerárquico, es decir, cada excepción tiene asociado un nivel de prioridad. De esta forma, una excepción puede interrumpir el tratamiento de otra excepción de menor prioridad.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

3.2.5. No determinismo

Una sentencia es determinista cuando el estado del sistema después de su ejecución es único y está totalmente determinado por el estado del mismo antes de la ejecución de dicha sentencia. En un principio, la totalidad de las sentencias de los lenguajes eran deterministas. Sin embargo, Dijkstra [Dij75] introdujo por primera vez el concepto de sentencias no deterministas a las que se añade una guarda, que es una expresión booleana que controla si una sentencia puede ser seleccionada para su ejecución. Dijkstra propuso una sentencia no determinista que permitía seleccionar entre varias sentencias con guardas. La semántica de dicha sentencia consistía en ejecutar alguna de las sentencias cuya guarda fuese cierta, en cuyo caso se seleccionaba una de las sentencias de forma aleatoria.

Este tipo de sentencias permite modelar sistemas cuyo comportamiento es correcto independientemente de las decisiones aleatorias que se tomen en ciertas partes de su especificación. De esta forma, un diseñador puede retrasar algunas decisiones dependientes de la implementación. Por ejemplo, supongamos que estamos especificando un sistema cliente-servidor. El orden en que se atienden las peticiones de los clientes podría ser especificado con sentencias no deterministas. De este modo, los detalles relativos a la política de prioridades seguida por el servidor pueden decidirse en las etapas posteriores del desarrollo del sistema.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

3.3. Lenguajes de especificación para codiseño

En este apartado se presentan los lenguajes más importantes que se utilizan en los entornos de codiseño. Existen diversos lenguajes para especificar sistemas. Cada lenguaje de especificación ha sido diseñado en función de una serie de necesidades derivadas principalmente del tipo de aplicaciones y de las actividades de codiseño (simulación, síntesis y verificación).

En función del tipo de las aplicaciones se puede diferenciar principalmente entre los lenguajes orientados a las aplicaciones de control (*control dominated*) y los orientados a las aplicaciones dominadas por los datos (*data dominated*). En los primeros, la especificación del comportamiento del sistema está normalmente basada en máquinas de estados, mientras que en los segundos se utilizan descripciones similares a los lenguajes de programación basados en múltiples procesos secuenciales.

3.3.1. Lenguajes basados en máquinas de estados

Los lenguajes basados en máquinas de estados describen el comportamiento de un sistema mediante la especificación de un conjunto de máquinas de estados finitos. Normalmente, estas máquinas de estados finitos son ampliadas para soportar características tales como concurrencia, comunicación, sincronización, jerarquía, etc. Algunos de los lenguajes más utilizados son los siguientes: *StateCharts*, *SpecCharts*, SDL y ESTELLE. Este tipo de lenguajes es adecuado para las aplicaciones dominadas por el flujo de control (*control dominated*). A continuación se presentan las características más relevantes de cada uno de los lenguajes mencionados.

StateCharts

StateCharts [Har87][Dru89] es un lenguaje que soporta la especificación de máquinas de estados jerárquicas y concurrentes que se comunican entre sí. Un estado puede descomponerse en estados paralelos (descomposición AND) o estados secuenciales (descomposición OR). Los estados paralelos se utilizan para definir las actividades concurrentes del sistema. Las acciones del sistema se especifican en los estados y en las transiciones entre estados. La comunicación entre estados se realiza mediante un *broadcast* de todas las señales, estados y variables del sistema. El lenguaje soporta métodos de sincronización mediante inicialización, estados y eventos.

El lenguaje permite transiciones entre estados pertenecientes a diferentes niveles de la jerarquía, y proporciona una transición especial denominada *timeout*. *StateCharts* también soporta la especificación del tiempo mínimo o máximo que el sistema puede permanecer en un determinado estado. Las excepciones se especifican mediante transiciones entre estados.

Statemate [Har88][Har90] es el entorno de codiseño que inicialmente propuso el lenguaje *StateCharts*.

SpecCharts

SpecCharts [Nar91][Vah91][Nar92b] está basado en los lenguajes *Statecharts* y VHDL [Vhd93]. Un sistema se describe como una máquina de estados jerárquica en la que cada nodo hoja puede especificar su comportamiento mediante sentencias secuenciales de VHDL. *SpecCharts* soporta comunicaciones mediante paso de mensajes a través de canales. Las transiciones sólo pueden realizarse entre estados de un mismo nivel jerárquico. Además de las transiciones heredadas de *StateCharts*, *SpecCharts* soporta nuevos tipos de transiciones entre estados:

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

- *TOC (transition on completion)*: se realiza cuando finaliza el procesamiento del estado actual.
- *TI (transition inmediatly)*: se realiza cuando se produce el evento al que la transición es sensible, independientemente del punto de procesamiento en que se encuentra el estado actual. Esta transición permite la especificación de excepciones.

SpecCharts se utiliza en el entorno de codiseño denominado SPECSYN [Gaj94][Gaj95]. Este entorno dispone de un compilador que genera código VHDL a partir de especificaciones *SpecCharts* [Nar90].

SDL y ESTELLE

SDL (*Specification and Description Language*) [Tur93] es un lenguaje estándar diseñado principalmente para sistemas de telecomunicación. Su semántica está formalmente definida, por lo que es adecuado para la utilización de técnicas de verificación formal. SDL describe un sistema mediante un conjunto de bloques interconectados entre sí a través de canales. Cada bloque puede describirse como un nuevo diagrama de bloques o como una máquina de estados. En las transiciones entre estados pueden ejecutarse una serie de acciones tales como modificación de señales, modificación de variables, y operaciones de *set* y *reset* sobre temporizadores.

La comunicación entre máquinas de estados se realiza mediante paso de mensajes, y responde a un esquema asíncrono con *buffer* ilimitado. La sincronización se realiza a través de señales globales. SDL también soporta el tratamiento de excepciones.

SDL es utilizado en entornos de codiseño tales como COSMOS [Ism95] y [Dou97]. Existen algunos trabajos que proponen traducciones de SDL a VHDL [Ism95][Fig95].

ESTELLE [Est87] es un lenguaje estándar muy parecido a SDL. Realmente, ESTELLE fue diseñado antes que SDL, aunque en la actualidad ha sido desplazado por este último. También existen traductores de ESTELLE a VHDL [Wyt95].

3.3.2. Lenguajes basados en múltiples procesos secuenciales

Los lenguajes basados en múltiples procesos secuenciales describen el comportamiento de un sistema mediante un conjunto de procesos secuenciales concurrentes que se comunican y sincronizan entre sí. Este tipo de lenguajes es adecuado para aplicaciones dominadas por los datos. Entre los lenguajes utilizados en los entornos de codiseño podemos citar principalmente los siguientes: *HardwareC*, OCCAM, LOTOS y VHDL. A continuación se presentan las características principales de cada uno de ellos.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

HardwareC

HardwareC [Ku88][Ku90] describe un sistema como un conjunto de procesos concurrentes especificados mediante un subconjunto de sentencias del lenguaje C y un conjunto de nuevas sentencias que permiten la comunicación entre los procesos y la especificación de restricciones temporales. Todas las operaciones de un proceso se ejecutan de forma sincronizada mediante un reloj de sistema, consumiendo cada operación un número determinado de ciclos de reloj. Cada proceso se ejecuta nuevamente desde el comienzo cada vez que termina su ejecución. Varios procesos se pueden agrupar en una entidad jerárquica superior llamada bloque.

La comunicación y sincronización entre los procesos se realiza mediante paso de mensajes, y se utiliza comunicación síncrona.

El diseñador puede especificar tres grados de concurrencia en la ejecución de las sentencias de un proceso: ejecución en serie, ejecución en paralelo, y ejecución en paralelo respetando las dependencias de datos.

HardwareC permite especificar retardos máximos y mínimos entre la ejecución de dos sentencias. El lenguaje también soporta el tratamiento de excepciones, pero su codificación es compleja [Gaj95].

Este lenguaje se utiliza en el entorno de codiseño VULCAN [Gup95] desarrollado en la Universidad de Stanford.

OCCAM

OCCAM [Pou87] es un lenguaje de programación concurrente basado en CSP (*Communicating Sequential Processes*) [Hoa85]. Un sistema se describe en OCCAM mediante una jerarquía de procesos secuenciales y paralelos. Por ello, los procesos se comunican y sincronizan a través de canales punto a punto síncronos y unidireccionales. La declaración de cada canal se acompaña de un protocolo de comunicación. Este protocolo indica el tipo y la secuencia de datos que transporta un canal. El lenguaje soporta sentencias para la transmisión y recepción de datos por los canales. OCCAM también soporta una sentencia no determinista (*Alt*) que permite seleccionar y ejecutar una acción de entre un conjunto de acciones precedidas de una guarda.

El lenguaje permite declarar temporizadores. Un temporizador incrementa cada cierto tiempo el valor de una variable entera. Cualquier proceso puede consultar el valor de dicha variable mediante sentencias de comunicación, así como esperar que se produzca un cierto número de incrementos en dicha variable. De este modo se permite la especificación de retardos en el diseño.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

OCCAM ha sido utilizado como lenguaje de especificación en trabajos como [Bar94][The97].

LOTOS

LOTOS [Lot89][Tur93] es un lenguaje de especificación que describe un sistema mediante su comportamiento externo observable. Se trata de un lenguaje estándar desarrollado para la especificación formal de protocolos de comunicación. LOTOS describe un sistema mediante un álgebra de procesos basada en CSP [Hoa85] y CCS [Mil89], y mediante tipos abstractos de datos. LOTOS dispone de varios operadores para expresar concurrencia. El comportamiento de un proceso se especifica mediante secuencias de eventos. Los procesos se comunican y sincronizan a través de puertas mediante un protocolo síncrono. El lenguaje soporta tratamiento de excepciones y sentencias no deterministas. Sin embargo, LOTOS no soporta información temporal.

Entre las publicaciones que lo utilizan podemos citar a [Klo95][Car96][Wol97]. La implementación de una especificación LOTOS es compleja, aunque en algunos se realiza una traducción restringida desde LOTOS a VHDL [Klo95][San98].

VHDL

VHDL [Lip93][Vhd93] es un lenguaje estándar de descripción de hardware desde 1987. VHDL fue diseñado inicialmente para la simulación y documentación de los diseños. Sin embargo, durante los últimos años su utilización se ha expandido hacia el resto de las actividades de codiseño tales como síntesis y verificación.

El comportamiento de un sistema se describe mediante un conjunto de procesos concurrentes que se comunican y sincronizan a través de señales globales. Cada proceso puede declarar sus propias variables locales. Un proceso ejecuta un conjunto de sentencias secuenciales, y comienza nuevamente su ejecución cada vez que se ejecuta la última sentencia del mismo. VHDL dispone de las sentencias típicas de cualquier lenguaje de programación secuencial. Además soporta dos sentencias específicas para la comunicación y sincronización entre procesos: la sentencia *wait* y la sentencia de asignación a señales. VHDL también soporta la especificación de jerarquía estructural.

La ejecución de una especificación VHDL se basa en un ciclo de simulación cuyas fases son las siguientes: *a) Primera fase*: todos los procesos se ejecutan hasta llegar a una sentencia *wait*. Cuando un proceso alcanza esta sentencia queda suspendido hasta que se cumple la condición de salida especificada en la misma. *b) Segunda fase*: una vez que todos los procesos han alcanzado una sentencia *wait* se procede a la actualización de

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

las señales, es decir, cualquier sentencia de asignación a señal que pudiera haberse ejecutado no modifica el valor de la señal hasta que se ejecuta una sentencia *wait. c) Tercera fase*: se avanza el tiempo de simulación, continuando la ejecución de aquellos procesos que satisfacen la condición de su sentencia *wait*. El resto de procesos queda esperando hasta el siguiente ciclo de simulación.

VHDL soporta excepciones, pero deben ser codificadas completamente por el diseñador. VHDL se utiliza como lenguaje de especificación o bien como lenguaje para la síntesis de hardware en trabajos tales como [Ele94][Gaj95][Ism95][Klo95][Eck97] [Car98][Rad98][Ash98].

VHDL es uno de los lenguajes de especificación de sistemas más utilizado debido principalmente a los siguientes motivos:

- VHDL soporta muchas de las características necesarias para la especificación de sistemas: jerarquía para especificar el comportamiento, jerarquía estructural, concurrencia, comunicación, sincronización, temporización y manejo de excepciones.
- VHDL permite que el diseñador represente los sucesivos refinamientos de un diseño, es decir, soporta el diseño *top-down* desde el nivel de sistema hasta el nivel lógico. Este aspecto permite, por un lado, reflejar los sucesivos resultados de los procesos de síntesis en el mismo lenguaje, y por otro lado, facilita la cosimulación del sistema en todos los niveles de abstracción
- VHDL facilita las tareas de codiseño, ya que es simulable, sintetizable (exceptuando algunas sentencias) y permite la aplicación de técnicas de verificación formal. Aunque VHDL no está formalmente definido, se han publicado una serie de trabajos orientados a la verificación formal en VHDL, tales como [Wil92][Tas93][Ard95][Bre95][Deh95][Goo95].
- La distancia semántica entre VHDL y las posibles implementaciones no es muy grande. Prueba de ello es que existen bastantes herramientas de síntesis basadas en VHDL. La distancia semántica de una especificación respecto a una posible implementación es un indicativo del número de decisiones de diseño que están implícitamente fijadas por la propia especificación.
- VHDL es un lenguaje estándar ampliamente utilizado, existiendo un amplio número de herramientas de diseño en el mercado. Esto favorece la implementación de los sistemas y la interoperabilidad entre las diferentes herramientas.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

Sin embargo, VHDL tiene las siguientes desventajas:

- La jerarquía de comportamiento es limitada, ya que no soporta la especificación de un proceso mediante una jerarquía de procesos. Es decir, un proceso secuencial de VHDL no puede ser descrito mediante una jerarquía de procesos secuenciales y/o concurrentes.
- El manejo de excepciones, la activación y desactivación jerárquica de procesos, la temporización, y las comunicaciones entre procesos necesitan especificaciones complejas, extensas y tediosas. Por tanto, el diseñador necesita un mayor tiempo de diseño y tiende a cometer errores con facilidad.
- El comportamiento de las variables compartidas no es determinista, ya que depende del orden en que el planificador (*scheduler*) de cada simulador de VHDL ejecute los procesos concurrentes. En definitiva, desde el punto de vista del diseñador no se soportan las variables compartidas.

En este trabajo se propone una ampliación de VHDL para resolver estos problemas. El lenguaje resultante de esta ampliación se denomina VSS y será presentado en el próximo capítulo.

3.4. Particionado hardware/software

Los trabajos más importantes que se han publicado sobre particionado hardware/ software pueden clasificarse atendiendo a múltiples aspectos. En este apartado se han realizado diversas clasificaciones atendiendo a los siguientes puntos: tipo de aplicaciones, modelo de representación, arquitectura, granularidad, estrategia, algoritmos, y parámetros considerados.

En primer lugar, los trabajos pueden clasificarse en dos grupos atendiendo al tipo de aplicaciones: a) orientados hacia aplicaciones dominadas por el control (*control dominated*) [Ern93][Bal96][Car96], y b) orientados hacia aplicaciones dominadas por los datos (*data dominated*) [Bec96][Ben96][Knu96].

En segundo lugar, los trabajos pueden clasificarse en tres grupos según el modelo de representación: a) trabajos que utilizan un grafo DAG (*Directed Acyclic Graph*) de procesos [Kal94][Ben96][Wol96], b) trabajos que utilizan un grafo CDFG (*Control and Data Flow Graph*) [Knu96], y c) trabajos que utilizan algún otro tipo de grafo [Ern93][Pen93][Ele94][Vah96].

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

En tercer lugar, los trabajos pueden clasificarse en dos grupos atendiendo al modelo de arquitectura: a) trabajos que utilizan un único procesador programable, y b) aquellos que utilizan varios procesadores programables [DAm94][Ben96][Wol96]. A su vez, los primeros pueden clasificarse en dos grupos: aquellos que utilizan el hardware específico como coprocesador/es [Ern93][Ele94][Bal96], y aquellos otros que lo utilizan como procesador/es paralelo/s [Gup93][Bar94][Kal94][Vah95b][Car96].

En cuarto lugar, los trabajos pueden clasificarse en cinco grupos según la granularidad que utilizan: a) granularidad a nivel de proceso [Ben96][Car96][Vah96] [Wol96][Vah97], b) granularidad a nivel de subprograma [Ath93][The94], c) granularidad a nivel de bloque básico [Ern93][Gup93][Bec96] [Knu96], d) granularidad a nivel de sentencia u operación [McF90][Lag91][Pen93], e) granularidad mixta, es decir, se soportan varios niveles [Bar94][Ele94][Ism94][Bal96][Hen97].

En quinto lugar, los trabajos pueden ser clasificados en dos grupos según la estrategia de particionado que utilizan: a) particionado manual/interactivo [Lag91][Ath93][Ism94][The94], y b) particionado automático. Estos últimos pueden ser clasificados en tres grupos según la orientación del particionado: a) métodos de orientación software, que inicialmente suponen que todo el sistema se implementa en software, y posteriormente realizan una migración hacia el hardware de algunas partes del sistema mientras no se verifiquen los requerimientos del diseñador [Ern93][Bec96][Ben96][Har96]. b) Métodos de orientación hardware, que inicialmente suponen que todo el sistema se implementa en hardware, y posteriormente realizan una migración hacia el software de algunas partes del sistema mientras se verifiquen los requerimientos del diseñador [Gup93][Kal93][Buc94]. c) Métodos mixtos, en los que no se presupone una distribución orientada hacia el software ni hacia el hardware [DAm94][Ele94][Car96][Knu96][Vah96].

En sexto lugar, los trabajos pueden clasificarse en dos grupos atendiendo a los algoritmos de optimización que utilizan: a) métodos que utilizan algoritmos que proporcionan soluciones óptimas, tales como ILP (*Integer Linear Programming*) [Ben96][Nie96][Nie97], *branch & bound* [DAm94], y programación dinámica [Jan94][Knu96][Mad97]. b) Métodos que utilizan algoritmos heurísticos basados en técnicas tales como *clustering* [McF90][Lag91][Bar94][Vah96][Wol96], migración de grupos [Car96][Vah96][Agr97][Vah97], *simulated annealing* [Ern93][Ele94][Ben96] [Vah96][Hen97], o bien heurísticas *ad-hoc* [Kal94][Kal97].

Finalmente, los trabajos pueden clasificarse en dos grupos según los parámetros considerados para optimizar el particionado: a) trabajos que sólo consideran el tiempo

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

de ejecución del diseño [Ern93][Ele94][The94][Ben96]. b) Trabajos que tienen en cuenta el tiempo de ejecución y el área del diseño [Gup93][Kal94][Bal96][Vah96] [Wol96].

Los trabajos anteriores presentan algunas carencias: en primer lugar, no consideran la ejecución segmentada de grafos DAG (*Directed Acyclic Graph*) de procesos durante la exploración del espacio de diseño [Kal94][Ben96][Wol96]. Algunos trabajos mencionan dicha posibilidad como trabajo futuro [Car96], o bien proponen la segmentación a nivel interno de un proceso (*software pipelining*) [Bak97]. En segundo lugar, la potencia consumida por el diseño no es considerada entre los parámetros utilizados para optimizar el particionado. Sin embargo, se trata de un parámetro cada vez más importante debido a la expansión de sistemas portátiles y a la densidad de integración de los circuitos integrados. Estas dos carencias limitan el espacio de diseño, y por tanto la posibilidad de obtener particiones más optimizadas. Por tanto, se concluye que existe la necesidad de diseñar e implementar métodos de particionado hardware/ software que consideren ambos aspectos.

3.5. Algoritmos de planificación

El método de particionado hardware/software que se propone en este trabajo necesita la evaluación previa de las estimaciones de una serie de parámetros del diseño tales como tiempo de ejecución, área y potencia. La evaluación de dichas estimaciones utiliza un algoritmo para planificar las operaciones de los procesos.

La planificación de las operaciones de un proceso es un problema de optimización combinatoria [Gaj92][Mic92][Dow95] cuyos principales parámetros son el número de pasos de control necesarios para ejecutar el proceso (tiempo de ejecución) y el coste de los recursos necesarios (área y potencia consumidas por los recursos).

En general, la planificación puede plantearse de diversas formas en función de los objetivos a optimizar. Básicamente, se pueden plantear dos tipos de planificación:

- *planificación para optimizar el coste de los recursos*: el número de pasos de control se fija inicialmente, y el objetivo es minimizar el coste de los recursos necesarios.
- *planificación para optimizar el tiempo de ejecución*: el coste máximo de los recursos se fija inicialmente, y el objetivo es minimizar el número de pasos de control. Este tipo de planificación es el que necesita nuestro método de particionado para la evaluación de las estimaciones.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

Existen dos formas básicas de resolver este tipo de problemas de optimización combinatoria [Gaj92][Mic92][Dow95]:

- Utilizar técnicas que proporcionan soluciones óptimas o cuasi-óptimas tales como la exploración exhaustiva y la programación lineal entera, respectivamente. La desventaja de este tipo de técnicas es que consumen un elevado tiempo de computación.
- Utilizar técnicas heurísticas que proporcionan soluciones *satisfactorias*, y que necesitan muy poco tiempo de computación en comparación con las técnicas anteriores. La desventaja de este tipo de técnicas es que no garantizan la obtención de una solución óptima. Estas técnicas son las más utilizadas para el problema de la planificación en una gran mayoría de los entornos de automatización de diseño.

A continuación se presentan las ideas básicas de los algoritmos que se utilizan en nuestro método de particionado hardware/software.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

3.5.1. Algoritmos de planificación ASAP y ALAP

En este apartado se presentan los algoritmos ASAP (*As Soon As Possible*) y ALAP (*As Late As Possible*) [Gaj92][Mic92].

Algoritmo de planificación ASAP

Dado el grafo acíclico (GA) de las operaciones de un bloque básico de un proceso, el algoritmo de planificación ASAP asigna a cada operación del GA el primer paso de control en el cual dicha operación puede ser planificada. Los pasos básicos de este algoritmo (Algoritmo 3.1) son los siguientes:

- *Paso 1*: planificar en el primer paso de control todas las operaciones que no tienen predecesoras.
- *Paso 2*: planificar aquellas operaciones cuyas predecesoras inmediatas han sido ya planificadas. Este paso debe repetirse hasta que se hayan planificado todas las operaciones.

La función PREDECESORAS retorna el conjunto de operaciones predecesoras inmediatas de una operación. La función PLANIFICADAS comprueba si todas las operaciones predecesoras inmediatas de una operación determinada han sido ya planificadas, retornando verdadero si están todas planificadas, o falso en caso contrario. La función MAXIMO_PASO, dado un conjunto de operaciones ya planificadas,

```
O= { conjunto de operaciones o_i del GA }
for cada operación o_i do
    if PREDECESORAS (o_i) = ϕ then
        paso (o_i) = 1
        O = O– {o_i}
    endif
endfor
while O <> ϕ do
    for cada operación o_i do
        if PLANIFICADAS (PREDECESORAS (o_i)) then
            paso (o_i) = MAXIMO_PASO (PREDECESORAS (o_i)) + 1
            O = O – {o_i}
        endif
    endfor
endwhile
```

Algoritmo 3.1. Algoritmo de planificación ASAP.

proporciona el mayor paso de control en que se han planificado las operaciones de dicho conjunto.

Algoritmo de planificación ALAP

El algoritmo de planificación ALAP asigna a cada operación de un GA el último paso de control en el cual dicha operación puede ser planificada. Los pasos básicos de este algoritmo (Algoritmo 3.2) son los siguientes:

- *Paso 1*: planificar en el último paso de control a todas las operaciones que no tienen sucesoras.
- *Paso 2*: planificar aquellas operaciones cuyas sucesoras inmediatas han sido ya planificadas. Este paso debe repetirse hasta que se hayan planificado todas las operaciones.

La función SUCESORAS retorna el conjunto de operaciones sucesoras inmediatas de una operación. La función PLANIFICADAS comprueba si todas las operaciones sucesoras inmediatas de una operación determinada han sido ya planificadas, retornando verdadero si están todas planificadas, o falso en caso contrario. La función

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

```
O = { conjunto de operaciones o_i del GA }
for cada operación o_i do
    if SUCESORAS (o_i) = ϕ then
        paso (o_i) = ultimo_paso
        O = O – {o_i}
    endif
endfor
while O <> ϕ do
    for cada operación o_i do
        if PLANIFICADAS (SUCESORAS (o_i)) then
            paso (o_i) = MINIMO_PASO (SUCESORAS (o_i)) - 1
            O = O – {o_i}
        endif
    endfor
endwhile
```

Algoritmo 3.2. Algoritmo de planificación ALAP.

MINIMO_PASO, dado un conjunto de operaciones ya planificadas, proporciona el menor paso de control en el que se han planificado las operaciones de dicho conjunto.

En las Figuras 3.1 y 3.2 se muestra un ejemplo de un GA y las correspondientes planificaciones ASAP y ALAP, respectivamente. Se observa claramente que la planificación ASAP tiene una tendencia a utilizar un mayor número de recursos en los primeros pasos de control, mientras que en la planificación ALAP sucede lo contrario.

3.5.2. Planificación basada en listas *(list-based)*

En este apartado se presenta un algoritmo basado en listas denominado *list-based* [Gaj92][Mic92]. Dado un número limitado de recursos, el objetivo del algoritmo de planificación *list-based* [Gaj92][Mic92] es minimizar el número de pasos de control necesarios para la planificación de un GA. Este algoritmo alcanza su objetivo planificando en cada paso de control el mayor número de operaciones posibles. Básicamente, se trata de una generalización del algoritmo de planificación ASAP. En particular, el resultado de una planificación ASAP y una planificación *list-based* es el mismo cuando se dispone de un número ilimitado de recursos. Este algoritmo se utiliza en trabajos como [The94][Knu96].

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

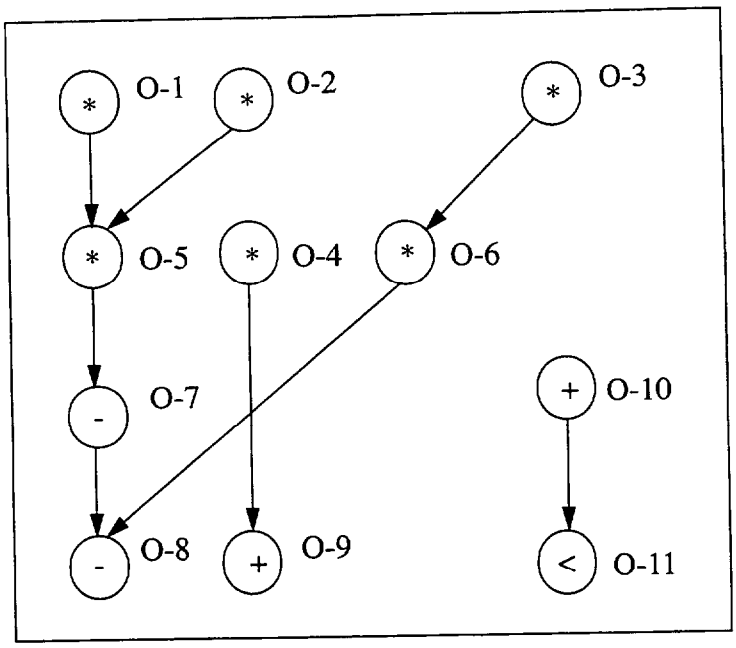


Figura 3.1. Grafo de flujo de datos acíclico.

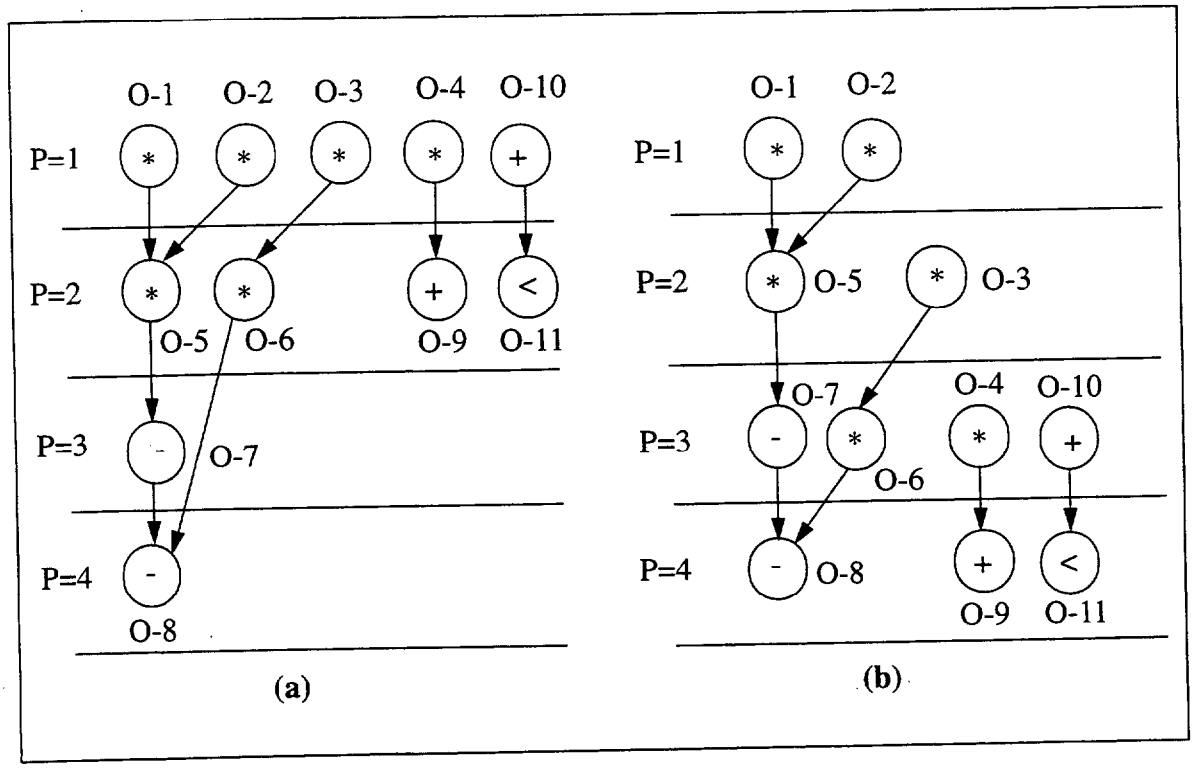


Figura 3.2. Planificaciones (a) ASAP y (b) ALAP.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

El algoritmo de planificación *list-based* tiene como características más importantes las siguientes:

- *es heurístico*, es decir, no garantiza la obtención de un resultado óptimo. El criterio heurístico que se utiliza está basado en la *urgencia de una operación*. Los recursos disponibles en un paso de control son asignados a aquellas operaciones con mayor *urgencia*. Un grado alto de *urgencia* en una operación indica la posibilidad de que el número de pasos de control de la planificación aumente de forma innecesaria si no se planifica pronto dicha operación.
- *es constructivo*, ya que en cada paso del algoritmo se planifican una o varias operaciones y la planificación de las mismas es definitiva, es decir, no es modificada en los sucesivos pasos del algoritmo. Por tanto, la planificación de las operaciones se realiza sin tener en cuenta los posibles efectos de las operaciones que aún no han sido planificadas.

Para cada recurso, el algoritmo mantiene una lista con prioridad con las operaciones que están preparadas para ser planificadas. Una operación está preparada cuando todas sus predecesoras han sido ya planificadas. En cada iteración, las operaciones situadas al principio de las listas con prioridad son planificadas hasta que los recursos disponibles se agotan, correspondiendo cada iteración a un paso de control. Las listas con prioridad están siempre ordenadas en función de la *urgencia* de cada operación. De esta forma se resuelve el conflicto entre las operaciones que pretenden utilizar los mismos recursos en un paso de control.

Una vez que una operación es planificada en un determinado paso de control, puede ocurrir que otras operaciones estén preparadas para ser planificadas. Estas operaciones son insertadas en las listas de operaciones preparadas de acuerdo con la *urgencia* de las mismas. La *urgencia* puede estar basada en criterios tales como los siguientes:

- La *movilidad* (número de pasos de control en los que se puede planificar una operación) de las operaciones, ya que el grado de *urgencia* de una operación crece a medida que los pasos de control se aproximan al último paso en el cual la operación puede ser planificada.
- La longitud del camino más largo de entre todos los caminos que van desde una operación hasta sus operaciones sucesoras, ya que esta longitud es proporcional al número de pasos de control adicionales que se necesitan para completar la planificación.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

- El número de operaciones sucesoras inmediatas de una operación. Una operación con muchas sucesoras inmediatas genera un mayor número de operaciones preparadas para ser planificadas en el próximo paso de control.

A continuación se comenta brevemente el pseudo-código que se muestra en el Algoritmo 3.3. El algoritmo utiliza una lista con prioridad para cada tipo de operación. Estas listas son denotadas por las variables LP_{t1} , LP_{t2} , ... , LP_{tm} , siendo *m* el número de tipos de operaciones diferentes, y denotando t_k el tipo de las operaciones. Las operaciones de estas listas son planificadas en los pasos de control en función del número de recursos disponibles para cada tipo de operación, que se denota mediante la variable N_{tk}. La función INSERTAR determina las operaciones que están preparadas y las inserta en las correspondientes listas. La función PLANIFICAR planifica la primera operación de una lista en el paso de control actual. La función BORRAR elimina la operación planificada de la lista correspondiente.

En la Figura 3.3 se muestra un pequeño ejemplo ilustrativo del proceso. En la Figura 3.3.a y 3.3.c se muestran el GA que se quiere planificar y los recursos disponibles, respectivamente. En la primera iteración del algoritmo existen cinco operaciones preparadas: O-1, O-2, O-3, O-4 y O-5. En la Figura 3.3.b se muestran las listas con prioridad con estas operaciones. Cada operación tiene asociada su *movilidad* inicial. Las

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

```
INSERTAR ( LPt1 , LPt2 , ... , LPtm)
Paso_control = 0
while (LPt1 <> ϕ) or ... or (LPtm<> ϕ) do
    Paso_control = Paso_control + 1
    for k = 1 to m do
        for recursos = 1 to Ntk do
            if LPtk <> ϕ then
                PLANIFICAR (LPtk)
                BORRAR (LPtk)
            endif
        endfor
    endfor
    INSERTAR (LPt1 , LPt2 , ... , LPtm)
endwhile
```

Algoritmo 3.3. Planificación *list-based.*

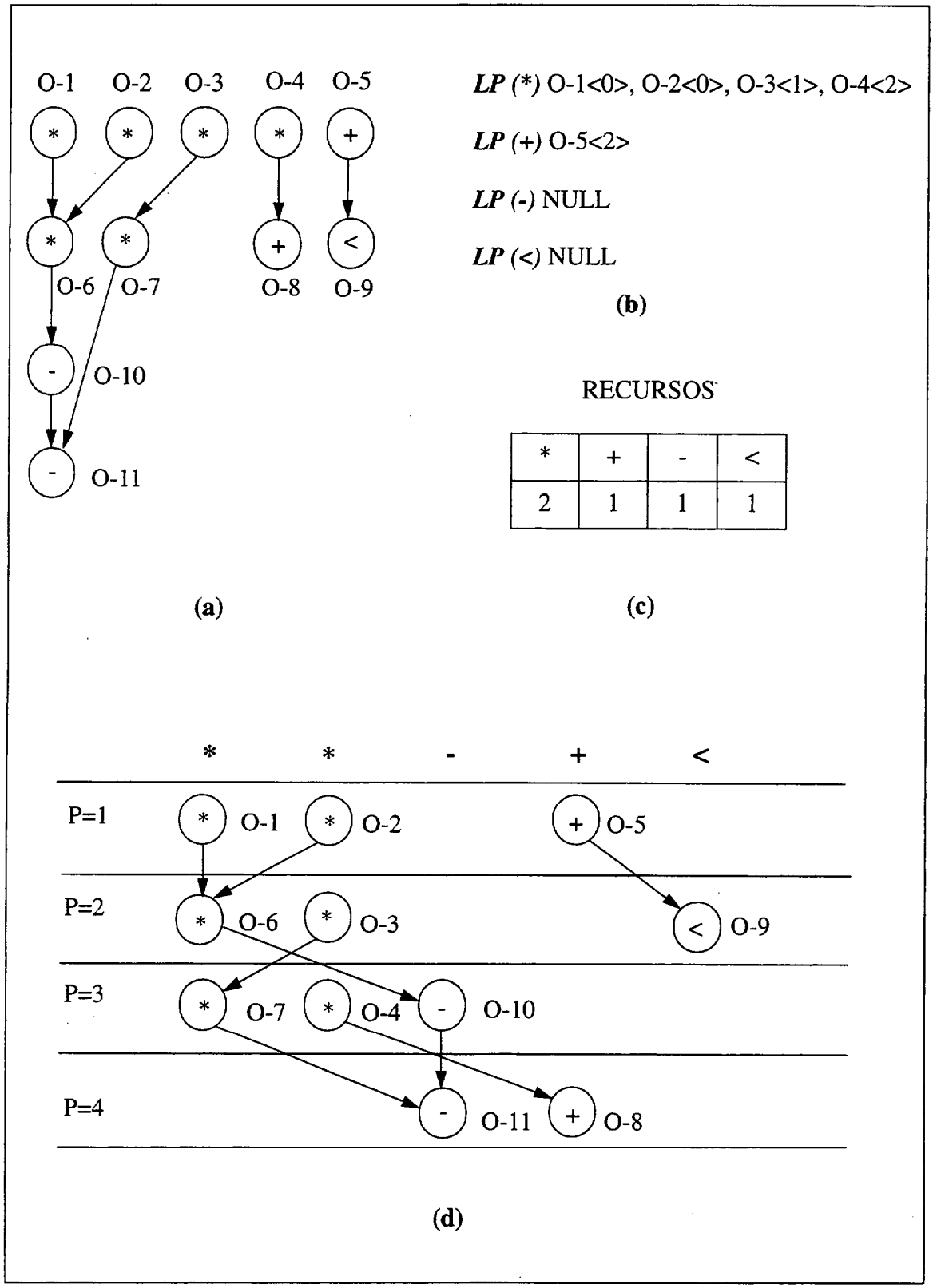


Figura 3.3. Ejemplo del proceso de planificación basado en el algoritmo *list-based*.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

listas están ordenadas en orden creciente de *movilidad*, es decir, las primeras operaciones de las listas tienen mayor *urgencia*. Por un lado, se observa que en la lista de las operaciones de multiplicar hay cuatro operaciones preparadas. Como solamente se dispone de dos multiplicadores, en la primera iteración se planifican las operaciones O-1 y O-2. Por otro lado, la operación O-5 es la única operación de suma que está preparada y también es planificada en la primera iteración. En la Figura 3.3.d se muestra el resultado final del proceso de planificación.

3.5.3. Planificación basada en el *simulated annealing*

En este apartado se presenta el algoritmo conocido como *simulated annealing* [Met53][Kir83][Dow95]. La utilización del *simulated annealing* como una técnica para la optimización de problemas combinatorios se remonta a comienzos de los años 80, cuando Kirkpatrick lo utilizaba para resolver diversos problemas de diseño VLSI [Kir83]. No obstante, las ideas principales en las que se fundamenta el *simulated annealing* fueron publicadas en 1953 como parte de un trabajo desarrollado por Metrópolis para estudiar el enfriamiento de materiales después de someterlos a un baño de calor [Met53]. Este proceso de enfriamiento es conocido como *annealing*. Si un material sólido es calentado hasta su fundición y luego es enfriado hasta que se solidifica nuevamente, las propiedades estructurales del material dependen de la velocidad del proceso de enfriamiento. Por ejemplo, con un enfriamiento muy lento pueden formarse redes cristalinas, pero si el enfriamiento es muy rápido las redes cristalinas tienen muchas imperfecciones.

El proceso de *annealing* puede ser simulado modelando el material como un sistema de partículas discretas. Básicamente, el algoritmo de Metrópolis simula el cambio energético del sistema mientras es sometido al proceso de enfriamiento hasta que el sistema converge a un estado de congelación estable. Treinta años después, Kirkpatrick sugiere que este tipo de simulación puede utilizarse como una técnica heurística para solucionar problemas de optimización.

La aproximación utilizada por Kirkpatrick se puede ver como una variante de los métodos heurísticos de optimización descendente, ya que en general se explora de forma descendente un subconjunto de todas las posibles soluciones, y la exploración se realiza utilizando un criterio de vecindad. La desventaja principal de los métodos descendentes radica en que la aceptación de una solución vecina como solución actual solamente se realiza si se produce un decremento en el valor de la función de coste. La diferencia fundamental del *simulated annealing* respecto de los métodos descendentes es que se aceptan soluciones vecinas que producen un incremento en el valor de la

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

función de coste. De esta forma existe la posibilidad de salir de los mínimos locales. Ahora bien, no siempre se admiten soluciones peores que la actual. La estrategia para aceptar este tipo de soluciones se basa en los trabajos sobre termodinámica estadística realizados por Metrópolis. Las leyes de la termodinámica establecen que a una temperatura t, la probabilidad de que en una partícula del sistema se produzca un incremento de energía de magnitud ΔE durante el proceso de enfriamiento viene dada por la Expresión 3.1 (k es la constante de Boltzmann).

El proceso de simulación de Metrópolis genera una perturbación sobre el estado actual del sistema y calcula la variación de energía que se produce en el mismo. Si la energía se decrementa, el nuevo estado del sistema es aceptado. Por el contrario, si la energía se incrementa, el nuevo estado es aceptado con una probabilidad que viene dada por la Expresión 3.1. Este proceso es repetido un número determinado de veces para cada temperatura, después de lo cual la temperatura es decrementada. El proceso completo se repite hasta que el sistema alcanza una temperatura final tan pequeña que permanece en un estado estable.

En 1983, Kirkpatrick [Kir83] publica que el algoritmo de Metrópolis se puede aplicar a los problemas de optimización combinatoria realizando una correspondencia entre los elementos del proceso físico de enfriamiento y los elementos de un proceso de optimización combinatoria (Tabla 3.1).

Desde la propuesta de Kirkpatrick se han publicado muchos estudios teóricos sobre la convergencia del *simulated annealing* [Dow95]. Los resultados muestran que se necesita un número de iteraciones superior a las que se necesitan en una exploración exhaustiva para garantizar la convergencia del algoritmo hacia una solución globalmente óptima. Aunque este resultado no es muy importante desde un punto de vista práctico, si es importante para conocer que el proceso tiene realmente una tendencia hacia la solución óptima global cuando se realiza un número elevado de iteraciones.

$$P(\Delta E,t) = e^{\frac{-\Delta E}{kt}}$$

Expresión 3.1. Probabilidad de incremento energético durante el proceso de *annealing*.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

Simulación del proceso de *annealing*	**Optimización combinatoria**
Estados del sistema	Soluciones posibles
Energía	Coste
Cambio de estado	Solución vecina
Temperatura	Parámetro de control
Estado estable	Solución heurística

Tabla 3.1: Correspondencia entre el proceso físico de *annealing* y los problemas de optimización combinatoria.

En el Algoritmo 3.4 se muestra el pseudo-código del *simulated annealing*. Como puede apreciarse, el método es similar al de un algoritmo descendente. La diferencia fundamental es que se añade una sentencia de control selectiva para tener en cuenta la posibilidad de aceptar soluciones vecinas con un coste superior al de la solución actual.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

```
Seleccionar una solución inicial So
S_actual = So
t = TEMPERATURA_INICIAL ( P_i)
repeat
    contador = 0
    repeat
        S_vecina = SELECCIONAR_VECINO (S_actual)
        ganancia = COSTE (S_vecina) - COSTE (S_actual)
        if ganancia < 0 then
            S_actual = S_vecina
        else
            r = RANDOM [0,1]
            if r < P(t,ganancia) then
                S_actual = S_vecina
            endif
        endif
        contador = contador + 1
    until contador = NUM_VECINOS
    t = TEMPERATURA (t)
until condición_de_parada = TRUE
```

Algoritmo 3.4. *Simulated Annealing.*

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

Capítulo 4

Lenguaje VSS

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

4.1. Introducción

En este capítulo se describe el lenguaje VSS (*VHDL-based system specification*) que se ha diseñado en este trabajo para la especificación de sistemas en el entorno de codiseño hardware/software GACSYS. VSS es un lenguaje que se basa en un amplio subconjunto de VHDL al que se han añadido un conjunto de sentencias para facilitar a los diseñadores la especificación de sistemas. El subconjunto de VHDL soportado por VSS puede consultarse en el Apéndice A de esta memoria.

VSS añade a VHDL un conjunto sentencias para la descripción de comportamiento jerárquico, la comunicación entre procesos, la temporización de las especificaciones, y el manejo de excepciones. Además, VSS soporta variables compartidas.

Además, en este trabajo también hemos desarrollado un compilador que traduce a VHDL una especificación VSS. En este capítulo se presenta el esquema de traducción que se ha utilizado. Para el desarrollo del compilador se han utilizado las herramientas LEX [Lev92] y YACC [Don91][Lev92]. La generación de código se realiza durante el proceso de reconocimiento sintáctico [Aho86]. Para una mayor información sobre el compilador puede consultarse [Cas98d]. También hemos diseñado una representación intermedia para el código VHDL generado, así como un compilador para la generación de la misma.

La estructura de este capítulo es la siguiente: la primera parte del capítulo describe el lenguaje VSS. En el apartado 2, se describen las sentencias para la descripción de comportamiento jerárquico. En el apartado 3, describimos las sentencias para la comunicación entre procesos. En el apartado 4, explicamos las sentencias para el manejo de excepciones. En el apartado 5, se explica el manejo de variables compartidas. En el apartado 6, se explica las sentencias relacionadas con la temporización de una especificación.

En la segunda parte del capítulo se describe el esquema de traducción de VSS a VHDL y la representación intermedia que hemos utilizado. En el apartado 7, se describe el esquema de traducción de las sentencias de la jerarquía de procesos. En el apartado 8, se presenta el esquema de las sentencias para la comunicación entre procesos. En el apartado 9, se describe el esquema de las sentencias para el manejo de excepciones. En el apartado 10, se presenta el esquema de las sentencias para el manejo de variables compartidas. En el apartado 11, se describe el esquema de las sentencias relacionadas con la temporización de una especificación VSS. Finalmente, en el apartado 12 se describe la representación intermedia utilizada.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

4.2. Jerarquía de procesos

VSS añade a VHDL un conjunto de sentencias para describir el comportamiento de un proceso como un conjunto de procesos jerárquicos. En primer lugar, VSS soporta sentencias (*seq_fork* y *par_fork*) para especificar jerarquías de procesos secuenciales y/ o concurrentes. En segundo lugar, VSS soporta un sentencia (*join*) que permite que un proceso espere por la finalización de cualquiera de sus procesos descendientes. Finalmente, VSS soporta sentencias para la activación y desactivación jerárquica de procesos. Un proceso puede ser activado (sentencia *start*) o desactivado (sentencias *kill* e *initialize*) por otro proceso. A continuación se explica cada una de las sentencias.

4.2.1. Composición secuencial y paralela de procesos: *seq_fork / par_fork*

La sentencia *seq_fork* permite que un proceso inicie la ejecución secuencial de un conjunto de procesos hijos, mientras que la sentencia *par_fork* inicia la ejecución concurrente de los mismos (Sintaxis 3.1). Los procesos sólo pueden trabajar con sus variables locales. Si un proceso necesita trabajar con las variables de otro proceso, debe utilizar las sentencias de comunicación entre procesos o las variables compartidas de VSS.

Para proporcionar una mayor flexibilidad al diseñador de sistemas, las sentencias *fork* se acompañan de una expresión *booleana* que indica si los procesos hijos deben ejecutarse. Cuando la expresión booleana es verdadera, se activa la ejecución de los procesos hijos. La sentencia *par_fork* activa simultáneamente a todos los procesos hijos, mientras que la sentencia *seq_fork* activa al primero de ellos, encargándose cada hijo de activar al siguiente cuando finaliza su ejecución. Por el contrario, los procesos no son activados si la expresión booleana es falsa.

```
stat_seqfork:
    fork_identifier: seq_fork bool_expression
        VSS_process_statement
        { || VSS_process_statement }
    end fork ;

stat_parfork:
    fork_identifier: par_fork bool_expression
        (simple/iterative) VSS_process_statement
        { || (simple/iterative) VSS_process_statement}
    end fork ;
```

Sintaxis 3.1. Sentencias *seq_fork* y *par_fork* para la composición jerárquica de procesos.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

A diferencia de VHDL, en VSS los procesos hijos pueden ejecutarse de dos modos diferentes: modo simple y modo iterativo (cláusulas *simple* e *iterative*, respectivamente). Un proceso que se ejecuta en modo simple solamente ejecuta su código una única vez. Un proceso que se ejecuta en modo iterativo vuelve a ejecutar su primera línea de código después de ejecutar su última línea. Este modo sólo puede utilizarse en la sentencia *par_fork*.

En general, la ejecución de una sentencia fork no es bloqueante, es decir, el proceso que la ejecuta continúa inmediatamente con la siguiente sentencia. Sin embargo, un proceso no ejecuta una sentencia *fork* si alguno de los procesos que la componen está activo. En ese caso, el proceso espera a que todos los procesos activados hayan finalizado su ejecución. Por tanto, VSS no soporta la creación dinámica de procesos.

En la Figura 3.1 se muestra una descripción VSS acompañada de una representación gráfica de su jerarquía de procesos.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

```
P0: process
begin
    sentencia_0;
    seq: seq_fork true
        P1: process
        begin
            par_1: par_fork true
                simple P3: process
                begin sentencia_3; end process;
            ||
                simple P4: process
                begin sentencia_4; end process;
            end fork;
        end process;
    ||
        P2: process
        begin
            par_2: par_fork true
                simple P5: process
                begin sentencia_5; end process;
            ||
                simple P6: process
                begin sentencia_6; end process;
            end fork;
        end process;
    end fork;
    sentencia_1;
    sentencia_2;
end process;
```

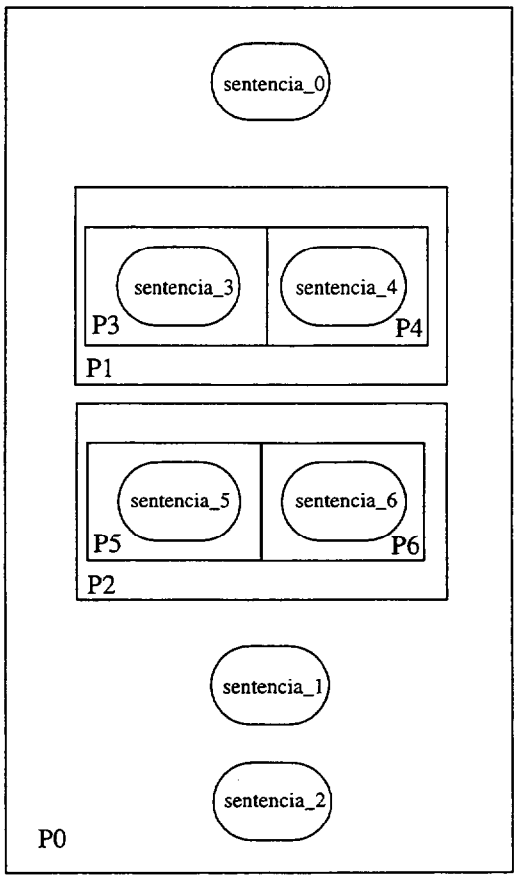


Figura 4.1. Descripción VSS y su correspondiente jerarquía de procesos.

4.2.2. Sincronización de procesos: *join*

La sentencia *join* (Sintaxis 3.2) permite que un proceso padre se sincronice con el final de la ejecución de todos o algunos de sus procesos descendientes ejecutados mediante sentencias *par_fork* o *seq_fork*. Un proceso hijo iterativo no finaliza nunca su actividad (excepto si es desactivado por otro proceso), de modo que su proceso padre permanecería esperando indefinidamente si ejecuta una sentencia *join* sobre dicho proceso hijo.

Si en una sentencia *join* sólo se indica el nombre de la etiqueta (*fork_identifier*), la sincronización se realiza con todos los procesos especificados en la correspondiente sentencia *fork*. Si sólo se especifica la lista de procesos hijos, la sincronización se realiza con el conjunto de procesos especificados, que pueden pertenecer a diferentes sentencias *fork*. Así mismo, una sentencia *join* sin parámetros es equivalente a una sentencia *join* sobre la sentencia *fork* previa.

En la Figura 3.2 se muestra una descripción VSS y un grafo que representa la sincronización especificada mediante las sentencias *join*.

4.2.3. Activación y desactivación de procesos: *start*, *kill* e *initialize*

VSS permite la activación y desactivación jerárquica de los procesos que se ejecutan mediante las sentencias *fork*. Para ello, el lenguaje soporta las sentencias *start*, *kill* e *initialize* (Sintaxis 3.3).

La sentencia *start* permite iniciar la ejecución de un proceso. Si el proceso que se quiere ejecutar no está activo, se inicia la ejecución del mismo independientemente del valor de la expresión booleana. En caso contrario, el comportamiento de la sentencia *start* depende de la expresión booleana: a) si la expresión es verdadera, la sentencia *start* es bloqueante, es decir, espera que el proceso que se quiere ejecutar finalice, y b) si la expresión es falsa, la sentencia no es bloqueante y no tiene efecto alguno. En cualquiera de los casos anteriores, independientemente del estado del proceso que se quiere ejecutar y del valor de la expresión booleana, si se ejecutan varias sentencias *start* sobre un mismo proceso, el efecto no es acumulativo, es decir, el comportamiento es análogo a la ejecución de una única sentencia *start* sobre dicho proceso.

```
stat_join:
    join [fork_identifier ][: process_identifier_list] ;
```

Sintaxis 3.2. Sentencia *join* para la sincronización de procesos.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

```
P0: process
begin
    sentencia_0;
    seq: seq_fork true
        P1: process
        begin
            par_1: par_fork true
                simple P3: process
                begin sentencia_3; end process;
            ||
                simple P4: process
                begin sentencia_4; end process;
            end fork;
            join par_1;
        end process;
    ||
        P2: process
        begin
            par_2: par_fork true
                simple P5: process
                begin sentencia_5; end process;
            ||
                simple P6: process
                begin sentencia_6; end process;
            end fork;
            join par_2;
        end process;
    end fork;
    sentencia_1; join seq; sentencia_2;
end process;
```

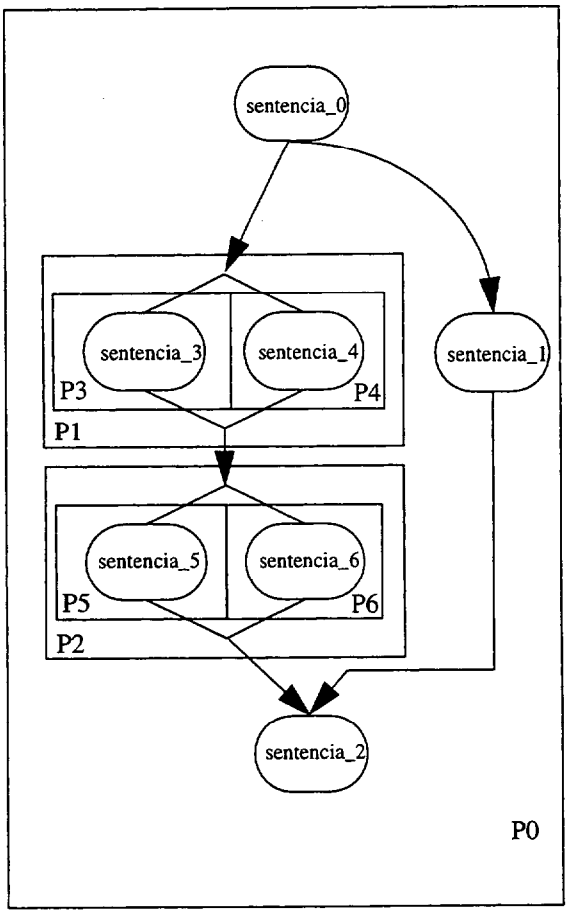


Figura 4.2. Descripción VSS y su correspondiente grafo de sincronización.

stat_start:
start bool_expression process_identifier ;

stat kill:
kill process_identifier ;

stat_initialize:
initialize process_identifier ;

initialize_body:
initialize begin
{*VHDL_statement* }
end ;

Sintaxis 3.3. Sentencias para la activación (*start*) y desactivación (*kill* e *initialize*) de procesos.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

La sentencia *kill* permite finalizar la ejecución de un proceso y de toda su descendencia. Por ejemplo, en la Figura 3.2, la ejecución de una sentencia *kill* sobre el proceso P2 también supondría la muerte de los procesos P5 y P6. Para evitar interbloqueos (*deadlocks*), un proceso no puede ejecutar una sentencia *kill* sobre otro proceso que sea antecesor suyo.

La sentencia *initialize*, al igual que la sentencia *kill*, finaliza con la ejecución de un proceso y de toda su descendencia, pero además el proceso inicializado ejecuta un conjunto de sentencias definidas en su cuerpo de inicialización. Una vez que el proceso finaliza la ejecución de su cuerpo de inicialización, permanece desactivado hasta una nueva ejecución. Para evitar interbloqueos (*deadlocks*), un proceso no puede ejecutar una sentencia *initialize* sobre otro proceso que sea antecesor suyo.

Las sentencias de desactivación (*kill* e *initialize*) no interrumpen la ejecución de un proceso inmediatamente, ya que un proceso sólo puede ser desactivado cuando se encuentra ejecutando alguna de las sentencias VSS sensibles a las sentencias de desactivación. Las sentencias sensibles son las siguientes: *join*, *delay*, *send*, *receive* y todas las variantes de la sentencia *check*. Estas sentencias se explicarán posteriormente.

Las sentencias de desactivación jerárquica son bloqueantes, es decir, los procesos que ejecutan una sentencia de desactivación permanecen bloqueados hasta que el proceso víctima y toda su descendencia están desactivados.

Finalmente, si un proceso dado es objeto de las sentencias *start*, *kill* e *initialize* de forma simultánea, el proceso actúa según la prioridad de las sentencias. El orden decreciente de prioridad es el siguiente: *kill*, *initialize* y *start*.

4.3. Comunicación entre procesos

VSS añade a VHDL un conjunto de sentencias específicas para la comunicación entre procesos basada en el paradigma de paso de mensajes. En general, los esquemas de comunicación basados en el paso de mensajes se pueden clasificar atendiendo a varios criterios, entre los cuales se pueden distinguir los que se muestran en la Tabla 3.1.

En la primera columna se distingue entre comunicación basada en canales y basada en puertos. En el primer caso, el mensaje se envía por un determinado canal al que se encuentran conectados los emisores y receptores. El canal es la abstracción de un medio físico a través del cual tiene lugar la comunicación. Los mensajes no necesitan incluir el destinatario de los mismos y sólo se pueden leer en el mismo orden en que se escriben. En el modelo de puertos, los mensajes incluyen el destinatario de los mismos, quedando

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

BIBLIOTECA GENERAL UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA

Medio	Modo de transmisión	emisores y receptores	Dirección
Canal	Síncrono	1:1	unidireccional
		1:N	
Puerto	Asíncrono	N:1	bidireccional
		N:M	

Tabla 3.1: Criterios para la clasificación de los esquemas de comunicación entre procesos basados en el paso de mensajes.

oculto el medio utilizado para hacer llegar los mensajes a su destino. Cada puerto tiene asociado un *buffer*, y los mensajes almacenados en el mismo pueden leerse en cualquier orden. Este modelo es normalmente utilizado en el diseño de sistemas distribuidos. VSS solamente soporta comunicación basada en canales, ya que está orientado al diseño de sistemas empotrados.

En la segunda columna se diferencia entre comunicación síncrona y asíncrona. En el primer caso, las comunicaciones son bloqueantes, es decir, el proceso transmisor debe esperar a que el proceso receptor reciba los datos antes de continuar con la ejecución de la siguiente sentencia. En el segundo caso, las comunicaciones no son bloqueantes, es decir, el proceso transmisor no espera por el proceso receptor. Esto requiere que el medio de comunicación disponga de memoria. VSS soporta ambos modelos de comunicación.

En la tercera columna se clasifican las comunicaciones atendiendo al número de transmisores y receptores involucrados en las mismas. El esquema [1:1] representa las conexiones punto a punto, en las cuales la comunicación se realiza exclusivamente entre un proceso transmisor y otro receptor. En el esquema [1:N], conocido como *broadcast*, un proceso transmisor se puede comunicar con una serie de procesos receptores. El esquema [N:1], denominado *incast*, permite que varios procesos comuniquen sus datos a un único proceso. El esquema [M:N], en el que existen M transmisores y N receptores, se utiliza básicamente con puertos de comunicación, es decir, normalmente no se soporta con canales. VSS soporta la definición de canales punto a punto, *broadcast* e *incast*.

Por último, en la cuarta columna se distingue entre comunicación unidireccional y bidireccional. La comunicación bidireccional se caracteriza porque la información

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

puede fluir en ambos sentidos por el mismo canal o puerto, mientras que en el caso unidireccional la información fluye siempre en el mismo sentido. Generalmente, la comunicación bidireccional se utiliza en comunicaciones basadas en puertos, mientras que la comunicación unidireccional solamente suele utilizarse en comunicaciones basadas en canales. VSS sólo soporta canales unidireccionales, necesitando dos canales lógicos diferentes para llevar a cabo una comunicación bidireccional.

Resumiendo, VSS soporta comunicaciones basadas en canales unidireccionales, tanto en modo síncrono como asíncrono, y con los siguientes esquemas: [1:1] (punto a punto), [1:N] (*broadcast*) y [N:1] (*incast*).

En una especificación VSS, los canales se definen en la parte declarativa de la arquitectura de una entidad de diseño, siendo su sintaxis la que se muestra en la Sintaxis 3.4. La cláusula opcional *buffer* permite definir una comunicación asíncrona con un *buffer* del tamaño deseado. El canal de comunicación es síncrono si se omite dicha cláusula.

Las sentencias de comunicación soportadas en VSS son *send* y *receive* (Sintaxis 3.4). La ejecución de dos sentencias de comunicación consecutivas precisa que se

```
channel_spec:
    channel channel_identifier_list:data_type [buffer size] (simple | broadcast | incast) ;

stat_send:
    send ( destinatario , VHDL_expression {; destinatario , VHDL_expression} ) ;

destinatario:
    channel_identifier |
    process_identifier <= channel_identifier |
    broadcast <= channel_identifier

stat_receive:
    receive ( channel_identifier , var_identifier {; channel_identifier , var_identifier} ) ;

VSS_bool_expressions:
    buffer_empty ( channel_identifier ); |
    buffer_full ( channel_identifier );

stat_resetbuffer:
    reset_buffer channel_identifier;
```

Sintaxis 3.4. Sentencias para la declaración de canales de comunicación (*channel*), la comunicación entre procesos (*send, receive*), y el manejo de los canales (*buffer_empty, buffer_full, reset_buffer*).

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

finalice la primera sentencia antes de iniciarse la segunda. Sin embargo, un diseñador de sistemas podría desear que varias sentencias de comunicación se ejecuten en paralelo. Por ello, VSS soporta sentencias *send* y *receive* que permiten la comunicación simultánea por varios canales. En una comunicación *broadcast*, además del nombre del canal, se debe indicar el nombre de los procesos receptores. Si se desea enviar la información a todos los procesos receptores se utiliza la palabra reservada *broadcast*.

En un canal *incast*, VSS no añade facilidad alguna para que el proceso receptor identifique al proceso transmisor de un dato, por lo que el diseñador debe hacerlo de forma explícita. Una forma sencilla de hacerlo es enviando el identificador del proceso transmisor junto con la información.

Finalmente, VSS soporta las siguientes sentencias para los canales de comunicación asíncronos: *buffer_full*, *buffer_empty* y *reset_buffer*. Las sentencias *buffer_full* y *buffer_empty* informan del estado del *buffer*. La primera función retorna verdadero cuando el *buffer* está lleno y falso en caso contrario. Análogamente, la función *buffer_empty* retorna verdadero cuando el *buffer* está vacío y falso en caso contrario. La sentencia *reset_buffer resetea* el *buffer* del canal indicado.

4.4. Excepciones

VSS añade a VHDL un conjunto de sentencias para el manejo de excepciones. Una excepción permite detener la ejecución de uno o varios procesos para ejecutar una o varias rutinas de tratamiento de la excepción. Una vez que las rutinas finalizan, los procesos continúan con su ejecución en el mismo punto donde se encontraban. La interrupción de un proceso no implica la interrupción de sus procesos descendientes

Las excepciones de un sistema especificado en VSS deben declararse en la parte declarativa de la arquitectura. La sintaxis de la declaración se muestra en la Sintaxis 3.5. La declaración de una excepción se acompaña de una constante entera positiva (*level*) que indica su nivel de prioridad, siendo más prioritarias las excepciones que tienen un mayor valor asociado.

Un proceso debe especificar los cuerpos de excepción (*exception body*) de aquellas excepciones a las que es sensible. Un cuerpo de excepción puede ser interrumpido por una excepción de mayor nivel. En caso de que dos excepciones de igual nivel se activen simultáneamente, se atiende antes la que se haya declarado en primer lugar. La sintaxis de un cuerpo de excepción se muestra en Sintaxis 3.5.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

```
exception_spec:
   exception exception_identifier_list priority level ;

exception_body:
   exception_body exception_identifier is
      {process_declarative_items}
   begin
      { VHDL_statement ¦ VSS_statement_notforks }
   end ;

stat_exception:
   exception (exception_identifier) bool_expression ;

stat_enable:
   vssenable exception_identifier_list ;

stat_disable:
   vssdisable exception_identifier_list ;
```

Sintaxis 3.5. Sentencias para el manejo de excepciones: declaración (*exception*), definición de los cuerpos (*exception_body*), activación (*exception*), habilitación (*vssenable*) y deshabilitación (*vssdisable*).

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

En un cuerpo de excepción no pueden incluirse sentencias *fork*. El objetivo de esta restricción es no generar múltiples jerarquías de comportamiento en un proceso, es decir, su propia jerarquía y la jerarquía correspondiente a cada cuerpo de excepción. Los cuerpos de excepción, aunque se declaran en el interior de los procesos, no pueden acceder a las variables locales de los mismos.

Las excepciones, al igual que las sentencias de activación y desactivación jerárquica, no actúan de forma inmediata. Un proceso no activa su cuerpo de excepción hasta que ejecuta una sentencia *delay* o una sentencia *check*.

La activación de una excepción se realiza mediante la sentencia *exception* (Sintaxis 3.5). Cuando un proceso activa una excepción y ya existe un proceso sensible a dicha excepción que está ejecutando su correspondiente cuerpo de excepción (debido a una activación previa), el proceso activador permanece bloqueado hasta que finalice la ejecución del cuerpo de excepción. Posteriormente, el proceso activa nuevamente la excepción. La expresión *booleana* que acompaña a la sentencia de activación permite que el proceso activador continúe con su ejecución (si es falsa), o bien espere a que los cuerpos de excepción finalicen su ejecución (si es verdadera).

VSS permite modificar la sensibilidad a las excepciones. Una excepción permanece habilitada (deshabilitada) hasta que se ejecuta una sentencia *vssdisable* (*vssenable*) (Sintaxis 3.5). Inicialmente, las excepciones están habilitadas por defecto.

4.4.1. Chequeo de excepciones: *check*

VSS hereda la semántica de simulación de VHDL, de modo que cada proceso ejecuta ininterrumpidamente su código hasta encontrar una sentencia *wait* o una sentencia VSS cuya traducción incluya sentencias *wait*. Este modo de ejecución de los procesos hace que las sentencias de desactivación jerárquica, así como las excepciones, sólo puedan actuar sobre los procesos cuando estos ejecutan las sentencias anteriores. Para no comprobar la desactivación jerárquica y las excepciones en todas las sentencias *wait* de una especificación, se añadieron al lenguaje las siguiente sentencias: *check on* y *check*.

La sentencia *check on* (Sintaxis 3.6), al igual que la sentencia *wait*, hace que un proceso detenga su ejecución. Un proceso permanece bloqueado hasta que se produce un cambio en el valor de alguna de las señales especificadas en la lista de señales. Además, si se utiliza la cláusula *until*, el proceso permanece bloqueado hasta que la expresión *booleana* sea verdadera. Mientras un proceso permanece bloqueado en una sentencia *check on*, el mismo es receptivo a cualquier sentencia de desactivación jerárquica y a cualquier excepción a la que es sensible.

La sentencia *check* (Sintaxis 3.6) sólo detiene la ejecución de un proceso si está activada alguna sentencia de desactivación jerárquica o alguna excepción a la que el proceso es sensible.

En definitiva, VSS dispone de un amplio espectro de sentencias para combinar la detención y continuación de un proceso con la detección de las sentencias de desactivación jerárquica y las excepciones (Tabla 3.2).

stat_check:
check on signal_identifier_list [**until** *bool_expresion*] ;
| **check ;**

Sintaxis 3.6. Sentencias para chequear excepciones y desactivación jerárquica.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

Sentencias	Desactivación y excepciones	Detención del proceso	Continuación del proceso
wait on/until	NO	SI	SI (condicional)
check on	SI	SI	SI (condicional) (después de atender una posible excepción)
			NO (si hay desactivación)
check	SI	NO	SI
		SI (si hay desactivación o excepción)	SI (después de atender una posible excepción)
			NO (si hay desactivación)
delay	SI	SI	Después de un intervalo de tiempo
			NO (si hay desactivación)

Tabla 3.2: Sentencias para combinar la detención y continuación de un proceso con la detección de las sentencias de desactivación jerárquica y las excepciones.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

4.5. Variables compartidas

VSS añade a VHDL un conjunto de sentencias para el manejo de variables compartidas. VSS permite declarar bloques de variables que pueden ser consultadas o modificadas por todos los procesos de una especificación. Estos bloques se definen en la parte declarativa de la arquitectura según la sintaxis mostrada en Sintaxis 3.7.

El acceso a las variables de un bloque *shared* debe tener lugar en el interior de regiones críticas, garantizándose así la exclusión mutua. Estas regiones se definen mediante la sentencia *lock* (Sintaxis 3.7), que va acompañada de una expresión booleana. Si el valor de la expresión es cierto, la sentencia *lock* es bloqueante, es decir, un proceso permanece bloqueado hasta que accede al bloque de variables compartidas. Si el valor de la expresión es falso y el proceso no accede al bloque en su primer intento, el proceso continúa con la ejecución de las sentencias posteriores a la región crítica. En el interior de una región crítica sólo pueden utilizarse las sentencias secuenciales de VHDL y las sentencias *get* y *put* de VSS. Como excepción a esta regla, en una región

```
shared_spec:
    shared block_identifier is
        {var_declaration}
    end shared;

stat_lock:
    lock (shared_block_identifier) bool_expression
        {VHDL_statement ¦ stat_get ¦ stat_put}
    end lock;

stat_get:
    get(shared_var_identifier => local_var_identifier);

stat_put:
    put(shared_var_identifier <= expression);
```

Sintaxis 3.7. Sentencias para el manejo de variables compartidas: declaración (*shared*), definición de regiones críticas (*lock*), y acceso a las variables (*get*, *put*).

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

crítica no pueden utilizarse las sentencias *exit* y *wait*. Además, como tampoco pueden utilizarse las sentencias *delay* y *check* de VSS, un proceso no es sensible a las sentencias de desactivación jerárquica ni a las excepciones en el interior de una región crítica.

El acceso a las variables compartidas debe realizarse a través de dos funciones predefinidas: para copiar el valor de una variable compartida en una variable local de un proceso se utiliza la sentencia *get*, mientras que para modificar el valor de una variable compartida se utiliza la sentencia *put*. La sintaxis se muestra en Sintaxis 3.7.

El lenguaje asegura que todos los procesos que solicitan acceso a una variable compartida consiguen acceder a la misma. Actualmente, la implementación realizada en el compilador de VSS se basa en un árbitro determinista que utiliza una política circular en la que todos los procesos tienen igual prioridad, es decir, un proceso que solicita acceder a una región crítica tiene la seguridad de que ningún otro proceso puede acceder dos veces a dicha región crítica mientras espera su turno.

4.6. Temporización

En la descripción de sistemas, muchas veces se necesita concretar la duración de algunas actividades. VHDL soporta este tipo de restricciones temporales mediante la sentencia *wait for time_expression* y la cláusula *after* de las sentencias de asignación a

señales. Sin embargo, las herramientas de síntesis de alto nivel no soportan la semántica de estas sentencias. Por tanto, VSS no soporta este tipo de sentencias.

No obstante, VSS soporta algunas sentencias para especificar restricciones temporales. Estas sentencias se basan en la cuenta de eventos de señales de reloj. Una señal de reloj (Sintaxis 3.8) viene definida por un nombre (*clock_identifier*) y por el período de la misma, siendo este último especificado mediante un valor entero (*value*) seguido de una unidad de tiempo (*time_unit*). Estas unidades son las mismas que se utilizan en el tipo de datos *physical_time* de VHDL.

Además de las señales de reloj, VSS soporta la declaración de temporizadores (*timer*) visibles a todos los procesos de una especificación. Cada temporizador tiene implícitamente asociada una excepción que se activa cada vez que finaliza una operación de cuenta. El nombre de esta excepción coincide con el nombre del temporizador. En la declaración de un temporizador se especifica el nombre del reloj al que está conectado, el máximo valor de cuenta al que se puede programar, y el nivel de prioridad de la excepción asociada al temporizador (Sintaxis 3.8).

VSS soporta las siguientes sentencias para la temporización de especificaciones: *delay*, *timeout* y *reset_timer* (Sintaxis 3.8). La sentencia *delay* permite la detención de un proceso durante el tiempo especificado en la misma. La sentencia *delay* es sensible a las sentencias de desactivación jerárquica y a las excepciones. El tiempo que transcurre durante la ejecución del cuerpo de una posible excepción también se contabiliza.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

clock_spec:
clocks (clock_identifier , value time_unit {; clock_identifier , value time_unit });

timer_spec:
timer timer_identifier_list : clock_identifier **max_cont** value **priority** level ;

stat_delay:
delay (clock_identifier, *expression* time_unit);

stat_timeout:
timeout (timer_identifier, *expression* time_unit);

stat_resettime:
reset_timer (timer_identifier);

Sintaxis 3.8. Sentencias relacionadas con la temporización de especificaciones VSS: definición de relojes (*clocks*), definición de temporizadores (*timer*), especificación de retardos (*delay*), especificación de *timeout* (*timeout*), y reseteado de temporizadores (*reset_timer*).

La sentencia *timeout* permite arrancar un temporizador, generándose una excepción al finalizar el proceso de cuenta que ha sido programado. Una vez arrancado el temporizador, el proceso continúa con la ejecución de la siguiente sentencia. Si el temporizador ya está activo cuando se ejecuta una sentencia *timeout*, el mismo es reseteado y arrancado con el nuevo valor de cuenta.

Finalmente, la sentencia *reset_timer* anula la operación de cuenta que se haya programado en un temporizador.

4.7. Traducción de las sentencias de la jerarquía de procesos

La nomenclatura utilizada para nombrar las señales que intervienen en la traducción de las sentencias es la siguiente: todos los nombres comienzan con los caracteres *vss_*. Estos caracteres van seguidos de otros dos caracteres que identifican el tipo de sentencia: *JE* para las sentencias de la jerarquía de procesos, *CO* para las sentencias de comunicación entre procesos, *EX* para las sentencias de manejo de excepciones, *SH* para las sentencias de variables compartidas, y *TI* para las sentencias de temporización.

Todos estos prefijos se han omitido ya que no aportan información útil para la comprensión del esquema de traducción y su omisión facilita la lectura. En todos los ejemplos, un nombre entre llaves referencia a dicho nombre precedido del prefijo adecuado. Por ejemplo, en este apartado en que se explica el esquema de traducción de las sentencias para la jerarquía de procesos, {*identifier*} realmente hace referencia a un identificador denominado *vss_je_identifier.*

VSS permite describir el comportamiento de un proceso como una jerarquía de procesos secuenciales y/o concurrentes (sentencias *seq_fork* y *par_fork*) que pueden ser activados y desactivados mediante las sentencias *start*, *kill* e *initialize*. Para cada uno de los procesos se genera un proceso VHDL en el que se distinguen, en general, tres secciones: a) la primera sección se encarga de reconocer la muerte del proceso y de ejecutar el código de inicialización que se pudiera haber especificado. b) La segunda sección se encarga de esperar a que el proceso sea arrancado o inicializado. c) La última sección incluye todas las sentencias del proceso original encerradas en un bucle etiquetado con el nombre *vss_live*.

La traducción de las sentencias de la jerarquía de comportamiento requiere la declaración de un conjunto de señales para la activación y desactivación jerárquica de procesos. Para cada proceso hijo se declaran, como mínimo, las siguientes señales:

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

- *{pr}_start*: esta señal la utiliza un proceso padre para iniciar la ejecución de sus procesos hijos.
- *{pr}_start_ack*: esta señal la utiliza un proceso hijo para informar si está activo. Inicialmente la señal se inicializa al valor 'Z', que indica que el proceso no está activo. Esta señal se activa una vez que el proceso inicia su ejecución, y sólo vuelve al estado de reposo cuando el proceso muere o acaba su ejecución.

Además, si un proceso hijo puede estar muerto o inicializado de un modo directo (no por herencia), se declaran las siguientes señales:

- *{pr}_extra_kill*: si esta señal está activa, fuerza la ejecución de la sentencia *exit vss_live*. Como el cuerpo de un proceso hijo se incluye en el interior de un bucle etiquetado con *vss_live*, la sentencia *exit* consigue que el proceso muera.
- *{pr}_extra_kill_ack*: esta señal la utiliza el proceso hijo como señal de reconocimiento para indicar que ha muerto.

Si un proceso hijo puede ser inicializado, se declaran también las siguientes señales:

- *{pr}_init*: esta señal indica a un proceso que debe ejecutar su cuerpo de inicialización.
- *{pr}_init_ack*: esta señal la utiliza un proceso como reconocimiento de la señal anterior.

Seguidamente se comenta la traducción a VHDL de cada uno de los procesos hijos del código VSS del Ejemplo 4.1. En la traducción del proceso *pr1*, como no puede ser muerto o inicializado, sólo se incluye una sentencia *if* en la que el proceso permanece a la espera de ser activado. Cuando el proceso sea arrancado, la señal *start_ack* indica al resto del sistema si el proceso está activo ('0') o inactivo ('Z'). Al final del proceso, esta señal se desactiva.

El proceso *pr2* sólo se diferencia del proceso *pr1* en que es iterativo. Por tanto, el cuerpo principal del proceso aparece en un bucle infinito etiquetado con el nombre *vss_live*.

El proceso *pr3*, aunque se ejecuta en modo simple, puede ser objeto de una sentencia *kill*. Por tanto, su código principal se encierra en una sentencia *for* de una única iteración. Esto permite que el proceso pase desde cualquier parte de su código al estado muerto mediante la ejecución de la sentencia *exit vss_live*. En este caso, al comienzo del proceso aparece una sección encargada de llevar a cabo el reconocimiento de la muerte

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

Proceso original VSS

```
padre:process begin
   fork1:par_fork false
      simple pr1:    process begin acc1; end process;
   || iterative pr2: process begin acc2; end process;
   || simple pr3:    process begin acc3; end process;
   || iterative pr4: process
                        initialize begin acc4; end;
                     begin acc5; end process;
   end fork;
   kill pr3; initialize pr4;
end process;
```

Traducción VHDL del proceso pr1

```
pr1:process
begin
   if {pr1}_start='z' then
      wait until {pr1}_start='0';
   end if;
   {pr1}_start_ack<='0';
   wait until {pr1}_start='z';
   --*****inicio cuerpo principal****
     acc1 ;
   --******fin cuerpo principal *******
   {pr1}_start_ack<='z';wait for 0 ns;
end process;
```

Traducción VHDL del proceso pr2

```
pr2:process
begin
   if {pr2}_start='z' then
      wait until {pr2}_start='0'; end if;
   {pr2}_start_ack<='0';
   wait until {pr2}_start='z';
   vss_live:while true loop
      --*****inicio cuerpo principal****
        acc2 ;
      --******fin cuerpo principal *******
   end loop;
   {pr2}_start_ack<='z'; wait for 0 ns;
end process;
```

Traducción VHDL del proceso pr3

```
pr3:process
begin

while {pr3}_extra_kill='0' loop
   {pr3}_extra_kill_ack<='0';
   wait until {pr3}_extra_kill='z';
   {pr3}_extra_kill_ack<='z';
   wait for 0 ns;
end loop;

if {pr3}_start='z' then
   wait until {pr3}_start='0' or {pr3}_extra_kill='0';
end if;

if {pr3}_extra_kill='z' then
   {pr3}_start_ack<='0';
   wait until {pr3}_start='z';

   vss_live:for i in 0 to 0 loop
      --*****inicio cuerpo principal****
        acc3 ;
      --******fin cuerpo principal *******
   end loop;

   {pr3}_start_ack<='z';
   wait for 0 ns;
end if;

end process;
```

Traducción VHDL del proceso pr4

```
pr4:process
begin
while {pr4}_init='0' or {pr4}_extra_kill='0' loop
   {pr4}_extra_kill_ack<='0';
   wait until {pr4}_extra_kill='z';
   {pr4}_extra_kill_ack<='z'; wait for 0 ns;
   if {pr4}_init='0' then
      {pr4}_init_ack<='0';
      wait until {pr4}_init='z';
      --**** cuerpo de inicializacion *******
      acc4 ;
      --**** fin cuerpo de inicializacion ****
      {pr4}_init_ack<='z'; wait for 0 ns;
   end if;
end loop;
if {pr4}_start='z' then
   wait until {pr4}_start='0' or {pr4}_extra_kill='0';
end if;
if {pr4}_extra_kill='z' then
   {pr4}_start_ack<='0';
   wait until {pr4}_start='z';
   vss_live:while true loop
      --*****inicio cuerpo principal****
      acc5 ;
      --******fin cuerpo principal *******
   end loop;
   vss_je_pr4_start_ack<='z'; wait for 0 ns;
end if;
end process;
```

Ejemplo 4.1. Código generado para procesos simples (pr1, pr3) e iterativos (pr2) sin cuerpo de inicialización, y para un proceso iterativo con cuerpo de inicialización (pr4).

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

del proceso. Esta sección es un bucle controlado por la activación de la señal *kill*, en el que se completa un protocolo de *handshaking* entre las señales de activación (*kill*) y reconocimiento (*kill_ack*) relacionadas con la sentencia *kill*.

El proceso *pr4* se diferencia del proceso *pr3* en dos aspectos: a) se ejecuta en modo iterativo, y b) puede ser inicializado. Como se ejecuta en modo iterativo, el cuerpo del proceso se encierra en un bucle infinito etiquetado como *vss_live*. Por otro lado, el cuerpo de inicialización se incluye en la primera sección del proceso, dentro del mismo bucle que completa el protocolo de *handshaking* de la sentencia *kill*. Esto se hace así porque, como se explicará posteriormente, la sentencia *initialize* se implementa como una sentencia *kill* seguida de la inicialización. La ejecución del cuerpo de inicialización y del protocolo de *handshaking* de la sentencia *initialize* se encuentra en el interior de una sentencia *if* controlada por la activación de la señal *init*.

Puesto que el proceso que ejecuta una sentencia *kill* debe esperar al reconocimiento del proceso víctima y de toda su descendencia, puede darse el caso de que un proceso que no está activo deba reconocer su muerte. Por esta razón, los procesos *pr3* y *pr4*, cuando están a la espera de ser activados, *testean* la señal *kill*, y en caso de que esté activa, ejecutan la primera sección sin ejecutar sus cuerpos principales.

4.7.1. Propagación de la señal *kill*

Para la desactivación jerárquica de procesos es preciso un esquema que propague adecuadamente las señales a través de la jerarquía de procesos. Puesto que la sentencia *initialize* se implementa inicialmente como una sentencia *kill*, el esquema sólo debe propagar las señales de desactivación y reconocimiento de la señal *kill* a través de las diferentes generaciones de procesos.

En la Figura 4.3 se muestran el esquema de propagación de las señales para la implementación de la desactivación jerárquica. Cada círculo representa a un proceso de la jerarquía. En este ejemplo se ha asociado un código binario a cada proceso en función de la rama (izquierda o derecha) de la jerarquía. Además, hemos supuesto que los procesos cuyo código acaba en 1 pueden ser muertos o inicializados directamente, mientras que los acabados en 0 sólo mueren cuando lo hace algún antecesor. En el caso de la propagación de la señal de desactivación (*kill*), los procesos que solamente pueden morir por herencia, *testean* la misma señal que su padre, por lo que no declaran señal alguna. Por el contrario, los procesos que son objeto directo de alguna señal *kill* o *initialize* deben *testear* tanto la muerte del proceso padre como las sentencias dirigidas directamente hacia ellos. Estos procesos declaran las señales *{pr}_extra_kill* y *{pr}_kill*. La propagación de la señal de reconocimiento (*kill_ack*) sólo debe producirse cuando el

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

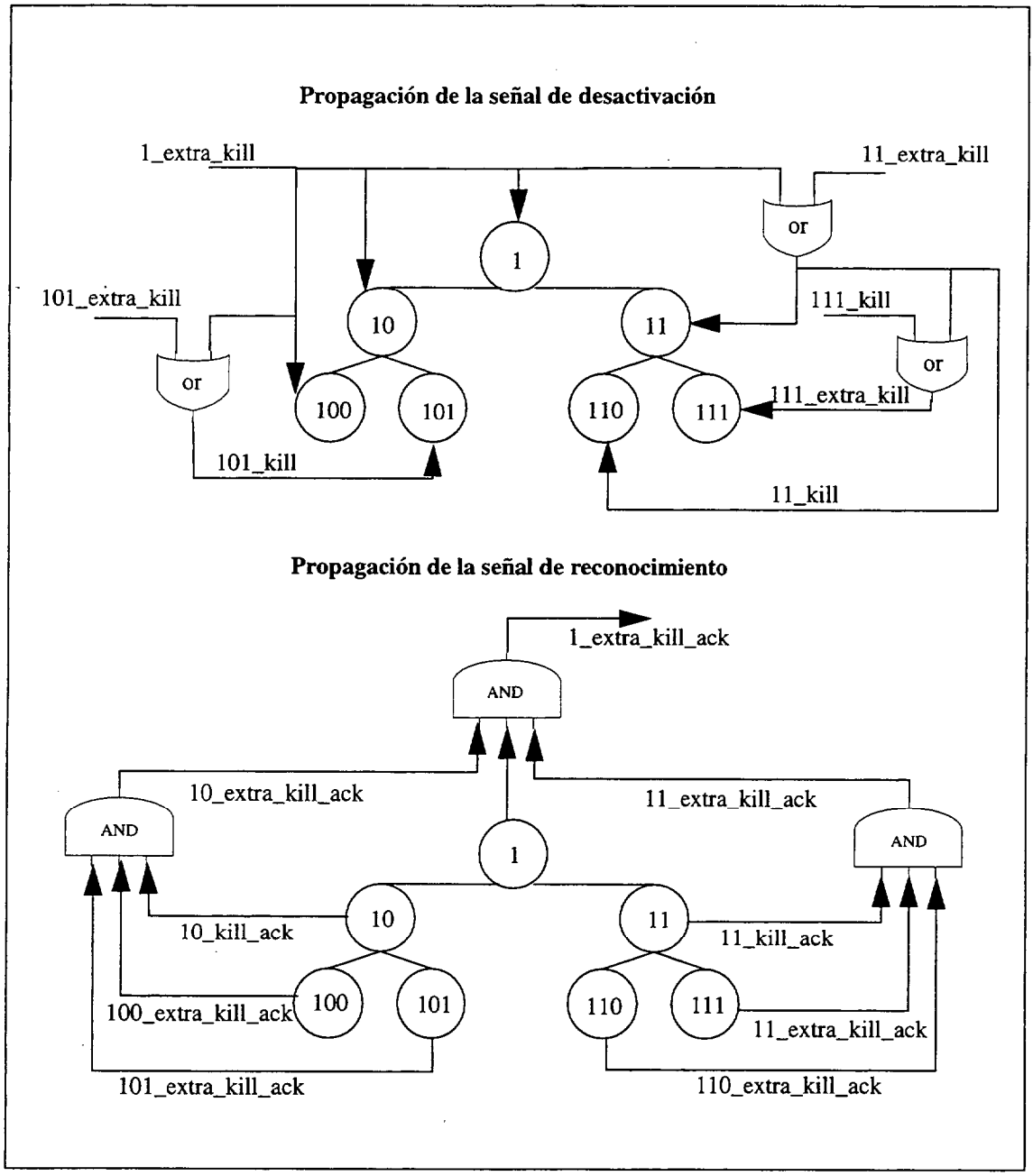


Figura 4.3. Esquema de propagación de las señales de activación y reconocimiento a través de la jerarquía de procesos.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

proceso que es objeto de una sentencia *kill* y toda su descendencia reconozcan su muerte. Los procesos hijos que carecen de descendencia sólo necesitan la señal *{pr}_extra_kill_ack*, mientras que los que tienen descendencia necesitan las señales *{pr}_extra_kill_ack* y *{pr}_kill_ack*.

4.7.2. Sentencia *par_fork*

La sentencia *par_fork* activa la ejecución de sus procesos hijos cuando la expresión booleana que la acompaña es cierta. En el Ejemplo 4.2. se muestra el código VHDL que se genera para una sentencia *par_fork*. El código que se muestra supone que la sentencia tiene dos procesos hijos (*pr1* y *pr2*), siendo sencilla la generalización para cualquier número de procesos. Como se observa en el ejemplo, la traducción se basa en un sencillo protocolo de *handshaking* para arrancar los procesos hijos.

4.7.3. Sentencia *seq_fork*

La sentencia *seq_fork* también se utiliza para especificar la jerarquía de procesos del sistema. La única diferencia con la sentencia *par_fork* es que el cuerpo de un proceso hijo finaliza activando al siguiente proceso hijo. De este modo se consigue ejecutar secuencialmente los procesos especificados en la sentencia. En el Ejemplo 4.3 se muestra el código de un proceso hijo especificado en una sentencia *seq_fork*.

El proceso que ejecuta la sentencia *seq_fork* activa al primer proceso hijo cuando la expresión booleana que acompaña a la sentencia es cierta. En el Ejemplo 4.4 se muestra el código generado para el proceso que ejecuta la sentencia.

Traducción del par_fork true

```
{pr1}_start<='0';
{pr2}_start<='0';

wait until {pr1}_start_ack='0'
  and {pr2}_start_ack='0' ;

{pr1}_start <='z';
{pr2}_start <='z';
wait for 0 ns;
```

Traducción del par_fork expression

```
if expression then
  {pr1}_start<='0';
  {pr2}_start<='0';
  wait until {pr1}_start_ack='0'
    and {pr2}_start_ack='0' ;
  {pr1}_start <='z';
  {pr2}_start <='z';
  wait for 0 ns;
end if;
```

Ejemplo 4.2. Traducción de la sentencia *par_fork* en los casos en que la expresión booleana que la acompaña es la constante *true* o bien es una expresión genérica.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

```
pr1:process
begin
  while {pr1}_init='0' or {pr1}_extra_kill='0' loop
    {pr1}_extra_kill_ack<='0'; wait until {pr1}_extra_kill='z';
    {pr1}_extra_kill_ack<='z'; wait for 0 ns;
    if {pr1}_init='0' then
      {pr1}_init_ack<='0'; wait until {pr1}_init='z';
      --**** cuerpo de inicializacion *******
      init5 ;
      --**** fin cuerpo de inicializacion *******
      {pr1}_init_ack<='z'; wait for 0 ns;
    end if;
  end loop;
  if {pr1}_start='z' then
    wait until {pr1}_start='0' or {pr1}_extra_kill='0';
  end if;
  if {pr1}_extra_kill='z' then
    {pr1}_start_ack<='0'; wait until {pr1}_start='z';
    vss_live:for i in 0 to 0 loop
      --*****inicio cuerpo principal****
        acc5 ;
      --******fin cuerpo principal *******
      --**** seq_fork arranque del hermano ****
        {pr2}_start<='0';
        wait until {pr2}_start_ack='0';
        {pr2}_start<='z'; wait for 0 ns;
      --**** fin seq_fork arranque
    end loop;
    {pr1}_start_ack<='z'; wait for 0 ns;
  end if;
end process;
```

Ejemplo 4.3. Código VHDL de un proceso hijo especificado en una sentencia *seq_fork*.

Traducción del seq_fork true

```
{pr1}_start<='0' ;
wait until {pr1}_start_ack='0';
{pr1}_start <='z' ;
 wait for 0 ns;
```

Traducción del seq_fork expression

```
if expression then
  {pr1}_start<='0' ;
  wait until {pr1}_start_ack='0';
  {pr1}_start <='z' ;
  wait for 0 ns;
end if;
```

Ejemplo 4.4. Traducción de la sentencia *seq_fork* para los casos en que la expresión booleana que la acompaña es la constante *true* o bien una expresión genérica.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

4.7.4. Sentencias *kill*, *initialize*, *start* y *join*

Para explicar la traducción de las sentencias *kill* e *initialize* se utiliza el Ejemplo 4.5. En la traducción de la sentencia *kill*, la orden de muerte se transmite mediante la señal *kill* asociada a cada uno de los procesos identificados en la sentencia. Los estados por los que pasan las señales *extra_kill* y *extra_kill_ack* son los típicos de un protocolo de *handshaking*. La sentencia *if* que aparece en la traducción impide el bloqueo en el caso de que otro proceso hubiese ejecutado la misma sentencia *kill*.

La sentencia *initialize* se traduce como una sentencia *kill* (que se propaga a todos sus procesos descendientes) seguida de un comando de inicialización que sólo afecta a los procesos implicados en la sentencia. Análogamente a la sentencia *kill*, la primera sentencia *wait* del protocolo de inicialización se encierra en una sentencia *if* para evitar el bloqueo de los procesos.

La traducción de la sentencia *start* depende del valor de la expresión booleana que la acompaña. Cuando la expresión es la constante *true*, la sentencia espera a que el proceso que se quiere arrancar muera y se encuentre a la espera de ser activado. Si la expresión

Traducción de la sentencia *kill son1, son2*

```
{son1}_extra_kill<='0';
{son2}_extra_kill<='0';
if {son1}_extra_kill_ack='z'or {son2}_extra_kill_ack='z' then
   wait until {son1}_extra_kill_ack='0'and {son2}_extra_kill_ack='0' ;
end if;
{son1}_extra_kill<='z';
{son2}_extra_kill<='z';
wait until {son1}_extra_kill_ack='z'and {son2}_extra_kill_ack='z';
```

Traducción de la sentencia *initialize son1, son2*

```
{son1}_extra_kill<='0'; {son1}_init<='0';
{son2}_extra_kill<='0'; {son2}_init<='0';
if {son1}_extra_kill_ack='z'or {son2}_extra_kill_ack='z' then
   wait until {son1}_extra_kill_ack='0'and {son2}_extra_kill_ack='0' ;
end if;
{son1}_extra_kill<='z';
{son2}_extra_kill<='z';
wait until {son1}_extra_kill_ack='z'and {son2}_extra_kill_ack='z' ;
if {son1}_init_ack='z'or {son2}_init_ack='z'then
   wait until {son1}_init_ack='0'and {son2}_init_ack='0' ;
end if;
{son1}_init<='z';
{son2}_init<='z';
wait until {son1}_init_ack='z'and {son2}_init_ack='z' ;
```

Ejemplo 4.5. Traducción de las sentencias *kill* e *initialize* actuando simultáneamente sobre dos procesosos.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

Traducción de la sentencia *start true son1*

```
if {padre}_kill='0' then
   exit vss_live;
end if;
{son1}_start<='0';
wait until {son1}_start_ack='0' or {padre}_kill='0';
{son1}_start<='z';
wait for 0 ns;
if {padre}_kill='0' then
   exit vss_live;
end if;
```

Traducción de la sentencia *start false son1*

```
if {padre}_kill='0' then exit vss_live;
end if;
if {son1}_start_ack='z' then
   {son1}_start<='0';
   wait until {son1}_start_ack='0' or {padre}_kill='0';
   {son1}_start<='z'; wait for 0 ns;
end if;
if {padre}_kill='0' then exit vss_live;
end if;
```

Traducción de la sentencia *start expresion son1*

```
if {padre}_kill='0' then
   exit vss_live;
end if;
if expresion then
   if {son1}_start_ack='0' then
      wait until {son1}_start_ack='z';
   end if;
   son1}_start<='0';
   wait until {son1}_start_ack='0' or {padre}_kill='0';
   {son1}_start<='z';
   wait for 0 ns;
else
   if {son1}_start_ack='z' then
      {son1}_start<='0';
      wait until {son1}_start_ack='0' or {padre}_kill='0';
      {son1}_start<='z';
      wait for 0 ns;
   end if;
end if;
if {padre}_kill='0' then exit vss_live; end if;
```

Ejemplo 4.6. Traducción de la sentencia *start* en función de la expresión *booleana*.

es la constante *false* y el proceso está activo, la sentencia no tiene efecto (se ignora). En cualquier otro caso, la sentencia se traduce como la suma lógica de las dos posibilidades anteriores. En el Ejemplo 4.6 se muestran las tres traducciones posibles.

Para evitar el bloqueo del proceso que arranca un hijo cuando con anterioridad, o simultáneamente, un antecesor suyo ejecuta una sentencia *kill* sobre él, la traducción de la sentencia *start* comprueba la señal *kill* antes de iniciar la activación del proceso hijo y durante la espera de la señal de reconocimiento del hijo.

La sentencia *join* permite sincronizar un proceso con la finalización de la ejecución de los procesos indicados en la sentencia. Si estos procesos no están activos, la sentencia se ignora. La sentencia se traduce tal como se muestra en el Ejemplo 4.7.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

Traducción de la sentencia *join :son1,son2*

```
if {son1}_start_ack='0' or {son2}_start_ack='0'then
   if {padre}_extra_kill='0' then
      exit vss_live;
   end if;
   wait until ({son1}_start_ack='z' and {son2}_start_ack='z') or {padre}_extra_kill='0';
   if {padre}_extra_kill='0' then
      exit vss_live;
   end if;
end if;
```

Ejemplo 4.7. Traducción de la sentencia *join*.

4.8. Traducción de las sentencias de comunicación

En este apartado se describe la traducción de las distintas sentencias de comunicación. En primer lugar, se describe la traducción de las sentencias de comunicación síncrona y asíncrona en modo simple. En segundo lugar, se describe la traducción de las sentencias de comunicación síncrona y asíncrona en modo *broadcast*. Finalmente, se describe la traducción de las sentencias de comunicación síncrona y asíncrona en modo *incast*.

4.8.1. Comunicación síncrona en modo simple

Todos los esquemas de comunicación soportados en VSS toman como base la comunicación síncrona en modo simple. El protocolo de *handshaking* implementado en todos los esquemas de comunicación necesita siempre de un bus de datos, una señal de indicación de dato preparado (*rdy*) y una señal de reconocimiento (*ack*). La longitud del bus de datos se determina siempre a partir del tipo de dato que transporta el canal. En el Ejemplo 4.8 se muestran las señales declaradas y la traducción de las sentencias *send* y *receive* en el caso de una comunicación síncrona.

La estructura de código generado en la traducción de las sentencias *send* y *receive* es siempre la misma, independientemente del esquema de comunicación utilizado. En el caso de la sentencia *send*, primero se depositan los datos en los canales y se activan las señales *rdy* de cada canal. Luego se ejecuta un bucle hasta que se completan todos los protocolos de comunicación de la sentencia *send* (todas las señales *rdy* y *ack* desactivadas) o hasta que el proceso muera. En el interior del bucle, para cada uno de los canales implicados en la sentencia, existe una sentencia *if* que actualiza las señales, espera por los cambios de estado de las señales de comunicación y comprueba la señal

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

Código VSS

```
entity comu is end;
architecture behav of comu is
   type byte is range -128 to 127;
   channel ch1,ch2:byte simple;
begin
   padre:process
   begin
      fork1:par_fork true

      simple son1:process
      begin
         send(ch1,expr1;ch2,expr2);
      end process;

      ||

      simple son2:process
      begin
         receive(ch1,var1;ch2,var2);
      end process;

      end fork;

      kill son1,son2;
   end process;
end;
```

Señales declaradas

```
signal {ch1}_dato:signed(7 downto 0):=(others=>'z');
signal {ch1}_rdy,{ch1}_ack:std_logic:='z';
signal {ch2}_dato:signed(7 downto 0):=(others=>'z');
signal {ch2}_rdy,{ch2}_ack:std_logic:='z';
```

Traducción del send(ch1,expr1;ch2,expr2)

```
{ch1}_dato<=to_signed(integer(expr1),8); {ch1}_rdy<='0';
{ch2}_dato<=to_signed(integer(expr2),8); {ch2}_rdy<='0';
wait for 0 ns;
while {ch1}_rdy='0' or {ch1}_ack='0' or {ch2}_rdy='0' or
      {ch2}_ack='0'loop
   if {ch1}_ack='0' then
      {ch1}_rdy<='z' ; {ch1}_dato<="zzzzzzzz";
   end if;
   if {ch2}_ack='0' then
      {ch2}_rdy<='z' ; {ch2}_dato<="zzzzzzzz";
   end if;
   wait on {ch1}_ack,{ch1}_rdy,{ch2}_ack,{ch2}_rdy,{son1}_extra_kill;
   if {son1}_extra_kill='0' then
      {ch1}_dato<="zzzzzzzz";
      {ch1}_rdy<='z'; {ch2}_dato<="zzzzzzzz";
      {ch2}_rdy<='z'; wait for 0 ns;
      exit vss_live;
   end if;
end loop;
```

Traducción del receive(ch1,var1;ch2,var2)

```
{ch1}_stop:=true;
{ch2}_stop:=true;
while {ch1}_stop or {ch2}_stop loop
   if {ch1}_stop and {ch1}_rdy='0' then
      var1:=byte(to_integer({ch1}_dato));
      {ch1}_ack<='0';
   else
      if {ch1}_stop and {ch1}_ack='0' then
         {ch1}_ack<='z';
         {ch1}_stop:=false;
      end if;
   end if;
   if {ch2}_stop and {ch2}_rdy='0' then
      var2:=byte(to_integer({ch2}_dato));
      {ch2}_ack<='0';
   else
      if {ch2}_stop and {ch2}_ack='0' then
         {ch2}_ack<='z';
         {ch2}_stop:=false;
      end if;
   end if;
   wait on {ch1}_ack,{ch1}_rdy,{ch2}_ack,{ch2}_rdy,{son2}_extra_kill;
   if {son2}_extra_kill='0' then
      {ch1}_ack<='z';
      {ch2}_ack<='z';
      wait for 0 ns;
      exit vss_live;
   end if;
end loop;
```

Ejemplo 4.8. Código generado para las sentencias *send* y *receive* de comunicación síncrona múltiple en modo simple.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

kill (permitiendo la muerte del proceso). La sentencia *if* final desactiva las señales *rdy* y *dato* de todos los canales cuando el proceso es muerto o inicializado. La última sentencia *if* y la señal *kill* de la sentencia *wait* no se generan cuando el proceso no puede ser muerto o inicializado.

La traducción de la sentencia *receive* es muy similar a la sentencia *send*. En un *bucle* que sólo finaliza cuando se completa la comunicación con todos los canales o cuando el proceso se muere, existen una serie de sentencias *if* que se encargan de completar la comunicación con cada canal, y una sentencia *wait* que espera por los cambios de estado de las señales de control o por el comando *kill*. La única diferencia con la sentencia *send* es que se utilizan las variables lógicas *(CH)_stop* para indicar que se ha completado la recepción por un canal. Cada proceso declara tantas variables de éstas como canales distintos haya en una sentencia *receive*.

4.8.2. Comunicación asíncrona en modo simple

Todos los esquemas de comunicación asíncronos se implementan mediante un *buffer* intermedio. En el caso de una comunicación asíncrona en modo simple, el esquema de traducción es equivalente a dos comunicaciones síncronas en modo simple, una entre el transmisor y el *buffer*, y otra entre el *buffer* y el receptor. El número de señales necesarias es el doble que en el caso anterior, añadiéndose el sufijo *_in* a las señales implicadas en la comunicación *transmisor-buffer* y el sufijo *_out* a las relacionadas con la comunicación *buffer-receptor*. En el Ejemplo 4.9 se muestran las señales declaradas y la traducción de las sentencias *send* y *receive* en el caso de que los canales del Ejemplo 4.8 fueran asíncronos. El *buffer* se encuentra en la librería VSS, y dependiendo del tamaño del *buffer* se instancia uno de los componentes de la Figura 4.4.

```
component buffer_1_simple
      generic (lineas:positive);
      port(rdy_in,ack_out:in std_logic; rdy_out,ack_in:out std_logic;
         dato_in:in signed(lineas-1 downto 0); dato_out:out signed(lineas-1 downto 0);
         reset:in std_logic; full,empty:out boolean);
end component;


component buffer_n_simple
      generic (lineas:positive;tama_buf:positive);
      port(rdy_in,ack_out:in std_logic; rdy_out,ack_in:out std_logic;
         dato_in:in signed(lineas-1 downto 0); dato_out:out signed(lineas-1 downto 0);
         reset:in std_logic;full,empty:out boolean);
end component;
```

Figura 4.4. Declaración VHDL de los componentes que intervienen en la implementación de los canales de comunicación asíncronos de tipo *simple* o *incast*.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

Señales del canal asíncrono channel ch1:byte buffer 1 simple

```
signal {ch1}_dato_in,{ch1}_dato_out:signed(7 downto 0):=(others=>'z');
signal {ch1}_rdy_in,{ch1}_rdy_out,{ch1}_ack_in,{ch1}_ack_out,{ch1}_reset:std_logic:='z';
signal {ch1}_full,{ch1}_empty:boolean:=false;
```

Traducción de la sentencia send(ch1,expr1;ch2,expr2)

```
{ch1}_dato_in<=to_signed(integer(expr1),8); {ch1}_rdy_in<='0';
{ch2}_dato_in<=to_signed(integer(expr2),8); {ch2}_rdy_in<='0';
wait for 0 ns;
while {ch1}_rdy_in='0' or {ch1}_ack_in='0' or {ch2}_rdy_in='0' or {ch2}_ack_in='0' loop
  if {ch1}_ack_in='0' then {ch1}_rdy_in<='z' ; {ch1}_dato_in<="zzzzzzzz"; end if;
  if {ch2}_ack_in='0' then {ch2}_rdy_in<='z' ; {ch2}_dato_in<="zzzzzzzz"; end if;
  wait on {ch1}_ack_in,{ch1}_rdy_in,{ch2}_ack_in,{ch2}_rdy_in,{son1}_extra_kill;
  if {son1}_extra_kill='0' then
    {ch1}_dato_in<="zzzzzzzz";
    {ch1}_rdy_in<='z';
    {ch2}_dato_in<="zzzzzzzz";
    {ch2}_rdy_in<='z';
    wait for 0 ns;
    exit vss_live;
  end if;
end loop;
```

Traducción de la sentencia receive(ch1,var1;ch2,var2)

```
{ch1}_stop:=true;
{ch2}_stop:=true;
while {ch1}_stop or {ch2}_stop loop
  if {ch1}_stop and {ch1}_rdy_out='0' then
    var1:=byte(to_integer({ch1}_dato_out));
    {ch1}_ack_out<='0';
  else
    if {ch1}_stop and {ch1}_ack_out='0' then
      {ch1}_ack_out<='z';
      {ch1}_stop:=false;
    end if;
  end if;
  if {ch2}_stop and {ch2}_rdy_out='0' then
    var2:=byte(to_integer({ch2}_dato_out));
    {ch2}_ack_out<='0';
  else
    if {ch2}_stop and {ch2}_ack_out='0' then
      {ch2}_ack_out<='z';
      {ch2}_stop:=false;
    end if;
  end if;
  wait on
{ch1}_ack_out,{ch1}_rdy_out,{ch2}_ack_out,{ch2}_rdy_out,{son2}_extra_kill;
  if {son2}_extra_kill='0' then
    {ch1}_ack_out<='z';
    {ch2}_ack_out<='z';
    wait for 0 ns;
    exit vss_live;
  end if;
end loop;
```

Ejemplo 4.9. Código generado para las sentencias *send* y *receive* de comunicación asíncrona múltiple en modo simple.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

Ambos componentes son genéricos en el número de líneas del bus de datos (*lineas*) y tienen la misma interfaz. Las señales *rdy_in*, *ack_in* y *dato_in* se utilizan para la comunicación simple síncrona entre transmisor y *buffer*, y las señales *rdy_out*, *ack_out* y *rdy_out* para la comunicación *buffer-receptor*. El *buffer_n_simple* se distingue del *buffer_1_simple* en que puede parametrizarse el tamaño del *buffer*, siendo su arquitectura algo más compleja. Ambos *buffers* disponen de una señal de *reset* que provoca la inicialización de las señales de control y datos, así como la pérdida de los datos que pudieran estar almacenados. Además, el *buffer* dispone de las salidas *full* y *empty* que son utilizadas para la traducción de las sentencias *channel_full* y *channel_empty*.

La traducción de las sentencias *send* y *receive* es idéntica al modelo síncrono, pero añadiendo los sufijos *_in* y *_out* a las señales de control y datos.

4.8.3. Comunicación síncrona en modo *broadcast*

El esquema de comunicación *broadcast* añade al esquema en modo simple un *bus* de identificación, y amplía la señal *ack* con un *bus* de tantas líneas de reconocimiento como procesos receptores estén conectados al canal. En el Ejemplo 4.10 se muestran las señales y constantes declaradas para este esquema de comunicación, así como la traducción de las sentencias *send* y *receive*.

Cada uno de los procesos receptores declara dos constantes: una le permite conocer si es el destinatario del mensaje (*{ch}_ProcessId_id*) y la otra le permite conocer la línea del *bus ack* (*{ch}_ProcessId_id*_nat) por la que tiene que efectuar el reconocimiento. Junto a las constantes anteriores aparece otra constante cuyo valor es cero que se utiliza para transmitir el mismo dato a todos los receptores (*{ch}_broadcast*) y otra que permite dejar el *bus* de identificación en alta impedancia (*{ch}_id_z*). Análogamente, y sólo en el modo síncrono, se declaran las constantes *{ch}_ack_z* y *{ch}_ack_0* que se utilizan para comprobar el estado del *bus ack* en la traducción de las sentencias *send*.

La traducción de la sentencia *send broadcast* se diferencia del *send* en modo simple en dos aspectos. Primero, el identificador del proceso destinatario se envía junto al dato. Segundo, el reconocimiento se realiza por la línea del *bus ack* asociada a ese proceso. Obsérvese que cuando se comunica el mismo dato a todos los receptores se espera por el reconocimiento de todos ellos.

La traducción de la sentencia *receive* en modo *broadcast* también se diferencia del *receive* en modo simple en dos aspectos. Primero, el proceso receptor debe comprobar que el dato va dirigido a él, comparando el identificador del bus con el suyo (*{ch}_ProcessId_id*) y con el identificador *broadcast* (*{ch}_ProcessId_broadcast*). Segundo, cada receptor efectúa el reconocimiento por su línea del *bus ack*.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

Código VSS

```
type byte is range -128 to 127;
channel ch1,ch2:byte broadcast;
begin
  padre:process
  begin
    fork1:par_fork true
      simple son1:process begin
        send(son2<=ch1,expr1;
          broadcast<=ch2,expr2);
      end process;
    || simple son2:process begin
        receive(ch1,var1;ch2,var2);
      end process;
    || simple son3:process begin
        receive(ch1,var1;ch2,var2);
      end process;
    end fork;
    kill son1,son2,son3;
  end process;
```

Señales que conforman el canal ch1

```
signal {ch1}_dato:signed(7 downto 0):=(others=>'z');
signal {ch1}_rdy:std_logic:='z';
signal {ch1}_ack:std_logic_vector(1 downto 0):=(others=>'z');
signal {ch1}_id:unsigned(1 downto 0):=(others=>'z');
```

Constantes de identificación de procesos para ch1

```
constant {ch1}_id_z:unsigned(1 downto 0):=(others=>'z');
constant {ch1}_ack_z:std_logic_vector(1 downto 0):=(others=>'z');
constant {ch1}_ack_0:std_logic_vector(1 downto 0):=(others=>'0');
constant {ch1}_son2_id:unsigned(1 downto 0):="01";
constant {ch1}_son2_id_nat:natural:=0;
constant {ch1}_son3_id:unsigned(1 downto 0):="10";
constant {ch1}_son3_id_nat:natural:=1;
constant {ch1}_broadcast:unsigned(1 downto 0):="00";
```

Traducción de la sentencia send(son2<=ch1,expr1;broadcast<=ch2,expr2)

```
{ch1}_dato<=to_signed(integer(expr1),8); {ch1}_rdy<='0';{ch1}_id<={ch1}_son2_id;
{ch2}_dato<=to_signed(integer(expr2),8); {ch2}_rdy<='0';{ch2}_id<={ch2}_broadcast;
wait for 0 ns;
while {ch1}_rdy='0' or {ch1}_ack /={ch1}_ack_z or {ch2}_rdy='0' or {ch2}_ack /={ch2}_ack_z loop
    if {ch1}_ack({ch1}_son2_id_nat)='0' then
      {ch1}_rdy<='z' ; {ch1}_id<={ch1}_id_z; {ch1}_dato<="zzzzzzzz";
    end if;
    if {ch2}_ack={ch2}_ack_0 then
      {ch2}_rdy<='z' ; {ch2}_id<={ch2}_id_z; {ch2}_dato<="zzzzzzzz";
    end if;
    wait on {ch1}_ack,{ch1}_rdy,{ch2}_ack,{ch2}_rdy,{son1}_extra_kill;
    if {son1}_extra_kill='0' then
      {ch1}_dato<="zzzzzzzz"; {ch1}_rdy<='z'; {ch1}_id<={ch1}_id_z;
      {ch2}_dato<="zzzzzzzz"; {ch2}_rdy<='z'; {ch2}_id<={ch2}_id_z;
      wait for 0 ns;exit vss_live;
    end if;
end loop;
```

Traducción de la sentencia receive(ch1,var1;ch2,var2)

```
{ch1}_stop:=true; {ch2}_stop:=true;
while {ch1}_stop or {ch2}_stop loop
  if {ch1}_stop and {ch1}_rdy='0' and ({ch1}_id={ch1}_son2_id or {ch1}_id={ch1}_broadcast) then
    var1:=byte(to_integer({ch1}_dato)); {ch1}_ack({ch1}_son2_id_nat)<='0';
  else
    if {ch1}_stop and {ch1}_ack({ch1}_son2_id_nat)='0' then
      {ch1}_ack({ch1}_son2_id_nat)<='z'; {ch1}_stop:=false;
    end if;
  end if;
  if {ch2}_stop and {ch2}_rdy='0' and ({ch2}_id={ch2}_son2_id or {ch2}_id={ch2}_broadcast) then
    var2:=byte(to_integer({ch2}_dato)); {ch2}_ack({ch2}_son2_id_nat)<='0';
  else
    if {ch2}_stop and {ch2}_ack({ch2}_son2_id_nat)='0' then
      {ch2}_ack({ch2}_son2_id_nat)<='z'; {ch2}_stop:=false;
    end if;
  end if;
  wait on {ch1}_ack,{ch1}_rdy,{ch2}_ack,{ch2}_rdy,{son2}_extra_kill;
  if {son2}_extra_kill='0' then
    {ch1}_ack({ch1}_son2_id_nat)<='z'; {ch2}_ack({ch2}_son2_id_nat)<='z';
    wait for 0 ns; exit vss_live;
  end if;
end loop;
```

Ejemplo 4.10. Código generado para las sentencias *send* y *receive* de comunicación síncrona múltiple en modo *broadcast*.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

4.8.4. Comunicación asíncrona en modo *broadcast*

Para este tipo de comunicaciones se intercala un *buffer* entre el proceso transmisor y todos los procesos receptores. La comunicación entre el transmisor y el *buffer* sólo se diferencia del esquema simple síncrono en que junto al dato se envía también el identificador del destinatario. La comunicación *buffer-receptores* es igual al esquema *broadcast* síncrono. Como en todos los esquemas asíncronos, las señales con el sufijo *_in* corresponden al canal de entrada al *buffer* y las señales con el sufijo *_out* al canal de salida del mismo. También se declaran las señales relacionadas con la sentencias *reset_buffer, buffer_full* y *buffer_empty.* A diferencia de la comunicación simple, estos *buffers* (Figura 4.5) memorizan tanto el dato como el identificador del destinatario, esperando por el reconocimiento del receptor apropiado (o de todos ellos en caso de un *broadcast*).

La traducción de la sentencia *receive* es idéntica a la del modelo *broadcast* síncrono añadiendo el sufijo *_out* a las señales de control y datos. La traducción de la sentencia *send* es igual a la del modelo simple síncrono, pero además del dato se envía el identificador del destinatario. El Ejemplo 4.11 muestra la traducción de estas sentencias así como la declaración de señales y constantes para el caso de una comunicación *broadcast* asíncrona.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

```
component buffer_1_broadcast
      generic (lineas_datos,lineas_id,receptores:positive);
      port(rdy_in:in std_logic;ack_out:in std_logic_vector(0 to receptores-1);
         rdy_out,ack_in:out std_logic; dato_in:in signed(lineas_datos-1 downto 0);
         dato_out:out signed(lineas_datos-1 downto 0);
         id_in:in unsigned(lineas_id-1 downto 0); id_out:out unsigned(lineas_id-1 downto 0);
         reset:in std_logic; full,empty:out boolean);
end component;


component buffer_n_broadcast
      generic (lineas_datos,lineas_id,tama_buf,receptores:positive);
      port(rdy_in:in std_logic;ack_out:in std_logic_vector(0 to receptores-1);
         rdy_out,ack_in:out std_logic; dato_in:in signed(lineas_datos-1 downto 0);
         dato_out:out signed(lineas_datos-1 downto 0);
         id_in:in unsigned(lineas_id-1 downto 0); id_out:out unsigned(lineas_id-1 downto 0);
         reset:in std_logic; full,empty:out boolean);
end component;
```

Figura 4.5. Declaración VHDL de los *buffers* que intervienen en la implementación de las comunicaciones asíncronas de tipo *broadcast.*

Código VSS

```
type byte is range -128 to 127;
channel ch1,ch2:byte buffer 1 broadcast;
begin
 padre:process
 begin
  fork1:par_fork true
     simple son1:process begin
       send(son2<=ch1,expr1;
       broadcast<=ch2,expr2);
     end process;
  || simple son2:process begin
       receive(ch1,var1;ch2,var2);
     end process;
  || simple son3:process begin
       receive(ch1,var1;ch2,var2);
     end process;
  end fork;
  kill son1,son2,son3;
end process;
```

Señales que conformal en canal ch1

```
signal {ch1}_dato_in,{ch1}_dato_out:signed(7 downto 0):=(others=>'z');
signal {ch1}_reset,{ch1}_ack_in,{ch1}_rdy_in, {ch1}_rdy_out:std_logic:='z';
signal {ch1}_full,{ch1}_empty:boolean:=false;
signal {ch1}_ack_out:std_logic_vector(1 downto 0):=(others=>'z');
signal {ch1}_id_in,{ch1}_id_out:unsigned(1 downto 0):=(others=>'z');
```

Constantes de identificación de procesos para ch1

```
constant {ch1}_id_z:unsigned(1 downto 0):=(others=>'z');
constant {ch1}_son2_id:unsigned(1 downto 0):="01";
constant {ch1}_son2_id_nat:natural:=0;
constant {ch1}_son3_id:unsigned(1 downto 0):="10";
constant {ch1}_son3_id_nat:natural:=1;
constant {ch1}_broadcast:unsigned(1 downto 0):="00";
```

Traducción de la sentencia send(son2<=ch1,expr1;broadcast<=ch2,expr2)

```
{ch1}_dato_in<=to_signed(integer(expr1),8); {ch1}_rdy_in<='0'; {ch1}_id_in<={ch1}_son2_id;
{ch2}_dato_in<=to_signed(integer(expr2),8); {ch2}_rdy_in<='0'; {ch2}_id_in<={ch2}_broadcast;
wait for 0 ns;
while {ch1}_rdy_in='0' or {ch1}_ack_in='0' or {ch2}_rdy_in='0' or {ch2}_ack_in='0'loop
  if {ch1}_ack_in='0' then
     {ch1}_rdy_in<='z' ; {ch1}_id_in<={ch1}_id_z; {ch1}_dato_in<="zzzzzzzz";
  end if;
  if {ch2}_ack_in='0' then
     {ch2}_rdy_in<='z' ; {ch2}_id_in<={ch2}_id_z; {ch2}_dato_in<="zzzzzzzz";
  end if;
  wait on {ch1}_ack_in,{ch1}_rdy_in,{ch2}_ack_in,{ch2}_rdy_in,{son1}_extra_kill;
  if {son1}_extra_kill='0' then
     {ch1}_dato_in<="zzzzzzzz"; {ch1}_rdy_in<='z'; {ch1}_id_in<={ch1}_id_z;
     {ch2}_dato_in<="zzzzzzzz"; {ch2}_rdy_in<='z'; {ch2}_id_in<={ch2}_id_z;
     wait for 0 ns; exit vss_live;
  end if;
end loop;
```

Traducción del receive(ch1,var1;ch2,var2)

```
{ch1}_stop:=true; {ch2}_stop:=true;
while {ch1}_stop or {ch2}_stop loop
  if {ch1}_stop and {ch1}_rdy_out='0' and ({ch1}_id_out={ch1}_son2_id or {ch1}_id_out={ch1}_broadcast) then
       var1:=byte(to_integer({ch1}_dato_out)); {ch1}_ack_out({ch1}_son2_id_nat)<='0';
  else
     if {ch1}_stop and {ch1}_ack_out({ch1}_son2_id_nat)='0' then
        {ch1}_ack_out({ch1}_son2_id_nat)<='z'; {ch1}_stop:=false;
     end if;
  end if;
  if {ch2}_stop and {ch2}_rdy_out='0' and ({ch2}_id_out={ch2}_son2_id or {ch2}_id_out={ch2}_broadcast) then
       var2:=byte(to_integer({ch2}_dato_out)); {ch2}_ack_out({ch2}_son2_id_nat)<='0';
  else
     if {ch2}_stop and {ch2}_ack_out({ch2}_son2_id_nat)='0' then
        {ch2}_ack_out({ch2}_son2_id_nat)<='z'; {ch2}_stop:=false;
     end if;
  end if;
  wait on {ch1}_ack_out,{ch1}_rdy_out,{ch2}_ack_out,{ch2}_rdy_out,{son2}_extra_kill;
  if {son2}_extra_kill='0' then
     {ch1}_ack_out({ch1}_son2_id_nat)<='z'; {ch2}_ack_out({ch2}_son2_id_nat)<='z';
     wait for 0 ns; exit vss_live;
  end if;
end loop;
```

Ejemplo 4.11. Código generado para las sentencias *send* y *receive* de comunicación asíncrona múltiple en modo *broadcast*

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

4.8.5. Comunicación síncrona en modo *incast*

Este esquema de comunicación se traduce añadiendo un árbitro al esquema de comunicación simple síncrona para solucionar los conflictos entre los transmisores. El árbitro es el mismo que se utiliza en la traducción de las variables compartidas, necesitándose una constante de identificación, una señal de petición (*lock*) y otra de reconocimiento (*locked*) por cada proceso transmisor. La constante la utilizan los procesos para conocer la línea de los buses *lock* y *locked* por la que deben efectuar su petición y esperar por el reconocimiento. En el Ejemplo 4.12 se muestran las señales y constantes generadas, así como la traducción de las sentencias *send* y *receive*, para el caso de una comunicación síncrona *incast*.

La traducción de la sentencia *send* comienza activando la señal de petición del canal. En el interior del *bucle*, las sentencias *if* llevan a cabo la comunicación síncrona simple cuando el árbitro les concede el recurso, liberándolo una vez que se completa la comunicación. Si un proceso resulta muerto o inicializado, además de dejar en reposo las señales del canal, también debe liberarlo (*{ch}_lock({ch}_ProcessId_id_nat)<='Z';*). La traducción de la sentencia *receive* es idéntica al modo síncrono simple.

4.8.6. Comunicación asíncrona en modo *incast*

Este esquema se traduce añadiendo un *buffer* entre los procesos transmisores y el proceso receptor. La comunicación *transmisores-buffer* es síncrona *incast* y la transmisión *buffer-receptor* es síncrona simple. Las señales y constantes declaradas son las necesarias para establecer estos dos esquemas de comunicación, añadiendo el sufijo *_in* a las señales del canal de entrada al *buffer*, y el sufijo *_out* a las del canal de salida. La traducción de la sentencia *send* es la misma que en una comunicación síncrona *incast*, pero añadiendo el sufijo *_in* a las señales de control y datos del canal. Análogamente, la sentencia *receive* se traduce como en una comunicación simple síncrona, pero añadiendo el sufijo *_out* a las señales de control y datos del canal.

4.8.7. Sentencias *resetbuffer*, *buffer_empty* y *buffer_full*

La traducción de las sentencias *buffer_empty* y *buffer_full* se realiza simplemente sustituyendo la llamada a la función por el nombre de la señal conectada a las salidas *empty* y *full* del *buffer*, respectivamente. La traducción de la sentencia *reset_buffer* mantiene activa la señal de *reset* del *buffer* durante un ciclo de simulación. En el Ejemplo 4.13 se muestra la traducción de estas sentencias.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

Código VSS

```
type byte is range -128 to 127;
channel ch1,ch2:byte incast;
begin
padre:process
begin
  fork1:par_fork true
    simple son1:process begin
      send(ch1,expr1;ch2,expr2);
    end process;
  || simple son2:process begin
      send(ch1,expr1;ch2,expr2);
    end process;
  || simple son3:process begin
      receive(ch1,var1;ch2,var2);
    end process;
  end fork;
  kill son1,son2,son3;
end process;
```

Constantes de los tranmisores del canal ch1

```
constant {ch1}_son1_id_nat:natural:=0;
constant {ch1}_son2_id_nat:natural:=1;
```

Señales relacionadas con el canal ch1

```
signal {ch1}_lock,{ch1}_locked:std_logic_vector(0 to 1):=(others=>'z');
signal {ch1}_dato:signed(7 downto 0):=(others=>'z');
signal {ch1}_ack,{ch1}_rdy:std_logic:='z';
```

Traducción de la sentencia send(ch1,expr1;ch2,expr2)

```
{ch2}_lock({ch2}_son1_id_nat)<='0';
wait for 0 ns;
while {ch1}_lock({ch1}_son1_id_nat)='0' or {ch1}_locked({ch1}_son1_id_nat)='0' or
      {ch1}_ack='0' or {ch2}_lock({ch2}_son1_id_nat)='0' or {ch2}_locked({ch2}_son1_id_nat)='0' or {ch2}_ack='0' loop
  if {ch1}_locked({ch1}_son1_id_nat)='0' and {ch1}_rdy='z' and {ch1}_ack='z' then
    {ch1}_dato<=to_signed(integer(expr1),8); {ch1}_rdy<='0';
  elsif {ch1}_locked({ch1}_son1_id_nat)='0' and {ch1}_rdy='0' and {ch1}_ack='0' then
    {ch1}_rdy<='z' ; {ch1}_dato<="zzzzzzzz"; {ch1}_lock({ch1}_son1_id_nat)<='z';
  end if;
  if {ch2}_locked({ch2}_son1_id_nat)='0' and {ch2}_rdy='z' and {ch2}_ack='z' then
    {ch2}_dato<=to_signed(integer(expr2),8); {ch2}_rdy<='0';
  elsif {ch2}_locked({ch2}_son1_id_nat)='0' and {ch2}_rdy='0' and {ch2}_ack='0' then
      {ch2}_rdy<='z' ; {ch2}_dato<="zzzzzzzz"; {ch2}_lock({ch2}_son1_id_nat)<='z';
  end if;
  wait on {ch1}_ack,{ch1}_rdy,{ch1}_lock,{ch1}_locked,{ch2}_ack,{ch2}_rdy,{ch2}_lock,{ch2}_locked,{son1}_extra_kill;
  if {son1}_extra_kill='0' then
    {ch1}_dato<="zzzzzzzz"; {ch1}_rdy<='z'; {ch1}_lock({ch1}_son1_id_nat)<='z';
    {ch2}_dato<="zzzzzzzz"; {ch2}_rdy<='z'; {ch2}_lock({ch2}_son1_id_nat)<='z';
    wait for 0 ns;exit vss_live;
  end if;
end loop;
```

Traducción del receive(ch1,var1;ch2,var2)

```
{ch1}_stop:=true;{ch2}_stop:=true;
while {ch1}_stop or {ch2}_stop loop
  if {ch1}_stop and {ch1}_rdy='0' then
    var1:=byte(to_integer({ch1}_dato)); {ch1}_ack<='0';
  else
    if {ch1}_stop and {ch1}_ack='0' then {ch1}_ack<='z'; {ch1}_stop:=false; end if;
  end if;
  if {ch2}_stop and {ch2}_rdy='0' then
    var2:=byte(to_integer({ch2}_dato)); {ch2}_ack<='0';
  else
    if {ch2}_stop and {ch2}_ack='0' then {ch2}_ack<='z'; {ch2}_stop:=false; end if;
  end if;
  wait on {ch1}_ack,{ch1}_rdy,{ch2}_ack,{ch2}_rdy,{son3}_extra_kill;
  if {son3}_extra_kill='0' then
    {ch1}_ack<='z';{ch2}_ack<='z'; wait for 0 ns; exit vss_live;
  end if;
end loop;
```

Ejemplo 4.12. Código generado para las sentencias *send* y *receive* de comunicación síncrona múltiple en modo *incast*

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

Sentencia VSS	Traducción VHDL

if **buffer_full**(ch3) then
var2:=**buffer_empty**(ch4);
reset_buffer ch3;
end if;
	if {ch3}_full then
	var2 := {ch4}_empty ;
	{ch3}_reset<='0';wait for 0 ns;
	(ch3)_reset<='z';wait for 0 ns;
	end if ;

Ejemplo 4.13. Código generado para la traducción de las sentencias *buffer_full*, *buffer_empty* y *reset_buffer*.

4.9. Traducción de las sentencias de manejo de excepciones

Los cuerpos de excepción de un proceso generan procesos hijo cuyos códigos son iguales al de un proceso hijo sin cuerpo de inicialización creado en modo simple con una sentencia *par_fork*. El código de estos procesos ya se ha comentado en el apartado de la jerarquía de procesos y aquí sólo se muestra un ejemplo (Ejemplo 4.14) en el que se observa un proceso de excepción que puede ser muerto y otro que no puede serlo. Lógicamente, los procesos de excepción sólo son muertos cuando su padre es muerto o inicializado.

El arranque de estos procesos sólo es posible en las sentencias *check* o *delay* de VSS, y puede ser realizado por su padre, o por otro cuerpo de excepción de menor prioridad.

Para implementar las excepciones, junto a los procesos anteriores y sus respectivas señales de control jerárquico, se necesitan las señales de activación y reconocimiento de las fuentes de excepción. La señal *{ex}_gl_st* permite que cualquier proceso active la excepción, mientras que la señal *{ex}_gl_ack* informa si algún proceso de excepción se está ejecutando. Para entender mejor cómo se relacionan estas señales con las de control jerárquico de procesos supóngase un sistema en el que existen dos procesos *A* y *B* sensibles a la excepción *EX1*, y supongamos que se produce la activación de dicha excepción. La evolución de las señales sería la que se ilustra en el cronograma de la Figura 4.6.

4.9.1. Sentencia *exception*

La traducción de esta sentencia depende del valor de la expresión que la acompaña. Cuando la expresión es la constante *true*, el proceso que activa la excepción espera hasta que todos los procesos de excepción arrancados acaben su tarea. Si por el contrario es la constante *false*, no espera a que los procesos de excepción arrancados acaben su tarea.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

Código VSS

```
entity except is end;
architecture behav of except is
   exception exa priority 1;
begin
padre:process
begin
   fork1:par_fork true
      simple son1:process
         exception_body exa is begin
            accion1; end;
      begin
         accion2;
      end process;
   || simple son3:process
         exception_body exa is begin
            accion3; end;
      begin
         accion3;
      end process;
   end fork;
   kill son1;
end process;
end;
```

Señales globales de excepción

```
signal {exa}_gl_st,{exa}_gl_ack:std_logic:='z';
```

Señales de control jerárquico de son3_exa

```
signal
{son3}_exa_start,{son3}_exa_start_ack:std_logicc:='z';
```

Proceso de excepción del proceso son3

```
son3_exa:process
begin
   if {son3}_exa_start='z' then
      wait until {son3}_exa_start='0';
   end if;
   {son3}_exa_start_ack<='0';
   wait until {son3}_exa_start='z';
      accion3 ;
   {son3}_exa_start_ack<='z'; wait for 0 ns;
end process;
```

Proceso de excepción del proceso son1

```
son1_exa:process
begin
   while {son1}_extra_kill='0' loop
      {son1}_exa_extra_kill_ack<='0';
      wait until {son1}_extra_kill='z';
      {son1}_exa_extra_kill_ack<='z';
      wait for 0 ns;
   end loop;
   if {son1}_exa_start='z' then
      wait until {son1}_exa_start='0' or{son1}_extra_kill='0';
   end if;
   if {son1}_extra_kill='z' then
      {son1}_exa_start_ack<='0';
      wait until {son1}_exa_start='z';
      vss_live:for i in 0 to 0 loop
         accion1 ;
      end loop;
      {son1}_exa_start_ack<='z';
      wait for 0 ns;
   end if;
end process;
```

Ejemplo 4.14. Código de los procesos correspondientes a los cuerpos de excepción.

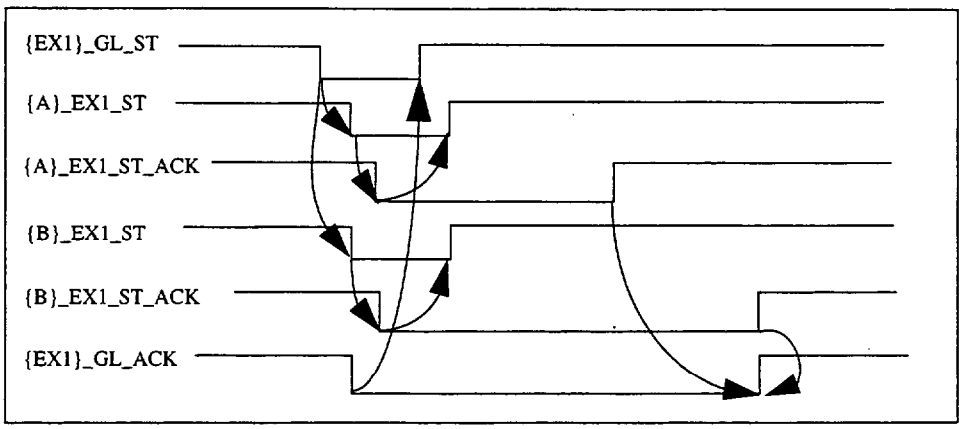


Figura 4.6. Cronograma de las señales de activacion y reconocimiento de una excepción.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

En cualquier otro caso, se traduce como una sentencia *if* en la que la ejecución de uno de los dos casos anteriores depende del valor de la expresión. En los tres casos anteriores se comprueba la señal *kill* para impedir el bloqueo del sistema cuando un proceso activa una fuente de excepción a la que sólo es sensible uno de sus hijos, y simultáneamente recibe un comando *kill*. Si no se realizara este test, el proceso hijo permanecería muerto hasta que muriese su padre, y el padre no podría morir hasta que el hijo activase su cuerpo de excepción. En el Ejemplo 4.15 se muestran las tres traducciones posibles.

4.9.2. Sentencia *check*

Esta es una de las dos sentencias que permiten la activación de los cuerpos de excepción. En el Ejemplo 4.16 se muestra la traducción de los tres tipos de sentencias *check*, así como la declaración de las variables implicadas en su traducción.

El enmascaramiento de las fuentes de excepción se implementa con ayuda de variables *booleanas*. Todos los procesos que utilicen las sentencias *check* o *delay* declaran tantas de estas variables como cuerpos de excepción hayan especificado.

Todas las sentencias *check* se traducen de modo que se comprueba, de mayor a menor prioridad, cada una de las fuentes de excepción. El proceso sensible a una excepción es el que activa el proceso de excepción (*{son1}_exa_start*) y el que maneja la señal global de reconocimiento (*{exa}_gl_ack*) a la espera de que el proceso de excepción finalice su tarea.

Las sentencias *check on* y *check on until* se traducen de manera que se comporten como una sentencia *wait* sensible a las sentencias de control jerárquico y a las excepciones. En ambos casos, estas sentencias se implementan mediante un *bucle* del que sólo se sale cuando el proceso es muerto o bien cuando se han atendido todas las excepciones y se cumple la condición de salida asociada a la sentencia *check*. En la sentencia *check on*, la condición de salida se cumple desde que se produzca un evento en alguna de las señales indicadas en la sentencia, mientras que en la sentencia *check on until* la condición de salida incluye también una *condición*. Las sentencias se traducen de modo que no se pierdan los eventos que llegan mientras se ejecuta el proceso de excepción. De este modo, la condición de salida se puede alcanzar aunque se estén ejecutando las excepciones.

Para cada fuente de excepción se genera una sentencia *if* en la que, tras comprobar que la excepción está activa y no enmascarada, se activa el proceso de excepción, se indica al sistema que la excepción está siendo atendida y se inicia un *bucle* en el que existe una única sentencia *wait* que espera por los eventos en las señales de control del

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

Código VSS

```
entity except is end;
architecture behav of except is
   exception exa priority 1;
begin
padre:process
begin
   fork1:par_fork true

      simple son1:process
         exception_body exa is
          begin accion1; end;
      begin
         accion2
      end process;

   || simple son2:process
      begin
         exception(exa) true;
         exception(exa) false;
         exception(exa) expresion;
      end process;

   end fork;
   kill son1,son2;
end process;
end;
```

Traducción del exception(exa) true

```
if {son2}_extra_kill='0' then
   exit vss_live;
end if;
{exa}_gl_st<='0';
wait until {exa}_gl_ack='0' or {son2}_extra_kill='0';
if {son2}_extra_kill='0' then
   {exa}_gl_st<='z';
   exit vss_live;
end if;
{exa}_gl_st<='z';
wait until {exa}_gl_ack='z';
if {son2}_extra_kill='0' then
   exit vss_live;
end if;
```

Traducción del exception(a) false

```
if {son2}_extra_kill='0' then
   exit vss_live;
end if;
{exa}_gl_st<='0';
wait until {exa}_gl_ack='0' or {son2}_extra_kill='0';
{exa}_gl_st<='z';
wait for 0 ns;
if {son2}_extra_kill='0' then
   exit vss_live;
end if;
```

Traducción del exception(a) expresion

```
if {son2}_extra_kill='0' then
   exit vss_live;
end if;
if expresion then
   {exa}_gl_st<='0';
   wait until {exa}_gl_ack='0' or {son2}_extra_kill='0';
   if {son2}_extra_kill='0' then
      {exa}_gl_st<='z';
      exit vss_live;
   end if;
   {exa}_gl_st<='z';
   wait until {exa}_gl_ack='z';
   if {son2}_extra_kill='0' then
      exit vss_live;
   end if;
else
   {exa}_gl_st<='0';
   wait until {exa}_gl_ack='0' or {son2}_extra_kill='0';
   {exa}_gl_st<='z';
   if {son2}_extra_kill='0' then
      exit vss_live;
   end if;
end if;
```

Ejemplo 4.15. Implementación VHDL de la sentencia *exception* en función del valor de la expresión booleana que la acompaña.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

Código VSS

```
entity except is end;
architecture behav of except is
 exception exa priority 1;
 exception exb priority 2;
begin
 padre:process
 begin
  fork1:par_fork true
   simple son1:process
   exception_body exa is begin
      accion1;
    end;
   exception_body exb is begin
    accion2;
   end;
   begin
    check;
    check on a,b;
    check on a,b until bool_expresion;
   end process;
  end fork;
  kill son1;
 end process;
end;
```

Traducción de la sentencia check

```
if {son1}_extra_kill='0' then exit vss_live; end if;
if {exb}_gl_st='0' and {exb} then
   {son1}_exb_start<='0'; {exb}_gl_ack<='0';
   wait until {son1}_exb_start_ack='0' or {son1}_extra_kill='0';
   if {son1}_extra_kill='0' then
      {son1}_exb_start<='z'; {exb}_gl_ack<='z';
      wait for 0 ns; exit vss_live;
   end if;
   {son1}_exb_start<='z'; wait until {son1}_exb_start_ack='z';
   {exb}_gl_ack<='z'; wait for 0 ns;
   if {son1}_extra_kill='0' then exit vss_live; end if;
end if;
if {exa}_gl_t='0' and {exa} then
   {son1}_exa_start<='0'; {exa}_gl_ack<='0';
   wait until {son1}_a_start_ack='0' or {son1}_extra_kill='0';
   if {son1}_extra_kill='0' then
      {son1}_exa_start<='z'; {exa}_gl_ack<='z';
      wait for 0 ns; exit vss_live;
    end if;
   {son1}_exa_start<='z'; wait until {son1}_exa_start_ack='z';
   {exa}_gl_ack<='z'; wait for 0 ns;
   if {son1}_extra_kill='0' then exit vss_live; end if;
 end if;
```

Traducción de las sentencia check on a,b; y check on a,b until bool_expresion;

```
if {son1}_extra_kill='0' then exit vss_live; end if;
{ex}_bucle:=true;
while {ex}_bucle loop
  if not (({exa}_gl_st='0' and {exa}) or ({exb}_gl_st='0' and {exb})) then
     wait on {exa}_gl_st,{exb}_gl_st,a , b,{son1}_extra_kill;
    if {son1}_extra_kill='0' then exit vss_live; end if;
    if a 'event or  b 'event then {ex}_bucle:=false ;end if;
  end if;
  if {exb}_gl_st='0' and {exb} then
     {son1}_exb_start<='0' ; {exb}_gl_ack<='0';
     while true loop
        wait on {son1}_exb_start_ack,a , b,{son1}_extra_kill;
        if {son1}_extra_kill='0' then {son1}_exb_start<='z'; {exb}_gl_ack<='z'; wait for 0 ns ;exit vss_live; end if;
        if a 'event or  b 'event then {ex}_bucle:=false; end if;
           --en el caso de la sentencia check on a,b until bool_expresion la sentencia anterior se sustituiría por la siguiente
           -- if a 'event or  b 'event then if bool_expresion then {ex}_bucle:=false;end if; end if;
        if {son1}_exb_start='0' then
           if {son1}_exb_start_ack='0' then {son1}_exb_start<='z';  end if;
        else
           if {son1}_exb_start_ack='z' then {exb}_gl_ack<='z'; exit; end if;
        end if;
     end loop;
  end if;
  if {exa}_gl_st='0' and {exa} then
     {son1}_a_start<='0' ; {exa}_gl_ack<='0';
     while true loop
        wait on {son1}_a_start_ack,a , b,{son1}_extra_kill;
        if {son1}_extra_kill='0' then {son1}_a_start<='z'; {exa}_gl_ack<='z'; wait for 0 ns; exit vss_live; end if;
        if a 'event or  b 'event then {ex}_bucle:=false;end if;
           --en el caso de la sentencia check on a,b until bool_expresion la sentencia anterior se sustituiría por la siguiente
           --if a 'event or  b 'event then if bool_expresion then {ex}_bucle:=false;end if; end if;
        if {son1}_a_start='0' then
           if {son1}_a_start_ack='0' then {son1}_a_start<='z';  end if;
        else
           if {son1}_a_start_ack='z' then {exa}_gl_ack<='z'; exit; end if;
        end if;
     end loop;
  end if;
end loop;
```

Ejemplo 4.16. Implementación VHDL de las tres variantes de la sentencia *check*.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

proceso de excepción, por los eventos de las señales especificadas en la sentencia *check* y por la señal *kill*. El conjunto de sentencias *if* que se generan a continuación se encargan de pasar al proceso al estado muerto cuando recibe un comando *kill*, de comprobar y actualizar la condición de salida de la sentencia *check*, y de completar el protocolo de ejecución del proceso de excepción. La existencia de una sola sentencia *wait* es lo que impide que se pierdan eventos.

4.9.3. Sentencias *vssenable* y *vssdisable*

Estas sentencias se traducen de modo que modifican las variables *booleanas* con las que se enmascararan las fuentes de excepción (Ejemplo 4.17).

Traducción de la sentencia vssdisable a,b	**Traducción de la setencia vssenable a,b**
{exa}:=false;{exb}:=false;	{exa}:=true;{exb}:=true;

Ejemplo 4.17. Traducción de las sentencias *vssenable* y *vssdisable*.

4.10. Traducción de las sentencias de manejo de variables compartidas

Para cada bloque de variables compartidas se construye el subsistema que se describe en la Figura 4.7, cuyos componentes principales son una memoria y un árbitro. En la memoria se almacenan todas las variables compartidas, y el árbitro asegura que los procesos acceden en exclusión mutua a las variables.

Cuando un proceso desea acceder a la memoria, activa su señal de petición (*lock*) y espera a que el árbitro le ceda el control del recurso activando la señal de reconocimiento (*locked*). El proceso mantiene su señal *lock* activa hasta que finaliza con su región crítica, y luego desactiva la señal *lock*. Entonces, el árbitro responde con la desactivación de la señal *locked* y continúa con la comprobación del resto de señales de petición de los procesos. El árbitro no atiende las peticiones de los procesos por orden de llegada sino que utiliza una política circular determinista. Este sistema garantiza la ausencia de *deadlock* y *starvation*.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

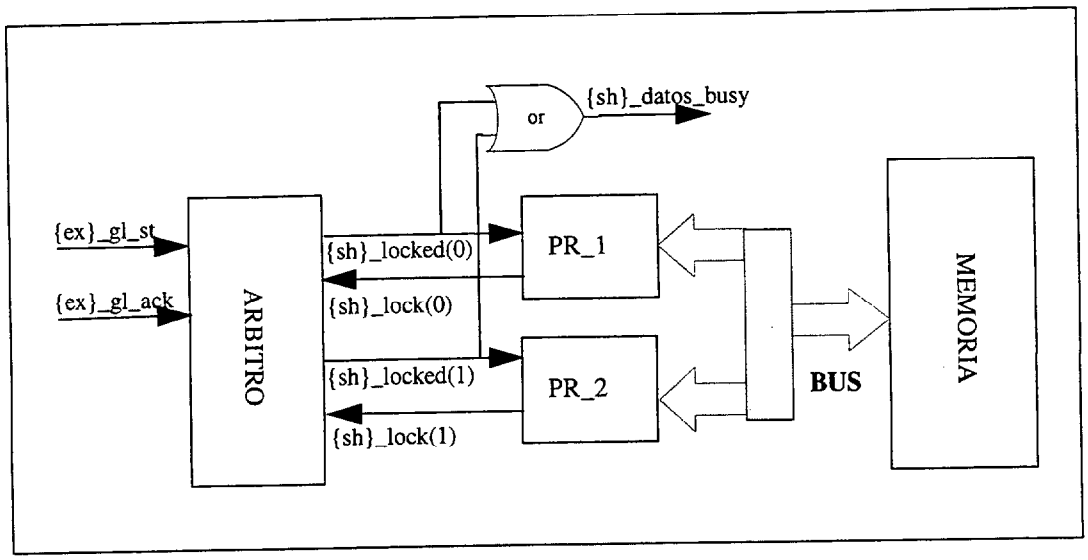


Figura 4.7. Diagrama de bloques para un sistema formado por dos procesos y una tarea que acceden a un bloque de variables compartidas.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

4.10.1.Sentencia *lock*

La traducción de esta sentencia depende del valor de la expresión booleana que la acompaña. En el Ejemplo 4.18 se muestra cada una de las tres traducciones posibles. Cuando la expresión es la constante *true*, el proceso entra en un *bucle* del que sólo sale cuando se le concede el recurso (*locked*='0') o cuando el proceso muere. En el primer caso, el proceso ejecuta la región crítica y luego libera el recurso desactivando la señal de petición (*lock*<='Z'). En el segundo caso, el proceso no finaliza la ejecución de la región crítica, desactivando la señal *lock* antes de morir.

La traducción es similar cuando la expresión es la constante *false*, pero en este caso la condición de salida del *bucle* está controlada por la señal *{sh}_datos_busy*. Esta señal se activa desde que el árbitro cede el control a cualquier recurso. En el interior del *bucle* la variable *vss_sh_stop*, inicializada a *false*, conmuta su valor sólo si el árbitro le concede el recurso. Después del *bucle* de espera se encuentra la región crítica encerrada en una sentencia *if* que está controlada por el valor de dicha variable. De este modo, un proceso no ejecuta la región crítica si el recurso se le concede a otro proceso.

Para cualquier otro valor de la expresión *booleana* la sentencia *lock* se traduce como la suma lógica de las dos traducciones anteriores mediante una sentencia *if* controlada por la expresión.

lock({SH}) true begin region_critica1; end lock;

```
{sh}_lock({sh}_son2_id_nat)<='0';
wait for 0 ns;
while {sh}_locked ({sh}_son2_id_nat)='z' loop
    wait on {sh}_locked({sh}_son2_id_nat),{pr}_extra_kill;
    if {pr}_extra_kill='0' then
       {sh}_lock({sh}_son2_id_nat)<='z';
       exit vss_live;
    end if;
end loop;
region_critica1 ;
{sh}_lock({sh}_son2_id_nat)<='z';
wait until {sh}_locked({sh}_son2_id_nat) ='z';
```

lock({SH}) false begin region_critica2; end lock;

```
vss_sh_stop:= false;
while not {sh}_datos_busy loop
    {sh}_lock({sh}_son2_id_nat)<='0';
    wait on {sh}_datos_busy,{sh}_locked({sh}_son2_id_nat),
       {son2}_extra_kill;
    if {pr}_extra_kill='0' then
       {sh}_lock({sh}_son2_id_nat)<='z';
       exit vss_live;
    end if;
    if {sh}_locked({sh}_son2_id_nat)='0' then
       vss_sh_stop:=true;
    end if;
end loop;
if vss_sh_stop then
    region_critica2 ;
end if;
{sh}_lock({sh}_son2_id_nat)<='z';
wait until {sh}_locked({sh}_son2_id_nat) ='z';
```

lock({SH}) expres begin region_critica3; end lock;

```
vss_sh_stop:= expres;
if not vss_sh_stop then

    while not {sh}_datos_busy loop
       {sh}_lock({sh}_son2_id_nat)<='0';
       wait on {sh}_datos_busy, {sh}1_locked({sh}_son2_id_nat),
          {pr}_extra_kill;
       if {pr}_extra_kill='0' then
          {sh}_lock({sh}_son2_id_nat)<='z';
          exit vss_live;
       end if;
       if {sh}_locked({sh}_son2_id_nat)='0' then
          vss_sh_stop:=true;
       end if;
    end loop;

else

    {sh}_lock({sh}_son2_id_nat)<='0';
    wait for 0 ns;

    while {sh}_locked ({sh}_son2_id_nat)='z' loop
       wait on {sh}_locked({sh}_son2_id_nat), {pr}_extra_kill;
       if {pr}_extra_kill='0' then
          {sh}_lock({sh}_son2_id_nat)<='z';
          exit vss_live;
       end if;
    end loop;

end if;

if vss_sh_stop then
    region_critica3 ;
end if;
{sh}_lock({sh}_son2_id_nat)<='z';
wait until {sh}_locked({sh}_son2_id_nat) ='z';
```

Ejemplo 4.18. Código generado para la sentencia *lock* en función de la expresión lógica que la acompaña.

4.10.2.Sentencias *get* y *put*

Para la lectura y escritura de las variables compartidas se generan procedimientos globales. El compilador genera tres tipos de procedimientos: a) *tipo base*, para la lectura y escritura de tipos enteros, rango de enteros o enumerados. b) *Tipo array*, para la lectura y escritura de *arrays*. c) *Tipo compuesto*, para la lectura y escritura de registros. Los Ejemplos 4.19, 4.20 y 4.21 ilustran la forma general de los procedimientos generados para cada uno de los tipos anteriores.

En los tres casos, los procedimientos de lectura se encabezan con el nombre de la variable en la que se desea dejar los *datos*. Luego continúan con la dirección base de la variable (*direccion*), los buses de dirección y datos, y las señales de control *cs* y *rd_wr*.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

```
type int is range -10000 to 10000;

procedure read_int(dato:out int;direccion:in natural;
      signal bus_direccion:out unsigned;
      signal bus_datos:in signed; signal cs,rd_wr:out std_logic) is
         constant dir_z:unsigned(bus_direccion'range):=(others=>'z');
begin
      bus_direccion<=to_unsigned(direccion,bus_direccion'length);
      cs<='0'; rd_wr<='1'; wait for 0 ns;wait for 0 ns;
       dato:=int(to_integer(bus_datos));
      bus_direccion<=dir_z; rd_wr<='z'; cs<='z'; wait for 0 ns;
end;

procedure write_int(expression:in int;direccion:in natural;
      signal bus_direccion:out unsigned;signal bus_datos:out signed;
      signal cs,rd_wr:out std_logic) is
         constant dat_z:signed(bus_datos'range):=(others=>'z');
         constant dir_z:unsigned(bus_direccion'range):=(others=>'z');
begin
      bus_direccion<=to_unsigned(direccion,bus_direccion'length);
      cs<='0'; rd_wr<='0'; bus_datos<=to_signed(integer(expression),bus_datos'length);
      wait for 0 ns; wait for 0 ns;
      bus_direccion<=dir_z; bus_datos<=dat_z; rd_wr<='z'; cs<='z'; wait for 0 ns;
end;
```

Ejemplo 4.19. Procedimientos de lectura y escritura declarados globalmente para el acceso a una variable compartida de tipo simple.

```
type record1 is record
    a:int; b:elarray1 ( 0 to 2 , 0 to 2 ) ;
end record;

procedure read_record1(dato:out record1;direccion:in natural;
      signal bus_direccion:out unsigned;signal bus_datos:in signed;
      signal cs,rd_wr:out std_logic) is
begin
      read_int(dato.a,direccion+0,bus_direccion,bus_datos,cs,rd_wr);
      read_elarray1(dato.b,direccion+1,bus_direccion,bus_datos,cs,rd_wr);
end;

procedure write_record1(variable dato:in record1;direccion:in natural;
      signal bus_direccion:out unsigned;signal bus_datos:out signed;
      signal cs,rd_wr:out std_logic) is
begin
      write_int(dato.a,direccion+0,bus_direccion,bus_datos,cs,rd_wr);
      write_elarray1(dato.b,direccion+1,bus_direccion,bus_datos,cs,rd_wr);
end;
```

Ejemplo 4.20. Procedimientos de lectura y escritura declarados globalmente para el acceso a una variable compartida de tipo compuesto.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

```
type elarray1 is array (int range <> , int range <> ) of int;

procedure read_elarray1(dato:out elarray1;direccion:in natural;
      signal bus_direccion:out unsigned; signal bus_datos:in signed;
      signal cs,rd_wr:out std_logic) is
         variable size2,size1,cont2,cont1,offset:natural:=0;
begin
      size1:=1;
      size2:=size1*dato'length(1);
      cont2:=0;
      for i2 in dato'left(2) to dato'right(2) loop
         cont1:=0;
         for i1 in dato'left(1) to dato'right(1) loop
            offset:=cont1*size1+cont2*size2;
            read_int(dato(i1,i2),direccion+offset,bus_direccion,bus_datos,cs,rd_wr);
            cont1:=cont1+1;
         end loop;
         cont2:=cont2+1;
      end loop;
   end;

procedure write_elarray1(variable dato:in elarray1;direccion:in natural;
      signal bus_direccion:out unsigned;signal bus_datos:out signed;
      signal cs,rd_wr:out std_logic) is
   variable size2,size1,cont2,cont1,offset:natural:=0;
begin
      size1:=1;
      size2:=size1*dato'length(1);
      cont2:=0;
      for i2 in dato'left(2) to dato'right(2) loop
         cont1:=0;
         for i1 in dato'left(1) to dato'right(1) loop
            offset:=cont1*size1+cont2*size2;
            write_int(dato(i1,i2),direccion+offset,bus_direccion,bus_datos,cs,rd_wr);
            cont1:=cont1+1;
         end loop;
         cont2:=cont2+1;
      end loop;
end;
```

Ejemplo 4.21. Procedimientos de lectura y escritura declarados globalmente para el acceso a una variable compartida de tipo *array*.

En el caso de la lectura de un *tipo base* (Ejemplo 4.19), el procedimiento programa el bus de direcciones y las señales de control *cs* y *rd_wr* para hacer una operación de lectura sobre la memoria. Luego espera a que la memoria retorne el dato, que es cargado en la variable *dato*. Por último, el procedimiento deja en alta impedancia los buses de direcciones y control de la memoria. Las funciones *to_signed*, *to_unsigned* y *to_integer* son las mismas que se proponen en el *standard 1076.3 numeric_std* para la conversión de tipos de datos. Análogamente se puede explicar el procedimiento de escritura.

Los procedimientos de lectura de un tipo *array* (Ejemplo 4.20) leen consecutivamente cada uno de los elementos que lo componen.

Por último, los procedimientos de lectura de un tipo compuesto (Ejemplo 4.21) leen cada uno de los campos que lo componen mediante llamadas a los procedimientos de lectura del tipo correspondiente.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

Una vez generados estos procedimientos, la traducción de las sentencias *get* y *put* es prácticamente inmediata. Como ejemplo considérese la traducción de las sentencias *get* y *put* del código VSS del Ejemplo 4.22.

La traducción no comprueba que la dirección base del dato al que se accede está dentro de las posibles. El diseñador debe cuidar que el acceso a los elementos de un *array* no sobrepase el rango de valores previstos en la declaración del mismo.

4.11. Traducción de las sentencias de temporización

Las señales de reloj declaradas en una especificación VSS se implementan con ayuda de una señal de igual nombre que el reloj, y un componente (*generador_reloj*) que hace oscilar la señal anterior a la frecuencia deseada. En el Ejemplo 4.23 se muestran la declaración de la señal de reloj y el generador de reloj.

Código VSS

```
entity test is end;
architecture pruebashared of test is
   type int is range -10000 to 10000;
   type elarray1 is
      array(int range<>, int range<>) of int;
   type record1 is record
      a:int; b:elarray1(3 to 5, 7 downto 2);
   end record;

   shared mem1 is
      variable var1:record1;
      variable var2:
         elarray1(3 to 5, 7 downto 2);
   end shared;

begin
   son2:process
      variable var1:record1;
      variable var2:.
         elarray1(3 to 5, 7 downto 2);
   begin
      lock(mem1) true begin
         get(var1 =>var1);
         get(var1.b=> var1.b);
         get(var1.b(i,j+k)=>var1.b(3,4));
         put(var2<=expresion);
         put(var2(i+j,k)<=expresion);
      end lock;
   end process;
end;
```

get(var1 =>var1)

```
vss_sh_dir:=0;
read_record1(var1, vss_sh_dir,
   {sh}_direccion,{sh}_datoinout,
   vss_sh_mem1_cs,{sh}_rd_wr);
```

get(var1.b=> var1.b)

```
vss_sh_dir:=0+1;
read_elarray1( var1 . b,vss_sh_dir,
   {sh}_direccion,{sh}_datoinout,
   {sh}_cs,vss_sh_mem1_rd_wr);
```

get(var1.b(i,j+k)=>var1.b(3,4))

```
vss_sh_dir:=0+( i-3)*1+( j + k-2)*3+1;
read_int( var1 .b( 3 , 4 ),vss_sh_dir,
   {sh}_direccion,{sh}_datoinout,
   {sh}1_cs,{sh}_rd_wr);
```

put(var2<=expresion)

```
vss_sh_dir:=19;
write_elarray1( expresion,vss_sh_dir,
   {sh}_direccion,{sh}_datoinout,
   {sh}_cs,{sh}_rd_wr);
```

put(var2(i+j,k)<=expresion)

```
vss_sh_dir:=19+(i+j-3)*1+(k-2)*3+0;
write_int( expresion,vss_sh_dir,
   {sh}_direccion,{sh}_datoinout,
   {sh}_cs,{sh}_rd_wr);
```

Ejemplo 4.22. Código generado para las sentencias *get* y *put*.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

clocks(clk1, 3 ns)

```
signal clk1:bit:='0';

component generador_reloj
  generic (semiperiodo:real);
  port(clock:inout bit);
end component;
```

Ejemplo 4.23. Código utilizado en la declaración del generador de reloj que se encarga de hacer oscilar la señal de reloj a la frecuencia deseada.

4.11.1.Sentencia *delay*

En el caso de la sentencia *delay*, el código generado realiza un proceso de espera hasta que se contabiliza el número de eventos deseados. Durante la espera, el proceso atiende las excepciones a las que es sensible, así como los comandos *kill* que recibe. Para la implementación de esta sentencia se necesita una variable auxiliar (*{clk}_cuenta*) que cuenta los eventos que se producen en la señal de reloj.

En el Ejemplo 4.24 se muestra la traducción de una sentencia *delay* y un supuesto código equivalente descrito en VSS. La traducción a VHDL del código VSS equivalente no contaría los eventos que se produjeran mientras se estuviera atendiendo una excepción. Por ello, la sentencia *delay* forma parte del lenguaje. La traducción se realiza como una sentencia *check* en la que la condición de salida se alcanza cuando la variable *cuenta* es cero, decrementándose el valor de esta variable cada vez que se produce un flanco de bajada en la señal de reloj. Obsérvese que cuando el proceso no es sensible a ninguna excepción, el código VSS propuesto sí implementaría la sentencia *delay*.

4.11.2.Sentencias *timeout* y *reset_timer*

Para cada temporizador declarado se instancia un *timer* y se declaran una señal de reset (*{temp}_reset*), una señal de control (*{temp}_data*) y una señal de datos (*{temp}_cuenta*) para programar el temporizador. Como cada temporizador es también una fuente de excepción, declara las mismas señales que cualquier otra fuente de excepción. Junto a estas señales aparece una constante (*{temp}_cuenta_z*) que se utiliza para dejar en alta impedancia el canal de datos del temporizador. En el Ejemplo 4.25 se muestran los dos modelos de traducción de las sentencias *timeout* y *reset_timer*, las señales y constantes declaradas, y el interfaz del temporizador.

El temporizador es un contador módulo N que se programa a un valor de cuenta, y una vez que se alcanza dicho valor se genera una excepción. El parámetro *modulo* define el módulo del contador. Las entradas son: la señal de reloj (*clk*), un reset asíncrono (*reset*), las señales de control (*data*) y datos (*cuenta*) para la programación del

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

Código VSS

```
entity timing is end;
architecture behav of timing is
   clocks(clk1, 3 ns);
   timer temp1:clk1
      max_cont 10 priority 1;
begin
padre:process
begin
   fork1:par_fork true
      simple son1:process
      begin
         delay(clk1, expression ns);
      end process;
   end fork;
   kill son1;
end process;
end;
```

Posible código VSS equivalente a delay(clk1, expression ns);

```
vss_cuenta:=eventos a contar;
while vss_cuenta>0 loop
      check on {clk1} until {clk1}='0';
      vss_cuenta:=vss_cuenta-1;
 end loop;
```

Variables declaradas a *son1*

```
variable vss_ti_clk1_cuenta:natural;
variable vss_ex_bucle:boolean;
variable {ex_temp1}:boolean:=true;
```

Traducción de la sentencia delay(clk1, expression ns)

```
vss_ti_clk1_cuenta:=natural(real( expression )* (1.000000e-09 / 3.000000e-09));
while vss_ti_clk1_cuenta>0 loop
    if {son1}_extra_kill='0' then exit vss_live; end if;
    vss_ex_bucle:=true;
    while vss_ex_bucle loop
       if not (({ex_temp1}_gl_st='0' and {ex_temp1})) then
          wait on {ex_temp1}_gl_st,clk1,{son1}_extra_kill;
          if {son1}_extra_kill='0' then exit vss_live; end if;
          if clk1'event and  clk1='0' and
             vss_ti_clk1_cuenta>0 then
               vss_ti_clk1_cuenta:=vss_ti_clk1_cuenta-1;
               if vss_ti_clk1_cuenta=0 then
                  vss_ex_bucle:=false;end if;
          end if;
       end if;
       if vss_ex_temp1_gl_st='0' and vss_ex_temp1 then
          {son1}_temp1_start<='0' ; vss_ex_temp1_gl_ack<='0';
          while true loop
             wait on {son1}_temp1_start_ack,clk1,{son1}_extra_kill;
             if {son1}_extra_kill='0' then
                {son1}_temp1start<='z';
                vss_ex_temp1_gl_ack<='z';
                wait for 0 ns; exit vss_live;
             end if;
             if clk1'event and  clk1='0' and
                vss_ti_clk1_cuenta>0 then
                   vss_ti_clk1_cuenta:=vss_ti_clk1_cuenta-1;
                   if vss_ti_clk1_cuenta=0 then
                      vss_ex_bucle:=false; end if;
             end if;
             if {son1}_temp1_start='0' then
                if {son1}_temp1_start_ack='0' then
                   {son1}_temp1_start<='z'; end if;
             else
                if {son1}_temp1_start_ack='z' then
                   vss_ex_temp1_gl_ack<='z'; exit; end if;
             end if;
          end loop;
       end if;
    end loop;
end loop;
```

Ejemplo 4.24. Implementación VHDL de la sentencia *delay*.

timer, y la señal de reconocimiento de excepción (*ack_int*). La única salida es la señal de activación de la excepción (*rdy_int*).

La sentencia *timeout* se traduce en la programación del temporizador al valor de cuenta deseado, mientras que la sentencia *reset_timer* provoca la inicialización asíncrona del temporizador, actuando sobre su señal de *reset*.

Un temporizador que puede ser programado por más de un proceso necesita un árbitro similar al comentado en el apartado de variables compartidas. En este caso, las sentencias *timeout* y *reset_timer* se traducen como las anteriores, pero van precedidas de la petición del recurso. Una vez finalizan la operación liberan el recurso.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

Código VSS

```
entity timing is end;
architecture behav of timing is
   clocks(clk1, 3 ns);
   timer temp1:clk1 max_cont 10
      priority 1;
   timer temp2:clk1 max_cont 5
      priority 2;
begin
padre:process
begin
   fork1:par_fork true

      simple son1:process
        exception_body temp1 is
          begin accion1; end;
      begin
        timeout(temp1, expression ns);
        reset_timer(temp1);
        timeout(temp2, expression ns);
        reset_timer(temp2);
      end process;

   end fork;
   kill son1;
   timeout(temp2, expression ns);
end process;
end;
```

Señales y constantes relacionadas con el temporizador temp2

```
signal {temp2}_reset,{temp2}_data:std_logic:='z';
signal {temp2}_cuenta:unsigned(2 downto 0):=(others=>'z');
constant {temp2}_cuenta_z:unsigned(2 downto 0):=(others=>'z');
constant {temp2}_id_z:unsigned(3 downto 0):=(others=>'z');
constant {temp2}_son1_id_nat:natural:=0;
constant {temp2}_son1_id_nat:natural:=1;
constant {temp2}_padre_id_nat:natural:=2;
signal {temp2}_lock,{temp2}_locked:std_logic_vector(0 to 2):=(others=>'z');
```

Traducción del timeout(temp1, expression ns);

```
{temp1}_reset<='0'; wait for 0 ns;
{temp1}_reset<='z'; wait for 0 ns;
{temp1}_data<='0';
{temp1}_cuenta<= to_unsigned
( natural(real( expression )* 1.000000e-09 / 3.000000e-09 ),4);
wait for 0 ns;
{temp1}_data<='z';
{temp1}_cuenta<={temp1}_cuenta_z; wait for 0 ns;
if {son1}_extra_kill='0' then exit vss_live; end if;
```

Traducción del reset_timer(temp1);

```
{temp1}_reset<='0';wait for 0 ns;
{temp1}_reset<='z';
wait for 0 ns;
if {son1}_extra_kill='0' then exit vss_live; end if;
```

Traducción del timeout(temp2, expression ns);

```
{temp2}_lock({temp2}_son1_id_nat)<='0';
wait until {temp2}_locked({temp2}_son1_id_nat)='0';
{temp2}_reset<='0'; wait for 0 ns;
{temp2}_reset<='z'; wait for 0 ns;
{temp2}_data<='0';
{temp2}_cuenta<=to_unsigned ( natural(real( expression )* 1.000000e-09 / 3.000000e-09 ),3);
wait for 0 ns;
{temp2}_data<='z';
{temp2}_cuenta<={temp2}_cuenta_z;
wait for 0 ns;
{temp2}_lock({temp2}_son1_id_nat)<='z';
wait until {temp2}_locked({temp2}_son1_id_nat)='z';
if {son1}_extra_kill='0' then exit vss_live; end if;
```

Traducción del reset_timer(temp2);

```
{temp2}_lock({temp2}_son1_id_nat)<='0';
wait until {temp2}_locked({temp2}_son1_id_nat)='0';
{temp2}_reset<='0';
wait for 0 ns;
{temp2}_reset<='z';
wait for 0 ns;
{temp2}_lock({temp2}_son1_id_nat)<='z';
wait until{temp2}_locked({temp2}_son1_id_nat)='z';
if {son1}_extra_kill='0' then exit vss_live; end if;
```

Ejemplo 4.25. Código generado para las sentencias *timeout* y *reset_timer* para los casos en que uno (temp1) o múltiples (temp2) procesos acceden al mismo temporizador.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

Como punto final del proceso de traducción de todas las sentencias de VSS a VHDL cabe hacer las siguientes puntualizaciones:

- La corrección del código generado ha sido comprobada mediante la simulación de multitud de ejemplos. En primer lugar, se utilizaron ejemplos sencillos en los que se comprobaron las sentencias de forma individualizada. Posteriormente, se utilizaron ejemplos más complejos en los que aparecen sentencias de varios tipos. Algunos de estos ejemplos pueden consultarse en [Cas98d].
- El ratio de líneas de código VHDL generadas respecto del código VSS es variable, ya que depende del número y tipo de sentencias del código VSS. Los ejemplos que se utilizaron para el proceso de comprobación tienen una relación que va desde 2/1 hasta 4/1.

4.12. Representación intermedia

En este apartado se presenta la representación intermedia que hemos utilizado en nuestro entorno de codiseño. Una de las representaciones más utilizadas es el grafo de flujo de datos utilizado en el proyecto europeo ASCIS (*Architectural Synthesis of Complex Integrated Systems*) [Eij91][Eij92]. Este grafo fue diseñado como estándar para el intercambio de información entre las herramientas de síntesis y verificación.

El grafo no soporta la semántica de simulación de VHDL, es decir, no soporta la sentencia *wait* ni la sentencia de asignación a señales de VHDL [Sch93]. En este trabajo hemos ampliado el grafo ASCIS para solucionar ambos problemas. Para ello hemos añadido nuevos nodos al grafo. Además, hemos desarrollado un compilador para generar el grafo desde especificaciones VHDL.

La estructura del capítulo es la siguiente: en el apartado 1, se presenta la semántica del grafo ASCIS. En el apartado 2, se presentan los nodos que hemos añadido al grafo. Finalmente, en el apartado 3 se muestra un ejemplo.

4.12.1.Semántica del grafo ASCIS

El grafo ASCIS está formado por nodos y flancos dirigidos. Los nodos representan operaciones y los flancos representan la transferencia de valores. Los valores se representan mediante *tokens*.

La *ejecución* de un nodo se define como el proceso mediante el cual los *tokens* de entrada son consumidos y los *tokens* con los resultados son generados en los flancos de

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

salida. Un nodo puede ejecutarse siempre que tenga un *token* en cada uno de sus flancos de entrada.

El grafo ASCIS tiene 4 clases de flancos: *a) flancos de datos*, que se utilizan para la transferencia de datos, *b) flancos de tipo source*, que se utilizan en algunos nodos para generar un *token* en su salida, *c) flancos de control*, que se utilizan en los nodos que controlan el flujo de control, y *d) flancos de tipo chain*, que se utilizan para establecer un orden de ejecución.

Los nodos principales del grafo ASCIS son los siguientes (Figura 4.8):

- *Nodos de operación*: realizan algún tipo de procesamiento sobre los datos. Los nodos de las funciones (*funct*) y procedimientos (*proc*) son jerárquicos.
- *Nodos constantes*: generan un *token* con un valor constante. Un flanco de entrada de tipo *source* indica cuando debe generarse el mismo.
- *Nodos de entrada/salida*: se utilizan en las interfaces de los subprogramas y procesos.
- *Nodo get*: se utiliza para la lectura de las señales y se encadenan a través de flancos de tipo *chain*.

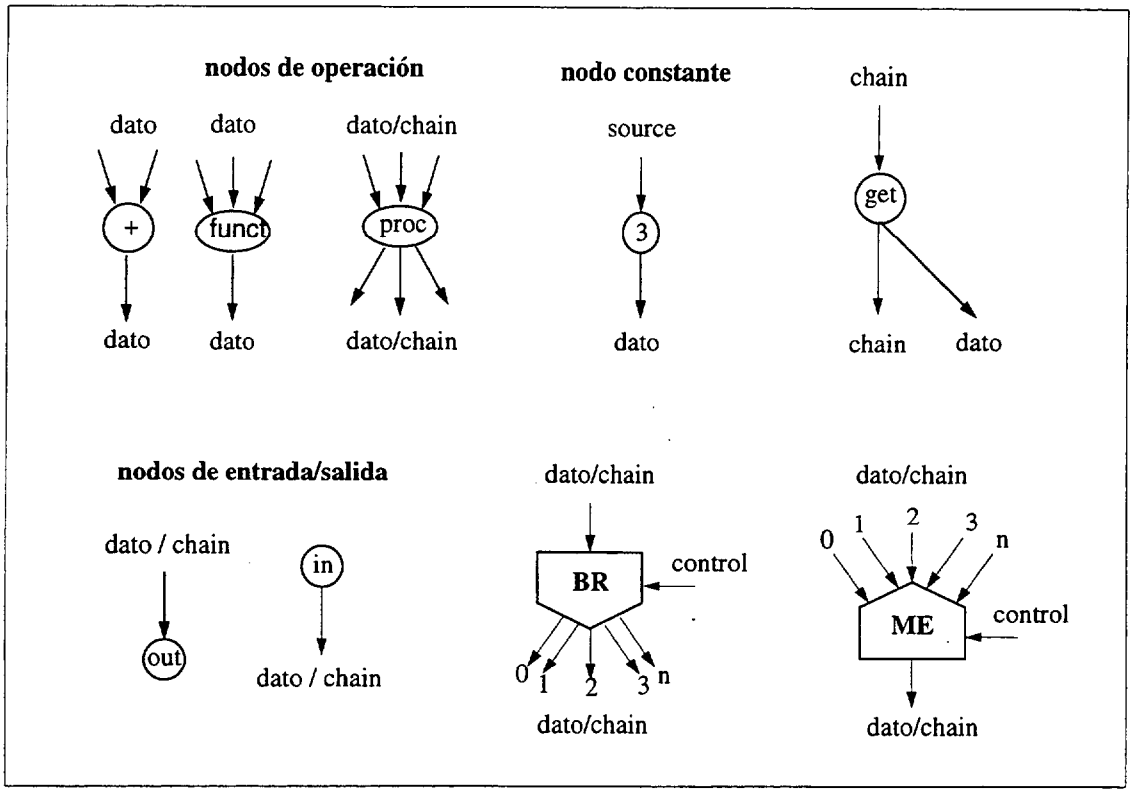


Figura 4.8. Nodos del grafo ASCIS.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

- *Nodos branch y merge (BR y ME)*: un nodo *branch* mueve un *token* desde el flanco de entrada hasta uno de los flancos de salida, que se selecciona a través del valor del *token* de la entrada de control. El nodo *merge* es dual al nodo *branch*. Estos nodos se utilizan para sentencias condicionales.
- *Nodos entry y exit (EN y EX)*: estos nodos son funcionalmente análogos a los nodos *merge* y *branch*, respectivamente. La única diferencia es que el nodo *entry* tiene un comportamiento especial frente al resto de nodos la primera vez que se ejecuta, ya que el *token* de la entrada pasa a la salida aunque el *token* de control no esté disponible. Estos nodos se utilizan para sentencias repetitivas.

4.12.2.Grafo ASCIS extendido

En este apartado se presentan los nodos que hemos añadido al grafo ASCIS para soportar la sentencia *wait* y la sentencia de asignación a señales. Los nuevos nodos son los siguientes (Figura 4.9.a):

- *Nodo put*: se utiliza para escribir en las señales y se encadena a través de flancos de tipo *chain*. La escritura en una señal tiene efecto cuando su *token* alcanza un nodo de actualización (*upd*).

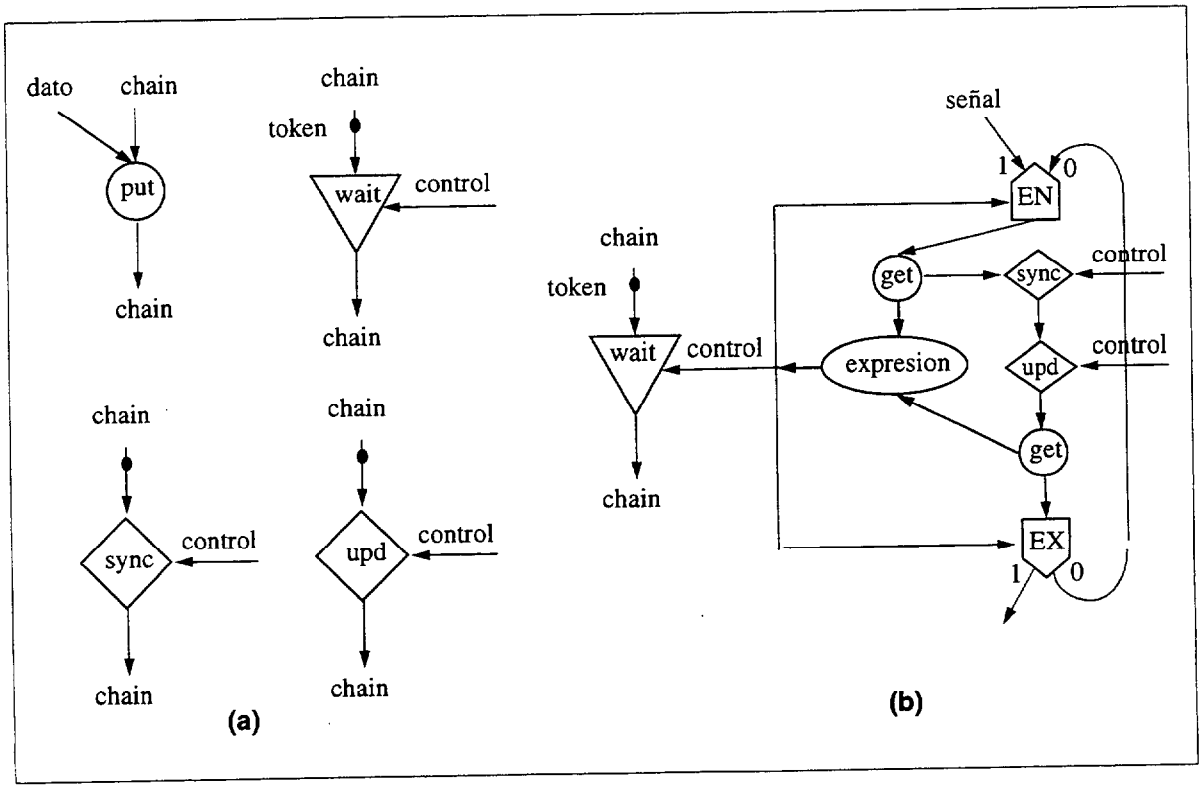


Figura 4.9. (a) Nodos que hemos añadido al grafo ASCIS. (b) Traducción de una sentencia *wait until expresion*.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004-

- *Nodo wait*: un *token* permanece esperando a la entrada del nodo hasta que el *token* del flanco de control sea *true*. Si el valor del *token* de control es *false*, el mismo es simplemente consumido por el nodo. Este nodo se utiliza para la sentencia *wait* (Figura 4.9.b).
- *Nodo sync*: un *token* permanece esperando a la entrada del nodo hasta que haya un *token* en el flanco de control. El *token* de control aparece cuando todos los procesos están parados en los nodos *wait*. Este nodo se utiliza para representar el ciclo de simulación de VHDL.
- *Nodo upd*: un *token* permanece esperando a la entrada del nodo hasta que haya un *token* en el flanco de control. El *token* de control aparece cuando se han realizado las actualizaciones de las señales en todos los procesos. Este nodo se utiliza para que los procesos no continúen su ejecución sin haberse actualizado todas las señales.

4.12.3.Ejemplo

En este trabajo hemos desarrollado un compilador para traducir desde VHDL al grafo ASCIS extendido. El compilador utiliza traducción dirigida por la sintaxis [Aho86] y ha sido desarrollado utilizando las herramientas Lex [Lev92] y Yacc [Don91].

En la Figura 4.10 se muestran dos procesos VHDL que se comunican, y en las Figuras 4.11 y 4.12 se muestra el grafo ASCIS generado por el compilador. Para una información detallada del compilador puede consultarse [Cas97a][Cas98e].

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

```
architecture BEHAVIOUR of MULT is
signal c: integer;
signal vss_c_request: boolean := false;
signal vss_c_ack: boolean := false;
begin

    P1: process
        variable mult: integer;
        variable count: integer;
    begin
        mult:=0;
        count:=X;
        while (count>0) loop
            mult:=mult+Y;
            count:=count-1;
        end loop;

        vss_c_request<=true;
        wait until (vss_c_ack);
        c<=mult;
        vss_c_request<=false;
        wait until (not vss_c_ack);
    end process;

P2: process
        variable mult,count,data: integer;
begin

        wait until (vss_c_request);
        vss_c_ack<=true;
        wait until (not vss_c_request);
        data:=c;
        vss_c_ack<=false;
        wait for 0ns;

        count:=data;
        mult:=0;

        while (count>0) loop
            mult:=mult+data;
            count:=count-1;
        end loop;

        O<=mult;
end process;

end BEHAVIOUR;
```

Figura 4.10. Ejemplo de dos procesos VHDL que se comunican.

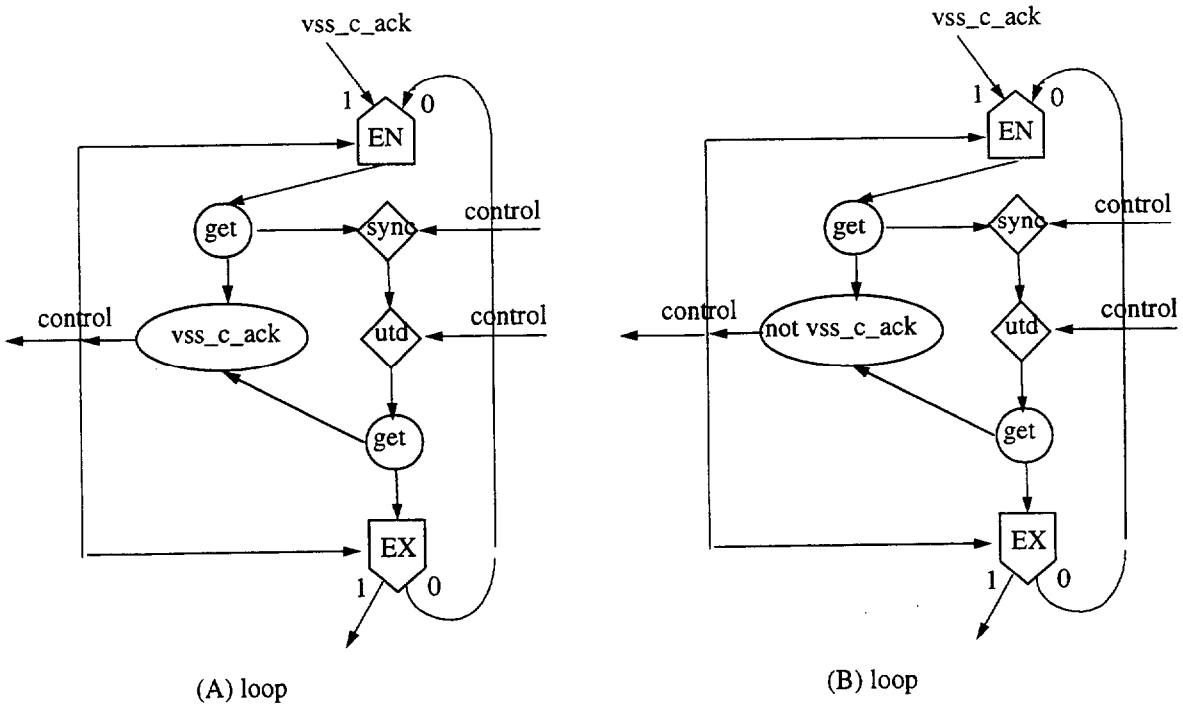


Figura 4.11. Bucles de la Figura 4.12.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

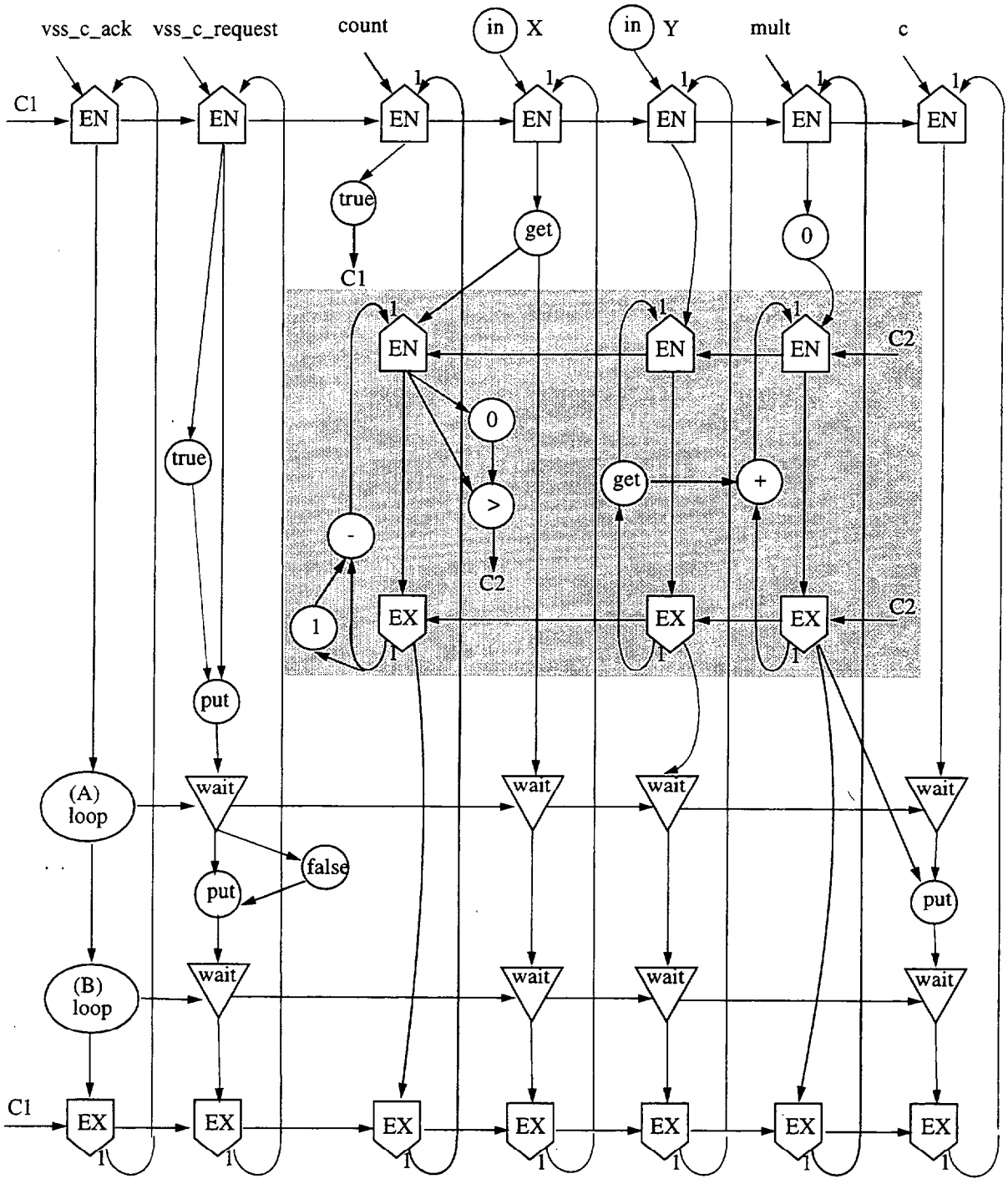


Figura 4.12. Grafo del proceso P1.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

Capítulo 5

Estimaciones

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

5.1. Introducción

La herramienta de particionado hardware/software de nuestro entorno de codiseño necesita las estimaciones de diversos parámetros relativos a la implementación de los procesos, tales como tiempo de ejecución, área y potencia. Estas estimaciones permiten que dicha herramienta explore el espacio de soluciones para optimizar el diseño.

Básicamente, existen dos tipos de estimaciones según el nivel de abstracción en el que se realizan: a) estimaciones de bajo nivel, y b) estimaciones de alto nivel.

Las estimaciones de *bajo nivel* son más precisas, ya que se tiene un conocimiento más detallado de la implementación del sistema. El problema de este tipo de estimaciones es que se necesita una descripción detallada del diseño. Por tanto, el diseñador necesita avanzar en la fase de diseño para obtener las estimaciones correspondientes a una determinada configuración hardware/software. Teniendo en cuenta que un diseño puede tener un elevado número de configuraciones hardware/ software, este proceso consume mucho tiempo de diseño. Por tanto, este tipo de estimaciones no resulta muy adecuado para tomar decisiones de diseño de alto nivel.

Las estimaciones de *alto nivel* [Nar92c][Ern93][Gup93][Kal94][The94][Bec96] [Ben96][Car96][Knu96][Nar96][Vah96][Wol96][Mac97] se realizan a partir de la especificación inicial del sistema. Este tipo de estimaciones se obtiene de forma muy rápida. De esta forma, el diseñador puede introducir modificaciones en el sistema y obtener rápidamente las correspondientes estimaciones. En el entorno de codiseño GACSYS se ha utilizado un esquema de estimaciones de alto nivel.

En este capítulo se presenta la fase de *estimaciones* que se ha desarrollado en este trabajo. En esta fase se generan las estimaciones del tiempo de ejecución, área y potencia para cada proceso especificado en la descripción del sistema. Estas estimaciones se generan para un amplio espectro de configuraciones hardware/software de los procesos. La principal aportación de la tesis en cuanto a estimaciones se refiere es la obtención de estimaciones para la potencia consumida por el diseño.

El método de estimaciones que se ha desarrollado está orientado a aplicaciones dominadas por los datos (*data dominated*) [Rab91][Bec96][Ben96][Knu96] que se pueden expresar como un grafo acíclico de procesos [Kal94][Ben96][Wol96]. En general, cada proceso de la especificación recibe unos datos de entrada, realiza un cierto procesamiento sobre los mismos, y finalmente los envía hacia otro proceso para que continúe procesándolos. Este tipo de aplicaciones es frecuente en el procesado digital de señales.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

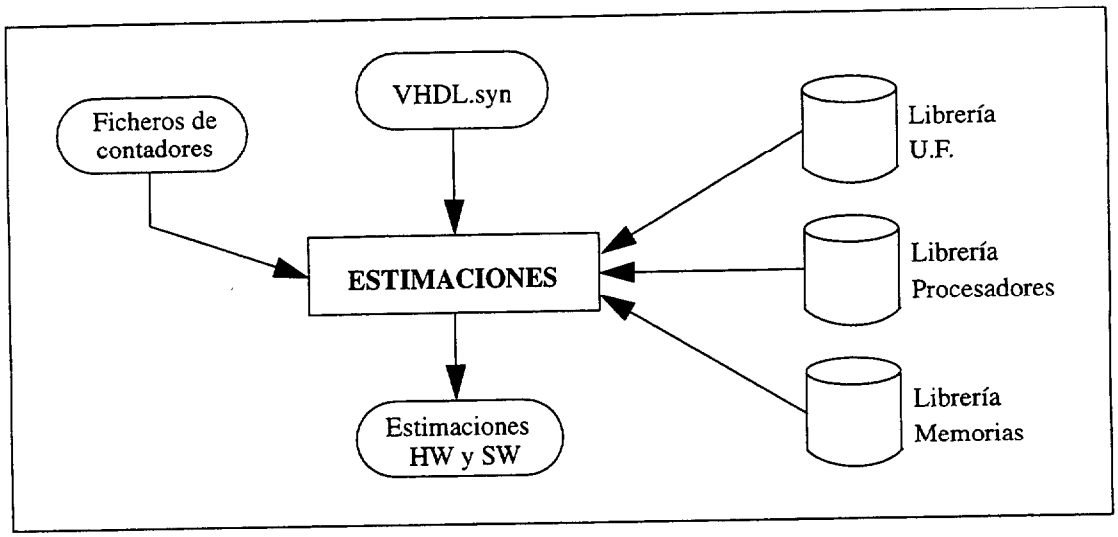


Figura 5.1. Esquema ilustrativo de la fase de estimaciones.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

En la Figura 5.1 se muestra un esquema de la fase de estimaciones desarrollada. Para generar las estimaciones se requiere: a) la especificación del diseño (descrita en VHDL), b) las librerías de unidades funcionales, procesadores y memorias que el diseñador quiere utilizar para diseñar el sistema, c) y un fichero de contadores que almacena información relativa a la ejecución del sistema.

Este capítulo se estructura de la siguiente forma: en los apartados 2 y 3 se describe la información de entrada y salida, respectivamente, de la fase de estimaciones. En el apartado 4 se explica el proceso seguido para la evaluación de las estimaciones. En el apartado 5 se explican los algoritmos de planificación que se han comparado para seleccionar el más adecuado para la fase de estimaciones. Finalmente, en el apartado 6 se presentan los experimentos realizados para seleccionar el algoritmo de planificación más adecuado.

5.2. Información de entrada

En este apartado se describe la información que se utiliza para el proceso de evaluación de las estimaciones. Básicamente se utilizan los siguientes ficheros:

- *Fichero de especificaciones*, que contiene la descripción VHDL del sistema. Esta descripción puede ser especificada directamente por el diseñador, o bien puede ser obtenida mediante un proceso de compilación a partir de una especificación descrita en el lenguaje VSS [Cas98d]. A partir de esta

descripción, se genera una representación interna en la que cada bloque básico es representado mediante un grafo acíclico que contiene las operaciones del bloque y sus dependencias [Cas99].

- *Ficheros de contadores*, que contienen información relativa al número de veces que se ejecuta cada bloque básico de la especificación VHDL. Estos ficheros se generan previamente mediante un proceso de simulación del fichero de especificaciones (*dynamic profiling*). De manera similar a otros trabajos [McF90] [Ern93] [Pen93] [Ele94] [The94] [Bec96] [Har96] [Knu96] [Nar96], la herramienta de estimaciones desarrollada utiliza esta información durante la evaluación de las estimaciones.
- *Fichero de unidades funcionales,* que especifica las unidades funcionales que el diseñador desea utilizar para implementar los operadores de la especificación.
- *Fichero de procesadores,* que especifica algunas características de los procesadores que el diseñador quiere utilizar para la implementación software de los procesos de la especificación VHDL (Figura 5.2).
- *Fichero de memorias*, que especifica las características de las memorias que el diseñador desea utilizar para almacenar el código y los datos de los procesos (Figura 5.3).

Nombre	*CPI*	*Área (mm^2)*	*Potencia (mw)*	*Frecuencia (MHz)*
80486	2.0	225.0	5000	50
pentium	1.0	296.0	16000	66
super_sparc	0.66	256.0	14200	60

Figura 5.2. Ejemplo de un fichero de procesadores.

Unidad	*Tamaño*	*T_acceso (s)*	*Área (mm^2)*	*Potencia (mw)*
RAM_1	64	15 E -9	0.414	5.0
RAM_2	128	15 E -9	0.458	7.0
RAM_3	256	20 E -9	0.548	10.0
RAM_4	512	20 E -9	0.714	15.0

Figura 5.3. Ejemplo de un fichero de memorias *(memorias.def)*.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

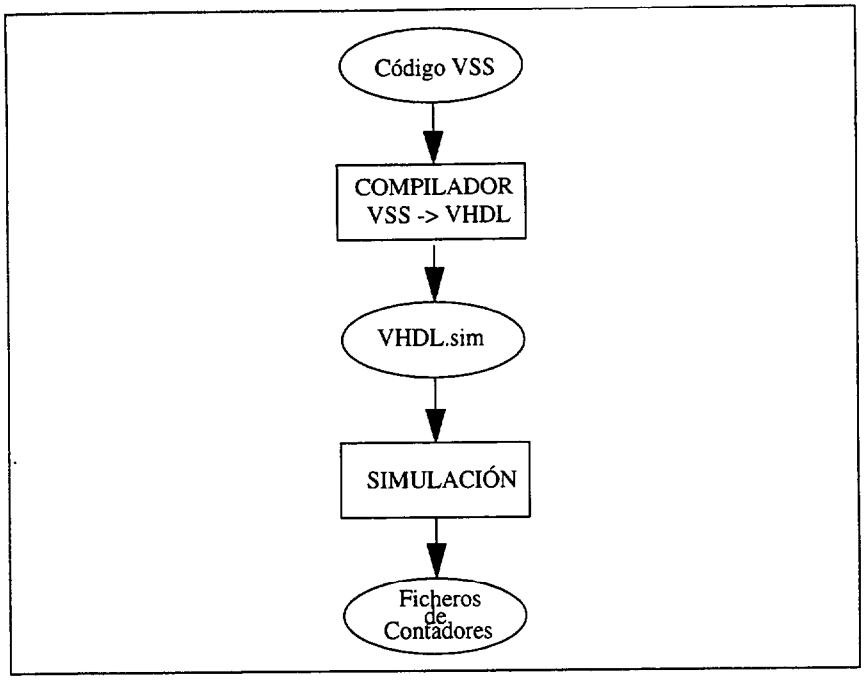


Figura 5.4. Flujo seguido para obtener los ficheros de contadores.

5.2.1. Ficheros de contadores

Estos ficheros se obtienen mediante la simulación del código VHDL del diseño. Este código es modificado ligeramente, de forma que durante el proceso de simulación se almacena en los ficheros de contadores la información correspondiente al número de veces que se ejecuta cada bloque básico de cada proceso [Cas98d]. Estos datos se guardan en ficheros diferentes, uno para cada proceso. En la Figura 5.4 se muestran los pasos seguidos para obtener esta información.

En general, un proceso tiene múltiples bloques básicos. Por tanto, es preciso indicar la relación que existe entre los contadores de un fichero y los bloques básicos del proceso correspondiente:

- El primer dato de un fichero de contadores corresponde al número de veces que se ha ejecutado el proceso.
- Los datos que aparecen a continuación corresponden a cada bloque básico del proceso (por orden de aparición en el fichero de especificaciones). Si el proceso define subprogramas, los primeros datos pertenecen a los bloques básicos de los mismos. Seguidamente se encuentran los contadores de los bloques básicos del cuerpo principal del proceso.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

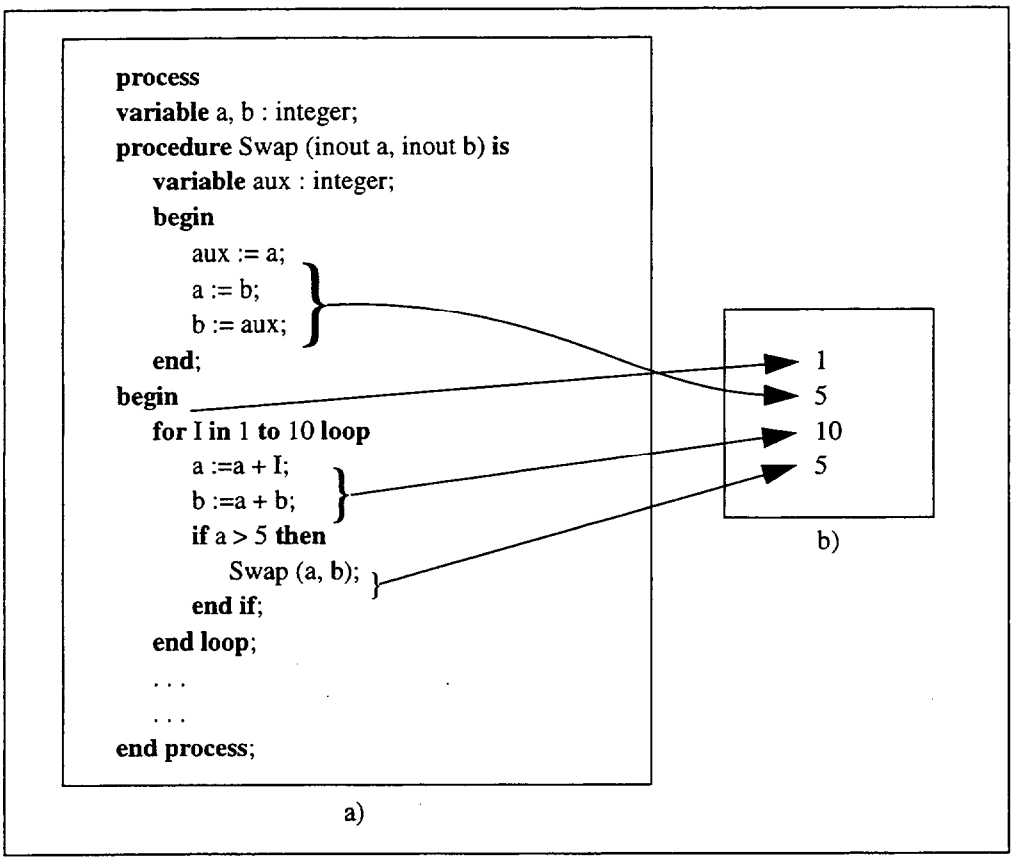


Figura 5.5. a) Código de un proceso VHDL. b) Ejemplo de un fichero de contadores.

En la Figura 5.5 se muestra un ejemplo que ilustra la correspondencia entre los datos de un fichero de contadores y el correspondiente proceso.

La información de los ficheros de contadores se utiliza para la evaluación de las estimaciones hardware y software de algunos parámetros de los procesos tales como tiempo de ejecución y potencia.

5.2.2. Fichero de unidades funcionales

En este fichero se especifica el conjunto de unidades funcionales que pueden utilizarse para implementar cada operador VHDL. Esta información es necesaria durante el proceso de obtención de las estimaciones hardware.

Un operador VHDL puede ser implementado por varias unidades funcionales diferentes, es decir, unidades que pueden tener diferente retardo, área y consumo de potencia (Figura 5.6). De esta forma pueden obtenerse soluciones más óptimas para el problema del particionado hardware/software, ya que se obtienen estimaciones para un espacio de diseño más amplio.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

Operador VHDL	*Unidad Funcional*
add_c	sum_1 sum_2 .
sub_c	sum_1 sum_2 .
mult_c	mult_1 mult_2 .
eq_c	comparador .
ne_c	comparador .
ls_c	comparador .
le_c	comparador .
gr_c	comparador .
ge_c	comparador .

a)

10 E -9 seg. /* Ciclo común */

Unidad	*Código*	*Retardo*	*Área (mm^2)*	*Potencia (mw)*	*Segmentada*	*Nº Etapas*
sum_1	1	2	0.127	22.7	0	0
sum_2	2	3	0.127	5.4	0	0
mult_1	3	16	0.363	84.0	0	0
mult_2	4	32	0.363	12.7	0	0
comparador	5	3	0.200	13.0	0	0

b)

Figura 5.6. a) Ejemplo de un fichero de unidades funcionales y b) ejemplo de un fichero de características de las mismas (*recursos.def*).

Las características de las unidades funcionales están definidas en un fichero de librería denominado *recursos.def* (Figura 5.6.b). En este fichero se especifican los siguientes datos para cada unidad funcional:

- *nombre* y *código* de identificación.
- *número de ciclos* (retardo) que utiliza para ejecutar una operación (si se trata de una unidad no segmentada), o número de ciclos utilizados en cada fase del *pipeline* si la unidad es *segmentada.*
- *área* y *potencia* consumida.
- *código* para indicar si la unidad es segmentada.
- *número de etapas* del *pipeline* (si la unidad es segmentada).

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

El retardo de todas las unidades funcionales viene dado en múltiplos de un ciclo común. El tiempo de este ciclo común es el máximo común divisor de los diferentes tiempos de ejecución de las unidades funcionales. El tiempo del ciclo común aparece especificado al comienzo del fichero.

5.3. Información de salida

El proceso de evaluación de las estimaciones proporciona un fichero en el que se especifica la siguiente información para cada proceso de la especificación:

- *Nombre del proceso.* Proceso al que pertenecen las estimaciones.
- *Estimaciones software.* Tiempo de ejecución, área y energía que se han estimado para la ejecución del proceso en el procesador seleccionado por el diseñador.
- *Memorias.* Indica las memorias que se han utilizado durante el proceso de evaluación de las estimaciones hardware y software del proceso.
- *Número de operadores.* Especifica el número de operadores diferentes que tiene la descripción VHDL del proceso.
- *Código de los operadores.* Indica el código de cada uno de los operadores VHDL presentes en el proceso.
- *Configuraciones de unidades funcionales.* Información relativa a cada una de las configuraciones de unidades funcionales que pueden utilizarse para una implementación hardware del proceso. La información especificada para cada configuración es la siguiente:
  - Número máximo de unidades funcionales para cada operador.
  - Código de las unidades funcionales utilizadas.
  - Tiempo de ejecución estimado para el proceso.
  - Área estimada que ocupa el proceso.
  - Energía media estimada que consume el proceso.

En la Figura 5.7 se muestra un ejemplo de la información de salida que genera el proceso de estimaciones.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

proceso_1 /* Nombre del proceso */

Estimaciones software /* Pentium */

Tiempo (seg)	Área (mm^2)	Energía (mj)
6.82e-07	2.96e+02	1.09e-2

Memoria : RAM_3

Estimaciones hardware

Memoria : RAM_1

3 /* Número de operadores */
* + >
6 4 22 /* Código de los operadores */

N° Unid.	Código	Tiempo (seg)	Área (mm^2)	Energía (mj)
* + >	* + + <			
1 1 1	3 1 5	1.900e-06	7.410e-01	1.973e-05
1 1 1	3 2 5	2.000e-06	7.410e-01	1.389e-05
1 2 1	3 1 1 5	1.900e-06	8.680e-01	1.973e-05
1 2 1	3 1 2 5	1.900e-06	8.680e-01	1.681e-05
1 2 1	3 2 1 5	2.000e-06	8.680e-01	1.681e-05
1 2 1	3 2 2 5	2.000e-06	8.680e-01	1.389e-05

Figura 5.7. Ejemplo de fichero generado durante el proceso de estimaciones.

5.4. Evaluación de las estimaciones

En este apartado se presenta el método utilizado para obtener las estimaciones hardware y software. Para cada uno de los procesos de la especificación VHDL se obtienen estimaciones de alto nivel de los siguientes parámetros: tiempo de ejecución, área y potencia.

Los primeros entornos de codiseño solamente obtenían estimaciones para el tiempo de ejecución y el área del diseño [Ele94][Kal94][Gup95][Ben96][Vah96][Wol96]. Sin embargo, la expansión de los equipos portátiles que se ha producido en los últimos años, y el aumento de la densidad de integración y la frecuencia de trabajo de los *chips*, han

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

generado la necesidad de estimar y considerar en mayor medida la potencia consumida por los diseños. De hecho, en los últimos años se han publicado trabajos sobre estimaciones y optimización del consumo de potencia de los diseños [Lan93][Meh94][Mar95][Joh97][Lin97][Mac97], aunque estos trabajos no han sido desarrollados en el marco de un entorno de codiseño.

El modelo de arquitectura utilizado (Figura 5.8) para evaluar las estimaciones es básicamente el siguiente:

- La implementación hardware de un proceso se lleva a cabo mediante una serie de unidades funcionales y una memoria local para almacenar sus datos.
- La implementación software de un proceso se lleva a cabo mediante el procesador especificado por el diseñador, así como una memoria para almacenar el código y los datos del proceso.

Este modelo de arquitectura es similar al que se utiliza en la mayoría de los entornos de codiseño [Ern93][Gup93][Ele94][Kal94][Bal96][Vah95b].

Como se observa en la Figura 5.8, el modelo de arquitectura utilizado para los procesos implementados en hardware solamente considera el *datapath* y la memoria de los mismos. Esta simplificación puede realizarse porque las estimaciones están

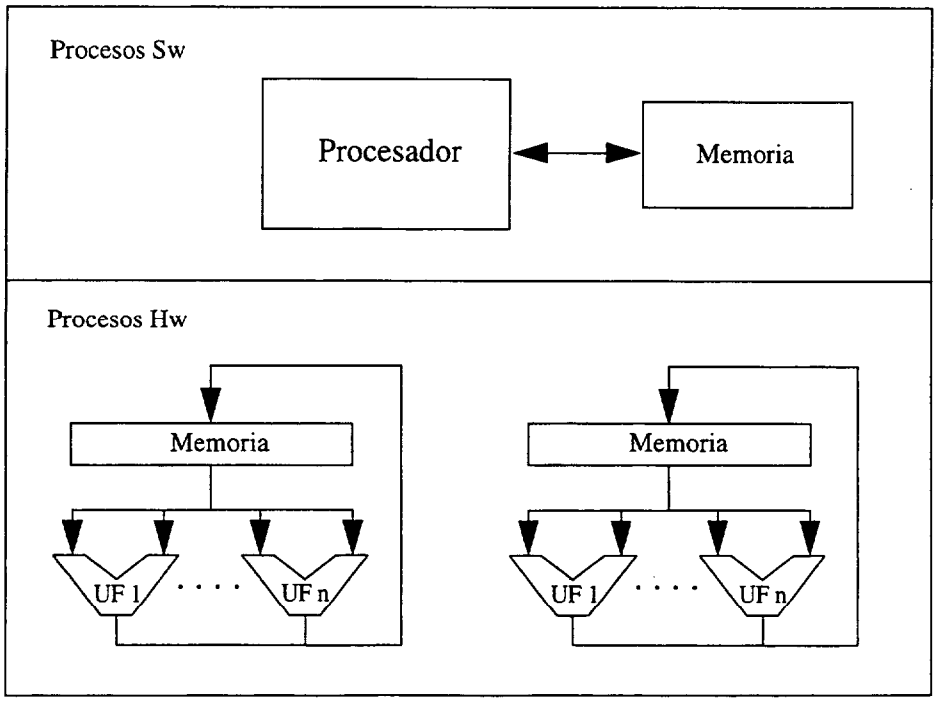


Figura 5.8. Modelo de arquitectura utilizado para la evaluación de las estimaciones.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

orientadas hacia aplicaciones dominadas por datos (*data dominated*), tal como se argumenta en trabajos como [Rab91][Rag94][Rag95][Joh97][Lin97]. En estos trabajos se considera que, en este tipo de aplicaciones, la variación que la unidad de control ocasiona en los parámetros estimados es pequeña en comparación con la que generan el *datapath* y la memoria.

5.4.1. Estimaciones hardware

El proceso de estimaciones hardware comienza obteniendo todas las configuraciones hardware para cada uno de los procesos de la especificación VHDL. Para generar estas configuraciones se realizan los siguientes pasos:

- En primer lugar, a partir de las dependencias de datos se calcula el número máximo de operadores de cada tipo que se pueden ejecutar en paralelo en cada bloque básico. Para cada operador se escoge el valor máximo de todos los bloques.
- En segundo lugar, se generan todas las configuraciones posibles de unidades funcionales teniendo en cuenta el número máximo de operadores de cada tipo que se pueden ejecutar en paralelo, así como las diferentes unidades que se pueden utilizar para implementar cada operador.

En la Figura 5.9 se ilustra con un ejemplo el método seguido para obtener todas las configuraciones hardware. Dado el código del proceso de la Figura 5.9.a, se determina que el número máximo de operadores que se pueden ejecutar en paralelo es de dos sumas y una resta (Figura 5.9.b). Por tanto, se tienen dos configuraciones posibles:

- 1 sumador y 1 restador.
- 2 sumadores y 1 restador.

Finalmente, dadas las unidades funcionales que pueden implementar cada operador (Figura 5.9.c) se obtienen todas las configuraciones de unidades funcionales (Figura 5.9.d).

Observando las configuraciones obtenidas en la Figura 5.9, podría parecer que la configuración número 7 es la misma que la número 9. Efectivamente, ambas configuraciones utilizan las mismas unidades funcionales, pero la diferencia es que las operaciones de suma tienen asignadas unidades diferentes. Como veremos posteriormente, durante el proceso de evaluación de las estimaciones se utilizan algoritmos de planificación de operaciones [Gaj92][Mic92]. Los resultados proporcionados por dichos algoritmos pueden ser diferentes según la asignación de las

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

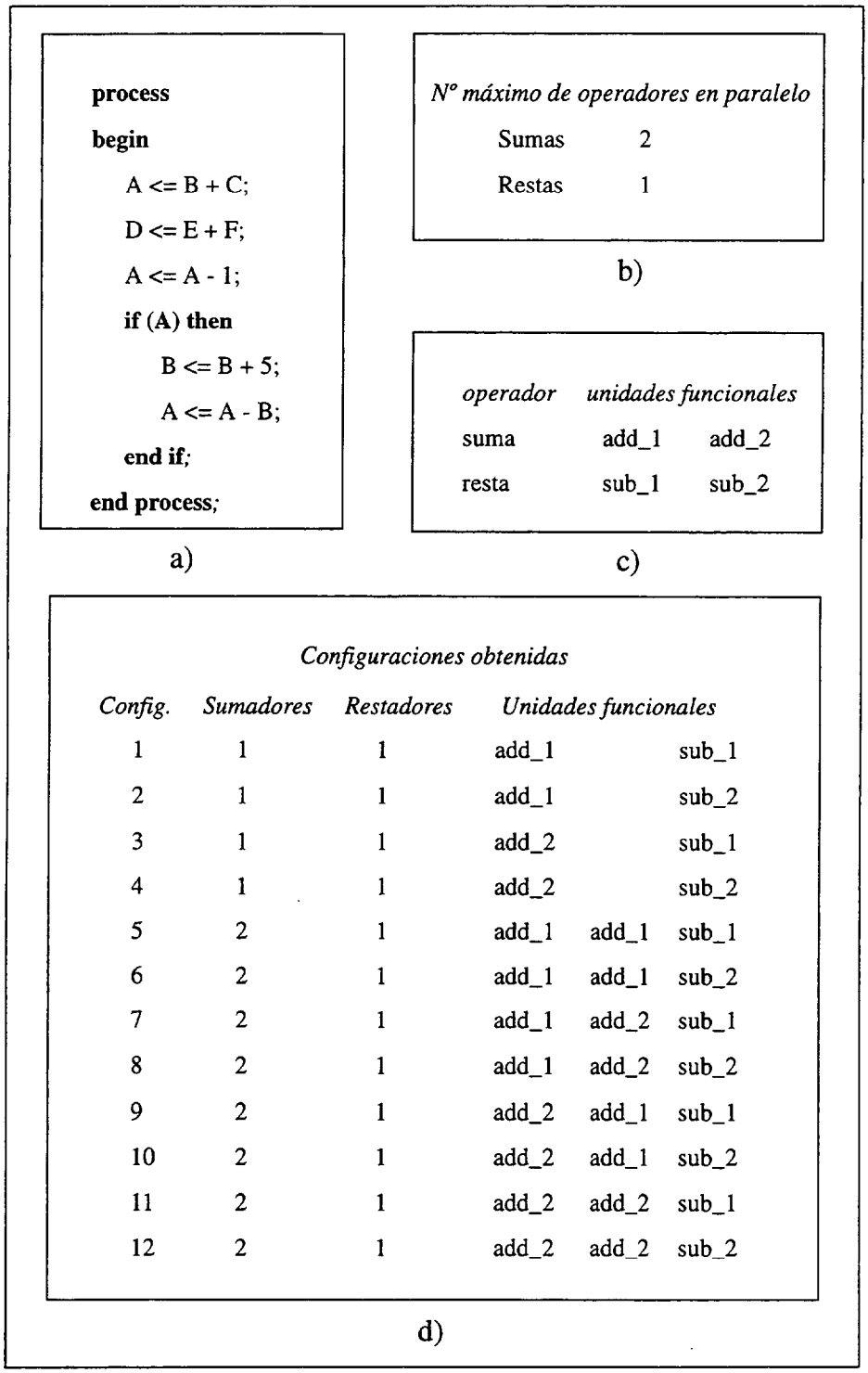


Figura 5.9. Ejemplo de obtención de las configuraciones hardware.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

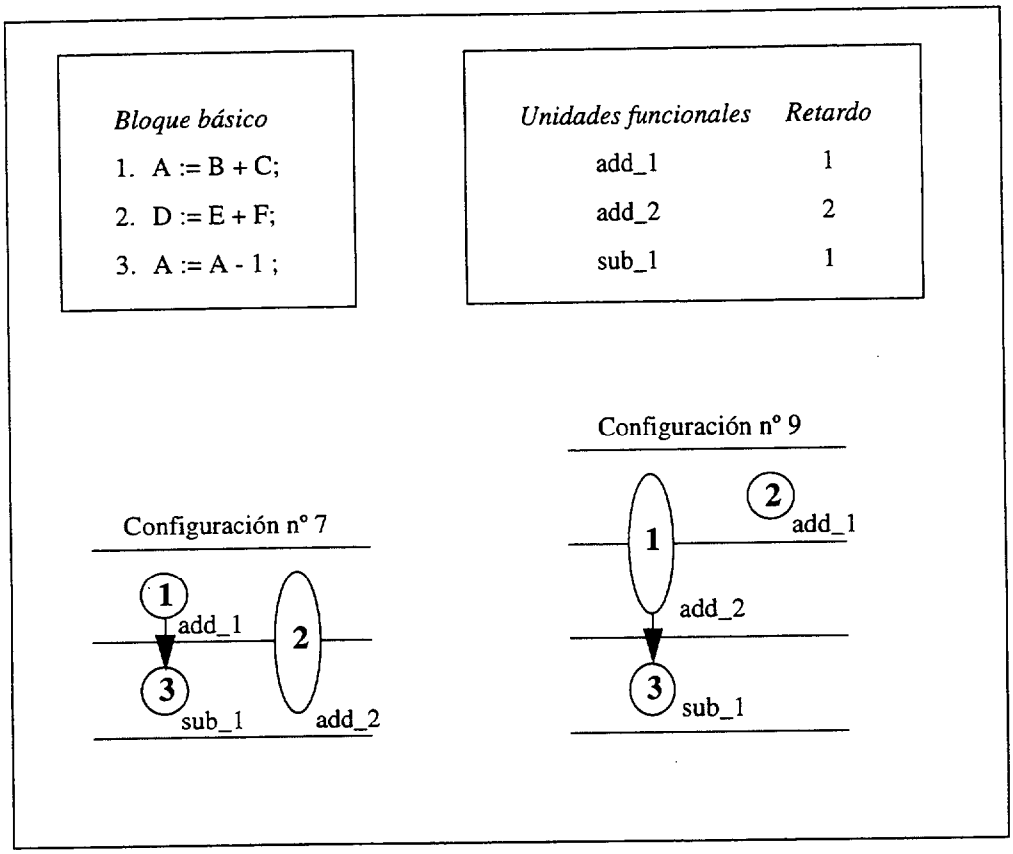


Figura 5.10. Ejemplo de planificaciones diferentes utilizando el mismo conjunto de unidades funcionales pero diferente asignación de las mismas.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

unidades funcionales. En la Figura 5.10 se ilustra un ejemplo de dos planificaciones realizadas sobre un determinado bloque básico utilizando las configuraciones número 7 y 9 de la Figura 5.9. Como se observa, aunque se utilizan las mismas unidades funcionales, el resultado de las planificaciones es diferente debido a la diferente asignación de las mismas.

En definitiva, el proceso de obtención de las configuraciones tiene en cuenta la asignación de las unidades funcionales (*allocation*), que es una de las principales tareas de las herramientas de síntesis de alto nivel [Gaj92][Mic92]. De esta forma, la herramienta de estimaciones explora un mayor espacio de diseño y permite que la herramienta de particionado hardware/software pueda realizar un mayor número de compromisos entre el tiempo de ejecución, el área y la potencia de un diseño.

Cuando el número de operaciones que se pueden ejecutar en paralelo es elevado, el número de configuraciones puede crecer de forma considerable. En ese caso, la fase de evaluación de las estimaciones puede necesitar un tiempo de computación elevado. Por

ello, se permite que el diseñador limite el número máximo de recursos de cada tipo que se pueden ejecutar en paralelo. De esta forma se puede disminuir el número de configuraciones, y por lo tanto acelerar el cálculo de las estimaciones. Por supuesto, este tipo de limitación elimina algunas configuraciones del espacio de diseño. Por tanto, la calidad del particionado hardware/software podría disminuir. El diseñador aplica estas limitaciones mediante un fichero en el que especifica los diferentes operadores de VHDL y el límite máximo de recursos para cada uno de ellos.

Una vez que se han obtenido todas las configuraciones de unidades funcionales, se procede a la selección de la memoria de cada proceso en función del tamaño que cada uno necesita. El tamaño de la memoria necesaria para almacenar los datos de un proceso se obtiene sumando los tamaños de sus variables junto con el tamaño necesario para almacenar los datos de la jerarquía de subprogramas de dicho proceso. Dada la jerarquía de llamadas a subprogramas, y dado el tamaño que necesita cada uno de ellos para sus variables locales, el tamaño de la memoria necesaria para un proceso se obtiene siguiendo el algoritmo de la Figura 5.11.

En la Figura 5.12 se muestra un ejemplo que ilustra el proceso de cálculo seguido por dicho algoritmo. En dicha figura se muestra una jerarquía de llamadas a subprogramas. El nodo raíz de la jerarquía representa a un proceso, mientras que el resto de los nodos representan a los subprogramas del proceso. En el interior de cada nodo se representa el tamaño necesario para almacenar las variables locales. Los valores situados en el exterior de los nodos son generados por el algoritmo y representan el tamaño total necesario para almacenar las variables de la jerarquía que depende de cada nodo. Por tanto, el valor exterior del nodo raíz indica el tamaño de la memoria necesaria para los datos del proceso.

```
L = { lista de nodos interiores de la jerarquía }
∀ nodo n de la lista L hacer[1]
   K = { conjunto de hijos del nodo n }
   tamaño_hijo_mayor = max{ tamaño(m) | m es un nodo de K }
   tamaño(n) = tamaño(n) + tamaño_hijo_mayor

[1] Los nodos deben visitarse con un recorrido bottom-up.
```

Figura 5.11. Algoritmo para obtener el tamaño de la memoria de datos necesaria para un proceso.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

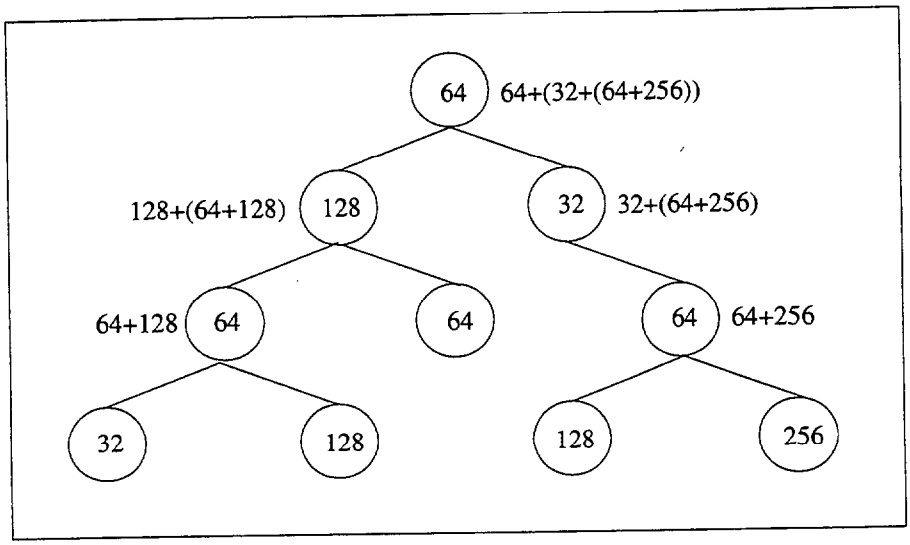


Figura 5.12. Ejemplo ilustrativo del Algoritmo de la Figura 5.11.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

Una vez que se han obtenido las configuraciones de unidades funcionales, y se han seleccionado las memorias, se procede a la evaluación de las estimaciones de los siguientes parámetros: tiempo de ejecución, área y potencia. Dichas estimaciones pueden evaluarse utilizando básicamente dos tipos de técnicas:

- técnicas que consideran los efectos de las tareas más importantes de la síntesis de alto nivel, es decir, la planificación de las operaciones de la especificación (*scheduling*), y la asignación de las mismas a las unidades funcionales (*allocation*). Este tipo de técnicas han sido utilizadas en trabajos como [Ern93] [The94][Rag95][Mon96][Joh97][Lin97]. La ventaja principal de estas técnicas es que proporcionan buenas estimaciones. Los algoritmos de planificación más utilizados en estos trabajos son el ASAP (*As Soon As Possible*) y la planificación basada en listas (*list-based scheduling*) [Gaj92][Mic92].
- técnicas que utilizan modelos simplificados de las anteriores. El objetivo de las mismas es obtener las estimaciones de forma más rápida, aunque la calidad de las mismas es inferior a la obtenida con los métodos anteriores. Así por ejemplo, en [Nar96] se presenta una técnica de este tipo que obtiene estimaciones cuya calidad es un 14% inferior a la obtenida con los métodos anteriores. Este tipo de técnicas se ha utilizado en trabajos como [Gup95][Ism95][Nar96].

En este trabajo se ha optado por utilizar una técnica no simplificada, es decir, la evaluación de las estimaciones se basa en la planificación (*scheduling*) de las operaciones de la especificación. La asignación de las unidades funcionales a las operaciones (*allocation*) se tiene en cuenta durante la generación de las distintas configuraciones, como ya se ha explicado anteriormente.

Para obtener las estimaciones de un proceso se planifican las operaciones de cada bloque básico del mismo. La planificación se realiza para cada una de las configuraciones de unidades funcionales que se hayan generado para el proceso. Es decir, el proceso de planificación se ejecuta bastantes veces durante la evaluación de las estimaciones. Por tanto, es importante que el algoritmo de planificación sea rápido.

En este trabajo se utiliza un algoritmo de planificación basado en listas (*list-based scheduling*) [Mic92][Gaj92]. La selección del mismo se realizó después de llevar a cabo una serie de experimentos para comparar la calidad y velocidad de diversos algoritmos de planificación. Los algoritmos y los experimentos se presentan en los siguientes apartados.

Una vez que se ejecuta el algoritmo de planificación se procede al cálculo de las estimaciones. A continuación se presenta el método de evaluación de cada parámetro.

La estimación del tiempo de ejecución de un proceso para una determinada configuración hardware se calcula sumando el tiempo de ejecución de cada bloque básico. Para ello, se considera el tiempo de ejecución debido a las unidades funcionales y el tiempo debido a los accesos a memoria.

El número de ciclos consumidos en un bloque básico *i-ésimo* debido a las unidades funcionales se obtiene multiplicando el número de ciclos que se obtiene al planificar el bloque (C_i) por el número de veces que se ejecuta el mismo (N_i). El tiempo de ejecución consumido por el bloque debido a las unidades funcionales se obtiene multiplicando el número de ciclos por el tiempo de un ciclo (t_c).

El tiempo de ejecución consumido en un bloque básico *i-ésimo* debido a los accesos a memoria se obtiene multiplicando el tiempo de acceso a memoria obtenido al planificar el bloque por el número de veces que se ejecuta el mismo (N_i). El primer factor se obtiene multiplicando el número de accesos del bloque a memoria ($N_{accesos,i}$) por el tiempo consumido en cada acceso (t_{acceso}). Cada operación binaria realiza tres accesos a memoria (dos accesos para leer y uno para escribir). Así mismo, las operaciones unarias realizan dos accesos a memoria (uno para leer y otro para escribir).

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

$$T_{hw} = \sum_i N_i \, (C_i \, t_c + N_{accesos,i} \, t_{acceso})$$

N_i	número de veces que se ejecuta el bloque básico i-ésimo.
C_i	número de ciclos obtenidos al planificar el bloque básico i-ésimo.
t_c	tiempo consumido por un ciclo.
$N_{accesos,i}$	número de accesos a memoria del bloque básico i-ésimo.
t_{acceso}	tiempo de acceso a memoria.

Expresión 5.1. Estimación del tiempo de ejecución de un proceso para una determinada configuración hardware.

Por lo tanto, el tiempo de ejecución estimado para un proceso viene dado por la Expresión 5.1.

En general, un bloque básico no contiene solamente operadores VHDL que pueden ser directamente implementados por las unidades funcionales, ya que también puede contener llamadas a subprogramas. A continuación se explica cómo se tienen en cuenta los subprogramas para las estimaciones del tiempo de ejecución.

Para una configuración hardware determinada, los subprogramas se implementan utilizando los recursos de dicha configuración, es decir, los mismos recursos que utiliza el proceso.

Los subprogramas son tratados como cualquier otro operador durante el proceso de planificación de un bloque básico. El número de ciclos que necesita un subprograma depende del tipo de planificación realizada: intraprocedural o interprocedural. La planificación interprocedural es compleja y conlleva excesivo tiempo de computación, por lo que solamente se ha realizado planificación intraprocedural. Por tanto, los ciclos utilizados para ejecutar un subprograma no pueden ser utilizados por el resto de las operaciones del bloque básico. Es decir, la planificación de un subprograma añade un número de ciclos exclusivos para la ejecución del mismo. Por tanto, la forma de contabilizar el tiempo de ejecución de los subprogramas para la estimación del tiempo de ejecución de un proceso es la siguiente:

- En primer lugar, cada ejecución de un subprograma puede tener un tiempo de ejecución diferente, dependiendo de los datos y del flujo de control seguido durante la ejecución. Por tanto, es preciso definir un tiempo medio de ejecución que hemos denominado *tiempo unitario*.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

- El *tiempo unitario* de un subprograma se obtiene contabilizando el tiempo de ejecución acumulado durante todas las llamadas al mismo (obtenido con la Expresión 5.1), y dividiendo este valor por el número de llamadas realizadas.
- El tiempo de ejecución debido a un subprograma se añade al tiempo de un bloque básico multiplicando el valor del *tiempo unitario* por el número de veces que se ejecuta el bloque.

Para finalizar con la evaluación del tiempo de ejecución de un proceso es preciso explicar cómo se tiene en cuenta el tiempo debido a las comunicaciones entre procesos.

Dado que este trabajo está orientado a aplicaciones dominadas por los datos (*data dominated*) que se pueden expresar como un grafo acíclico de procesos, consideramos que cada proceso de la especificación recibe unos datos de entrada, realiza un cierto procesamiento sobre los mismos, y finalmente los envía hacia otro proceso para que continúe procesándolos.

Dado este tipo de aplicaciones, las comunicaciones entre los procesos se encuentran desacopladas de los procesos de cálculo, ya que las mismas se producen al principio y al final de cada proceso, tal como se muestra en la Figura 5.13. Por lo tanto, no se producen tiempos de espera aleatorios en los procesos debido a las comunicaciones.

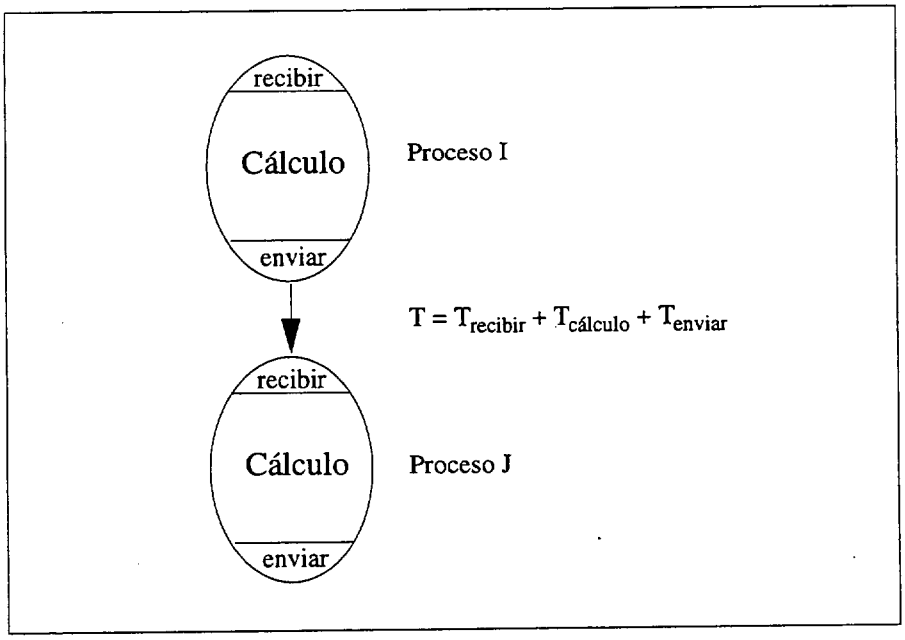


Figura 5.13. El cálculo y las comunicaciones de los procesos están desacoplados.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

Por tanto, el tiempo debido a las comunicaciones se estima como el tiempo consumido por las diferentes sentencias utilizadas para enviar y recibir los datos. Este tiempo ya ha sido considerado en el proceso de estimaciones que se ha explicado anteriormente para cada proceso, ya que las sentencias utilizadas para las comunicaciones son consideradas como el resto de sentencias durante el proceso de estimación del tiempo de ejecución (Expresión 5.1).

La estimación del área utilizada por un proceso para una configuración hardware determinada se obtiene sumando el área ocupada por la memoria seleccionada para el proceso junto con el área ocupada por cada unidad funcional utilizada en dicha configuración (Expresión 5.2).

$$A_{hw} = A_{mem} + \sum_i A_{uf,i}$$

A_{mem}	Área ocupada por la memoria del proceso.
$A_{uf,i}$	Área ocupada por la unidad funcional i-ésima.

Expresión 5.2. Estimación del área de un proceso para una determinada configuración hardware.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

En cuanto a las estimaciones de potencia se refiere, en primer lugar debemos puntualizar 3 aspectos:

- Para la evaluación de las estimaciones se utiliza la energía (potencia · tiempo) en lugar de la potencia [Cha92a][Cha92b][Mar95].
- Las estimaciones realizadas suponen una implementación con tecnología CMOS, en la que el principal consumo de potencia se produce durante el régimen dinámico de los transistores *(switching power)* [Cha92a][Cha92b] [Rag94][Cha95][Mar95][Rag95][Rab96].
- Se ha supuesto la utilización de técnicas de *power-down*, es decir, se deshabilita la señal de reloj de algunos *latches* para que las unidades funcionales que no son utilizadas en un determinado ciclo no consuman potencia [Ell91][Cha92a][Mar95][Mon96][Mac97].

La estimación de la energía media que consume un proceso para una determinada configuración hardware se calcula sumando la energía consumida en cada bloque básico. Para ello, se considera el consumo debido a las unidades funcionales y el consumo debido a los accesos a memoria.

El consumo energético en un bloque básico *i-ésimo* debido a las unidades funcionales se obtiene sumando el consumo de cada unidad funcional que ejecuta operaciones del bloque ($E_{uf,i}$), y multiplicando este valor por el número de veces que se ejecuta el mismo (N_i).

El consumo energético en un bloque básico *i-ésimo* debido a los accesos a memoria se obtiene, en primer lugar, multiplicando el número de accesos a memoria realizados en el bloque ($N_{accesos,i}$) por el número de veces que se ejecuta el mismo (N_i). De esta forma se obtiene el número total de accesos a memoria, que multiplicado por el consumo energético de cada acceso (e_{acceso}) proporciona el consumo del bloque. Cada operación binaria realiza tres accesos a memoria (dos accesos para leer y uno para escribir). Así mismo, las operaciones unarias realizan dos accesos a memoria (uno para leer y otro para escribir).

Por lo tanto, el consumo de energía estimado para un proceso viene dado por la Expresión 5.3.

El tratamiento de los subprogramas es similar al realizado para las estimaciones del tiempo de ejecución. En primer lugar, se obtiene la *energía unitaria* consumida por el subprograma. Esta energía se obtiene sumando la energía consumida por todas las llamadas al mismo, y dividiendo la misma por el número de llamadas realizadas. La energía que consume un subprograma dentro de un bloque básico se obtiene multiplicando la *energía unitaria* por el número de veces que se ejecuta el bloque.

$$E_{hw} = \sum_i N_i \left(N_{accesos,i}\, e_{acceso} + \sum_{uf} E_{uf,i} \right)$$

N_i	número de veces que se ejecuta el bloque básico i-ésimo.
$N_{accesos,i}$	número de accesos a memoria del bloque básico i-ésimo.
e_{acceso}	energía consumida en un acceso a memoria.
$E_{uf,i}$	energía consumida por la unidad funcional uf-ésima en el bloque i-ésimo

Expresión 5.3. Estimación de la energía que consume un proceso para una determinada configuración hardware.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

5.4.2. Estimaciones software

La obtención de estimaciones software de calidad requiere el desarrollo de herramientas complejas. En esta primera versión del entorno de codiseño no se han desarrollado este tipo de estimaciones. Actualmente, el modelo de estimaciones software que se utiliza es simple. La idea es estimar los mismos parámetros que en el caso de las estimaciones hardware: tiempo de ejecución, área y potencia. Aunque la calidad de dichas estimaciones no es óptima, es necesario disponer de un primer modelo sencillo que nos permita la posibilidad de comenzar a realizar experimentos con la herramienta de particionado. A continuación se explica cómo se estima cada parámetro.

Para estimar el tiempo de ejecución de un proceso en el procesador seleccionado por el diseñador, nos hemos basado en la fórmula presentada en [Pat90] (Expresión 5.4).

Cada operación de un bloque básico, tanto de cómputo como de lectura/escritura en memoria, se contabiliza como una instrucción. Por lo tanto, el valor del parámetro N viene dado por la Expresión 5.5.

El valor del parámetro CPI y la frecuencia del procesador son especificados por el diseñador.

$$T_{sw} = N \cdot CPI \cdot 1/f$$

N	número de instrucciones ejecutadas.
CPI	número promedio de ciclos por instrucción.
f	frecuencia del procesador.

Expresión 5.4. Estimación del tiempo de ejecución de un proceso ejecutado en un procesador.

$$N = \sum_i N_i \, (O_i + N_{accesos,i})$$

N_i	número de veces que se ejecuta el bloque básico i-ésimo.
O_i	número de operaciones del bloque básico i-ésimo.
$N_{accesos,i}$	número de accesos a memoria del bloque básico i-ésimo.

Expresión 5.5. Número de instrucciones ejecutadas por el procesador.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

Los datos necesarios para estimar el área de una implementación software de un proceso son los siguientes: a) el área del procesador, que la especifica el diseñador, y b) el área de la memoria en la que se almacenan el código y los datos del proceso. Por un lado, el tamaño de la memoria necesario para almacenar los datos se obtiene de forma similar a la explicada para la obtención de las estimaciones hardware. Por otro lado, el tamaño de la memoria necesaria para almacenar el código del proceso (M_{codigo}) se obtiene mediante la Expresión 5.6.

$$M_{codigo} = \omega \sum_i O_i + N_{accesos,i}$$

O_i	número de operaciones del bloque básico i-ésimo.
$N_{accesos,i}$	número de accesos a memoria del bloque básico i-ésimo.
w	tamaño de instrucción del procesador.

Expresión 5.6. Número de instrucciones estimadas para el código de un proceso.

Finalmente, la energía media consumida por un proceso implementado en software viene dada por el producto de la potencia media consumida por el procesador y del tiempo de ejecución estimado para el proceso.

5.5. Algoritmos de planificación

La planificación de operaciones de un proceso tiene una influencia muy importante sobre el coste (área y potencia) y el tiempo de ejecución de la implementación hardware del mismo. Por un lado, las operaciones asignadas a un mismo paso de control requieren, para su ejecución, una unidad funcional para cada operación. Por otro lado, el número total de pasos de control necesarios para la ejecución de un proceso disminuye conforme aumenta el número de operaciones que se planifican en cada paso de control. Esta planificación se lleva a cabo mediante un algoritmo que asigna a las operaciones de cada proceso el paso de control en el que deben ejecutarse.

La fase de estimaciones de GACSYS ejecuta muchas veces el algoritmo de planificación. En concreto, el algoritmo es ejecutado para cada bloque básico de cada proceso tantas veces como configuraciones hardware se hayan obtenido. Por tanto, es muy importante que el algoritmo sea rápido, por lo que se ha optado por utilizar un algoritmo heurístico.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

A continuación se presentan los algoritmos heurísticos de planificación que se estudiaron como candidatos para ser utilizados en nuestro entorno de codiseño.

5.5.1. Planificación basada en listas *(list-based)*

Dado un número limitado de recursos, el objetivo del algoritmo de planificación *list-based* [Gaj92][Mic92] es minimizar el número de pasos de control necesarios para la planificación de un GA. Este algoritmo alcanza su objetivo planificando en cada paso de control el mayor número de operaciones posibles.

El algoritmo que hemos diseñado es una ampliación del algoritmo básico que se ha presentado en el capítulo 3. Esta ampliación consiste básicamente en soportar unidades funcionales multiciclo y unidades segmentadas. En la Figura 5.14 se muestra una unidad multiciclo (tres ciclos) de multiplicación y una unidad segmentada de suma.

El criterio que se ha seleccionado para el algoritmo implementado en este trabajo está basado en la *movilidad* de las operaciones, ya que en general es el criterio que mejores resultados proporciona [Gaj92][Mic92]. Como consecuencia de esta elección es preciso realizar las planificaciones ASAP y ALAP para el cálculo de las movilidades iniciales de cada operación.

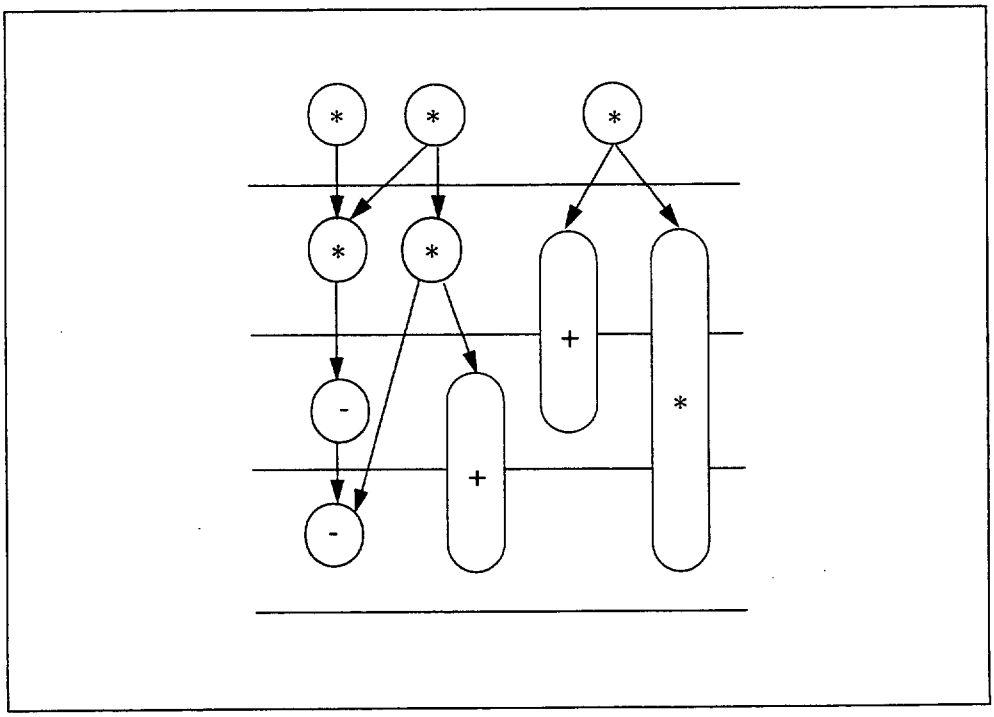


Figura 5.14. Planificación con unidades multiciclo y segmentadas.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

5.5.2. Planificación basada en el *simulated annealing*

En este apartado se explica un algoritmo de planificación para la optimización del número de pasos de control que está basado en el algoritmo iterativo conocido como *simulated annealing* [Met53][Kir83][Dow95].

La utilización del *simulated annealing* para el problema de la planificación requiere la fijación de una serie de parámetros del algoritmo. El conjunto de parámetros puede dividirse en dos grupos:

- *Parámetros relacionados con la metodología de annealing*: la temperatura inicial, la probabilidad de aceptar soluciones peores que la actual, el esquema de enfriamiento, y la condición de parada.
- *Parámetros relacionados con el problema de la planificación*: el espacio de soluciones, la estructura de la vecindad, y la función de coste.

La función TEMPERATURA_INICIAL proporciona la temperatura inicial con la que comienza el proceso de enfriamiento. La temperatura inicial debe seleccionarse de forma que al comienzo del proceso de *annealing* sea muy alta la probabilidad de aceptar una solución vecina con un coste peor que la solución actual. De esta forma se evita que el proceso quede atrapado desde el principio en el *valle* del mínimo local correspondiente a la solución inicial. Para inicializar correctamente dicha temperatura es preciso estimar la peor ganancia que puede haber entre cualquier pareja de soluciones. Una vez que se estima la peor ganancia, la temperatura inicial se obtiene mediante la Expresión 5.7, en la que P_i es la probabilidad inicial que se desea para la aceptación de soluciones peores que la actual. Por tanto, P_i debe tener valores próximos a la unidad.

En la Expresión 5.7 se asume que la función P que se utiliza, y que proporciona la probabilidad de aceptación de una solución vecina que tiene un coste superior a la solución actual, es la que se muestra en la Expresión 5.8.

En la Figura 5.15 se muestra la forma que tiene la función P. Claramente se observa que a medida que la temperatura *t* tiende a cero, también tiende a cero la probabilidad de

$$t \geq \frac{\Delta E}{Ln\,(Pi)}$$

Expresión 5.7. Selección de la temperatura inicial.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

$$p(\Delta E) = e^{\frac{-\Delta E}{t}}$$

Expresión 5.8. Función P utilizada para aceptar soluciones vecinas peores que la solución actual.

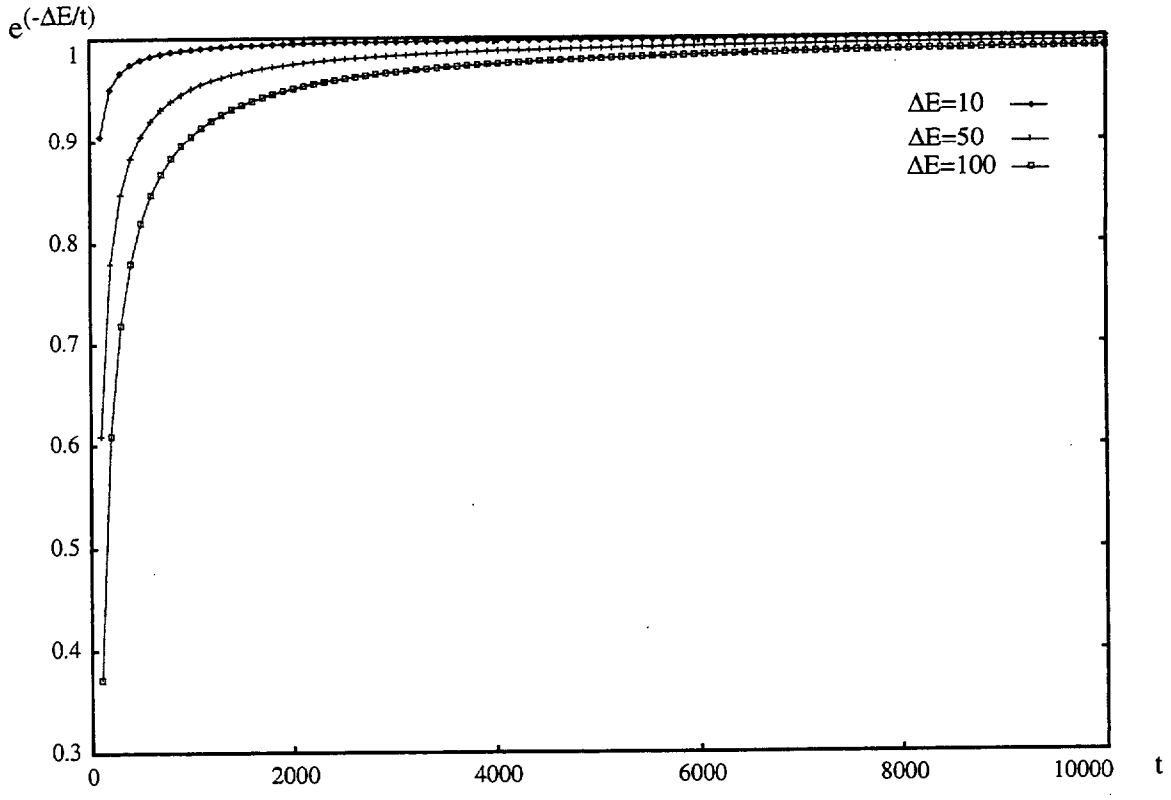


Figura 5.15. Forma de la función P que se utiliza para la aceptación de una solución con un coste superior al de la solución actual.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

aceptar una solución vecina peor que la actual. Esta tendencia se produce independientemente del valor de la ganancia (ΔE). La ganancia solamente acelera en mayor o menor medida esta tendencia.

El esquema de enfriamiento del algoritmo está gobernado por los parámetros NUM_VECINOS y por la función TEMPERATURA, y es de suma importancia para el proceso de *annealing*.

El parámetro NUM_VECINOS controla el número de vecinos que se visitan para una misma temperatura. La función TEMPERATURA controla el descenso de la temperatura. La calidad de las soluciones es mayor si la temperatura se reduce muy lentamente y el número de iteraciones es muy elevado para cada temperatura, aunque el tiempo de computación también crece de forma considerable. Si se quiere disminuir el tiempo de computación se puede optar por reducir la temperatura más rápidamente, o disminuir el número de vecinos visitados para una misma temperatura. En este trabajo se ha seleccionado un número de $2n$ vecinos, siendo n el número total de operaciones a planificar. Para la reducción de la temperatura se ha escogido una función de reducción geométrica ($t=\alpha \cdot t$, $0<\alpha<1$), que es la que se utiliza con más frecuencia debido a que no es computacionalmente muy costosa [Dow95].

La condición de parada, según el algoritmo de Metrópolis, debe garantizar la convergencia del sistema hacia un estado de congelación estable. Para un proceso de optimización, esto quiere decir que el sistema debe alcanzar un óptimo local estable. La condición de parada que se ha utilizado en este trabajo se alcanza cuando la temperatura está muy próxima a cero, de tal forma que el valor de la función P es prácticamente cero.

Una vez que se han explicado los parámetros que están intrínsecamente relacionados con el proceso de *annealing*, a continuación se explican los parámetros relacionados con el problema de la planificación.

Las alternativas principales que se pueden utilizar para obtener una planificación inicial son las siguientes:

- obtenerla planificando cada operación en un paso de control. Su principal ventaja es que se obtiene muy rápidamente. Sin embargo, la calidad de la solución no es muy buena.
- obtenerla mediante algoritmos de planificación más avanzados tal como el *list-based*. De esta forma se obtiene una mejor planificación inicial, pero el tiempo de ejecución para obtenerla es mayor que en el caso anterior.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

El algoritmo de *simulated annealing* que se ha desarrollado en este trabajo utiliza la primera alternativa para la obtención de la solución inicial, es decir, se ha optado por una solución inicial que no tiene mucha calidad, pero que se obtiene de forma rápida.

La función SELECCIONAR_VECINO selecciona de forma aleatoria una solución vecina de la solución actual. El criterio de vecindad que se ha escogido se basa en que una solución vecina se diferencia de la solución actual en la planificación de una de sus operaciones. No obstante, se han seleccionado tres tipos de vecindad diferentes en función de los movimientos que puede realizar la operación de la solución actual que se mueve hacia otro paso de control. Distinguimos tres tipos de movimientos diferentes:

- *Movimientos correctos*: el conjunto de vecinos de una solución está formado por soluciones correctas, es decir, soluciones que respetan las dependencias de datos de las operaciones del GA (Figura 5.16.a).
- *Movimientos amplios*: el conjunto de vecinos de una solución puede incluir soluciones incorrectas, es decir, soluciones que no respetan las dependencias de las operaciones del GA. Las operaciones que no respetan las dependencias pueden estar planificadas en cualquier paso de control (Figura 5.16.b).

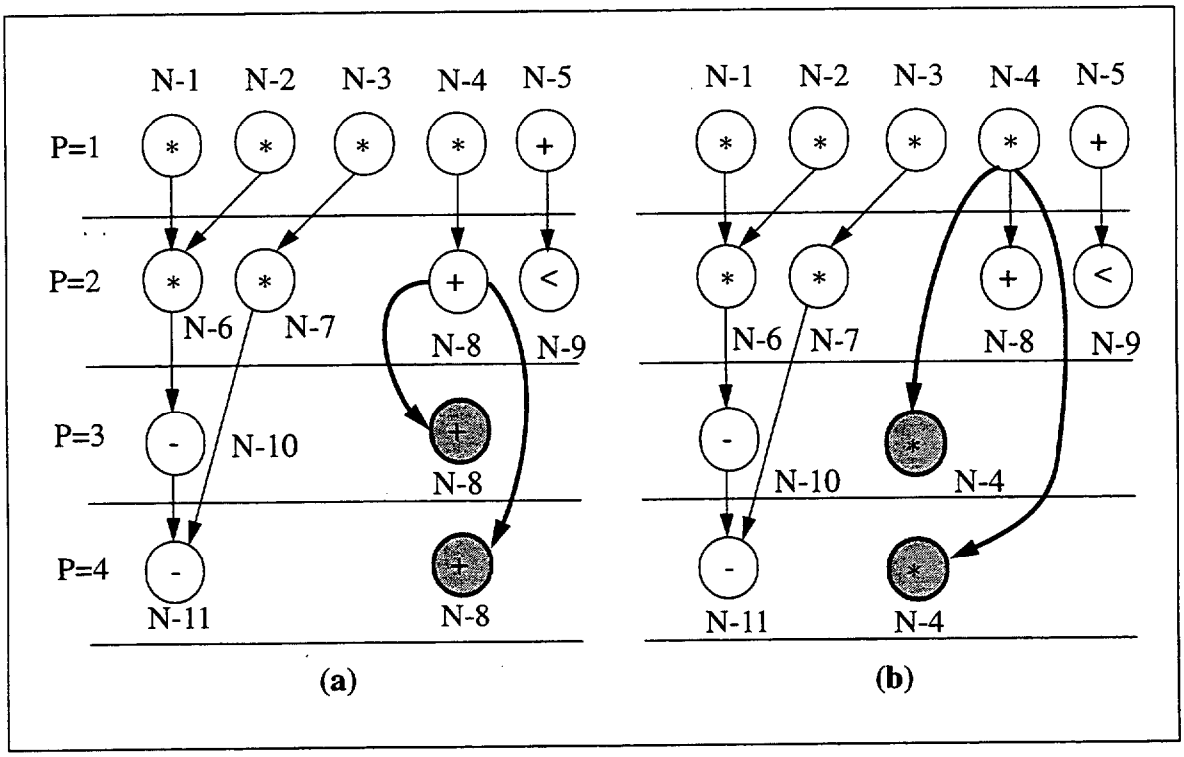


Figura 5.16. Algunos movimientos (a) correctos y (b) amplios para obtener soluciones vecinas.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

- *Movimientos restringidos*: el conjunto de vecinos de una solución puede incluir soluciones incorrectas, aunque no se permiten todo tipo de soluciones incorrectas. En concreto, aquellas operaciones que no respetan las dependencias no pueden estar planificadas en un paso de control inferior al de sus operaciones predecesoras, ni en un paso superior al de sus operaciones sucesoras (Figura 5.17).

La vecindad basada en movimientos correctos tiene la ventaja de que se tiene la certeza que la solución final siempre respeta las dependencias. Sin embargo, tiene como desventaja que los resultados pueden ser de peor calidad debido a que no se mantiene la continuidad del espacio de soluciones. Esto puede ocasionar que algunas soluciones óptimas no sean nunca alcanzadas. En la Figura 5.18 se ilustra este problema.

Por el contrario, la vecindad basada en movimientos amplios tiene la ventaja de mantener la continuidad del espacio de soluciones. Sin embargo, tiene la desventaja de que no se tiene la certeza que la solución final sea correcta. Teóricamente, la solución final debería ser siempre correcta si la función de coste penaliza adecuadamente aquellas soluciones incorrectas y además el proceso de enfriamiento se ralentiza lo suficiente. La vecindad basada en movimientos restringidos es una opción que intenta evitar este problema soportando soluciones incorrectas que están siempre muy próximas a las soluciones correctas.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

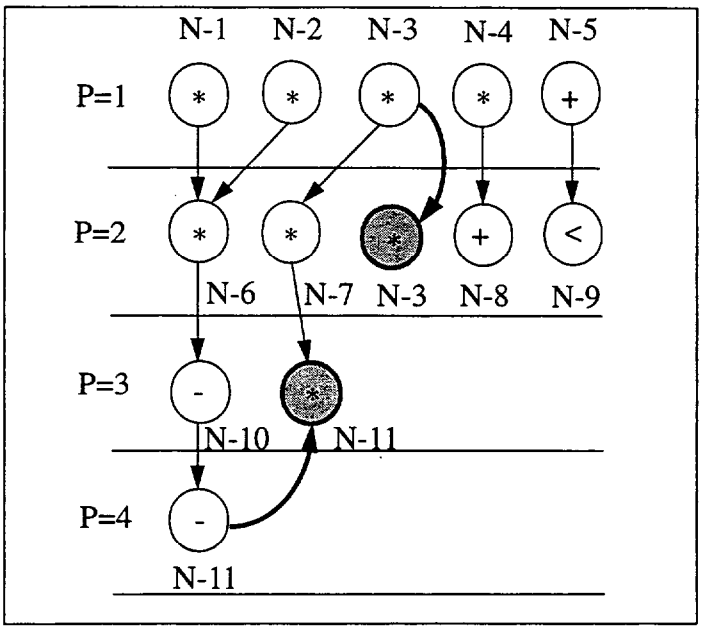


Figura 5.17. Algunos movimientos restringidos para obtener soluciones vecinas.

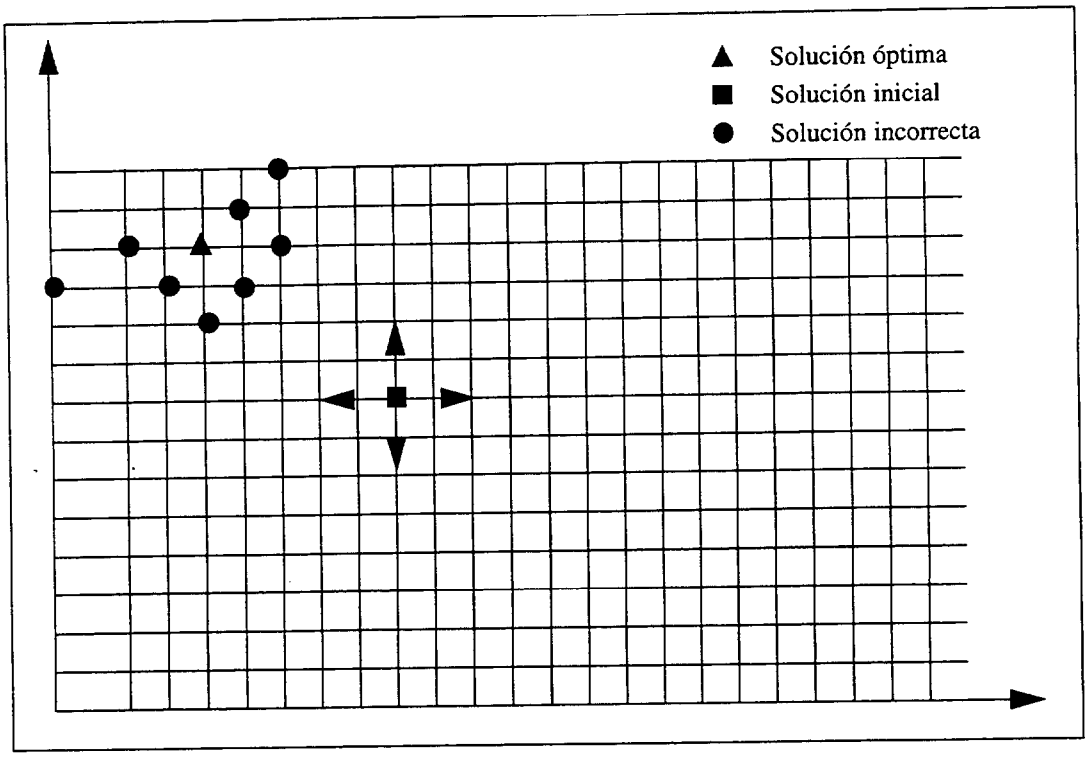


Figura 5.18. La solución óptima no puede ser alcanzada al no haber continuidad en el espacio de soluciones.

Por lo que a la función de coste se refiere, se ha utilizado una función sencilla que contabiliza el número de pasos de control que se utilizan para una planificación. Ahora bien, esta función de coste debe ser modificada en función del tipo de movimientos que una operación puede realizar para la obtención de una solución vecina. La modificación consiste en añadir algún tipo de penalización a la función de coste para que la solución final respete las dependencias del GA. El valor de la penalización debe ser mayor que el valor máximo que pueda tener la función de coste. La función de coste debe ser modificada para los siguientes tipos de movimientos:

- *Movimientos amplios*: deben añadirse una o varias penalizaciones por cada una de las dependencias del GA que haya sido violada. El número de penalizaciones para una dependencia viene dado por la diferencia que existe entre los pasos de control en que se han planificado las operaciones origen y destino de dicha dependencia.
- *Movimientos restringidos*: debe añadirse una penalización por cada una de las dependencias del GA que haya sido violada.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

5.6. Selección del algoritmo de planificación

En este apartado se presentan los experimentos que se han realizado para seleccionar el algoritmo más adecuado para la fase de estimaciones.

La selección del algoritmo de planificación debe tener en cuenta dos aspectos: la calidad de los resultados y el tiempo de ejecución. Por un lado, no resulta adecuado un algoritmo que proporciona buenos resultados si su tiempo de ejecución no es viable para el entorno de diseño. Por otro lado, un algoritmo muy rápido no es adecuado si sus resultados son muy pobres. Por tanto, se entiende que un método adecuado es aquel que proporciona los mejores resultados en un tiempo aceptable.

Los experimentos que se han realizado para seleccionar el algoritmo de planificación se han dividido en dos fases:

- *Primera fase:* en esta fase se han comparado los algoritmos de *simulated annealing* utilizando los tres tipos de vecindades mencionadas en el apartado anterior. De esta forma se obtiene el tipo de vecindad más adecuada.
- *Segunda fase:* en esta fase se ha comparado el algoritmo de planificación basado en listas (*list-based*) con el algoritmo de *simulated annealing*, utilizando este último el tipo de vecindad seleccionada en la primera fase.

Los algoritmos ASAP y ALAP fueron descartados porque producen planificaciones poco optimizadas. A continuación se presenta el diseño de los experimentos, y posteriormente se muestran los resultados y conclusiones más importantes.

5.6.1. Diseño de los experimentos

El algoritmo de planificación que se pretende seleccionar es utilizado en la fase de estimaciones para planificar los bloques básicos de los procesos del sistema bajo diseño. Las dependencias de datos de estos bloques básicos están representadas mediante un grafo acíclico. Para que los resultados de los experimentos sean significativos debe seleccionarse un amplio espectro de grafos acíclicos, es decir, los grafos deben representar a la mayoría de los bloques básicos que se pueden encontrar en un proceso.

Para realizar los experimentos se han definido 28 clases diferentes de grafos acíclicos en función del número de operaciones (tamaño de los bloques básicos) y del porcentaje de flancos de los mismos (número de dependencias). Para cada una de estas clases se han generado 10 grafos de forma aleatoria. Los valores que se han utilizado para definir las clases son los siguientes:

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

- *Número de operaciones*: **4, 8, 12, 16, 20, 32 y 64** operaciones. Se ha seleccionado este rango de operaciones teniendo en cuenta el número de operaciones que normalmente tienen los bloques básicos.
- *Porcentaje de flancos*: **10%, 25%, 50% y 75%** de flancos respecto del número máximo de flancos posible. De esta forma se tienen en cuenta desde bloques básicos con muy pocas dependencias (10%) hasta bloques con muchas dependencias (75%).

Los algoritmos desarrollados para estos experimentos se han codificado en el lenguaje C y se han ejecutado bajo el sistema operativo Linux en un computador personal (PC) con las siguientes características:

- Procesador: Pentium 200 MHz.
- Memoria RAM: 64 Mbytes.

A continuación se muestran los resultados y conclusiones para las dos fases de experimentos.

5.6.2. Primera fase: selección de la vecindad

El objetivo de esta primera fase de experimentos es comparar los algoritmos de *simulated annealing* utilizando las tres clases de vecindades (movimientos correctos, amplios y restringidos). De esta forma se obtiene la vecindad que proporciona mejores resultados.

En esta fase de los experimentos se ha utilizado un límite de uno y dos recursos para cada operador. Los motivos son los siguientes: en primer lugar, un número elevado de recursos no permite comparar la calidad de los resultados correctamente, ya que existen recursos suficientes. Sin embargo, un número escaso de recursos permite comprobar si un algoritmo los utiliza de forma más eficiente que otro. En segundo lugar, en la implementación de un diseño real se utiliza pocas veces un número superior de unidades funcionales para cada operador.

A continuación se comentan los resultados obtenidos, aunque primeramente deben realizarse dos puntualizaciones: en primer lugar, se ha comprobado que el algoritmo que utiliza una vecindad de movimientos amplios no siempre proporciona una solución correcta. Por lo tanto, este tipo de vecindad ha sido descartada y solamente se comentan los resultados obtenidos para los algoritmos que utilizan una vecindad basada en movimientos correctos y en movimientos restringidos. En segundo lugar, aunque se han realizado experimentos con diversos valores del parámetro α de la función de

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004.

enfriamiento de la temperatura (α= 0.99, 0.94, 0.89 y 0.84), solamente se muestran los resultados obtenidos para α=0.99, ya que se obtienen los mejores resultados. No obstante, en los Apéndices B y C se puede observar el resto de experimentos.

Por lo que al tiempo de ejecución respecta, en las Figuras 5.19 y 5.20 se observa que el algoritmo que utiliza una vecindad de movimientos correctos consume menos tiempo de ejecución que el algoritmo de movimientos restringidos. Resultados similares se presentan en [Ern93], trabajo en el que se utilizan penalizaciones en la función de coste.

Por lo que a la calidad de los resultados respecta, en las Figuras 5.21, 5.22, 5.23 y 5.24 se observa que el número de pasos de control obtenido por ambos algoritmos es prácticamente igual, independientemente del número de nodos de los grafos y del porcentaje de flancos de los mismos.

Conclusiones

La conclusión de esta primera fase de experimentos es la siguiente: dado que ambos tipos de vecindades proporcionan resultados de igual calidad, la vecindad seleccionada debe ser aquella que consume un menor tiempo de ejecución, es decir, la vecindad de movimientos correctos.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

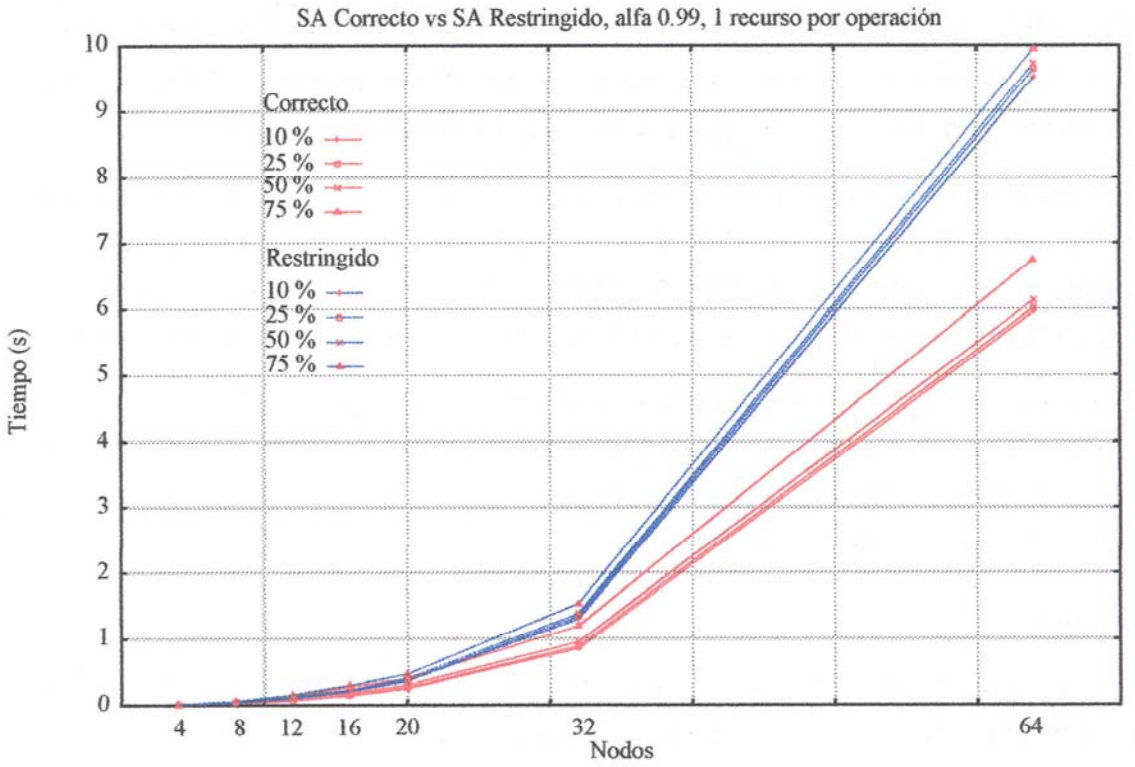


Figura 5.19. Tiempo de ejecución para 1 recurso por operación.

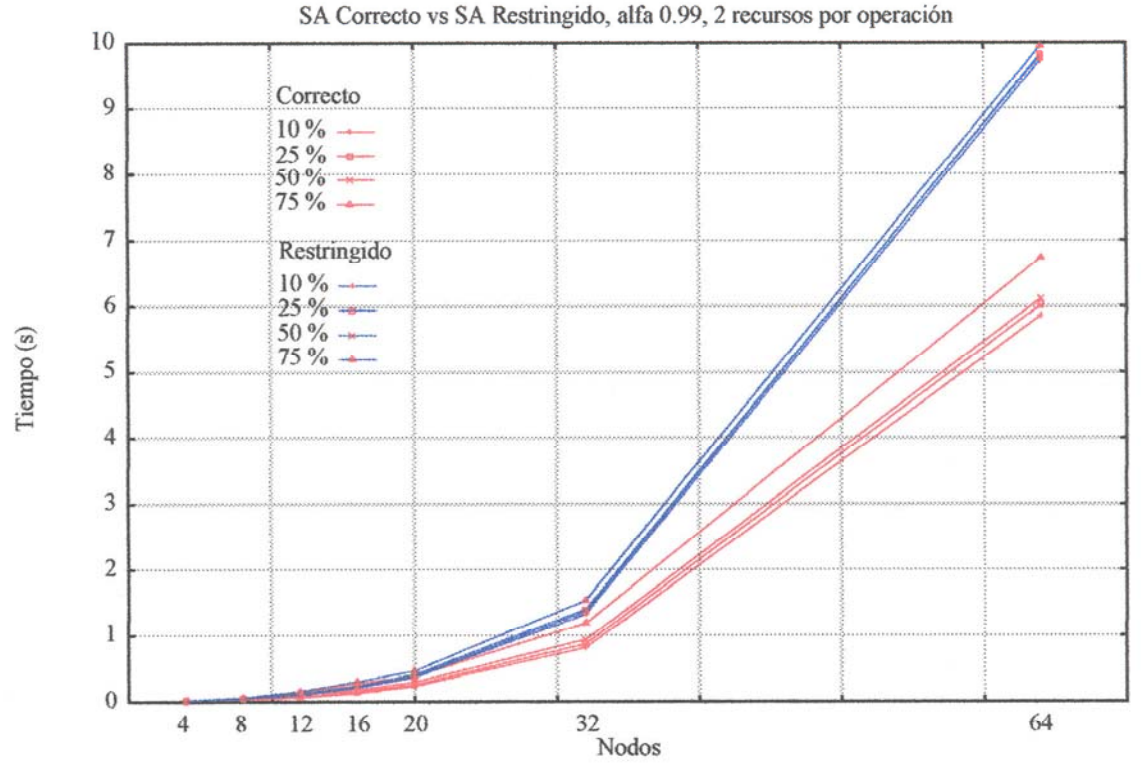


Figura 5.20. Tiempo de ejecución para 2 recursos por operación.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

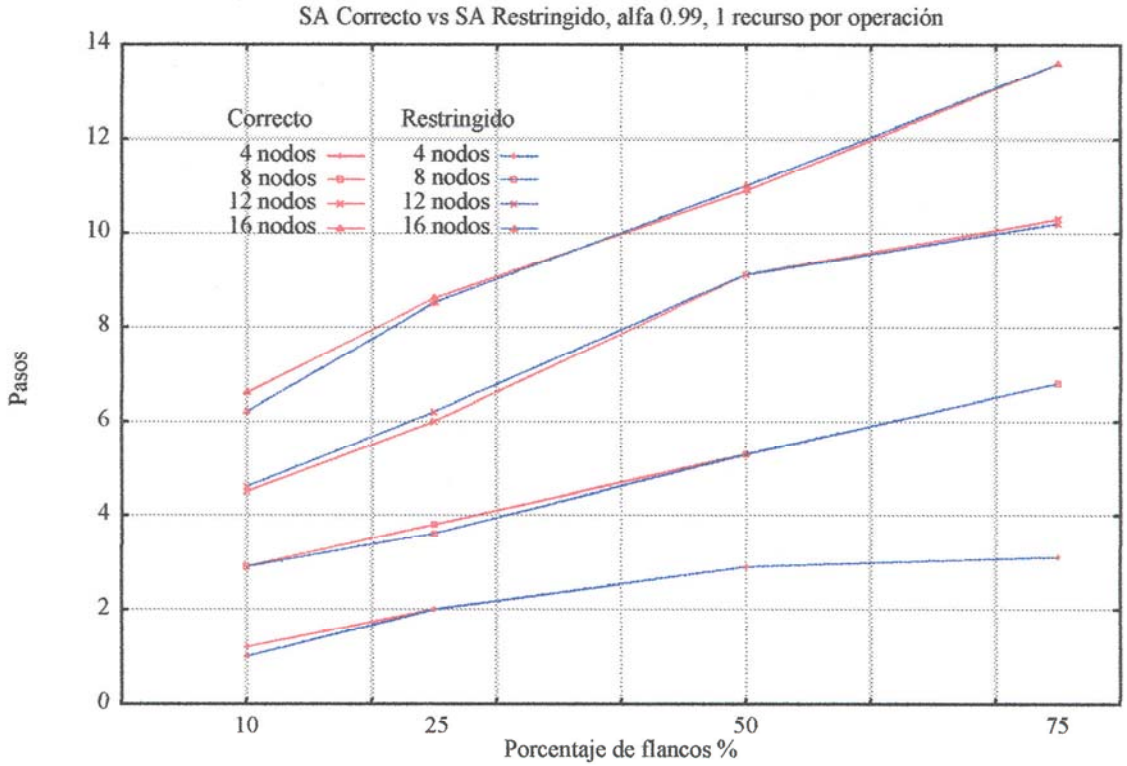


Figura 5.21. Pasos de control para 4-16 nodos (1 recurso por operación).

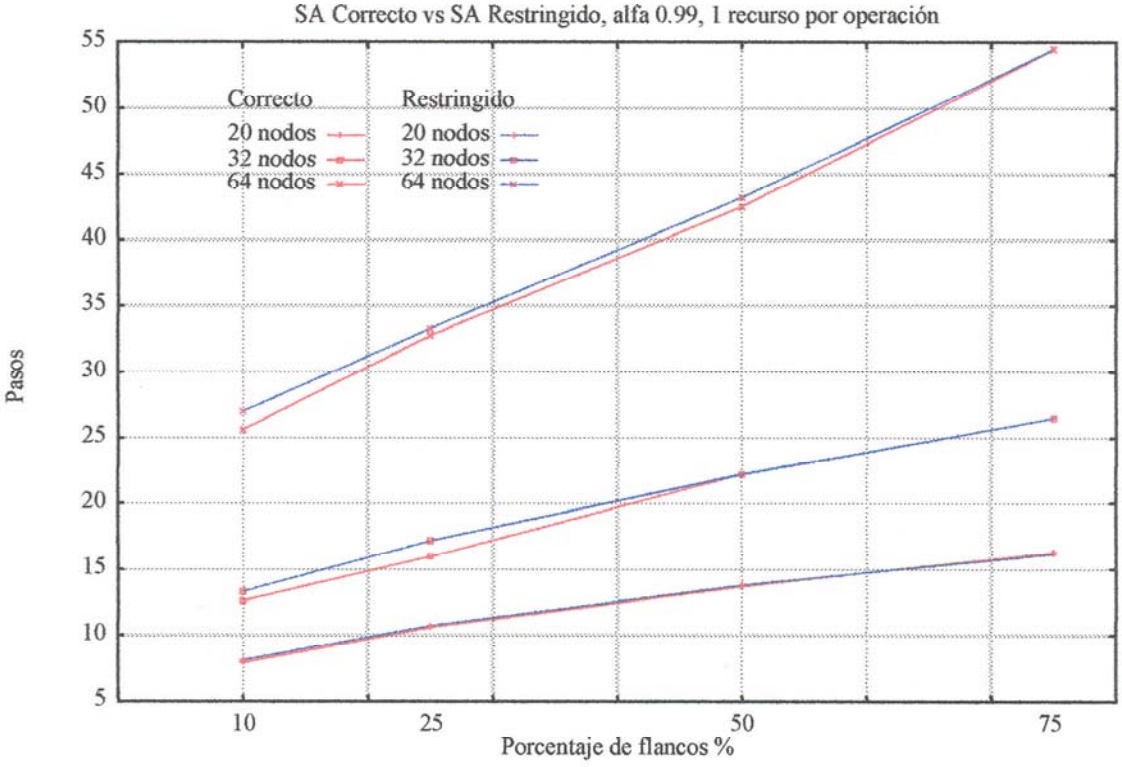


Figura 5.22. Pasos de control para 20-64 nodos (1 recurso por operación).

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

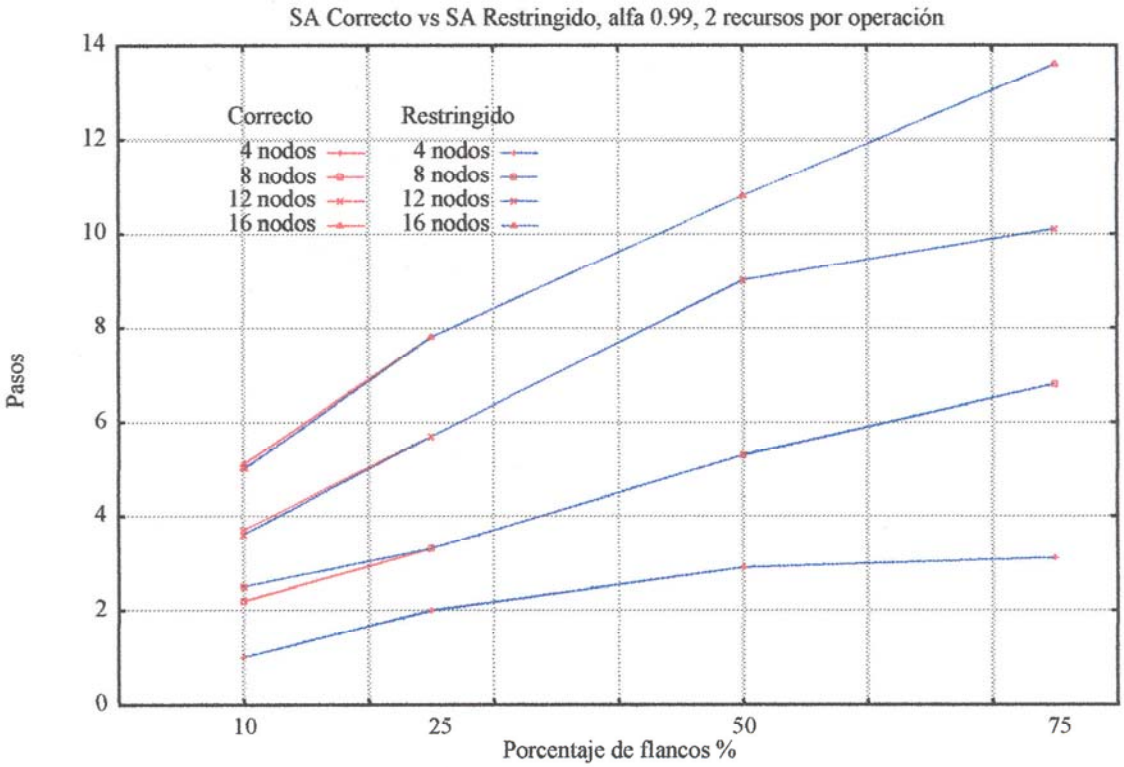


Figura 5.23. Pasos de control para 4-16 nodos (2 recursos por operación).

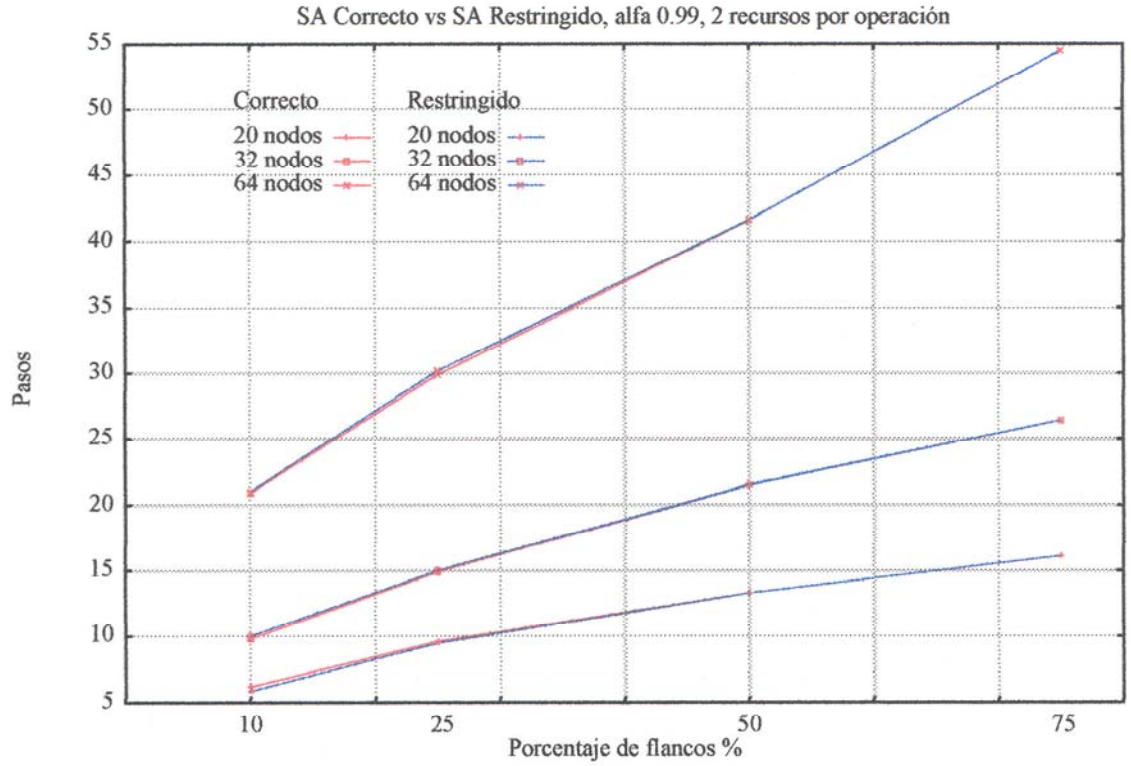


Figura 5.24. Pasos de control para 20-64 nodos (2 recursos por operación).

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

5.6.3. Segunda fase: selección del algoritmo

El objetivo de esta segunda fase de experimentos es comparar el algoritmo de planificación basado en listas (*list-based*) con el algoritmo de *simulated annealing*, utilizando este último la vecindad de movimientos correctos. Esta comparación nos permite decidir que algoritmo es el mejor para la fase de estimaciones de GACSYS.

En esta fase de los experimentos se ha utilizado un límite de uno, dos y tres recursos para cada operador. Los motivos son los mismos que los expuestos en la primera fase de experimentos.

Aunque se han realizado experimentos con diversos valores del parámetro α de la función de enfriamiento de la temperatura (α= 0.99, 0.94, 0.89 y 0.84), solamente se muestran los resultados obtenidos para α=0.99, ya que se obtienen los mejores resultados. No obstante, en los Apéndices B y C se puede observar el resto de experimentos.

A continuación se comentan los resultados obtenidos. En las Figuras 5.25, 5.26, 5.27, 5.28, 5.29 y 5.30 se observa que el tiempo de ejecución del algoritmo de *simulated annealing* es notablemente mayor que el tiempo del algoritmo *list-based*. Esta diferencia se incrementa conforme crece el número de nodos. En concreto, existe una diferencia entre dos y tres órdenes de magnitud.

Por lo que al número de pasos de control respecta (calidad de los resultados), de las Figuras 5.31, 5.32, 5.33, 5.34, 5.35 y 5.36 se extrae la siguiente información:

- Para grafos con muchas dependencias (50% y 75% de flancos), ambos algoritmos obtienen el mismo número de pasos de control.
- Para grafos con pocas dependencias (10% y 25% de flancos), el algoritmo *list-based* obtiene resultados mejores que el *simulated annealing*. En concreto, para grafos con un 10% de flancos, los resultados son entre un 10% y un 20% mejores. Para grafos con un 25% de flancos, los resultados son entre un 5% y un 10% mejores.

Conclusiones

El algoritmo seleccionado para la fase de estimaciones del entorno de codiseño GACSYS es el *list-based*, ya que obtiene resultados mejores, o iguales, que el *simulated annealing*, y además su tiempo de ejecución es muy inferior al que necesita este último.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

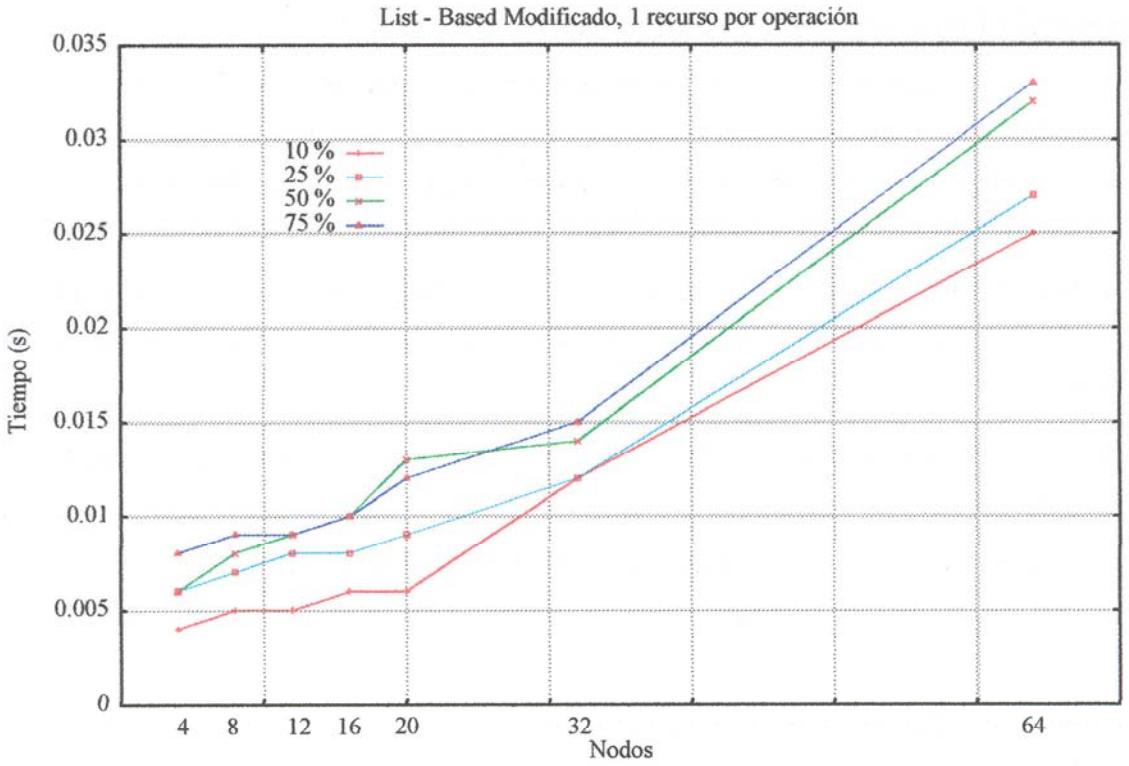


Figura 5.25. *List-Based.* Tiempo de ejecución (1 recurso por operación).

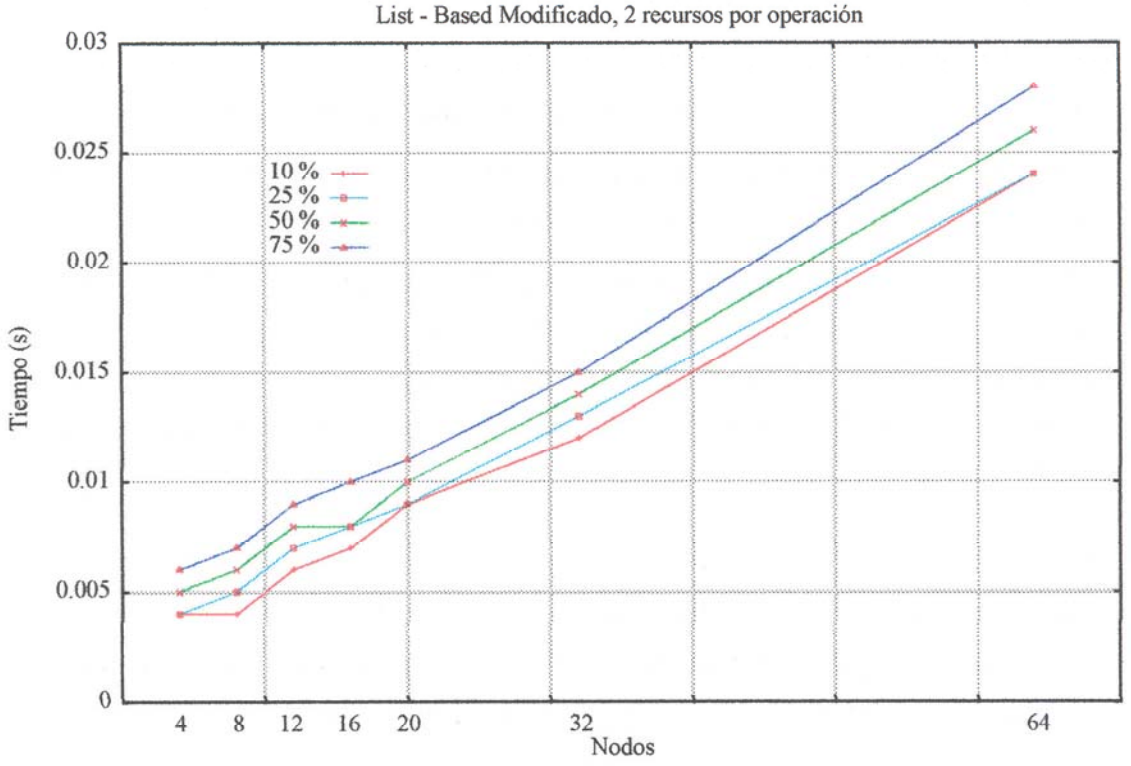


Figura 5.26. *List-Based.* Tiempo de ejecución (2 recursos por operación).

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

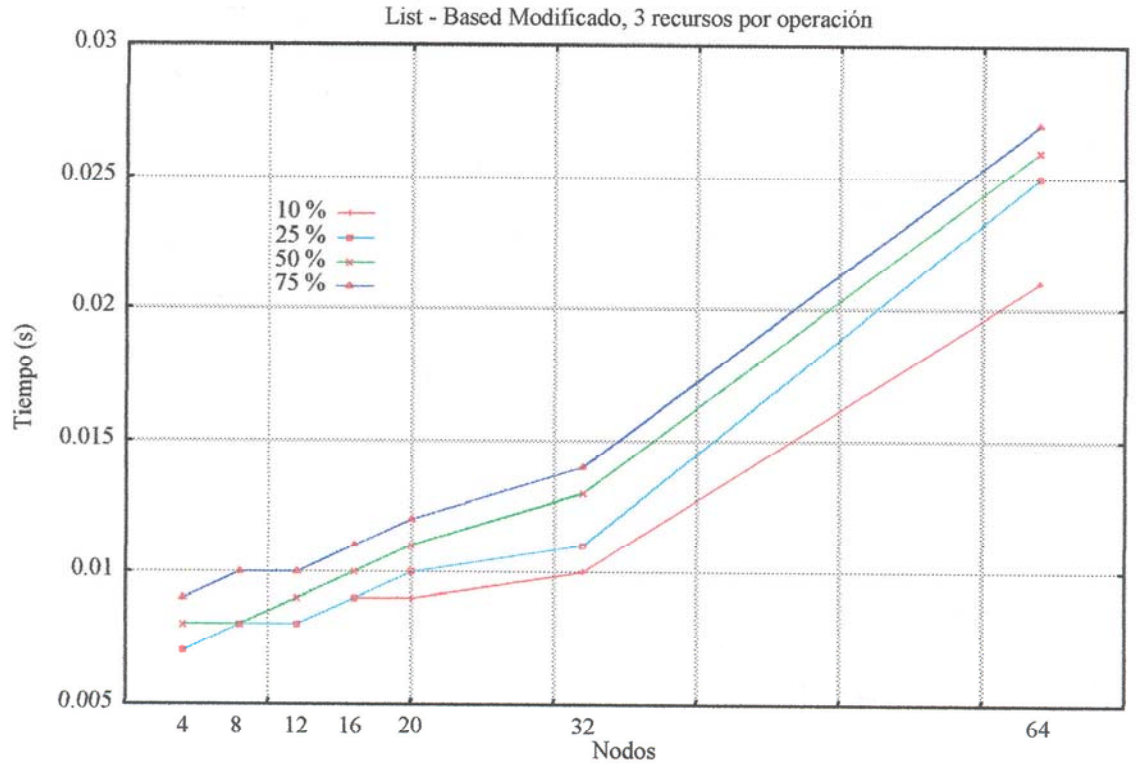


Figura 5.27. *List-Based.* Tiempo de ejecución (3 recursos por operación).

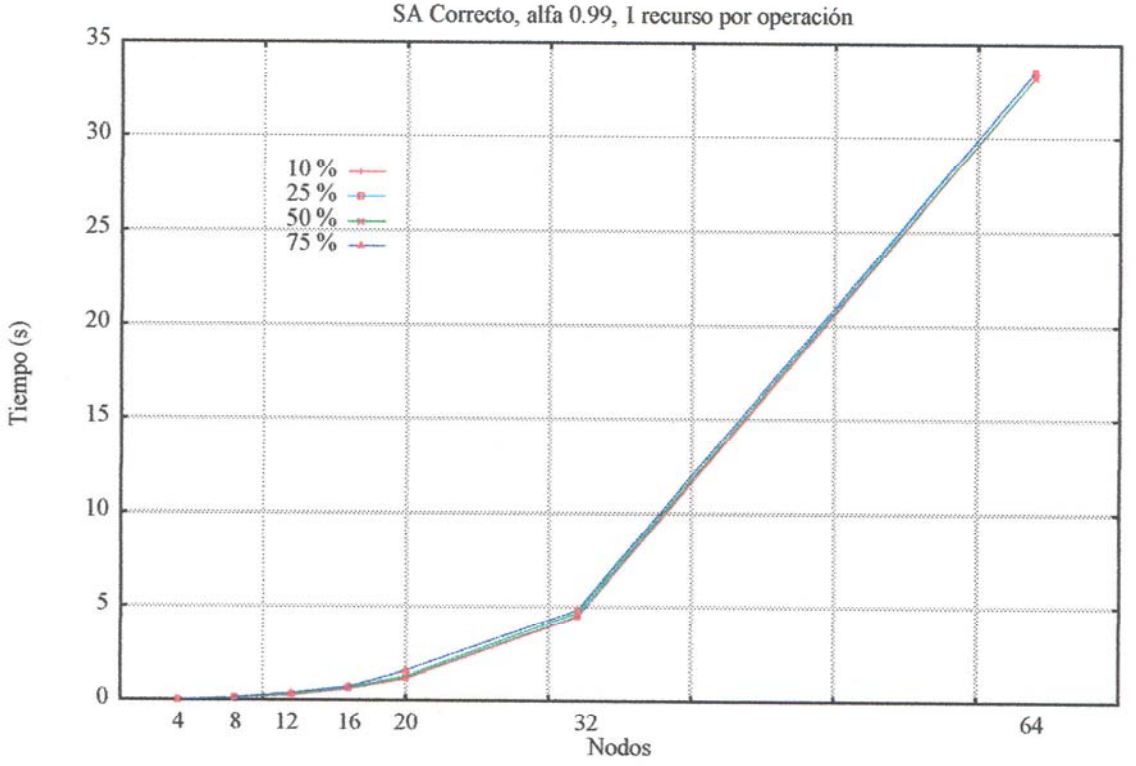


Figura 5.28. *Simulated Annealing.* Tiempo de ejecución (1 recurso por operación).

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

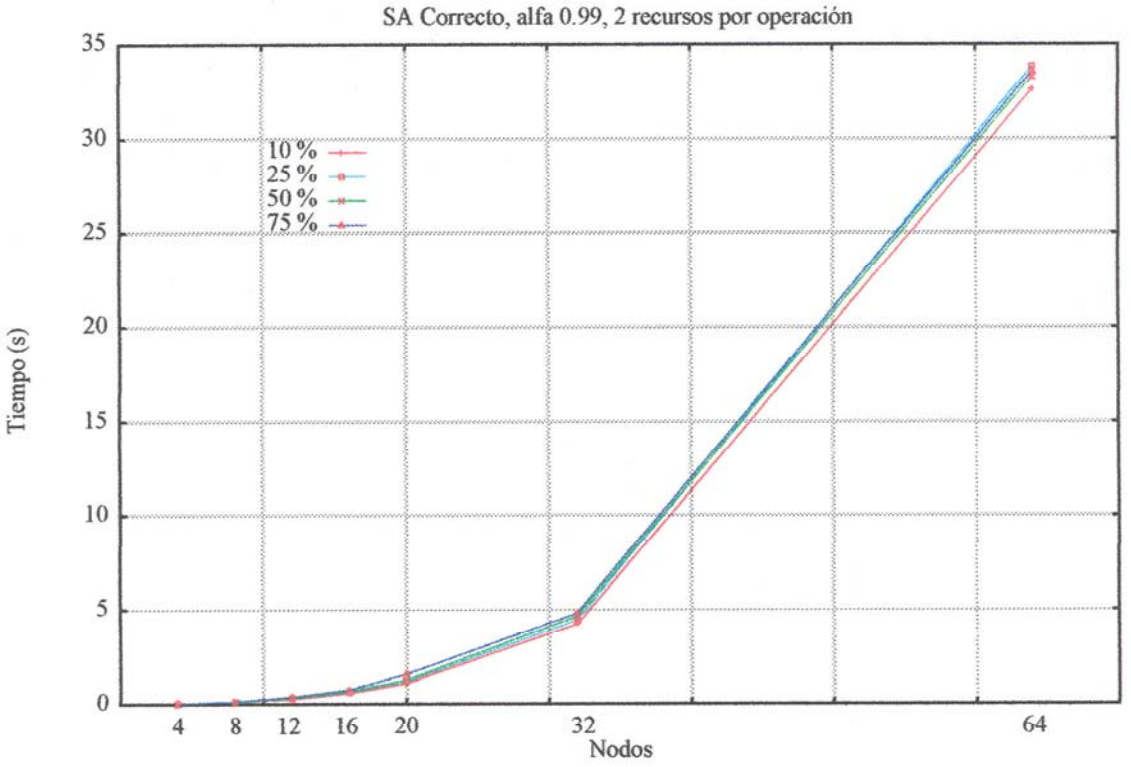


Figura 5.29. *Simulated Annealing.* Tiempo de ejecución (2 recursos por operación).

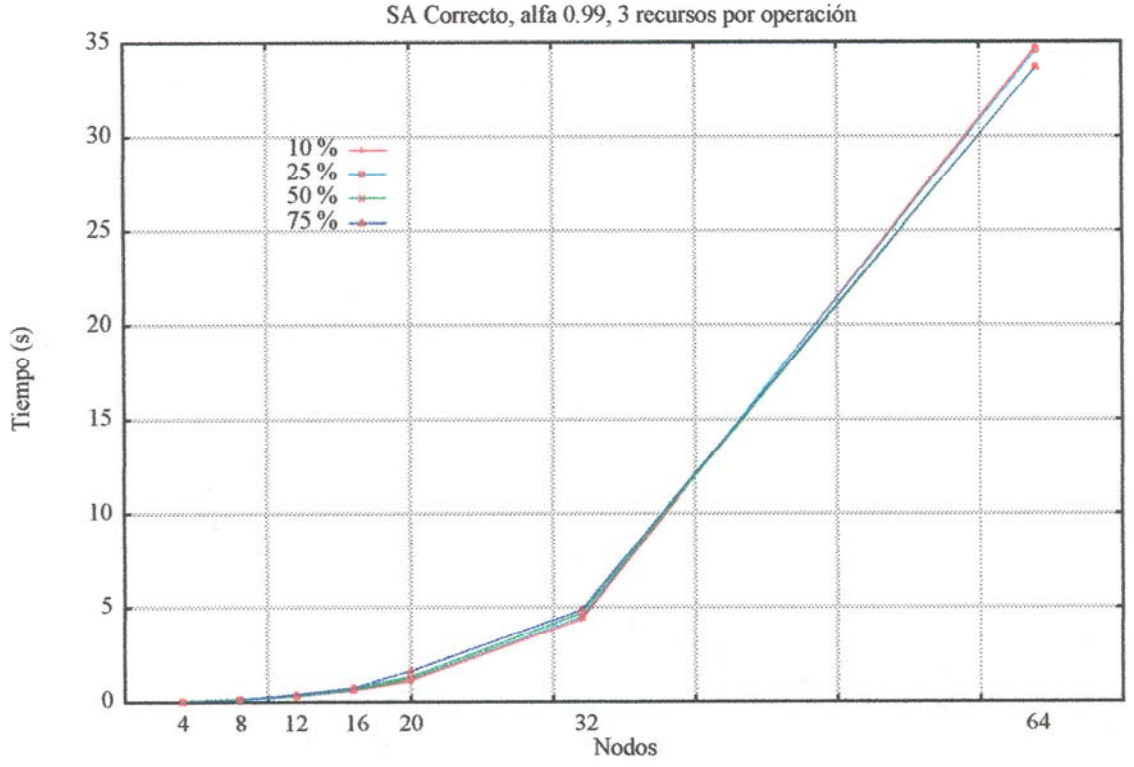


Figura 5.30. *Simulated Annealing.* Tiempo de ejecución (3 recursos por operación).

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

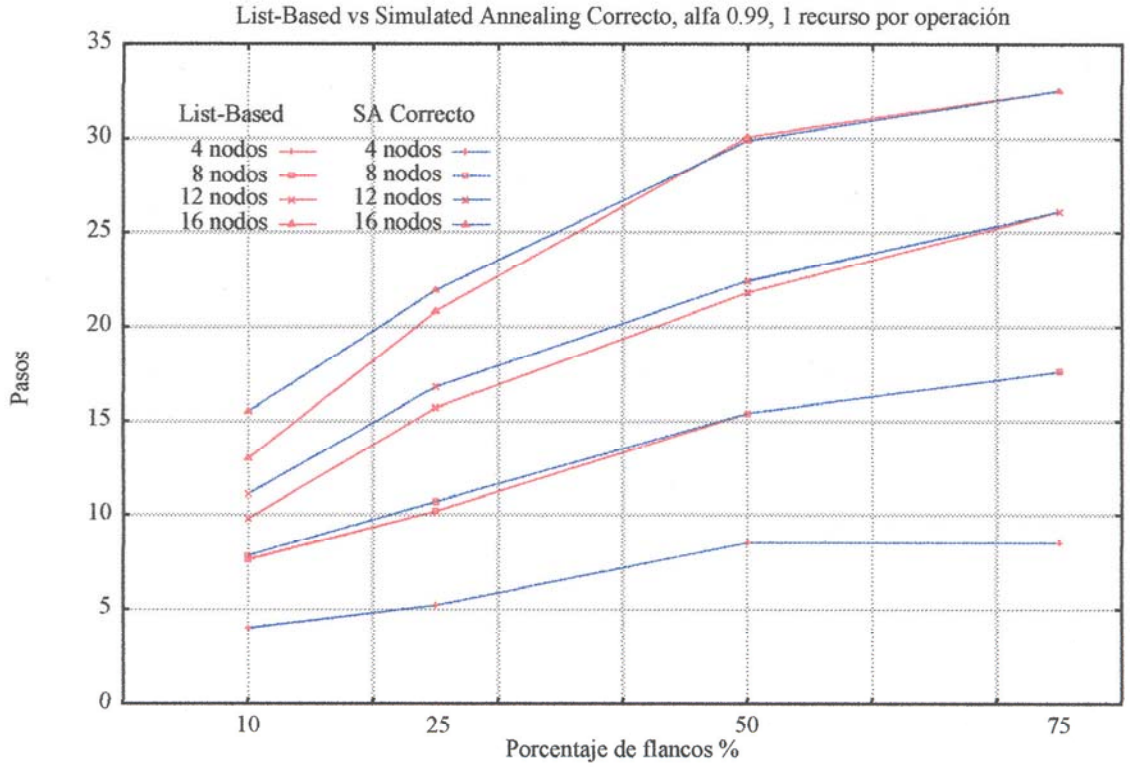


Figura 5.31. Pasos de control para 4-16 nodos (1 recurso por operación).

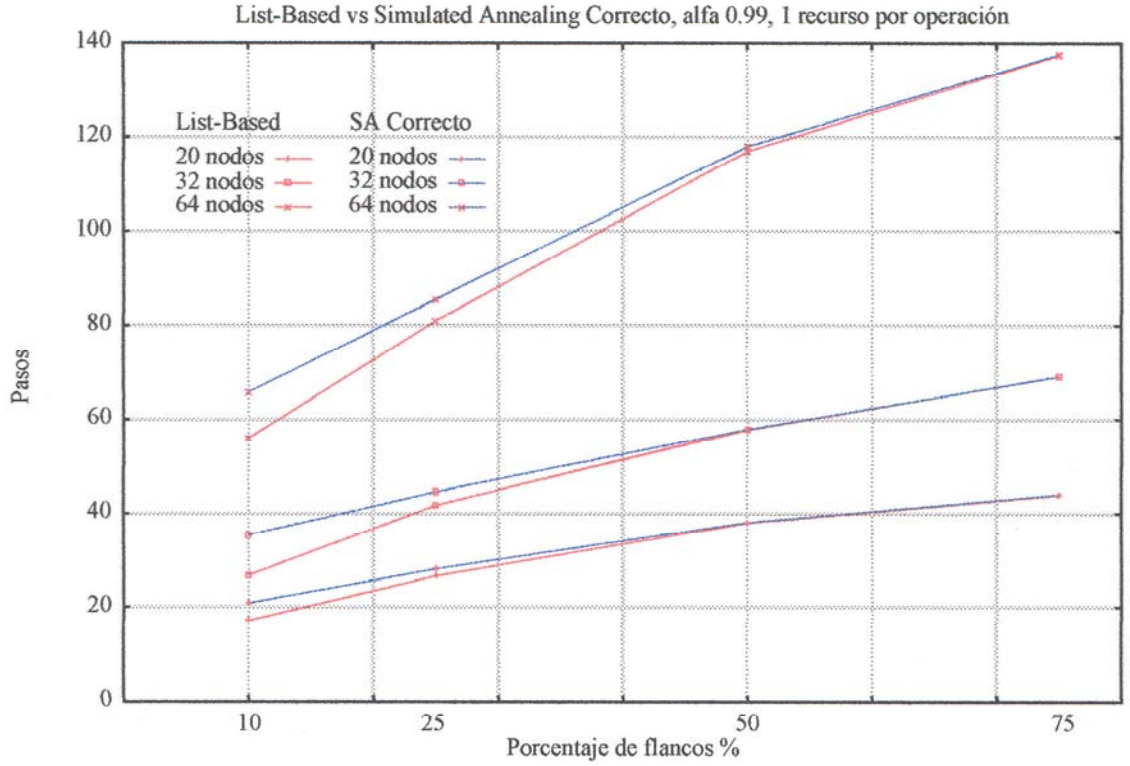


Figura 5.32. Pasos de control para 20-64 nodos (1 recurso por operación).

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

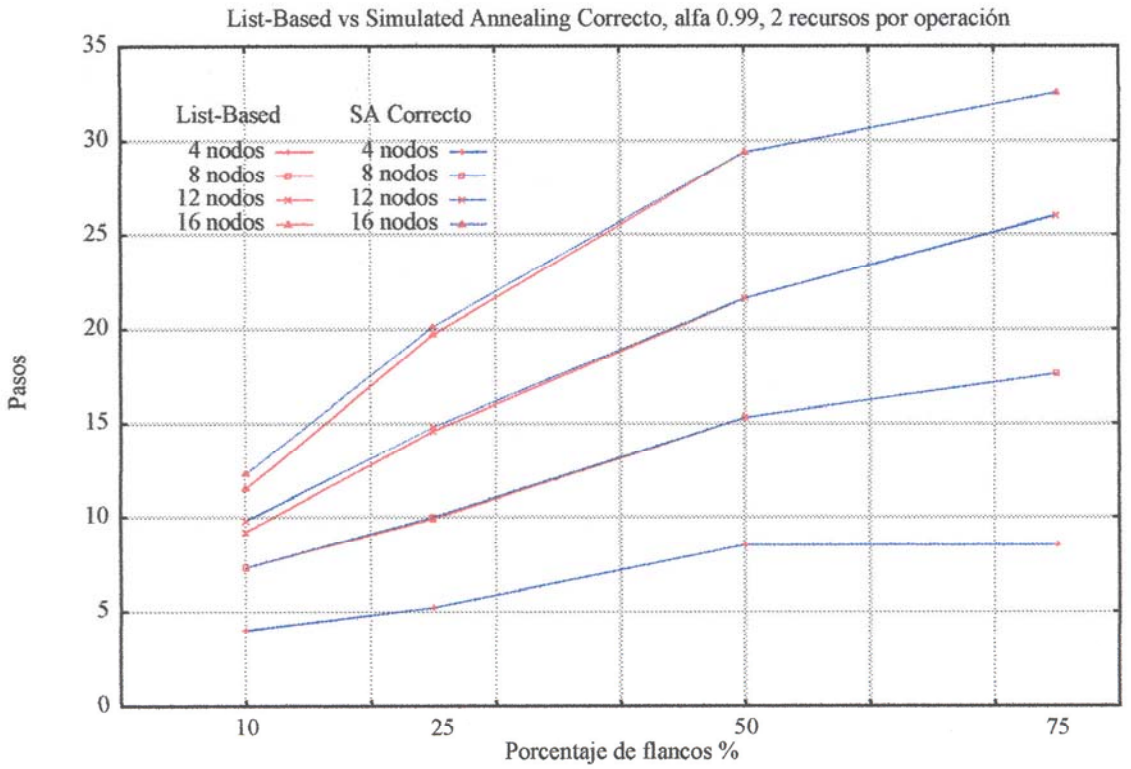


Figura 5.33. Pasos de control para 4-16 nodos (2 recursos por operación).

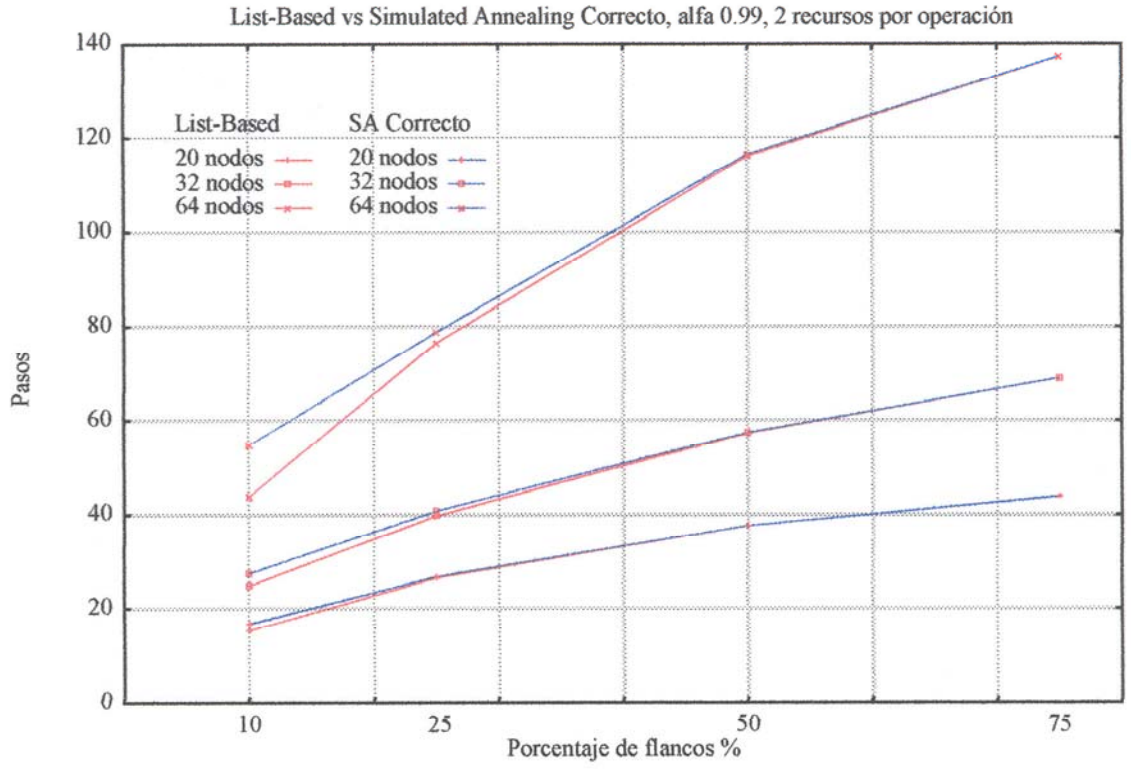


Figura 5.34. Pasos de control para 20-64 nodos (2 recursos por operación).

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

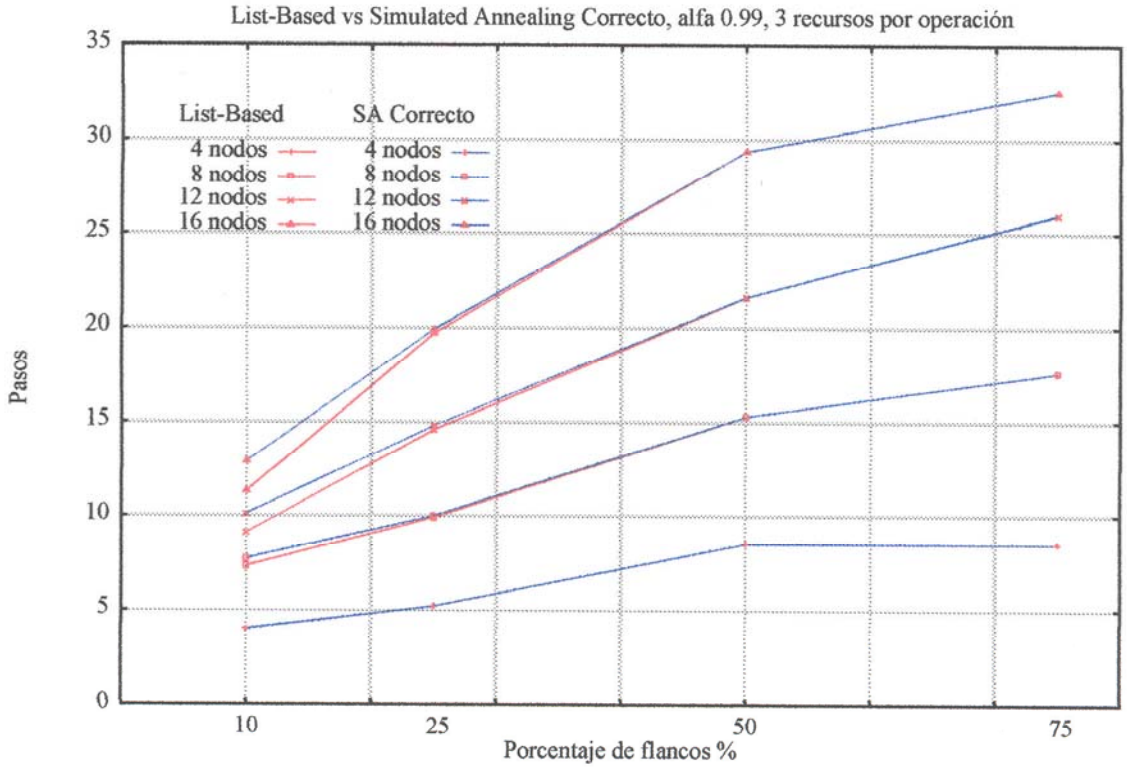


Figura 5.35. Pasos de control para 4-16 nodos (3 recursos por operación).

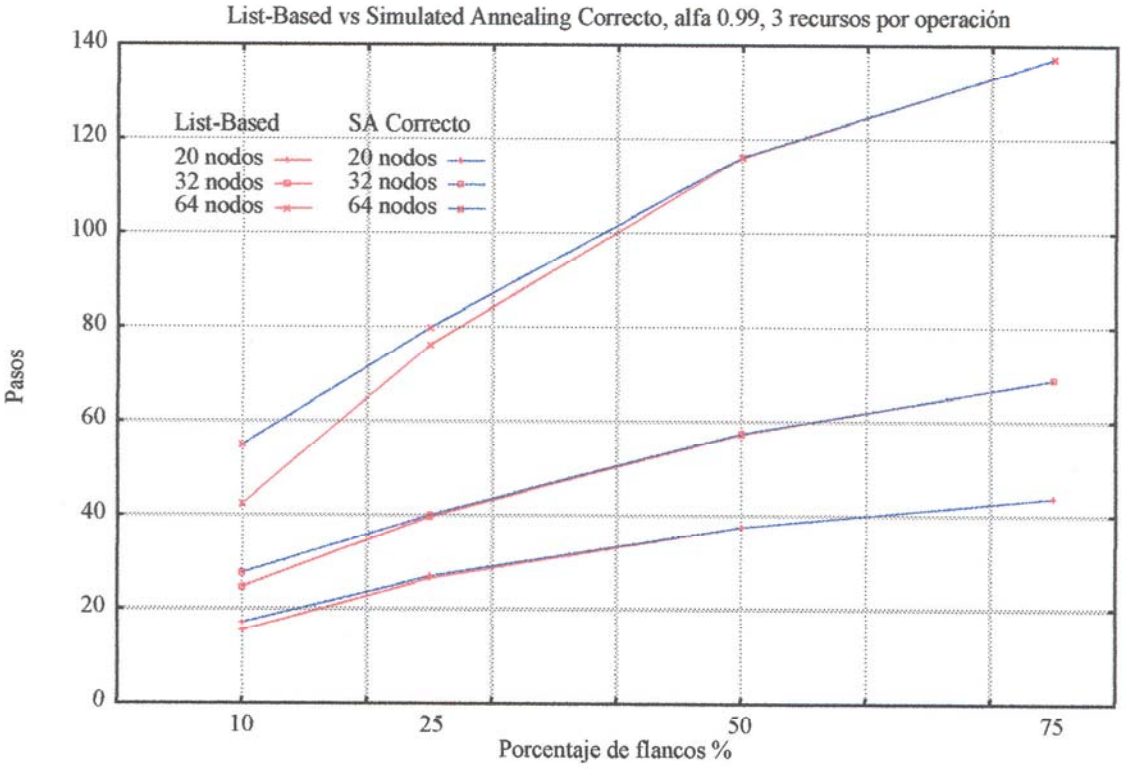


Figura 5.36. Pasos de control para 20-64 nodos (3 recursos por operación).

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

Capítulo 6

Particionado Hardware/Software

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital. 2004

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

6.1. Introducción

En este capítulo se describe la etapa de *particionado Hw/Sw* que ha sido desarrollada en este trabajo para el entorno de codiseño GACSYS. El objetivo de dicha etapa es encontrar una partición Hw/Sw que optimice unos determinados objetivos y satisfaga unas determinadas restricciones. Los objetivos y restricciones están relacionados directamente con el tiempo de ejecución del sistema, el área necesaria para su implementación, y la potencia consumida por el mismo.

En la Figura 6.1 se muestra un esquema de la herramienta desarrollada. Dado el grafo de procesos de la especificación del sistema, dadas las estimaciones Hw y Sw de los diferentes parámetros de los procesos (tiempo de ejecución, área y potencia), y dados los objetivos y restricciones requeridos por el diseñador, la herramienta de particionado distribuye los procesos en dos clases, hardware y software, de tal forma que se optimicen los objetivos y se satisfagan las restricciones.

En general, la aproximación al problema del particionado Hw/Sw utilizada en la mayoría de los trabajos publicados hasta ahora contempla la optimización del problema considerando que los procesos no se ejecutan de forma segmentada. Una de las principales aportaciones de nuestro método es que permite al diseñador la posibilidad de implementar el grafo de procesos de forma segmentada, es decir, mediante un *pipeline* de procesos. Esta aportación permite al diseñador la realización de compromisos entre el coste de las soluciones y la latencia necesaria para ejecutar el grafo de procesos. De esta forma, el diseñador dispone de un espacio de diseño más amplio.

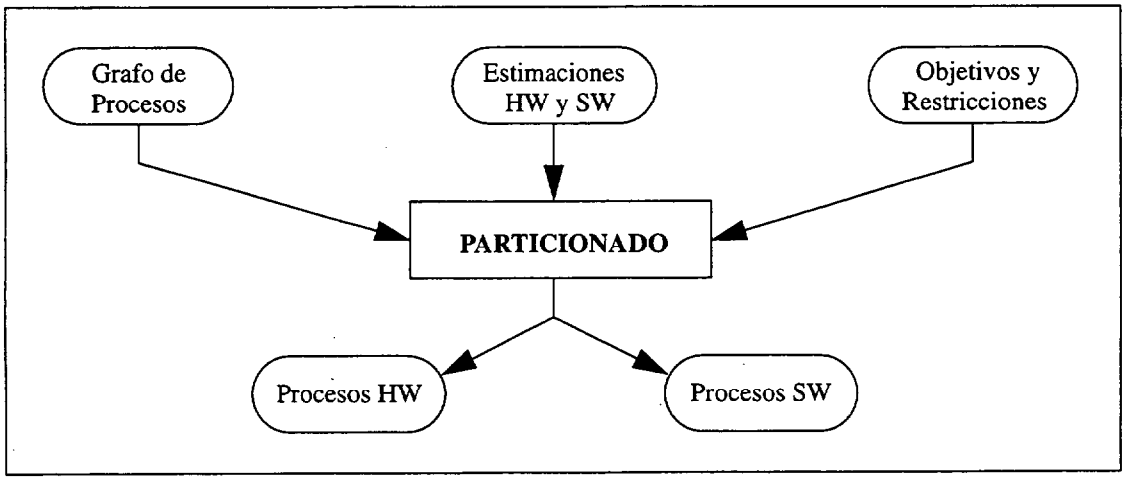


Figura 6.1. Herramienta de particionado Hw/Sw del entorno de codiseño GACSYS.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

El particionado Hw/Sw es un problema de optimización combinatoria NP-completo [Gar79][Hoc97]. Por un lado, cada proceso de la especificación puede ser implementado mediante un amplio número de configuraciones Hw y Sw. Por otro lado, el número de procesos de un sistema es normalmente elevado. Por ejemplo, un sistema con 20 procesos y 100 configuraciones Hw y Sw para cada proceso genera un espacio de diseño del orden de 10^{40} posibles soluciones. Además, la herramienta que se ha diseñado permite al diseñador utilizar la segmentación a nivel de procesos, por lo que el grado de complejidad del problema aumenta aún más. Por tanto, en general no se pueden utilizar métodos que obtienen soluciones óptimas debido a que el tiempo de computación necesario es tan elevado que resultan inviables.

La alternativa viable para solucionar el problema del particionado Hw/Sw es la utilización de algoritmos heurísticos. Estos algoritmos permiten realizar compromisos entre la calidad de las soluciones y el tiempo de cómputo necesario para obtenerlas. En definitiva, se trata de obtener soluciones *satisfactorias* utilizando tiempos de ejecución *razonables*. Uno de los algoritmos heurísticos más utilizados para problemas de optimización combinatoria es el denominado *simulated annealing* [Met53][Kir83] [Dow95]. En el capítulo anterior se ha realizado una descripción de las características del mismo.

Los algoritmos de particionado Hw/Sw que se han desarrollado para la herramienta que se presenta en este capítulo se basan en el algoritmo de *simulated annealing*. El motivo principal para seleccionar este algoritmo es que los estudios experimentales realizados en [Vah95b][Vah96] demuestran que el *simulated annealing* obtiene mejores resultados que otros métodos heurísticos tales como los basados en *clustering* [Joh67] o en migración de grupos [Ker70]. Los algoritmos que se han desarrollado son los siguientes:

- *Métodos A y A/N*: se utilizan cuando el diseñador especifica una latencia unitaria, es decir, el grafo de procesos se implementa sin utilizar la segmentación a nivel de procesos.
- *Métodos B y B/N*: se utilizan cuando el diseñador desea implementar el grafo de procesos utilizando la segmentación a nivel de procesos.
- *Métodos C y C/N*: estos métodos se han desarrollado para evaluar la calidad de los resultados de los métodos heurísticos anteriores. Inicialmente se desarrolló un método exhaustivo para obtener soluciones óptimas. Sin embargo, el tiempo de ejecución del mismo era tan elevado que no nos permitió obtener resultados para los sistemas que se han utilizado para comprobar la

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

herramienta. Para resolver el problema se ha optado por un método semi-exhaustivo. Por una parte, se trata de un método exhaustivo porque analiza todas las planificaciones posibles del *pipeline* de procesos. Por otra parte, se trata de un método heurístico porque aplica los métodos *Métodos A y A/N* a cada una de estas planificaciones. En general, como se mostrará en el capítulo de resultados experimentales, este método no es aplicable debido al excesivo tiempo de computación que requiere. Sin embargo, es muy interesante su desarrollo para comparar sus resultados con los obtenidos mediante los métodos heurísticos.

Este capítulo está estructurado de la siguiente forma: en el apartado 2, se describe la información de entrada y salida de la herramienta de particionado que se ha desarrollado. En el apartado 3, se presenta la segmentación a nivel de procesos. En el apartado 4, se describe la función de coste que se utiliza en los algoritmos de optimización de la herramienta. Finalmente, en el apartado 5 se describen los algoritmos de particionado que se han desarrollado.

6.2. Información de entrada y salida

En este apartado se describe la información de entrada que utiliza la herramienta de particionado, así como la información de salida que proporciona.

La información de entrada que utiliza la herramienta es la siguiente:

- *Grafo acíclico de procesos.* Cada nodo del grafo representa a un proceso de la especificación del sistema, y cada flanco del mismo indica el flujo de datos que existe entre los procesos.
- *Fichero de estimaciones.* En este fichero se encuentran todas las estimaciones relativas al tiempo de ejecución, área y potencia de cada una de las configuraciones Hw y Sw de cada proceso. Este fichero se genera en la fase de estimaciones, que se ha explicado en el capítulo anterior de esta memoria.
- *Tiempo de ejecución.* Tiempo máximo especificado por el diseñador para la ejecución del grafo de procesos. Durante este tiempo el sistema debe procesar una muestra o un conjunto de muestras.
- *Latencia.* Indica el número de fases que desea el diseñador para el *pipeline* de procesos. La duración de cada fase está restringida al tiempo de ejecución especificado anteriormente.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

- *Objetivos.* En primer lugar, debe optimizarse el tiempo de ejecución de cada fase del *pipeline*, es decir, el tiempo de ejecución de cada fase debe aproximarse al tiempo máximo permitido. En segundo lugar, debe optimizarse el área o la potencia. En este caso, el diseñador puede plantearse dos objetivos distintos:
  - Acotar el área y minimizar la potencia.
  - Acotar la potencia y minimizar el área.
- *Restricciones.* En primer lugar, el tiempo de ejecución de cada fase del *pipeline* no debe superar el tiempo máximo especificado por el diseñador. En segundo lugar, el área o la potencia, según sea el objetivo a minimizar, no debe superar la cota máxima especificada por el diseñador.

Como resultado de la fase de particionado se genera la siguiente información:

- *Configuraciones Hw y Sw.* Indican la configuración seleccionada para cada uno de los procesos del sistema.
- *Planificación del pipeline.* Especifica la fase del *pipeline* en la que ha sido planificado cada proceso.
- *Planificación virtual.* Indica la planificación de los procesos dentro de cada fase del *pipeline*.
- *Área estimada que ocupa el diseño.*
- *Potencia estimada que consume el diseño.*

6.3. Segmentación a nivel de procesos

La aproximación al problema del particionado Hw/Sw utilizada en la mayoría de los trabajos publicados hasta ahora contempla la optimización del problema considerando que los procesos no se ejecutan de forma segmentada. Una de las principales aportaciones de nuestro método es que permite al diseñador la posibilidad de implementar el grafo de procesos de forma segmentada, es decir, mediante un *pipeline* de procesos. En la Figura 6.2.a se ilustra un grafo de procesos no segmentado, y en la Figura 6.2.b se ilustra una posible segmentación del mismo.

La segmentación del grafo de procesos proporciona al diseñador la posibilidad de obtener soluciones con menor coste, o bien obtener soluciones que sin el *pipeline* de procesos no serían posibles.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

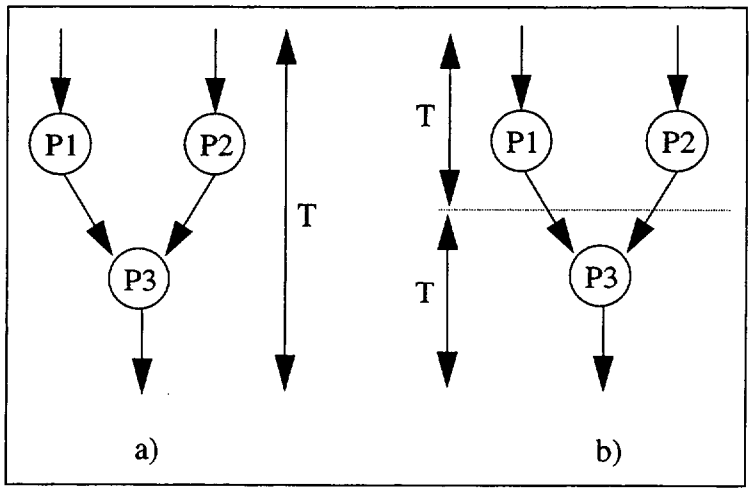


Figura 6.2. (a) Grafo de procesos no segmentado y (b) posible segmentación del grafo en dos fases.

En el primer caso, la segmentación del grafo de procesos permite que las restricciones temporales requeridas por el diseñador puedan verificarse con diseños menos costosos, ya que el tiempo de ejecución de los procesos puede relajarse. Por tanto, los procesos ocupan menos área y/o consumen menos potencia. Por ejemplo, en la Figura 6.2.b, los procesos tienen una exigencia de tiempo de ejecución claramente inferior a los procesos de la Figura 6.2.a.

En el segundo caso, la segmentación del grafo de procesos permite al diseñador obtener soluciones que de otra forma serían inviables. Imaginemos que el tiempo requerido para la ejecución de un grafo es muy exigente. Por tanto, los tiempos de ejecución de los procesos serían muy exigentes, y podría suceder que los mismos no pudiesen implementarse con las librerías disponibles. La segmentación del grafo proporciona una alternativa al diseñador para solucionar este problema.

En cualquier caso, el diseñador consigue los beneficios anteriores aumentando la latencia del sistema. Por tanto, se trata de una alternativa de la que el diseñador dispone pero que puede utilizar dependiendo de los requerimientos del sistema en cuanto a latencia. En definitiva, la segmentación a nivel de procesos proporciona al diseñador un espacio de diseño más amplio, y por tanto una mayor posibilidad para optimizar el particionado Hw/Sw del sistema.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

6.4. Diseño de la función de coste

Independientemente de los algoritmos de particionado que se utilicen, en cualquier problema de optimización combinatoria es necesario definir la función de coste que se pretende minimizar. La función de coste debe tener en cuenta aquellos parámetros que son objeto de la minimización. En nuestro caso son los siguientes:

- En primer lugar, debe optimizarse el tiempo de ejecución de cada fase del *pipeline*, es decir, el tiempo de ejecución de cada fase debe aproximarse al máximo permitido. En la Figura 6.3 se ilustran dos configuraciones diferentes para una determinada planificación del grafo de procesos. La configuración de la Figura 6.3.a utiliza procesos más rápidos que la configuración de la Figura 6.3.b. Sin embargo, ambas configuraciones satisfacen el tiempo de ejecución. Por tanto, la configuración de la Figura 6.3.b es preferible debido a que los procesos consumen menos área y/o potencia.
- En segundo lugar, debe optimizarse el área o la potencia, según sea el objetivo especificado por el diseñador.

A continuación se explica cómo se obtienen los valores de estos parámetros que se pretende optimizar.

En primer lugar, la Expresión 6.1 evalúa el área consumida por todos los procesos ($Area_{Pi}$ representa el área ocupada por el proceso *i-ésimo*). En el caso de los procesos implementados en hardware se contabiliza el área de las unidades funcionales y de la

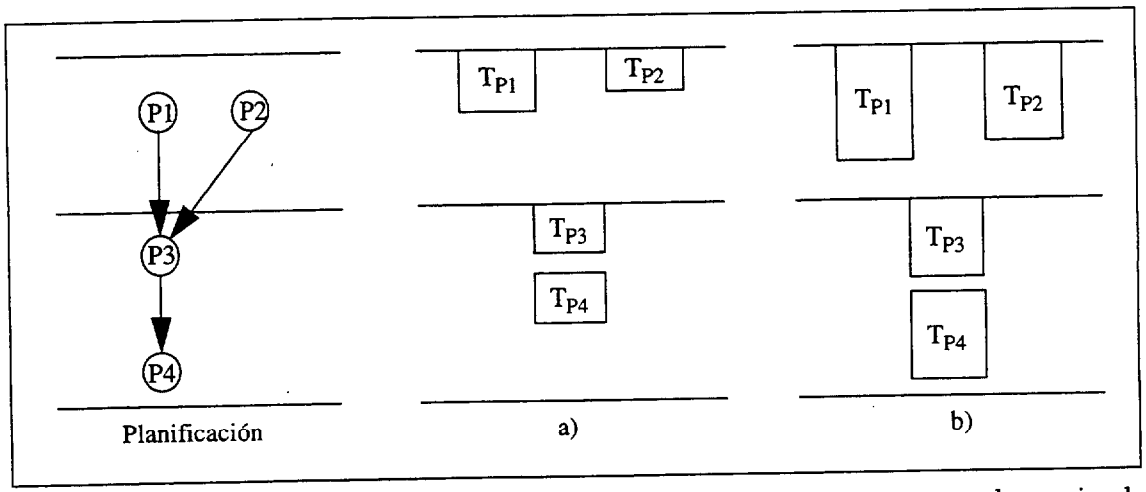


Figura 6.3. Ejemplo de dos configuraciones diferentes para una determinada planificación del *pipeline* de procesos. El alto de cada rectángulo indica el tiempo de ejecución (T_{Pi}) de cada proceso.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

$$Area = \sum_{i}^{Procesos} Area_{P_i}$$

Expresión 6.1. Área consumida por el diseño.

$$Energia = \sum_{i}^{Procesos} Energia_{P_i}$$

Expresión 6.2. Energía consumida por el diseño.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

memoria. En el caso de haber procesos implementados en software hay que añadir el área que ocupan el procesador y la memoria.

En segundo lugar, la Expresión 6.2 evalúa la energía consumida por todos los procesos ($Energia_{P_i}$ representa la energía consumida por el proceso *i-ésimo*).

En tercer lugar, la Expresión 6.3 evalúa el tiempo de ejecución no utilizado en cada fase del *pipeline*, es decir, la diferencia existente entre el tiempo máximo permitido ($T_{ejecución}$) para la ejecución de una fase y el tiempo de ejecución utilizado por los procesos en dicha fase (T_F).

Todos los datos que se necesitan para evaluar estas expresiones son conocidos, exceptuando el valor de T_F. Dada una determinada planificación del *pipeline* de procesos, y dada una configuración Hw o Sw para cada proceso, el tiempo de ejecución de una fase depende de la planificación de los procesos en dicha fase. Esta planificación interna a cada fase la hemos denominado planificación virtual.

$$Tiempo = \sum_{i}^{Fases} (T_{Ejecucion} - T_{Fi})$$

Expresión 6.3. Tiempo de ejecución no utilizado en las fases del *pipeline*.

Por tanto, el cálculo de T_F requiere de una planificación virtual concreta. En lugar de seleccionar una planificación virtual aleatoria, los algoritmos que se presentan posteriormente obtienen todas las planificaciones virtuales y seleccionan la mejor, es decir, aquella planificación virtual que satisface el tiempo máximo permitido para cada fase ($T_{ejecución}$), pero que al mismo tiempo se aproxima más a dicho tiempo. Se ha decidido realizar una búsqueda exhaustiva debido al número reducido de planificaciones que en general existe en cada fase, ya que el número de procesos en las mismas suele ser pequeño. En la Figura 6.4 se muestran las planificaciones virtuales para una determinada planificación del *pipeline* de procesos.

Dada una planificación virtual, el tiempo de ejecución de la fase *i-ésima* se obtiene mediante la Expresión 6.4.

En general, en cada paso de una planificación virtual pueden haber procesos implementados en hardware y software. Por tanto, el tiempo de ejecución de un paso virtual viene dado por el máximo valor de entre los dos siguientes:

- Por un lado, el tiempo de ejecución del proceso más lento de entre los implementados en hardware ($max\{T_{Hw}\}$).
- Por otro lado, el tiempo de ejecución de los procesos implementados en software. Como todos estos procesos utilizan un único procesador, dicho tiempo se obtiene sumando el tiempo de cada proceso (ΣT_{Sw}).

Una vez que se conocen los valores de los parámetros a optimizar, es preciso proceder a la normalización de los mismos, ya que tienen unidades diferentes. Para ello se evalúan los valores mínimo y máximo que puede tener cada parámetro y se obtiene el rango del mismo. Estos rangos se utilizan en la función de coste para la normalización.

La Expresión 6.5 evalúa el rango del área que puede consumir el diseño. $Area_{max}$ representa la máxima área que puede consumir una solución. $Area_{max}$ se obtiene mediante la suma de las n (siendo n el número de procesos) áreas mayores elegidas entre el conjunto formado por:

$$T_{Fi} = \sum_{p}^{PasosVirtuales} max\{max\{T_{Hw}\}, \sum T_{Sw}\}_p$$

Expresión 6.4. Tiempo de ejecución de los procesos de la fase *i-ésima* del *pipeline*.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

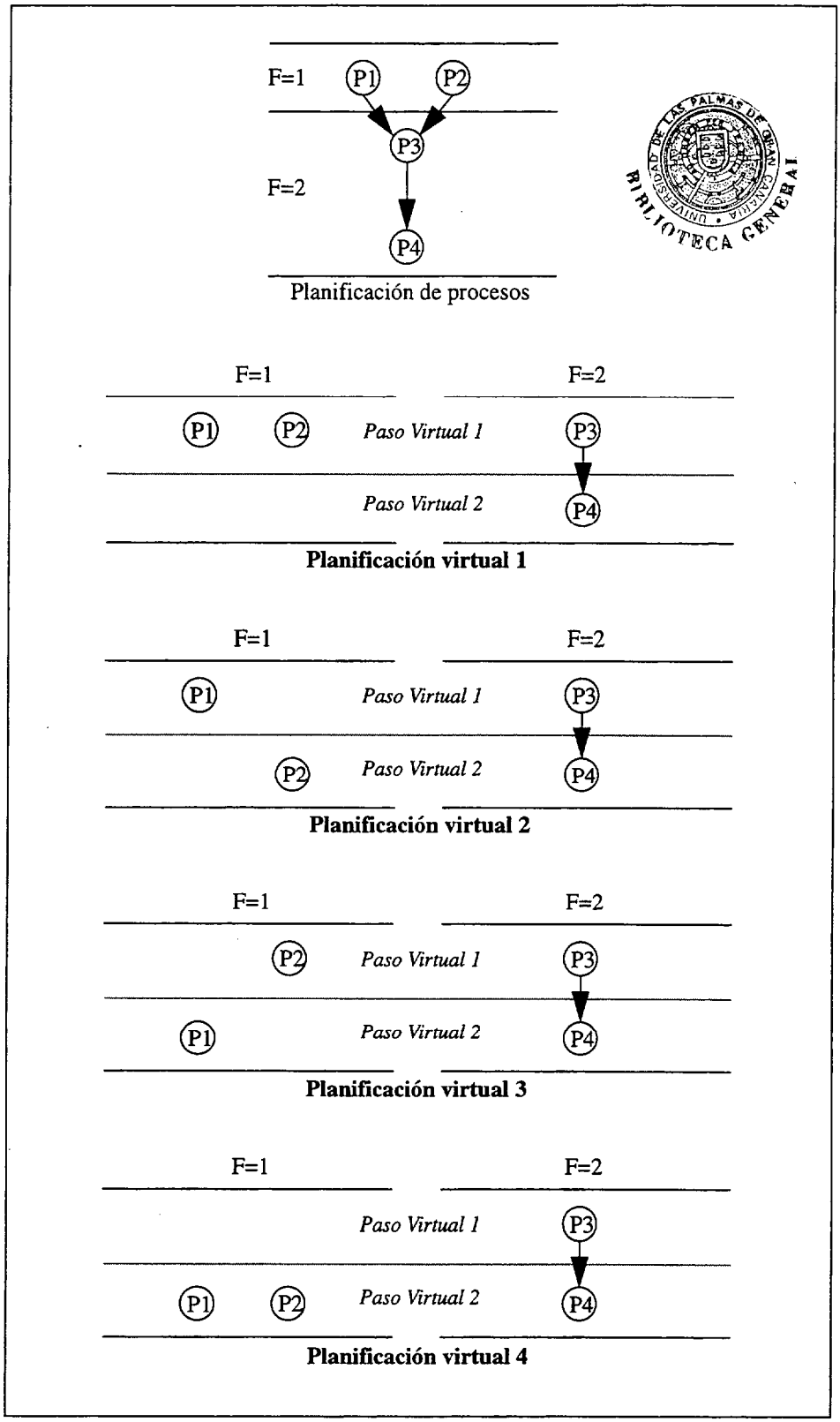


Figura 6.4. Planificaciones virtuales para una determinada planificación del *pipeline* de procesos.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

$$RangoArea = Area_{max} - Area_{min}$$

$$Area_{max} = \sum max^{n} \{A_{Hw_{maxP1}}, A_{Hw_{maxP2}}, \ldots, A_{Hw_{maxPn}}, A_{\mu P}\}$$

Expresión 6.5. Rango del parámetro área.

- las áreas máximas de cada proceso para su implementación en hardware ($A_{HWmaxPi}$).
- el área del procesador y su memoria ($A_{\mu P}$).

La obtención de $Area_{min}$ es análoga, pero utilizando áreas mínimas.

La Expresión 6.6 evalúa el rango de la energía que puede consumir el diseño. E_{maxPi} y E_{minPi} representan la energía máxima y mínima que consume cada proceso, respectivamente.

La Expresión 6.7 evalúa el rango del parámetro relativo a la optimización del tiempo de cada fase del *pipeline*.

Una vez calculados los valores de los parámetros y los rangos de los mismos, la función de coste que se utiliza en los algoritmos de particionado de la herramienta es la que se muestra en la Expresión 6.8.

$$RangoEnergia = \sum_{i}^{Procesos} (E_{max_{Pi}} - E_{min_{Pi}})$$

Expresión 6.6. Rango del parámetro energía.

$$RangoTiempo = N°Fases \times T_{Ejecucion}$$

Expresión 6.7. Rango del tiempo de ejecución no utilizado en las fases del *pipeline*.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

$$COSTE_{Potencia} = \alpha \frac{Energia - \sum_{i}^{Procesos} E_{min_{Pi}}}{RangoEnergia} + \beta \frac{Tiempo}{RangoTiempo}$$

$$COSTE_{Area} = \alpha \frac{Area - Area_{min}}{RangoArea} + \beta \frac{Tiempo}{RangoTiempo}$$

Expresión 6.8. Función de coste según el objetivo sea la potencia ($COSTE_{potencia}$) o el área ($COSTE_{area}$).

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

6.5. Algoritmos de particionado

En este apartado se describen los algoritmos heurísticos que hemos desarrollado para nuestro método de particionado Hw/Sw. En primer lugar, se describen los métodos que no utilizan segmentación a nivel de procesos, que hemos denominado *Métodos A y A/N*. En segundo lugar, se describen los métodos que utilizan segmentación a nivel de procesos, denominados *Métodos B y B/N*. Finalmente, se describen los *Métodos C y C/N* que se han desarrollado para contrastar la calidad de los métodos anteriores.

6.5.1. Métodos A y A/N

Estos métodos se utilizan para los casos en que el diseñador especifica latencia unitaria. Por tanto, no se utiliza la técnica de segmentación a nivel de procesos.

Estos métodos están basados en un algoritmo de *simulated annealing* (véase Algoritmo 6.1). Para la utilización del *simulated annealing* para el problema del particionado Hw/Sw es necesario fijar una serie de parámetros. Estos parámetros pueden dividirse en dos grupos:

- *Parámetros relacionados con la metodología de annealing*: la temperatura inicial, la probabilidad de aceptar soluciones peores que la actual, el esquema de enfriamiento, y la condición de parada.

```
C_actual = OBTENER_SOLUCION_INICIAL ( )
t = TEMPERATURA_INICIAL ( Pi )
do
    Contador = 0
    do
        if ( SELECCIONAR_VECINO ( C_vecina ) = TRUE ) then
            ganancia = COSTE (C_vecina ) - COSTE (C_actual )
            if ganancia < 0 then
                C_actual = C_vecina
            else
                r = RANDOM [ 0 , 1 ]
                if r < P ( t , ganancia ) then
                    C_actual = C_vecina
                end if
            end if
        end if
        Contador = Contador + 1
    while Contador <= N_Repeticiones
    t = TEMPERATURA ( t )
while Condicion_de_parada = FALSE
```

Algoritmo 6.1. Pseudo-código del algoritmo utilizado para los *Métodos A y A/N.*

- *Parámetros relacionados con el problema de particionado*: la obtención de la solución inicial, la selección de soluciones vecinas, y la función de coste.

Los parámetros relacionados con la metodología de *annealing* se han seleccionado de la forma explicada en capítulo 3. No obstante, es preciso recordar que para obtener la temperatura inicial hay que evaluar la máxima diferencia de coste que puede existir entre dos soluciones. En el problema del particionado Hw/Sw, la solución con el máximo coste es aquella donde el área o la energía consumida por el diseño es máxima, y el tiempo de cómputo no utilizado en cada fase del *pipeline* tiende al tiempo máximo de cada fase. La solución con el mínimo coste es aquella donde el área o la energía es mínima, y el tiempo de cómputo no utilizado en cada fase se aproxima a cero.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

Los parámetros relacionados con el problema de particionado se explican a continuación, a medida que se comenta el algoritmo.

En primer lugar, el algoritmo de *simulated annealing* comienza obteniendo una solución inicial correcta mediante la función OBTENER_SOLUCION_INICIAL. La obtención de dicha solución (C_{actual}) se realiza de forma aleatoria hasta encontrar la primera configuración que satisfaga las restricciones. La verificación de las restricciones consiste en:

- comprobar que el área ocupada o la potencia consumida es menor que la cota especificada por el diseñador.
- comprobar que el tiempo de cómputo del grafo de procesos es inferior al especificado por el diseñador.

En segundo lugar, la función SELECCIONAR_VECINO (véase el Algoritmo 6.2) selecciona un proceso de forma aleatoria en cada iteración del proceso de *annealing*. La elección del proceso se realiza mediante la función ELEGIR_PROCESO. Una vez seleccionado el proceso, se analizan las configuraciones del mismo y se selecciona aquella para la que se obtiene un menor valor de la función de coste. Para ello se examinan todas las configuraciones válidas del proceso. Esta tarea es realizada por la función ELEGIR_CONF (véase el Algoritmo 6.3). La validez de una configuración es determinada por la función VERIFICAR_PLANIFICACION. La función SELECCIONAR_VECINO retorna *true* si ha encontrado una configuración vecina válida (C_{vecina}), o bien retorna *false* en caso contrario.

Este método tiene una variante dependiendo de la función ELEGIR_CONF. En lugar de analizar todas las configuraciones de un proceso y seleccionar la mejor, se pueden

```
SELECCIONAR_VECINO ( Cvecina )
Proceso = ELEGIR_PROCESO ( )
if ( ELEGIR_CONF ( Proceso , Cvecina ) = TRUE ) then
    return (TRUE)
end if
return (FALSE)
```

Algoritmo 6.2. Pseudo-código de la función SELECCIONAR_VECINO.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

```
ELEGIR_CONF (Proceso , C_vecina)

Mejor_Coste = ∞
for Configuracion_i (Proceso) do
    C_aux ( Proceso ) = Configuracion_i
    if ( VERIFICAR_PLANIFICACION (C_aux ) = TRUE ) then
        if ( Mejor_Coste > COSTE ( C_aux ) ) then
            Mejor_Coste = COSTE (C_aux )
            C_vecina (Proceso) = C_aux (Proceso)
        end if
    end if
end for
if ( Mejor_Coste = ∞) then
    return ( FALSE )
else
    return ( TRUE )
end if
```

Algoritmo 6.3. Pseudo-código de la función ELEGIR_CONF.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

analizar un subconjunto aleatorio de las configuraciones del mismo y seleccionar la mejor de dicho conjunto. Esta variante da lugar a los *Métodos A/N* (siendo N el porcentaje de configuraciones que se analizan aleatoriamente).

Si la función SELECCIONAR_VECINO encuentra una configuración vecina válida, el algoritmo de *annealing* compara el coste de la misma con el coste de la configuración actual. Si el coste de la configuración vecina es menor que el coste de la actual, la primera es aceptada como nueva configuración actual. Si no es así, existe aún una probabilidad de aceptarla que viene dada por la función P. Esta función depende de la temperatura del proceso de *annealing* y de la diferencia entre los costes de ambas configuraciones. La función P que se utiliza es la misma que se ha explicado en el capítulo 3.

6.5.2. Métodos B y B/N

Estos métodos se utilizan cuando el diseñador especifica una latencia no unitaria, es decir, el diseñador quiere utilizar la técnica de segmentación de procesos. Análogamente a los *Métodos A y A/N*, los *Métodos B y B/N* están basados en un algoritmo de *simulated annealing* (véase Algoritmo 6.4).

En primer lugar, el algoritmo de *simulated annealing* comienza con la obtención de una solución inicial correcta. La segmentación a nivel de procesos requiere una solución inicial formada por dos componentes: por un lado, debe determinarse una planificación inicial del *pipeline* de procesos (P_{actual}), y por otro lado, debe seleccionarse una configuración inicial de los mismos (C_{actual}).

```
( Pactual , Cactual ) = OBTENER_SOLUCION_INICIAL ( )
t = TEMPERATURA_INICIAL ( Pi )
do
    Contador = 0
    do
        if ( SELECCIONAR_VECINO ( Pactual , Cactual , Pvecina , Cvecina ) = TRUE ) then
            ganancia = COSTE (Pvecina , Cvecina ) - COSTE ( Pactual , Cactual )
            if ganancia < 0 then
                ( Pactual , Cactual ) = ( Pvecina , Cvecina )
            else
                r = RANDOM [ 0 , 1 ]
                if r < P ( t , ganancia ) then
                    ( Pactual , Cactual ) = ( Pvecina , Cvecina )
                end if
            end if
        end if
        Contador = Contador + 1
    while Contador <= N_Repeticiones
    t = TEMPERATURA ( t )
while Condicion_de_parada = FALSE
```

Algoritmo 6.4. Pseudo-código del algoritmo utilizado para los *Métodos B y B/N*.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

La planificación inicial del *pipeline* de procesos se obtiene aplicando el algoritmo de planificación ASAP (*As Soon As Possible*) [Gaj92][Mic92] al grafo de procesos. En el caso que la latencia especificada por el diseñador sea inferior al número de pasos de control obtenidos mediante el algoritmo ASAP, se procede a una agrupación de los procesos hasta conseguir la latencia deseada. Este proceso consiste en la agrupación de fases consecutivas. Cada fase solamente puede ser agrupada una única vez. Si después de un primer intento no se ha conseguido la latencia que el diseñador ha especificado, entonces se repite el proceso. En la Figura 6.5.a se muestra la planificación ASAP de un

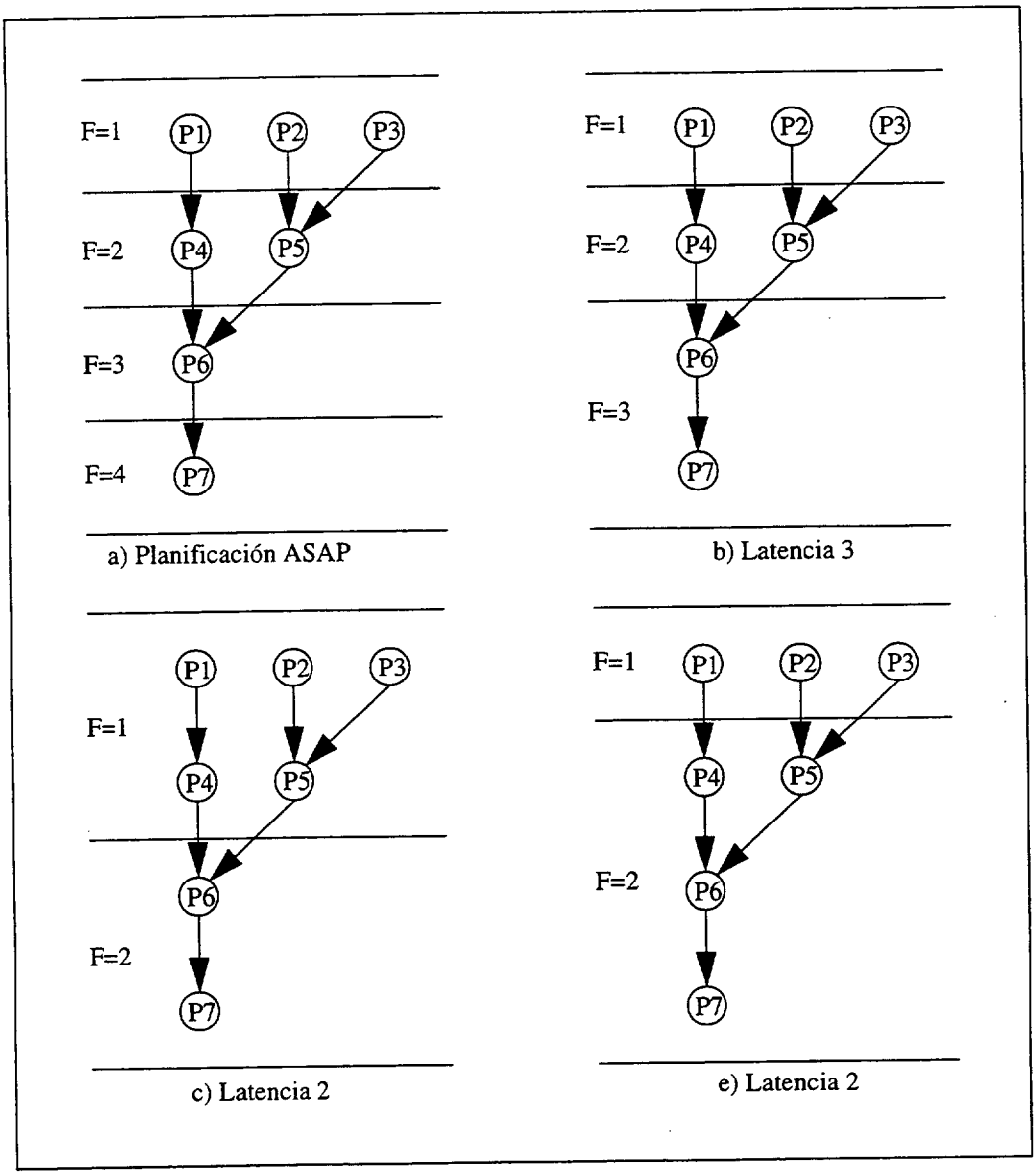


Figura 6.5. Ejemplo de agrupación de procesos para obtener una planificación inicial con una latencia inferior a la obtenida por el algoritmo ASAP.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

grafo, resultando un total de cuatro fases. En las Figuras 6.5.b y 6.5.c se muestra el proceso de agrupamiento para obtener tres y dos fases, respectivamente. Hay que tener en cuenta que para pasar de la planificación de la Figura 6.5.b a la planificación de la Figura 6.5.c se agrupan la primera y la segunda fase. De lo contrario, obtendríamos una planificación menos equilibrada (Figura 6.5.e).

La obtención de la configuración inicial de cada proceso se realiza de forma aleatoria hasta encontrar la primera configuración que satisfaga las restricciones. Para comprobar si se cumplen dichas restricciones se utiliza la función VERIFICAR_PLANIFICACION comentada en el apartado anterior.

Una vez obtenidas una planificación y una configuración iniciales, la función SELECCIONAR_VECINO (Algoritmo 6.5) proporciona una planificación vecina y una configuración vecina. En primer lugar, esta función utiliza la función ELEGIR_Y_MOVER_PROCESO para seleccionar un proceso de la planificación actual (P_{actual}) de forma aleatoria. Una vez seleccionado, el proceso es cambiado de fase de *pipeline*, pasando de forma aleatoria a otra fase previa o posterior a la actual. Este movimiento de fase de un proceso no se realiza en todas las iteraciones del proceso de *annealing*. De esta forma se consigue que cada planificación analizada por el algoritmo permanezca estable durante una serie consecutiva de iteraciones. Esto permite que el resto de procesos se adapten al cambio brusco que se produce cuando un proceso es cambiado de fase.

El movimiento de un proceso de una fase a otra del *pipeline* debe respetar las dependencias. En la Figura 6.6 se muestra un ejemplo con algunos movimientos de procesos. El proceso P1 no puede planificarse en la tercera fase (Figura 6.6.b) porque

```
SELECCIONAR_VECINO ( P_actual , C_actual , P_vecina , C_vecina )
(Proceso, P_vecina) = ELEGIR_Y_MOVER_PROCESO ( P_actual )
if ( ELEGIR_CONF ( Proceso, P_vecina , C_vecina ) = TRUE ) then
    return (TRUE)
end if
return ( FALSE )
```

Algoritmo 6.5. Pseudo-código de la función SELECCIONAR_VECINO.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

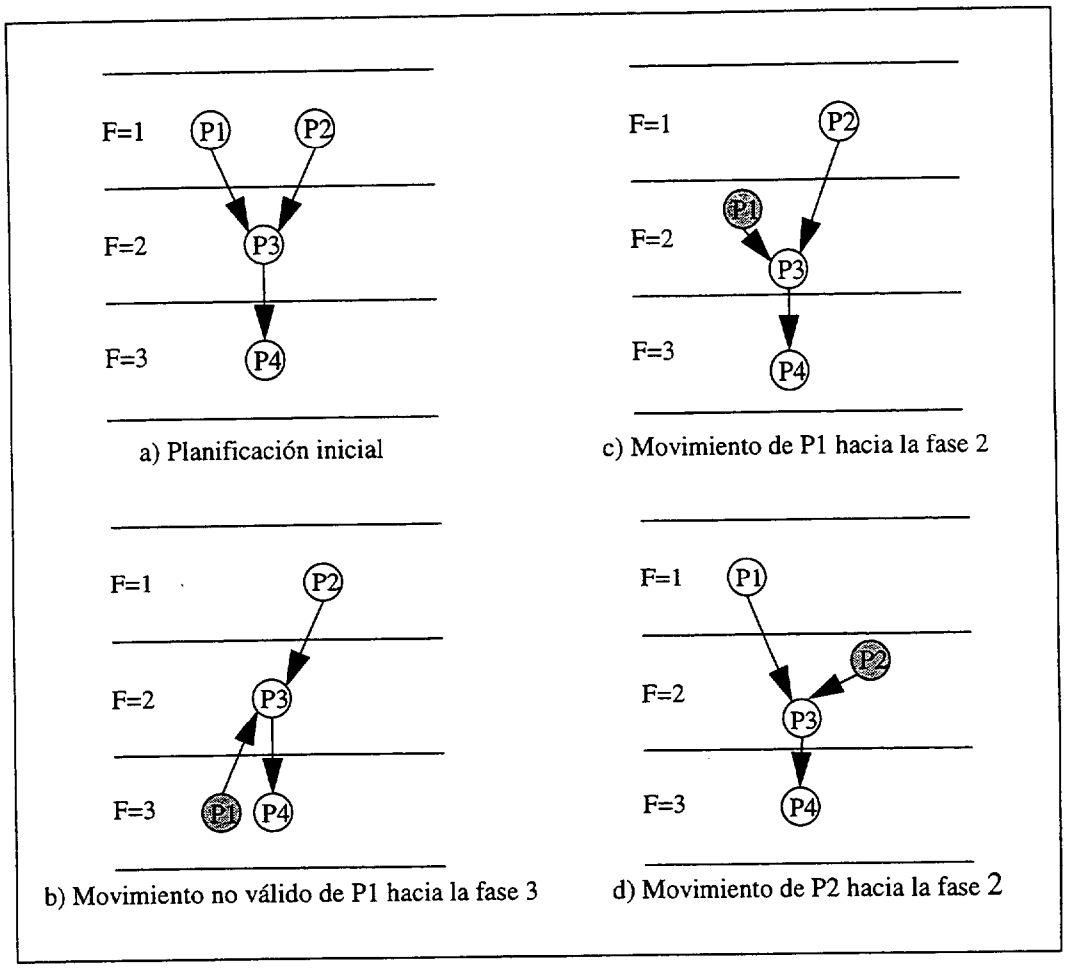


Figura 6.6. Ejemplo de movimientos de fase de los procesos del *pipeline*.

violaría la dependencia que tiene con el proceso P3. Sin embargo, tanto el proceso P1 como el proceso P2 pueden planificarse en la segunda fase del *pipeline* (Figuras 6.6.c y 6.6.d).

Una vez que un proceso ha sido seleccionado y cambiado de fase, la función SELECCIONAR_VECINO obtiene la configuración del proceso que proporciona el menor coste. Para ello se tienen en cuenta todas las configuraciones del proceso. Para realizar esta tarea se utiliza la función ELEGIR_CONF (Algoritmo 6.6).

Análogamente al *Método A*, este método tiene una variante dependiendo de la función ELEGIR_CONF. En lugar de analizar todas las configuraciones de un proceso y seleccionar la mejor, se pueden analizar un subconjunto aleatorio de las mismas y seleccionar la mejor de dicho conjunto. Esta variante da lugar a los *Métodos B/N* (siendo N el porcentaje de configuraciones que se analizan aleatoriamente).

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

```
ELEGIR_CONF (Proceso, P_vecina , C_vecina)

Mejor_Coste = ∞
for Configuracion_i (Proceso) do
    C_aux = Configuracion_i
    if ( VERIFICAR_PLANIFICACION ( P_vecina , C_aux) = TRUE ) then
        if ( Mejor_Coste > COSTE ( P_vecina , C_aux ) ) then
            Mejor_Coste = COSTE (P_vecina , C_aux )
            C_vecina = C_aux
        end if
    end if
end for
if ( Mejor_Coste = ∞) then
    return ( FALSE )
else
    return ( TRUE )
end if
```

Algoritmo 6.6. Pseudo-código de la función ELEGIR_CONF.

Si la función SELECCIONAR_VECINO encuentra una solución vecina válida, el algoritmo de *annealing* compara el coste de la misma con el coste de la solución actual. Si el coste de la solución vecina es menor que el coste de la solución actual, la primera es aceptada como solución actual. Si no es así, aún existe una probabilidad de aceptarla que viene dada por la función P. Esta función es la misma que la utilizada en los *Métodos A y A/N.*

6.5.3. Métodos C y C/N

Estos métodos se han desarrollado para evaluar la calidad de los resultados de los métodos heurísticos anteriores. Inicialmente se desarrolló un método completamente exhaustivo para obtener soluciones óptimas. Sin embargo, el tiempo de ejecución era tan elevado que no nos permitía obtener resultados para los sistemas que se han utilizado para *testear* la herramienta. Para resolver el problema se ha optado por un método semi-exhaustivo (Algoritmo 6.7). Por una parte, se trata de un método exhaustivo porque analiza todas las planificaciones posibles del *pipeline* de procesos. Por otra parte, se trata de un método heurístico porque aplica los *Métodos A y A/N* a

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

```
SEMI_EXHAUSTIVO ( Proceso )
begin
   L = { Lista de fases F donde se puede colocar el proceso Proceso }
   for ( F de L ) do
      P_actual ( Proceso ) = F
      if ( Todos_Procesos_Planificados ) then
         if (Metodo_A_N ( P_actual , C_actual ) = TRUE ) then
            if ( Mejor_Coste > COSTE ( C_actual ) ) then
               Mejor_Coste = COSTE ( C_actual )
               ( P_mejor , C_mejor ) = ( P_actual , C_actual )
            end if
         end if
      else
         SEMI_EXHAUSTIVO ( Proceso + 1 )
      end if
   end for
end
```

Algoritmo 6.7. *Métodos C y C/N.*

cada una de estas planificaciones. En general, como se mostrará en el capítulo de resultados experimentales, el diseñador no debe utilizar este método debido al excesivo tiempo de computación que requiere. Sin embargo, es muy interesante su desarrollo para comparar sus resultados con los obtenidos mediante los métodos anteriores.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

Capítulo 7

Resultados experimentales

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

7.1. Introducción

En este capítulo se presentan los resultados más importantes de los experimentos que se han realizado en este trabajo. Por un lado, los experimentos ilustran las posibilidades que tiene el diseñador para explorar el espacio de diseño al tener en cuenta la potencia consumida por el diseño así como la segmentación de los grafos de procesos. Por otro lado, se realiza una comparación entre los métodos heurísticos que se utilizan y se analiza la calidad de los resultados obtenidos mediante los mismos.

Los experimentos se han realizado en un computador personal (PC) con las siguientes características:

- Procesador: Pentium II 300 MHz.
- Memoria RAM: 64 Mbytes.

La estructura del capítulo es la siguiente: en el apartado 2 se describe el ejemplo que se ha utilizado para los experimentos. En el apartado 3 se indican las librerías utilizadas. En el apartado 4 se comentan las estimaciones obtenidas. Finalmente, en el apartado 5 se presentan los resultados más importantes.

7.2. Descripción del ejemplo

El sistema que se ha utilizado como ejemplo forma parte de un sistema de reconocimiento de voz. El objetivo del sistema es la obtención de un conjunto de coeficientes a partir de una señal de voz muestreada a 8 Khz y dividida en tramas de 128 muestras.

El sistema está formado por ocho procesos concurrentes que se comunican a través de un canal de tipo *broadcast* y otro de tipo *incast*. El proceso *padre* lee una trama de voz y la procesa aplicando una ventana de Hamming y calculando su FFT (*Fast Fourier Transform*). Seguidamente, el proceso *padre* envía los resultados a los procesos *son00*, *son01*, *son02*, *son20*, *son21* y *son22* a través del canal *broadcast*. Estos procesos filtran la señal y envían los resultados hacia el proceso *son3* a través del canal *incast*. Finalmente, el proceso *son3* calcula los coeficientes.

El flujo de datos de los procesos del sistema se ilustra en el grafo de procesos de la Figura 7.1. La especificación VSS del sistema, así como el código VHDL generado por el compilador desarrollado en este trabajo, se encuentran en los Apéndices D y E, respectivamente. Consultando estos apéndices se puede observar claramente que la especificación VSS es bastante más pequeña y sencilla que la especificación VHDL.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

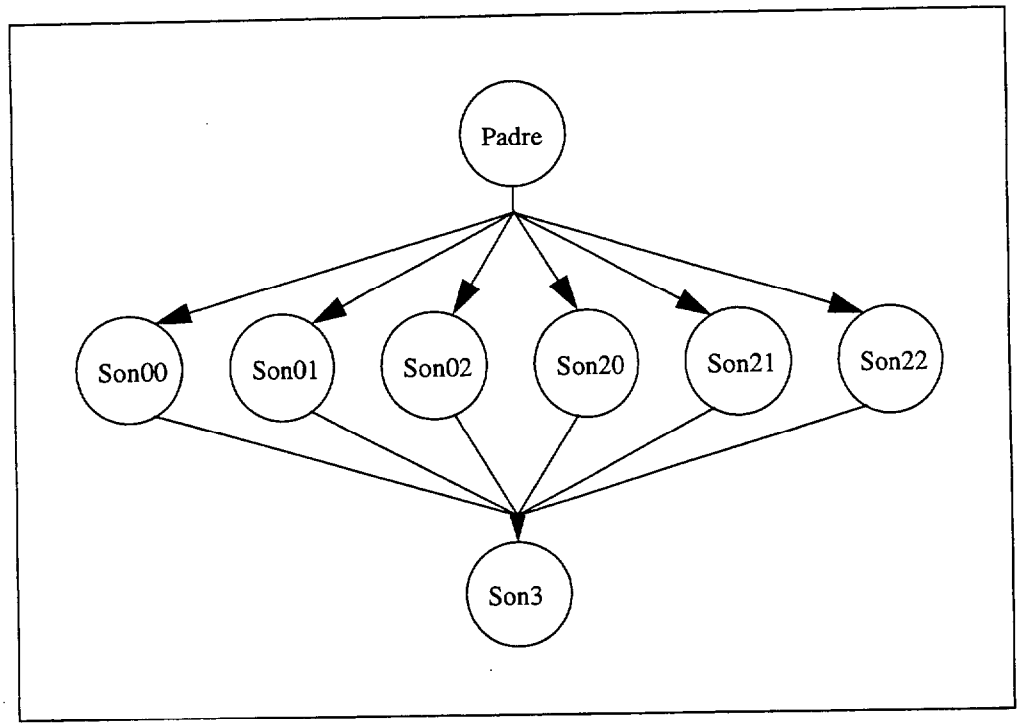


Figura 7.1. Grafo de procesos.

7.3. Librerías utilizadas

Las librerías que se han utilizado para el diseño del sistema son las siguientes:

- *recursos.def*: unidades funcionales (Figura 7.2).
- *memorias.def*: memorias (Figura 7.3).
- *processor.def*: procesadores (Figura 7.4).

Por un lado, los datos de la librería *recursos.def* se han tomado de [Mar96]. Se ha decidido utilizar cuatro sumadores y tres multiplicadores porque son los operadores más frecuentes en la especificación (situación típica en el procesado digital de señales). En la Figura 7.5 se muestran las unidades funcionales que pueden implementar los operadores VHDL.

Por otro lado, los datos de la librería *memorias.def* se han tomado de [Sen96]. Finalmente, los datos de la librería *processor.def* se han tomado de [Spe93]. El procesador seleccionado para el diseño ha sido el Pentium.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

10 E -9 /* Ciclo HW en seg. */

Unidad	*Código*	*Delay*	*Área (mm^2)*	*Potencia (mW)*	*Segmentada*	*Nº Etapas*
sum_1	1	2	0.127	22.7	0	0
sum_2	2	2	0.222	10.5	0	0
sum_3	3	3	0.127	5.4	0	0
sum_4	4	1	0.200	22.7	0	0
mult_1	5	16	0.363	84.0	0	0
mult_2	6	16	0.463	60.0	0	0
mult_3	7	32	0.363	12.7	0	0
div	8	32	0.500	95.0	0	0
and_32	9	1	0.002	0.5	0	0
or_32	10	1	0.002	0.5	0	0
comparador	11	3	0.200	13.0	0	0

Figura 7.2. Librería *recursos.def* utilizada.

Unidad	*Bytes*	*T_Acceso (s)*	*Área (mm^2)*	*Potencia (mW)*
RAM_1	64	15 E -9	0.414	5.0
RAM_2	128	15 E -9	0.458	7.0
RAM_3	256	20 E -9	0.548	10.0
RAM_4	512	20 E -9	0.714	15.0
RAM_5	1024	30 E -9	1.060	20.0
RAM_6	2048	30 E -9	1.730	25.0
RAM_7	4096	40 E -9	2.400	30.0
RAM_8	8192	40 E -9	3.700	40.0

Figura 7.3. Librería *memorias.def* utilizada.

Nombre	*CPI*	*Área (mm^2)*	*Potencia (mW)*	*Frecuencia (MHz)*
pentium	1.33	296.0	16000	66
80486	2.00	225.0	5000	50
super_sparc	0.66	256.0	14200	60

Figura 7.4. Librería *processor.def* utilizada.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

```
add_c     sum_1 sum_2 sum_3 sum_4 .
sub_c     sum_1 sum_2 sum_3 sum_4 .
mult_c    mult_1 mult _2 mult_3 .
div_c     div .
eq_c      comparador .
ne_c      comparador .
ls_c      comparador .
le_c      comparador .
gr_c      comparador .
ge_c      comparador .
and_c     and_32 .
or_c      or_32 .
```

Figura 7.5. *Mapping* de operadores VHDL a unidades funcionales.

7.4. Estimaciones

Una vez que dispone de la especificación del sistema, de los ficheros de contadores, y de las librerías de diseño, el diseñador procede a la obtención de las estimaciones hardware y software necesarias para la herramienta de particionado. Las estimaciones pueden obtenerse sin limitación alguna en cuanto al número de unidades funcionales que se pueden utilizar en paralelo. Sin embargo, el diseñador puede estar interesado en limitar dichas unidades. De esta forma, se puede eliminar inicialmente una parte del espacio de diseño que el diseñador no desea considerar, contribuyendo de esta forma a que la herramienta de particionado realice una mayor exploración del espacio de diseño restante.

El número de configuraciones Hw/Sw del sistema que se ha utilizado como ejemplo es elevado si se realizan las estimaciones sin limitar los recursos (Tabla 7.1). La herramienta de particionado se encontraría con un espacio de diseño del orden de 10^{37} soluciones posibles. Los resultados que se muestran en el resto del capítulo se han obtenido limitando el número máximo de unidades de cada tipo que se pueden ejecutar en paralelo. El fichero utilizado para la limitación de recursos se muestra en la Figura 7.6.

En la Tabla 7.2 se muestra el número de configuraciones que se generan para cada proceso utilizando la limitación de recursos mencionada. Se observa claramente que el espacio de diseño ha sido reducido, de forma que la herramienta de particionado se encuentra con un espacio de diseño del orden de 10^{24} soluciones posibles.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

Proceso	Nº Configuraciones
Padre	10.752.001
Son00	28.801
Son01	28.801
Son02	28.801
Son20	28.801
Son21	28.801
Son22	28.801
Son3	1.344.001

Tabla 7.1: Número de configuraciones sin limitación de recursos.

Proceso	Nº Configuraciones
Padre	1601
Son00	1601
Son01	1601
Son02	1601
Son20	1601
Son21	1601
Son22	1601
Son3	1601

Tabla 7.2: Número de configuraciones generadas para las estimaciones realizadas.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

add_c	2
sub_c	2
mult_c	1
div_c	1
eq_c	1
ne_c	1
ls_c	1
le_c	1
gr_c	1
ge_c	1
and_c	1
or_c	1

Figura 7.6. Fichero de limitación de recursos.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

La obtención de las estimaciones de la Tabla 7.2 ha necesitado un tiempo de computación de 8 minutos.

7.5. Particionado hardware/software

Como se ha comentado en el capítulo anterior, el particionado del sistema se puede realizar utilizando dos objetivos diferentes:

- Acotar el área y minimizar la potencia.
- Acotar la potencia y minimizar el área.

Para el particionado del sistema de reconocimiento de voz se ha utilizado el primer objetivo. Por tanto, es necesario fijar los siguientes parámetros de entrada:

- *Tiempo de ejecución.* La elección de este valor se ha realizado de la siguiente forma: se parte de una señal de voz muestreada a 8 Khz y dividida en tramas de 128 muestras. Por tanto, el tiempo de ejecución del grafo de procesos no debe ser superior a *1.6E-2* segundos ($128 \cdot 8000^{-1}$).
- *Cota de área.* Para observar el comportamiento según el área se han realizado particionados con diversas cotas de área: 296, 310, 320, 330 y 340 mm^2. Estas cotas fueron seleccionadas para tener en cuenta todo el espectro de diseño, desde diseños con área mínima hasta diseños con área máxima.

En los siguientes apartados se presentan los resultados más importantes que se extraen de los experimentos que se han realizado con la herramienta de particionado. En primer lugar, se muestran las posibilidades que tiene el diseñador para explorar el espacio de diseño. En segundo lugar, se presentan los resultados obtenidos al utilizar la técnica de segmentación a nivel de procesos. En tercer lugar, se comparan los métodos heurísticos que se utilizan en la herramienta. Finalmente, se analiza la calidad de los resultados obtenidos mediante los algoritmos heurísticos.

7.5.1. Espacio de diseño

Los algoritmos heurísticos que se utilizan en la herramienta de particionado permiten que el diseñador explore ampliamente el espacio de diseño. Así, el diseñador puede optar por diferentes particiones Hw/Sw en función del área y la potencia.

En la Figura 7.7 se muestran diferentes soluciones del espacio de diseño en función del área y la potencia. La figura muestra claramente que la potencia consumida por el diseño disminuye conforme aumenta el área permitida para el mismo. Por una parte, al aumentar el área permitida se pueden utilizar unidades funcionales con mayor área y menor voltage. Por otro lado, al permitirse un área mayor se pueden implementar un mayor número de procesos en hardware, disminuyendo de esta forma la potencia consumida por el microprocesador.

El número de procesos implementados en hardware y software para las soluciones presentadas en la Figura 7.7 se muestran en la Figura 7.8. En la misma, se observa claramente que el número de procesos implementados en hardware crece conforme lo hace el área que puede utilizar el diseño.

En definitiva, el diseñador puede realizar compromisos entre el área y la potencia del diseño, por lo que puede orientar el particionado Hw/Sw según las necesidades del diseño. Así por ejemplo, si el grafo de procesos representa a un diseño que va a ser implantado en un sistema portátil, el diseñador puede optar por un particionado que minimice el consumo de potencia.

7.5.2. Segmentación a nivel de procesos

En el capítulo anterior se presentó la segmentación a nivel de procesos como una de las principales aportaciones de la herramienta de particionado. Esta aportación permite al diseñador la realización de compromisos entre la latencia para ejecutar el grafo de procesos y el área o potencia consumidas por el mismo.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

En las Figuras 7.9, 7.10 y 7.11 se ilustra la disminución de la potencia consumida por el diseño si se utiliza un *pipeline* de dos fases (latencia 2). En dichas figuras se observa claramente que las particiones que se obtienen utilizando el *pipeline* de dos fases consumen aproximadamente 0.5 watios menos que las particiones realizadas sin la segmentación (latencia 1).

Ahora bien, no siempre es beneficioso aumentar el número de fases del *pipeline*. En la Figura 7.12 se muestra que la potencia consumida por las particiones obtenidas para un *pipeline* de tres fases es la misma que la consumida para un *pipeline* de dos fases. Este resultado se debe a la estructura del grafo de procesos del sistema que se ha utilizado en los experimentos (apartado 7.2).

7.5.3. Comparación de los métodos

En este apartado se muestran los resultados comparativos de los métodos heurísticos utilizados en la herramienta de particionado. Básicamente, se comparan los parámetros más importantes: el grado de optimización de la función de coste, y el tiempo de ejecución de los algoritmos.

La comparación de los métodos se ha realizado fijando una determinada latencia. Es decir, en primer lugar se han comparado los *Métodos A/N* (latencia 1), y en segundo lugar, se han comparado los *Métodos B/N* (latencias mayores que uno).

Por lo que a la minimización de la función de coste se refiere, los *Métodos A/N* obtienen resultados prácticamente iguales independientemente del valor del parámetro N, como se ilustra en las Figuras 7.13 y 7.14. Sin embargo, este parámetro influye notablemente en el tiempo de ejecución, como se observa en las Figuras 7.15 y 7.16. Por tanto, según los experimentos realizados el diseñador debe utilizar el *Método A/1*. Por lo que a los *Métodos B/N* respecta, ocurre exactamente lo mismo que con los *Métodos A/N*.

7.5.4. Calidad de los resultados

La calidad de los métodos heurísticos utilizados debe ser medida comparando sus resultados con los resultados óptimos. Sin embargo, la obtención de los resultados óptimos para el diseño que se ha utilizado en los experimentos fue descartada debido al excesivo tiempo de computación. De hecho, varios experimentos fueron abortados después de varias semanas de ejecución.

Como una forma alternativa de comprobar que la heurística utilizada proporciona buenos resultados se desarrollaron los *Métodos C/N*, comentados en el capítulo anterior. Estos métodos realizan un particionado heurístico para cada una de las planificaciones

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

posibles de un grafo de procesos. Esto nos permite comprobar si los *Métodos B/N*, que solamente analizan un subconjunto de todas las planificaciones, se aproximan a los resultados obtenidos por los *Métodos C/N*.

En las Figuras 7.17, 7.18, 7.19, 7.20, 7.21, 7.22 y 7.23 se muestran los resultados obtenidos para los *Métodos B/N* y *C/N* para latencia dos. En las mismas, se observa claramente que la optimización de la función de coste realizada por los *Métodos B/N* se aproxima a la realizada por los *Métodos C/N*. Sin embargo, como se muestra en las Figuras 7.24, 7.25. 7.26 y 7.27, el tiempo de ejecución de los *Métodos B/N* es notablemente menor (entre 4 y 8 veces) que el correspondiente a los *Métodos C/N*. Además, el tiempo de ejecución de los *Métodos C/N* crecería de forma exponencial si aumentásemos el número de procesos del grafo, mientras que en los *Métodos B/N* crecería de forma lineal.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

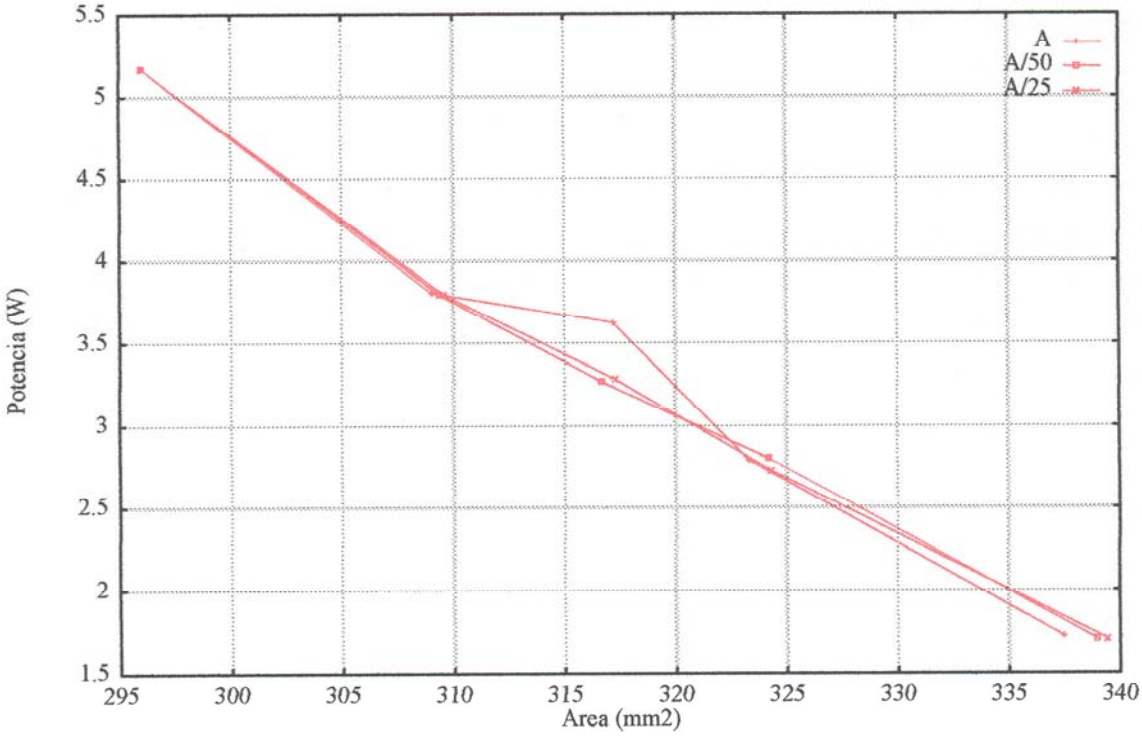


Figura 7.7. Espacio de diseño.

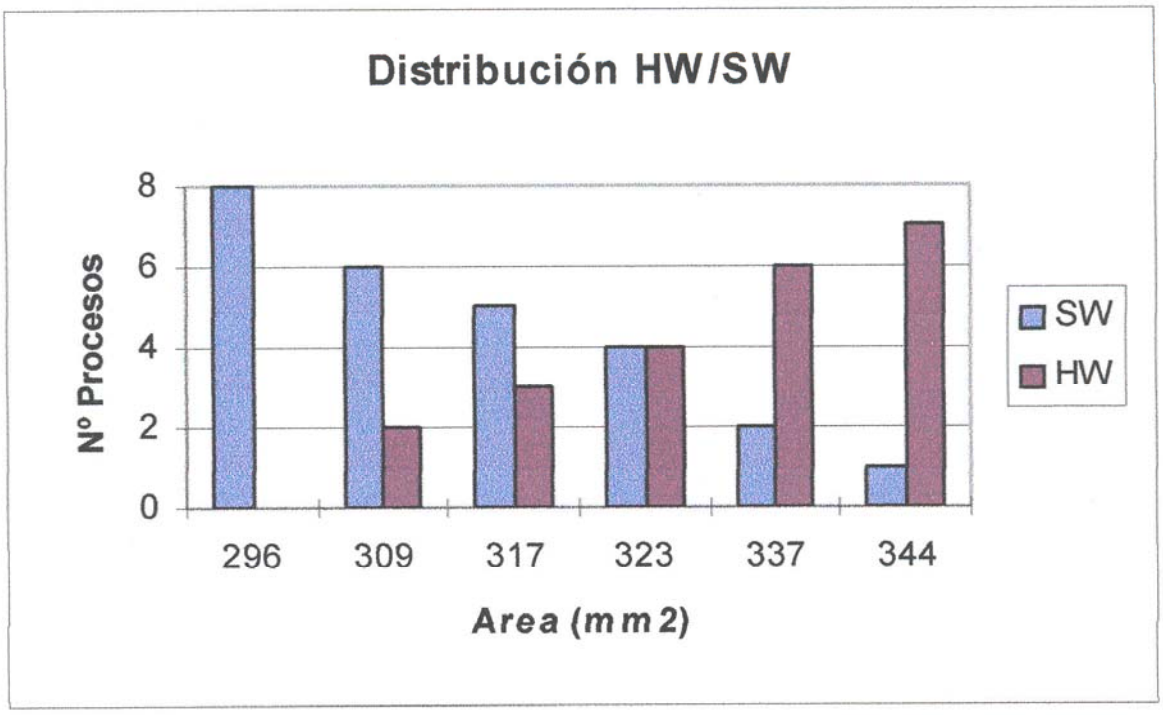


Figura 7.8. Distribución de procesos Hw / Sw.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

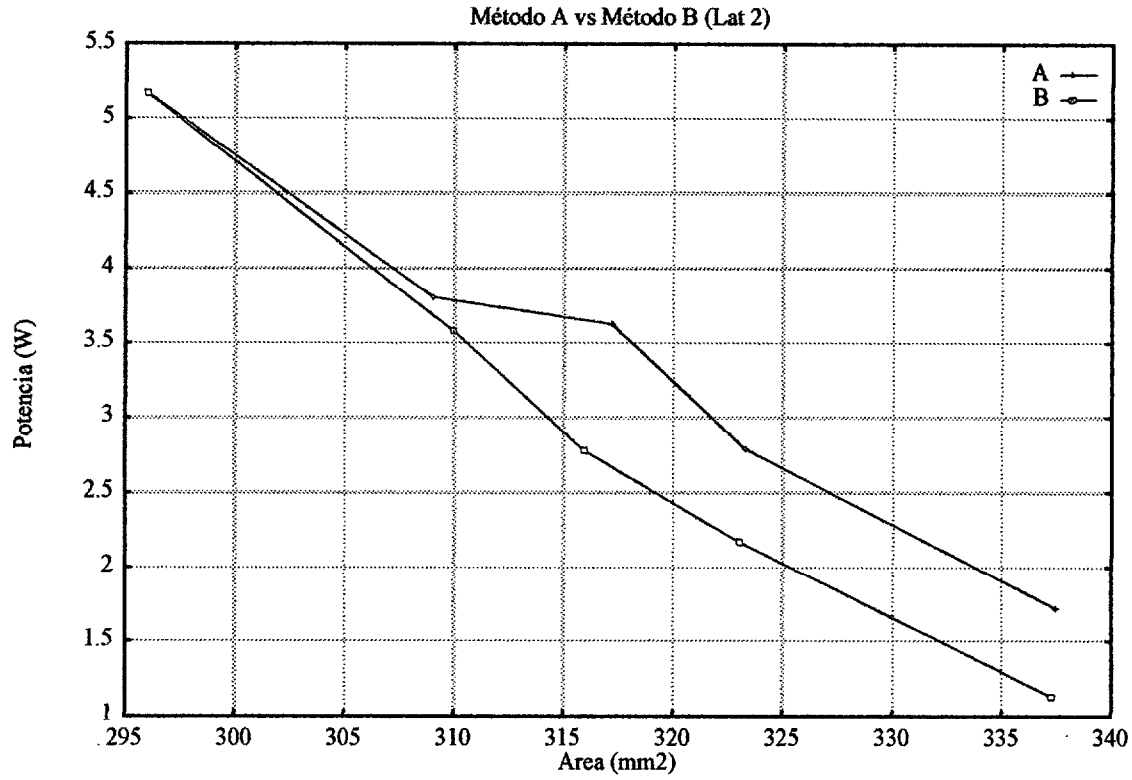


Figura 7.9. Segmentación a nivel de procesos.

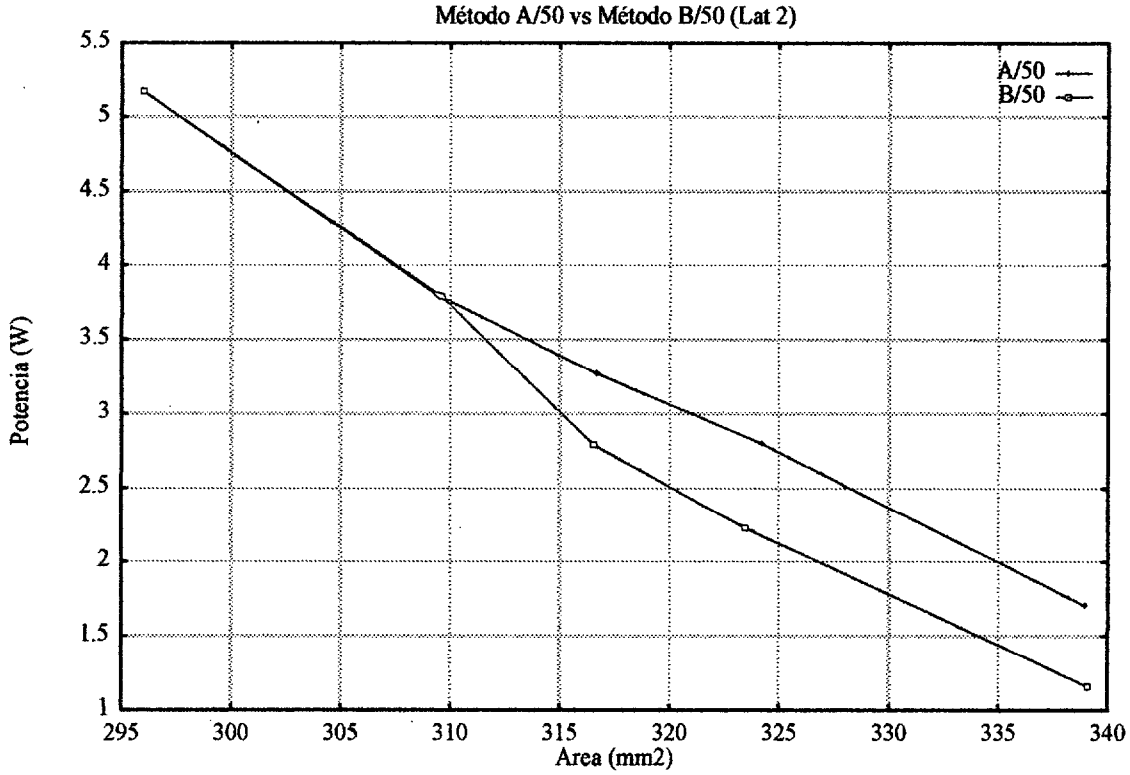


Figura 7.10. Segmentación a nivel de procesos.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

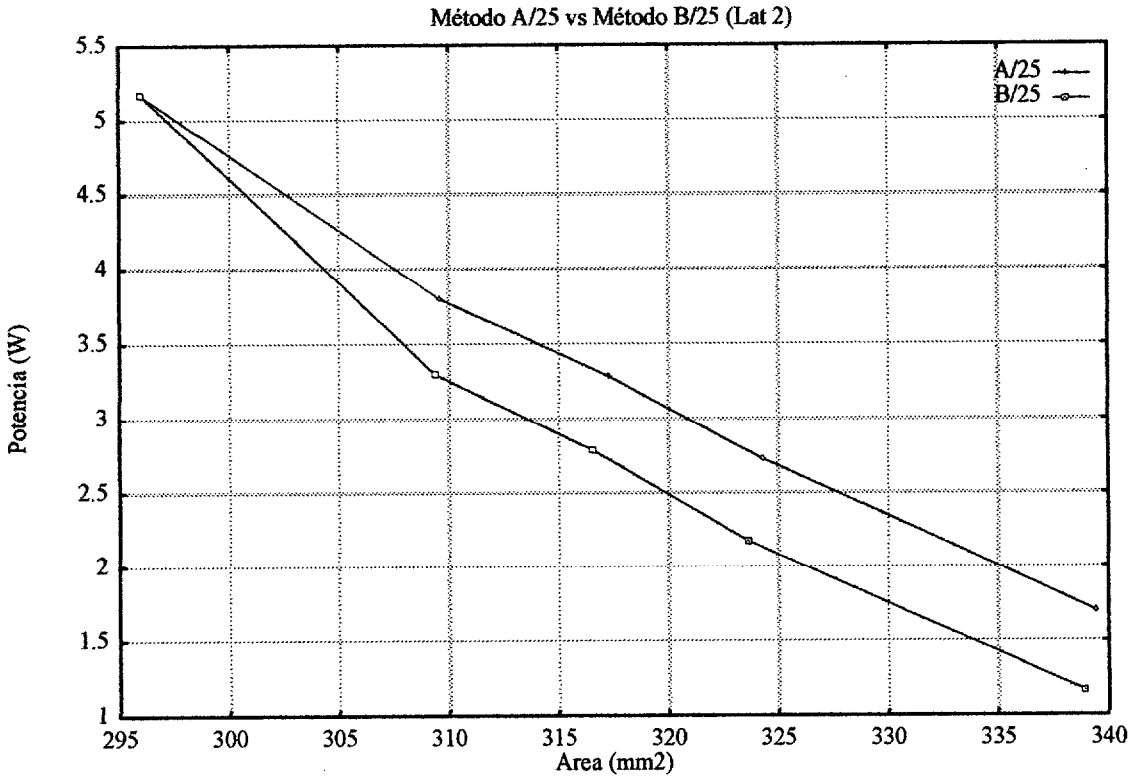


Figura 7.11. Segmentación a nivel de procesos.

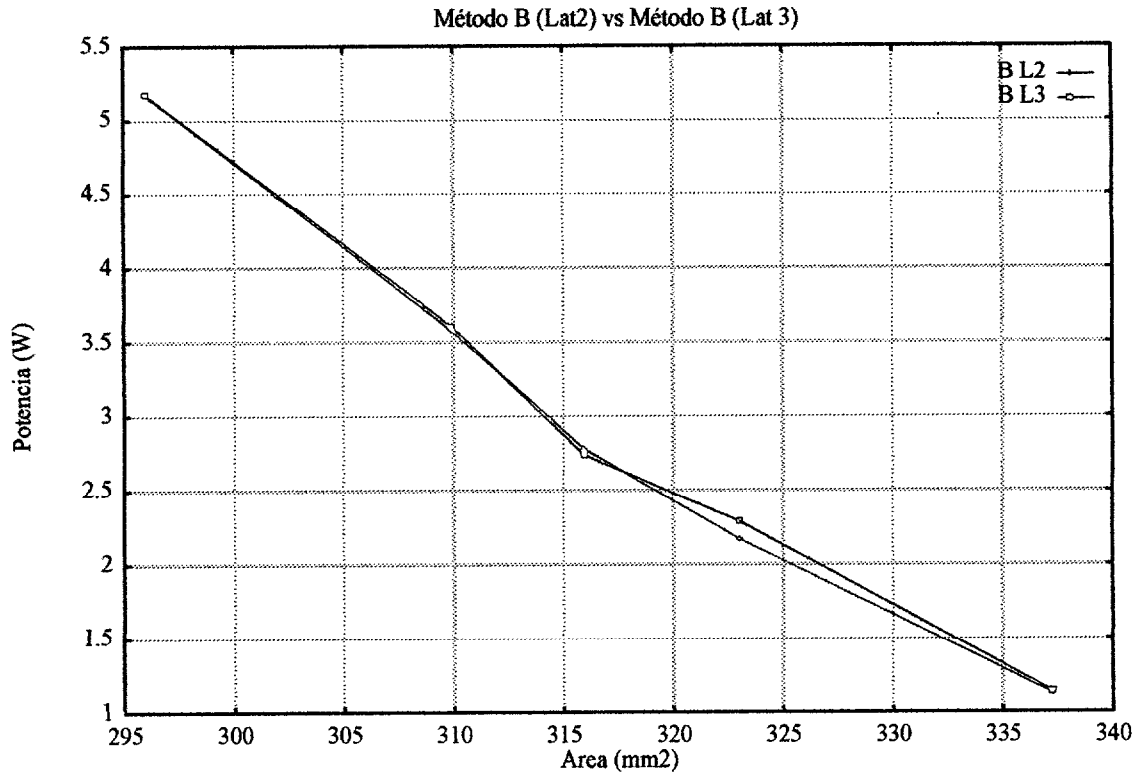


Figura 7.12. Segmentación a nivel de procesos.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

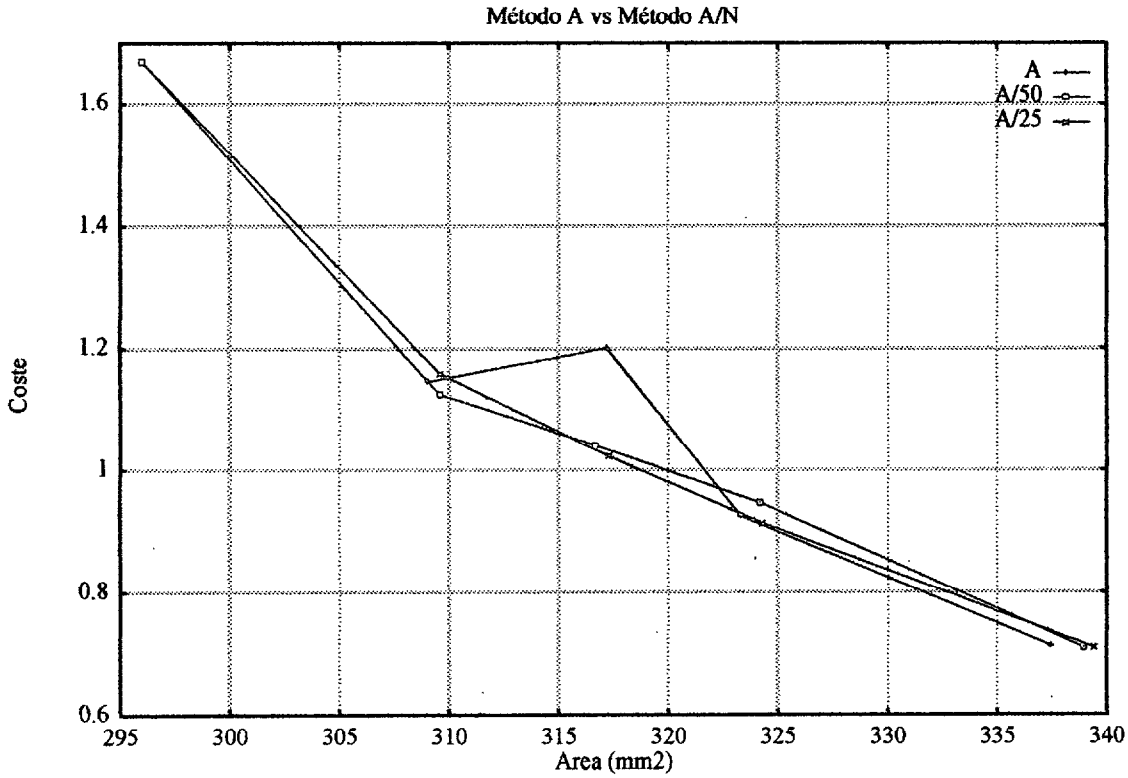


Figura 7.13. Comparación de los métodos.

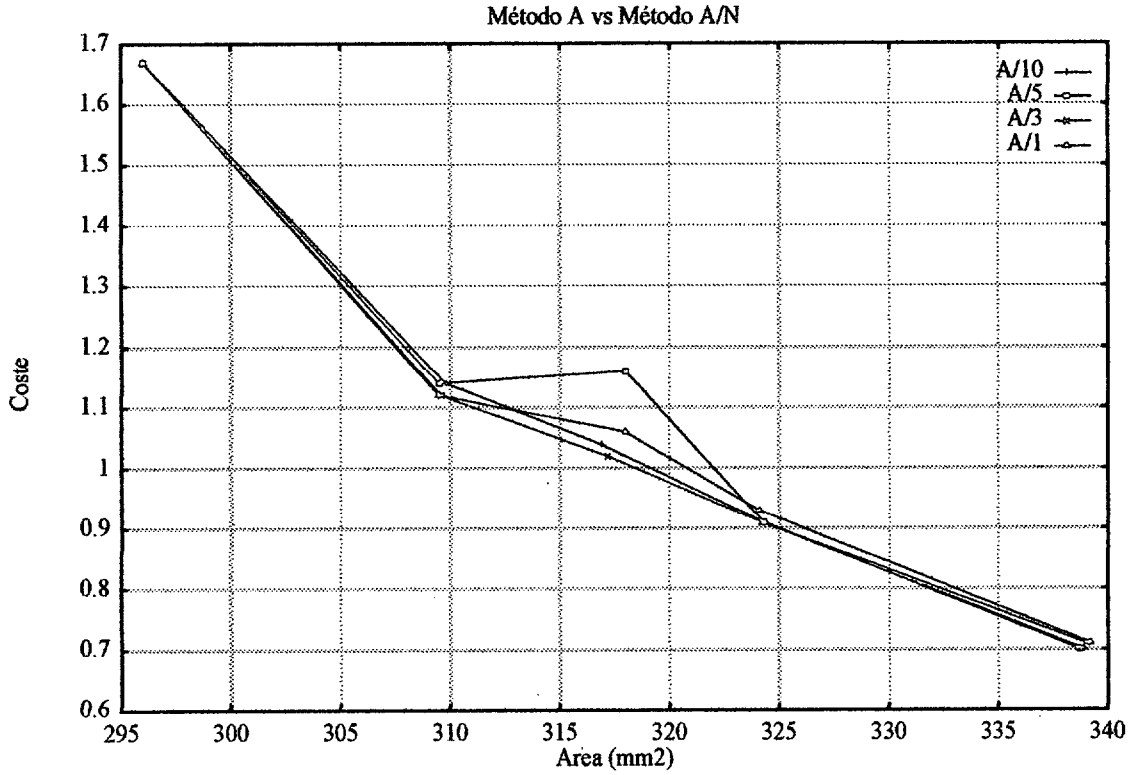


Figura 7.14. Comparación de los métodos.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

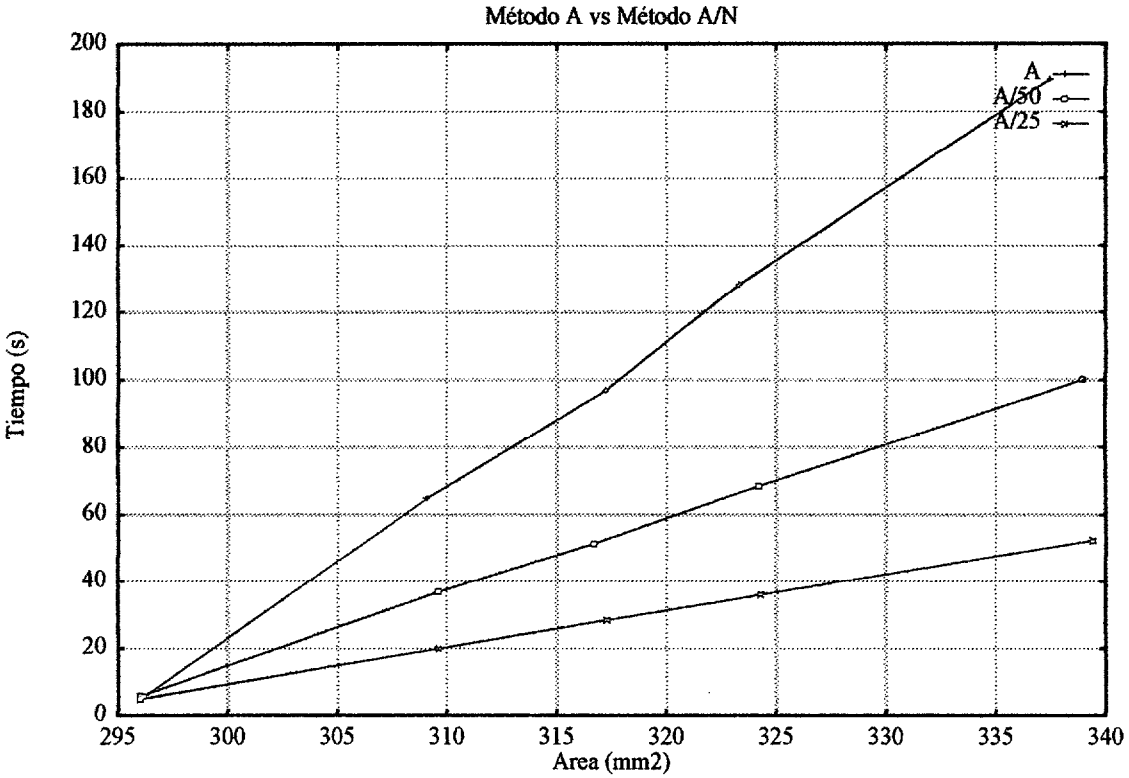


Figura 7.15. Comparación de los métodos.

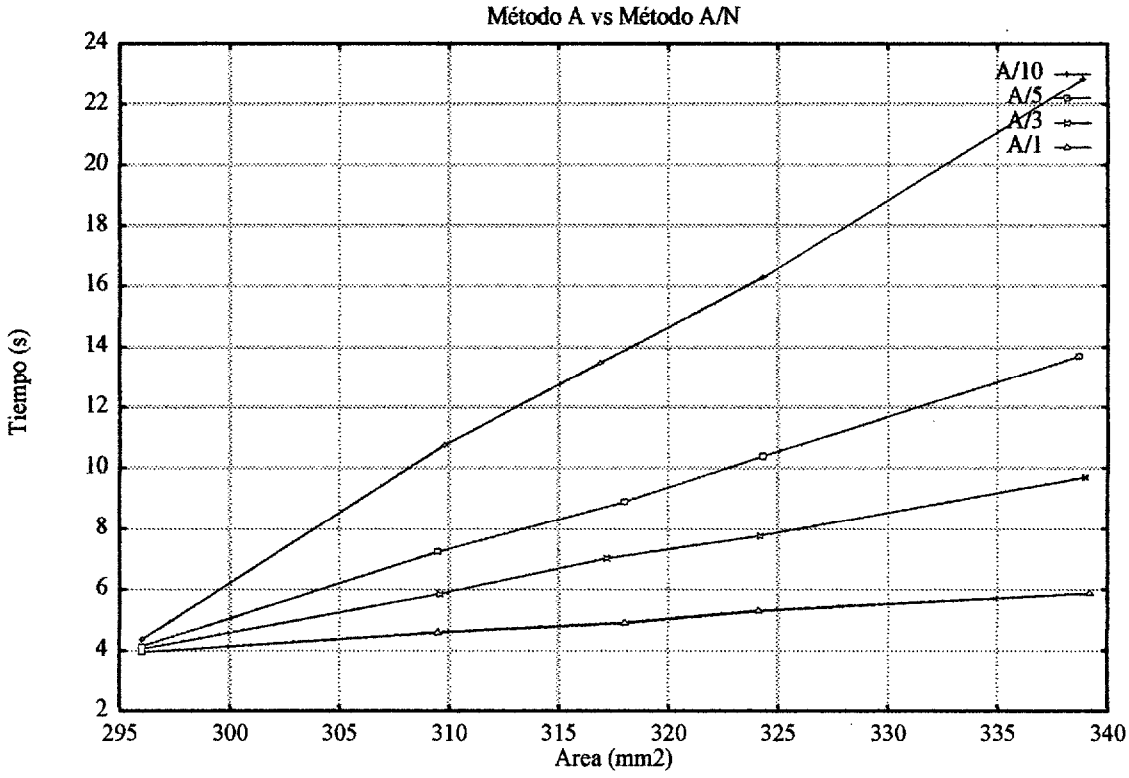


Figura 7.16. Comparación de los métodos.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

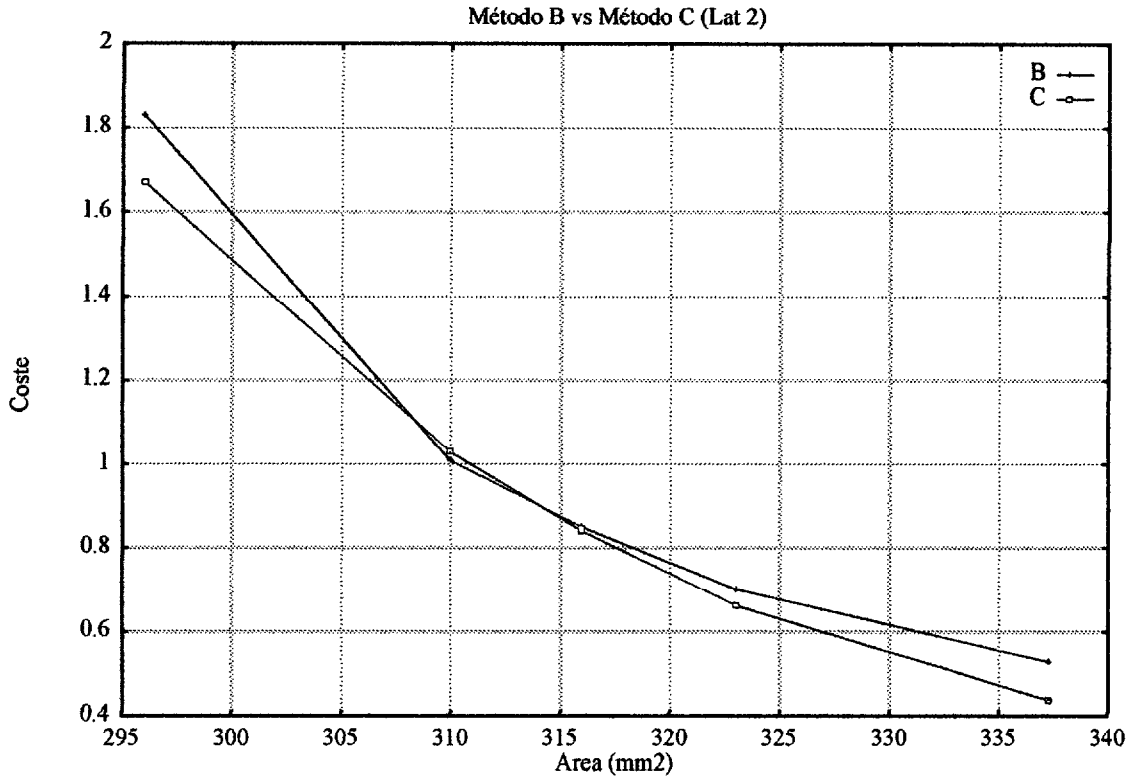


Figura 7.17. Calidad de los resultados.

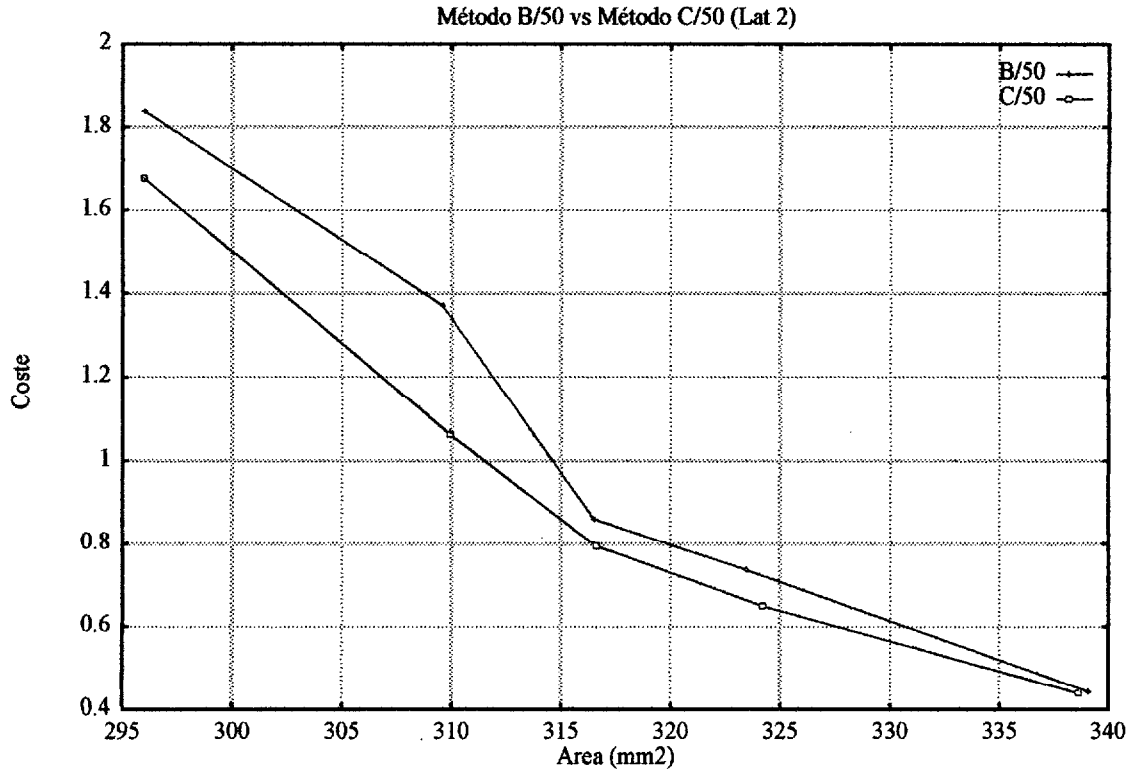


Figura 7.18. Calidad de los resultados.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

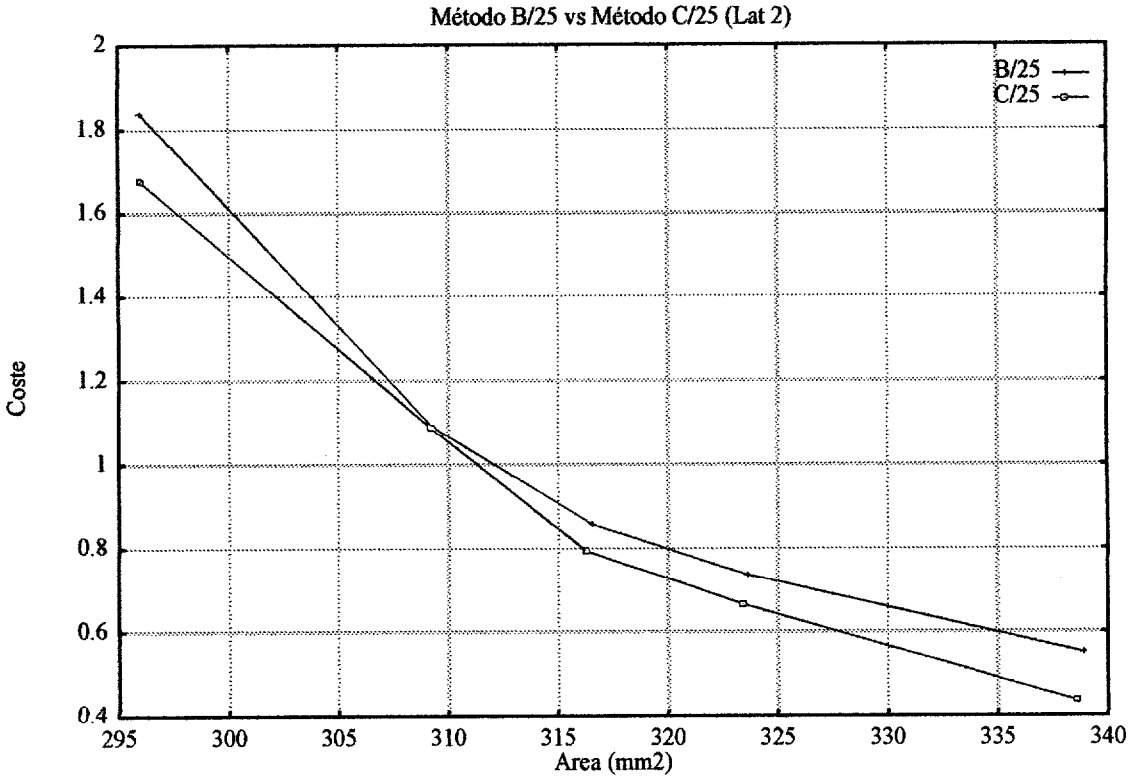


Figura 7.19. Calidad de los resultados.

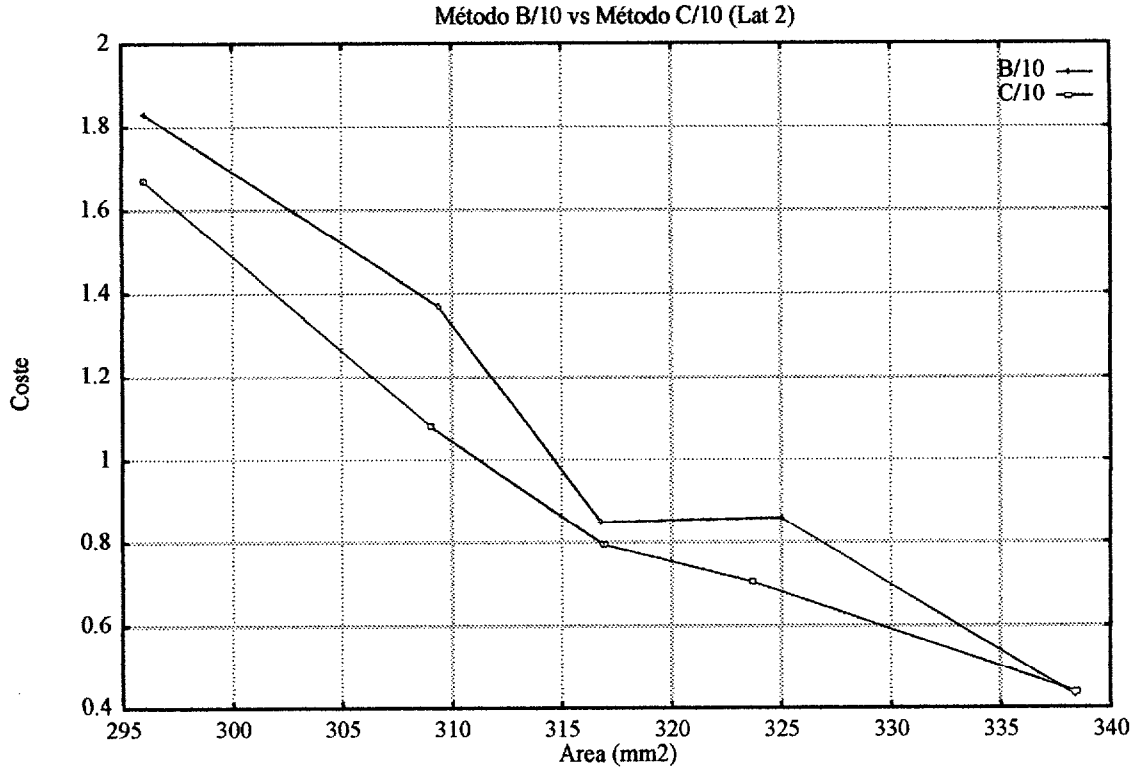


Figura 7.20. Calidad de los resultados.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

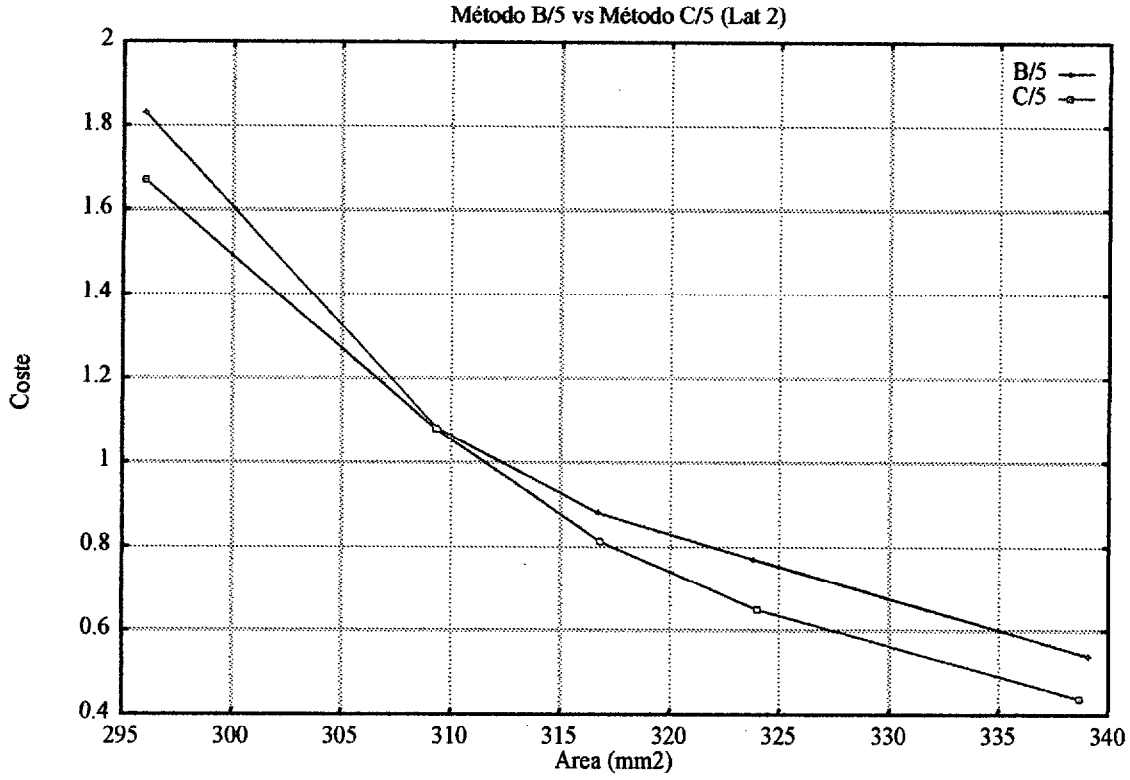


Figura 7.21. Calidad de los resultados.

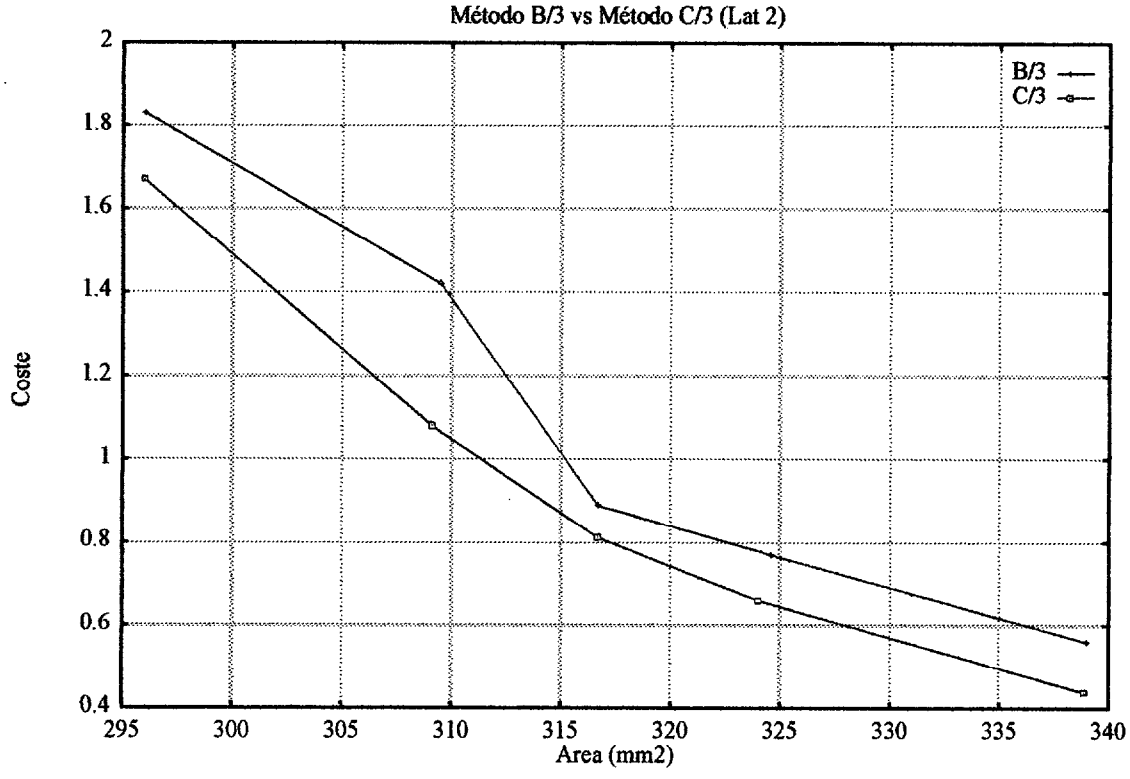


Figura 7.22. Calidad de los resultados.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

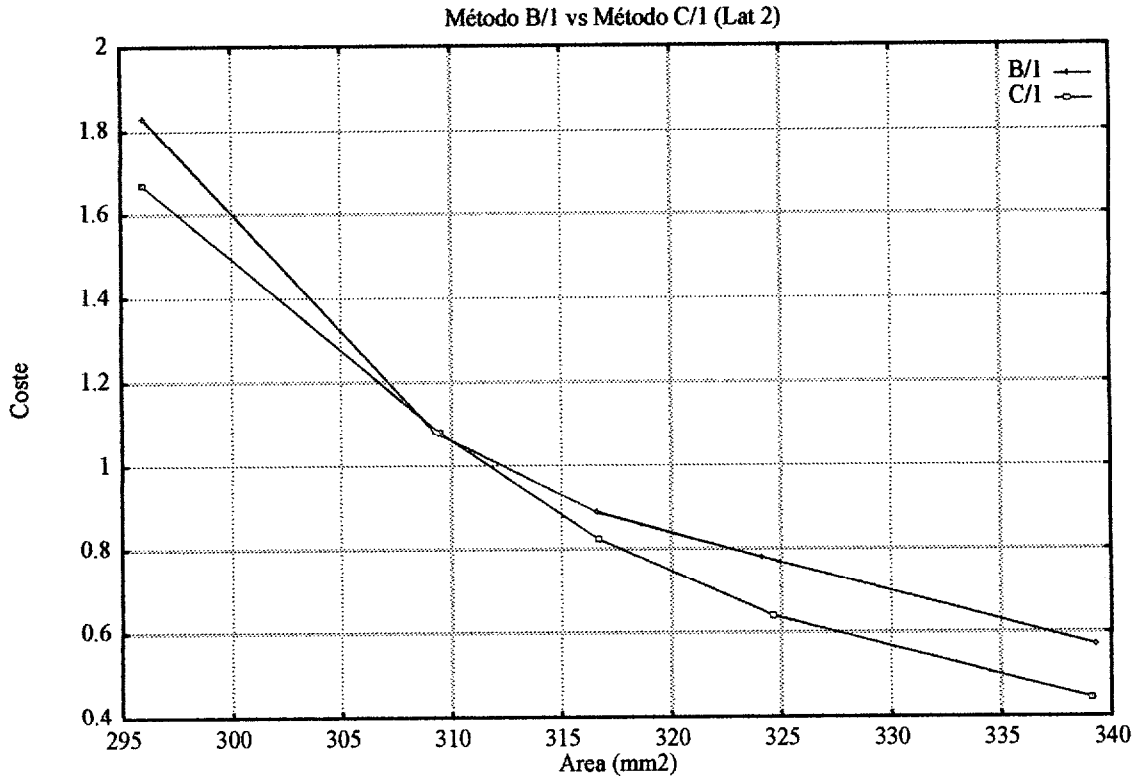


Figura 7.23. Calidad de los resultados.

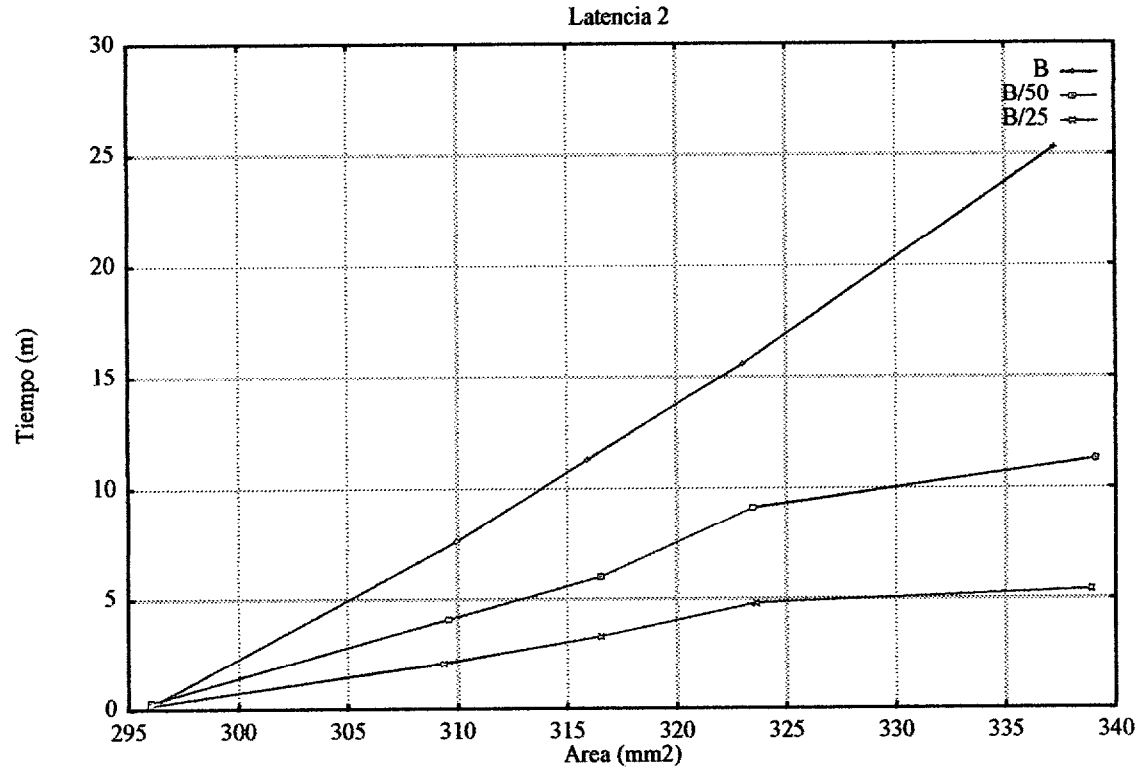


Figura 7.24. Calidad de los resultados.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

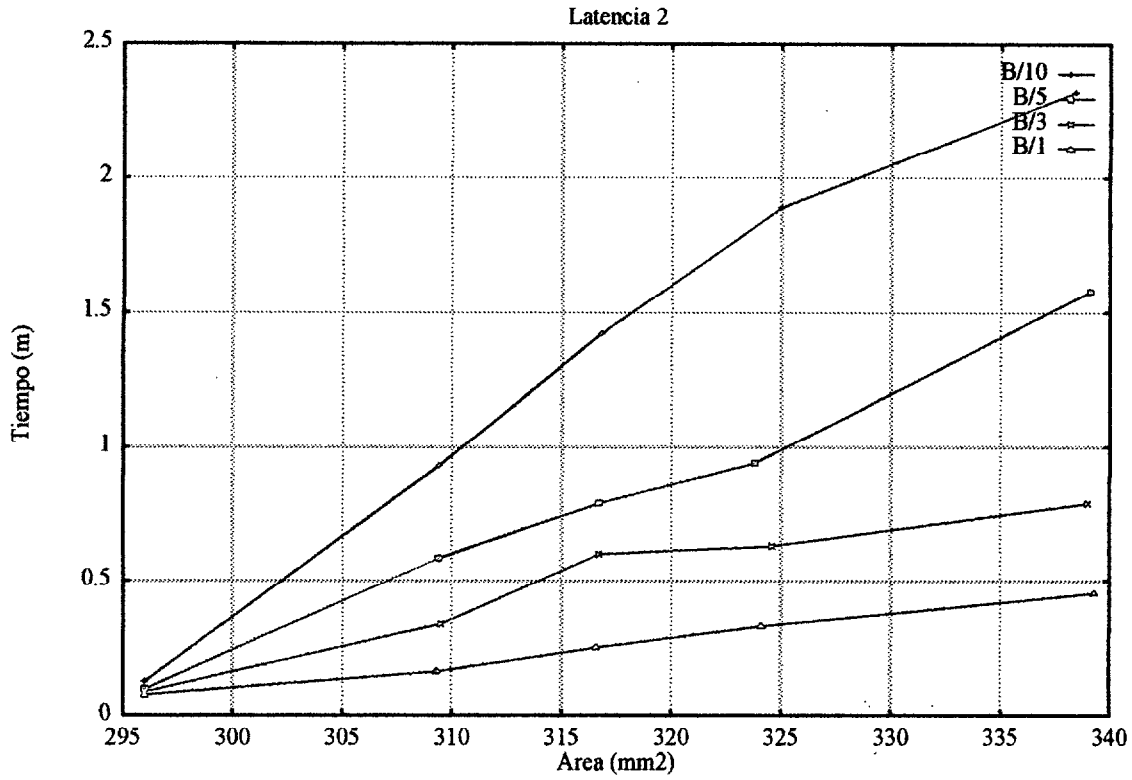


Figura 7.25. Calidad de los resultados.

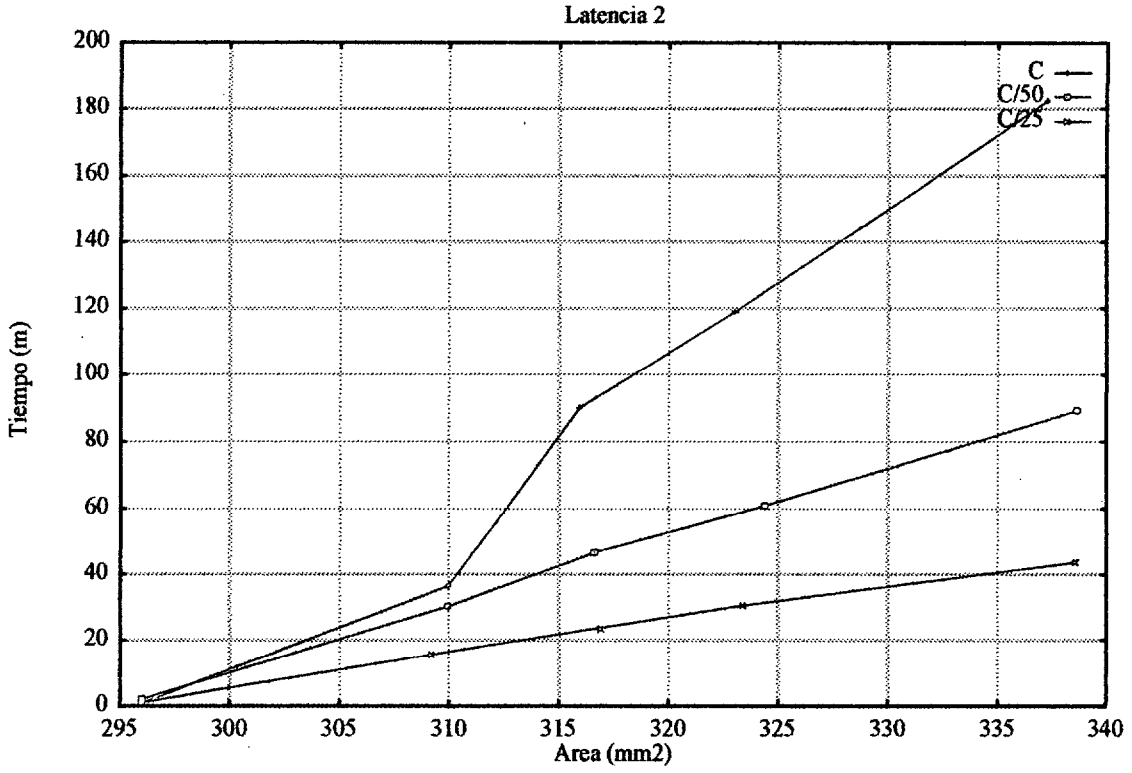


Figura 7.26. Calidad de los resultados.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

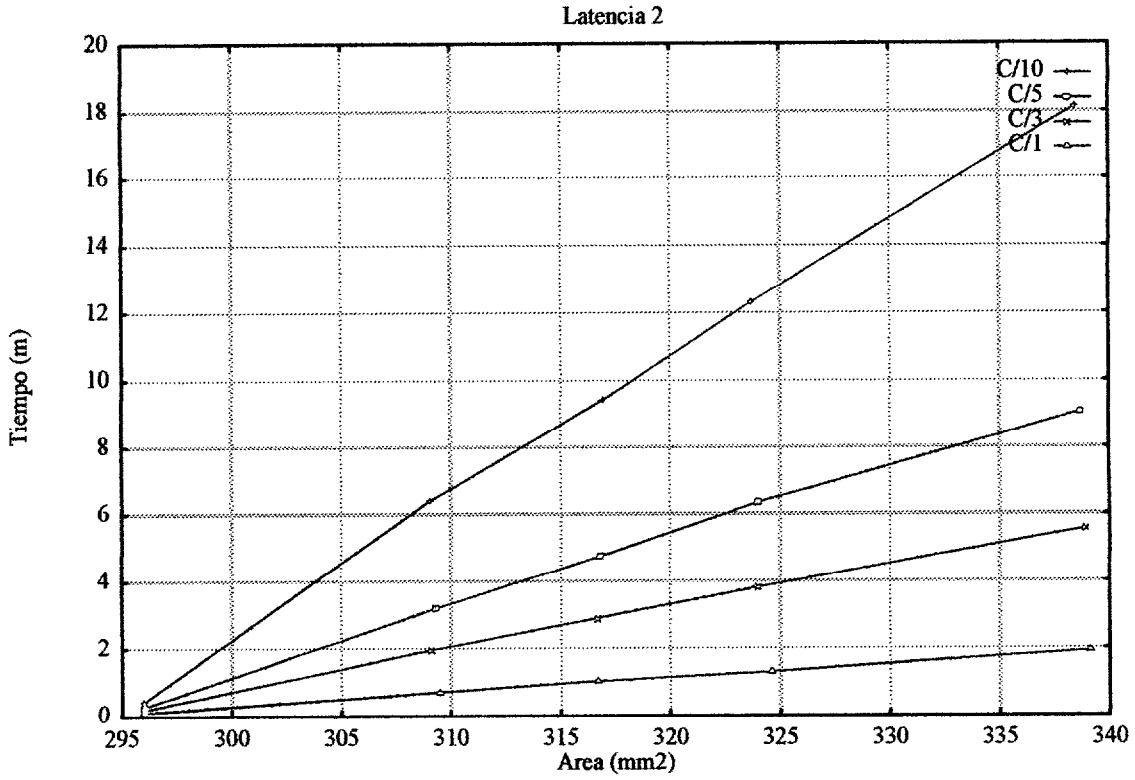


Figura 7.27. Calidad de los resultados.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

Capítulo 8

Conclusiones y trabajo futuro

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

8.1. Conclusiones

La tecnología de fabricación de circuitos integrados permite el desarrollo de sistemas empotrados cada vez más complejos. El aumento de la complejidad dificulta el diseño de los sistemas, apareciendo algunos problemas como los siguientes:

- Se incrementa la aparición de errores al realizarse la integración final de hardware y software. La causa principal de este problema es que el proceso de diseño se realiza a partir de una especificación informal del sistema. Este problema genera ciclos de rediseño que aumentan el coste y el tiempo de diseño de los sistemas.
- El espacio de diseño crece de forma considerable, y por tanto es muy poco probable que el particionado hardware/software realizado manualmente por un diseñador sea óptimo.

En los últimos años, las metodologías de codiseño hardware/software han aportado algunas soluciones para resolver estos problemas. Entre ellas se pueden mencionar la siguientes:

- Se han propuesto lenguajes de especificación de sistemas que permiten la simulación y verificación conjunta del hardware y software de los diseños.
- Se han propuesto métodos automáticos o semi-automáticos para optimizar el particionado hardware/software.

Por lo que a lenguajes de especificación respecta, VHDL [Vhd93] es uno de los lenguajes más utilizados [Sri92][Ern93][Ele94][The94][Eck97]. Las razones principales son las siguientes: a) soporta la mayoría de las características para la especificación de sistemas, tales como jerarquía, concurrencia, comunicación, sincronización, temporización y manejo de excepciones, b) soporta el diseño *top-down*, c) soporta las diversas tareas de codiseño, y d) es un lenguaje estándar para el que se han desarrollado muchas herramientas de diseño.

Sin embargo, VHDL tiene algunos problemas: a) la jerarquía de comportamiento es limitada, b) el manejo de excepciones, la jerarquía de comportamiento, la temporización y las comunicaciones necesitan especificaciones complejas, extensas y tediosas, y c) no soporta variables compartidas. Estos problemas aumentan la complejidad de las especificaciones, y por tanto aumenta el tiempo de diseño.

Por lo que a trabajos de particionado hardware/software respecta, los entornos de codiseño actuales presentan algunas carencias: en primer lugar, dado un grafo DAG

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

(*Directed Acyclic Graph*) de procesos a particionar, los principales trabajos publicados no soportan la ejecución segmentada del mismo [Kal94][Ben96][Wol96]. Otros trabajos, como [Car96], mencionan la segmentación como trabajo futuro, o bien proponen la segmentación a nivel interno de un proceso (*software pipelining*) [Bak97]. En segundo lugar, la potencia consumida por el diseño no es considerada entre los parámetros utilizados para optimizar el particionado. Sin embargo, se trata de un parámetro cada vez más importante debido a la expansión de sistemas portátiles y a la densidad de integración de los circuitos integrados. Estas dos carencias limitan el espacio de diseño, y por tanto la posibilidad de obtener particiones más optimizadas.

El objetivo principal de este trabajo ha sido solucionar los problemas anteriores, mejorando así dos aspectos importantes de las metodologías de codiseño hardware/ software: los lenguajes de especificación y el particionado hardware/software. En particular, en este trabajo se han cubierto los siguientes objetivos:

- Se ha diseñado un lenguaje de especificaciones denominado VSS (*VHDL-based System Specification*) que resuelve los problemas de VHDL mencionados anteriormente. La consecución de este objetivo contribuye principalmente a obtener especificaciones más sencillas y disminuir el tiempo de diseño.
- Se ha diseñado un método automático de particionado hardware/software que soporta la ejecución segmentada de grafos de procesos y que considera la potencia consumida por el diseño. La consecución de este objetivo contribuye a explorar un mayor espacio de diseño, es decir, el diseñador puede realizar un mayor número de compromisos hardware/software. De esta forma puede obtener particiones más óptimas.

Para un desarrollo integral de estos objetivos, en este trabajo se ha desarrollado el núcleo principal de un entorno de codiseño denominado GACSYS (*GAC's Codesign System*). Las tareas principales que se han llevado a cabo para alcanzar nuestros objetivos son las siguientes:

- Se ha diseñado el lenguaje de especificaciones VSS que aporta las siguientes características: a) extiende la jerarquía de comportamiento de VHDL, b) soporta especificaciones sencillas para el manejo de excepciones, la jerarquía de procesos, la temporización y las comunicaciones, y c) soporta variables compartidas. Además, se ha desarrollado un compilador que genera código VHDL a partir de VSS.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

- Se ha diseñado una representación interna basada en un grafo de flujo de datos denominado ASCIS. En este trabajo se ha ampliado dicho grafo para soportar especificaciones VHDL multiproceso. Además, se ha desarrollado un compilador que genera el grafo a partir de una especificación VHDL.
- Se ha diseñado e implementado un método automático de particionado hardware/software que permite una mejor exploración del espacio de diseño. Este método soporta la segmentación a nivel de procesos y tiene en cuenta la potencia consumida por el diseño. Para su implementación se ha desarrollado, por un lado, una herramienta de evaluación de las estimaciones de diversos parámetros relativos a la implementación de un sistema, tales como tiempo de ejecución, área y potencia. Por otro lado, para llevar a cabo el particionado hardware/software se han desarrollado algoritmos heurísticos de optimización que soportan la segmentación de los grafos de procesos.

8.2. Trabajo futuro

En este apartado presentamos aquellas líneas de trabajo que bajo nuestro punto de vista sería interesante abordar como continuación del trabajo que hemos realizado. Los trabajos que proponemos abordan las dos áreas de codiseño en las que hemos realizado nuestras aportaciones: a) los lenguajes de especificación, y b) el particionado hardware/ software.

Por lo que a lenguajes de especificación respecta, la experiencia ganada a través de los años en el diseño de lenguajes indica que se trata de un proceso que nunca finaliza, ya que constantemente aparecen nuevas necesidades. Con el diseño de VSS ha sucedido lo mismo. La complejidad de los sistemas empotrados está alcanzando un nivel tan elevado que el mantenimiento y la reutilización de los diseños es cada vez más difícil.

Las técnicas orientadas a objetos han demostrado en el diseño de software que son adecuadas para abordar estos problemas. Por tanto, sería interesante modificar VSS para soportar especificaciones orientadas a objetos. De hecho, existen algunos trabajos recientes que proponen estas técnicas. Entre las propuestas más relevantes podemos citar a los lenguajes *Objective VHDL* [Rad98][eis99] y *SUAVE* (*SAVANT and University of Adelaide VHDL Extensions*) [Ash98][Ash99]. *Objective VHDL* propone una extensión de VHDL utilizando el concepto de clase para tipos y entidades de diseño. *SUAVE* propone una extensión de VHDL basada en las características de ADA'95 [Whe96]. Otro trabajo que propone VHDL y ADA para especificar sistemas es [Lop98]. También existen algunas propuestas que no están basadas en VHDL. Por ejemplo, en

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

[Kuh98][You98] se propone el lenguaje Java [Gos96]. Existe un grupo de trabajo denominado *OOVHDL* (*Object-Oriented VHDL*) [eda99a] cuyo objetivo es obtener una propuesta que permita estandarizar una ampliación de VHDL orientada a objetos.

Finalmente, cabe mencionar que existen algunos grupos de trabajo tales como *SDDL* (*System Design and Description Languages*) [eda99b] y *SLDL* (*System Level Design Language*) [fdl98] que trabajan activamente en el diseño de un lenguaje estándar para la especificación de sistemas.

Por lo que al particionado hardware/software respecta, los trabajos que consideramos sería interesante abordar son los siguientes: a) generar código para mejorar las estimaciones software, b) soportar especificaciones con comunicaciones no desacopladas, y c) considerar la utilización de arquitecturas basadas en varios procesadores.

En primer lugar, las estimaciones software de nuestro entorno de codiseño son sencillas, por lo que sería interesante obtener estimaciones de mayor calidad mediante la generación del código software que tiene que ejecutar el procesador. Para procesadores de propósito general existen compiladores disponibles que son eficientes. Sin embargo, estos compiladores no son adecuados para procesadores DSP, ya que estos últimos tienen *data-path* irregulares y disponen de pocos registros. Por tanto, se necesitan compiladores específicos para generar código para procesadores DSP. De hecho, recientemente se han publicado algunos trabajos al respecto [Ara98][Lia98].

En segundo lugar, nuestro método de particionado soporta comunicaciones desacopladas de los cálculos. Sin embargo, una parte importante de los sistemas empotrados están orientados a sistemas de control en los cuales las comunicaciones no están desacopladas, ya que los procesos pueden comunicarse en cualquier instante de tiempo. Por tanto, para extender la utilización de nuestro entorno a este tipo de aplicaciones sería interesante modificar la obtención de estimaciones para soportar este tipo de comunicaciones. Este objetivo no es sencillo debido a los tiempos de espera que se producen en la sincronización entre procesos. Algunos trabajos recientes [Mat98][Vec98] abordan este problema, aunque siguen utilizando comunicaciones restringidas.

Finalmente, nuestro trabajo supone una arquitectura basada en un único procesador. La experiencia en el diseño de procesadores nos indica que el aumento de *performance* que se obtiene mediante paralelismo de grano fino (por ejemplo, añadiendo unidades funcionales) está limitado. Por tanto, sería interesante soportar arquitecturas con varios procesadores. Algunos trabajos como [Hsi98][Kar98] abordan este tipo de arquitectura,

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

en la que aparecen nuevos factores de diseño a tener en cuenta, tales como el número de procesadores, la red de interconexión entre los mismos, la distribución del código y los datos entre los procesadores y las memorias, respectivamente, etc. La consecuencia principal es que el espacio de diseño crece de tal manera que la exploración del mismo consume tiempos de ejecución enormes aunque se utilicen algoritmos heurísticos. Nosotros pensamos que una forma de abordar este problema sería la utilización de máquinas paralelas para llevar a cabo la exploración.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

Bibliografía

[Agr97] S. Agrawal, R.K. Gupta. *Data-flow Assisted Behavioral Partitioning for Embedded Systems*. Proceedings of the Design Automation Conference, June 1997, pp. 709-712.

[Aho86] A.V. Aho, R. Sethi, J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1986.

[Alu93] R. Alur, C. Courcoubetis, D. Hill. *Model-Checking in Dense Real Time*. Information and Computation, Vol. 104, N° 1, May 1993, pp. 2-34.

[And91] G.R. Andrews. *Concurrent Programming: Principles and Practice*. Benjamin/Cummings, 1991.

[Ara98] G. Araujo, S. Malik. *Code Generation for Fixed-Point DSPs*. ACM Transactions on Design Automation of Electronic Systems, Vol. 3, N° 2, April 1998, pp. 136-161.

[Ard95] L. Arditi and H. Collavizza. *Towards verifying VHDL descriptions of processors*. In European Design Automation Conference with EURO-VHDL, 1995.

[Ash98] P.J. Ashenden, P.A. Wilsey, D.E. Martin. *SUAVE: Object-Oriented and Genericity Extensions to VHDL for High-Level Modeling*. Forum on Design Languages, FDL'98, Lausanne, Switzerland, pp. 109-118.

[Ash99] P. J. Ashenden and P. A. Wilsey. *Principles for Extensions to VHDL for High-Level Modeling*. VLSI Design, pendiente de publicación.

[Ath93] P. Athanas, H.F Silverman. *Processor Reconfiguration Through Instruction-Set Metamorphosis*. IEEE Computer, Vol. 26, N° 3, March 1993, pp. 11-18.

[Bak97] S. Bakshi, D. Gajski. *Hardware/Software Partitioning and Pipelining*. Proceedings of the Design Automation Conference, June 1997, pp. 713-716.

[Bal96] A. Balboni, W. Fornaciari, D. Sciuto. *Partitioning and Exploration Strategies in the TOSCA Co-Design Flow*. Proceedings of the International Workshop on Hardware/Software Codesign, 1996, pp. 62-69.

[Bal97] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, B. Tabbara. *Hardware-Software Codesign of Embedded Systems: The POLIS Approach*. Kluwer Academic Publishers, 1997.

[Bar94] E. Barros, A. Sampaio. *Towards Provably Correct Hardware/Software Partitioning Using Occam*. Proceedings of the International Workshop on Hardware/Software Codesign, 1994, pp. 210-217.

[Bec96] J. Becker, R.W. Hartenstein, R. Kress. *Two-level Partitioning of Image Processing Algorithms for the Parallel Map-oriented Machine*. Proceedings of the International Workshop on Hardware/Software Codesign, 1996, pp. 77-84.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

[Ben91] A. Benveniste, G. Berry. *The Synchronous Approach to Reactive and Real-Time Systems*. Proceedings IEEE, Vol. 79, Nº 9, 1991, pp. 1270-1282.

[Ben93] T. Benner, J. Henkel, R. Ernst. *Internal Representation of Embedded Hardware/Software Systems*. Proceedings of the Second International Workshop on Hardware/Software Codesign, Geneva, 1993.

[Ben96] A. Bender. *MILP Based Task Mapping for Heterogeneous Multiprocessor Systems*. Proceedings of the EURODAC, September 1996, pp. 190-197.

[Ber91] G. Berry, P. Couronné, G. Gonthier. *The synchronous approach to reactive and real-time systems*. IEEE Proceedings, 79, September, 1991.

[Bha94] S. Bhattacharya, S. Dey, F. Brglez. *Performance analysis and optimization of schedules for conditional and loop-intensive specifications*. Proceedings of the DAC, 1994.

[Bra91] R.K. Brayton, M. Chiodo, R. Hojati, T. Kam, K. Kodandapani, R.P. Kurshan, S. Malik, A.L. Sangiovanni-Vincentelli, E.M. Sentovich, T. Shiple, H.Y. Wang. *BLIF-MV: an interchange format for design verification and synthesis*. Tech. Report UCB/ERL M91/97, University of California at Berkeley, November, 1991.

[Bre95] Breuer, P., Sánchez, L. and Kloos, C.D. *A simple denotational semantics, proof theory and validation condition generator for unit delay VHDL*. Formal Methods in System Design, 1995.

[Buc93] J.T. Buck. *The Ptolemy Kernel*. Memorandum UCB/ERL M93/8, University of California at Berkeley, January, 1993.

[Buc94] J. Buck, S. Ha, E.A. Lee, D.G. Masserschmitt. *Ptolemy: a framework for simulating and prototyping heterogeneous systems*. Journal of Computer Simulation, special issue on Simulation Software Development, January 1994.

[Cal93] J. P. Calvez. *Embedded Real-Time Systems. A Specification and Design Methodology*. John Wiley & Sons, 1993.

[Car96] C. Carreras, J.C. López, M.L. López, C. Delgado-Kloos, N. Martínez, L. Sánchez. *A Co-Design Methodology Based on Formal Specification and High-level Estimation*. Proceedings of the International Workshop on Hardware/Software Codesign, 1996, pp. 28-35.

[Car98] A. Carpenter, N. Messer. *The use of VHDL+ in the Specification Level Modelling of an Embedded System*. Forum on Design Languages, FDL'98, Lausanne, Switzerland.

[Cas94] Juan F. Pérez Castellano, A. Suárez, Luis M. Benítez. *Codiseño Hardware/ Software para Algoritmos con Bucles Vectoriales*. Reunión Técnica del Grupo de Usuarios de VHDL, Maspalomas, Gran Canaria, Noviembre, 1994.

[Cas96] J. F. Pérez Castellano, Álvaro Suárez, J. C. Bordón. *Representaciones Intermedias para Especificaciones Concurrentes basadas en VHDL*. VII Jornadas de Paralelismo, Santiago de Compostela, 11-13 Septiembre, 1996, pp.377-395.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

[Cas97a] J. F. Pérez Castellano, Álvaro Suárez, J. C. Bordón. *Intermediate Representations of Concurrent VHDL-based Specifications*. Proceedings of 5th EUROMICRO Workshop on Parallel and Distributed Processing, January 22nd-24th, 1997, London, United Kingdom, pp. 66-71. ISBN 0-8186-7770-8.

[Cas97b] J. F. Pérez Castellano, O. Cazorla, J. C. Bordón, Álvaro Suárez. *A VHDL-based Hardware/Software Codesign Approach.* VHDL User's Forum in Europe, SIG-VHDL Spring'97 Working Conference, Toledo, Spain, April, 1997, pp. 159-170. ISBN 84-8102-158-X.

[Cas97c] Juan P. Castellano, Mario Marrero, Álvaro Suárez, Onassis Cazorla, Juan C. Bordón. *Un lenguaje concurrente para el codiseño de sistemas complejos*. V Jornadas de Concurrencia, Vigo, Junio 1997, pp. 193-204. ISBN 84-8158-065-1.

[Cas98a] Juan P. Castellano, David Sánchez, Onassis Cazorla, Juan C. Bordón, Álvaro Suárez, Elsa Macías, Mario Marrero. *Codiseño de sistemas desde especificaciones concurrentes basadas en VHDL.* VI Jornadas de Concurrencia, Pamplona, Junio 1998, pp. 53-64.

[Cas98b] Juan P. Castellano, David Sánchez, Onassis Cazorla, Juan Bordón, Álvaro Suárez, Elsa Macías, Mario Marrero. *GACSYS: a VHDL-based Hw/Sw Codesign Tool.* IV Jornadas de Informática, Las Palmas de G.C., 13-17 Julio, 1998, pp. 163-172. ISBN 84-87526-61-6.

[Cas98c] Juan P. Castellano, David Sánchez, Onassis Cazorla, Juan Bordón, Álvaro Suárez. *GACSYS: a VHDL-based Hw/Sw Codesign Tool.* The Second International Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS'98). Szczyrk, Poland, September (2nd - 4th), 1998, pp. 293-299. ISBN 83-908409-6-0.

[Cas98d] O. Cazorla, J.P. Castellano. *Diseño de un lenguaje de especificación de sistemas y desarrollo de un compilador para su traducción a VHDL.* Proyecto Fin de Carrera, ETSIT, ULPGC, Junio 1998.

[Cas98e] J. Bordón, J.P. Castellano. *Desarrollo de un compilador para representar descripciones VHDL mediante un grafo de flujo de datos*. Proyecto Fin de Carrera, ETSIT, ULPGC, Julio 1998.

[Cas99] D. Sánchez, J.P. Castellano. *Herramienta de Particionado Hardware/ Software para Codiseño de Sistemas*. Proyecto Fin de Carrera, ETSIT, ULPGC, Enero 1999.

[Cha92a] A. Chandrakasan, A. Sheng, R.W. Brodersen. *Lower-Power CMOS Design.* IEEE Journal of Solid-State Circuits, 1992, pp. 472-484.

[Cha92b] A.P. Chandrakasan, M. Potkonjak, J. Rabaey, R.W. Brodersen. *HYPER-LP: A system for power minimisation using architectural transformations.* Proceedings of the International Conference on Computer-Aided Design, November 1992, pp. 300-303.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

[Cha95] A.P. Chandrakasan, M. Potkonjak, R. Mehra, J. Rabaey, R.W. Brodersen. *Optimizing power using transformations*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 14, n° 1, January 1995, pp. 12-31.

[Chi93a] M. Chiodo, et al. *A Formal Specification Model for Hardware/Software Codesign*. Tech. Report UCB/ERL M93/18, University of California at Berkeley, June, 1993.

[Chi93b] M. Chiodo, P. Giusto, A. Jurecska, L. Lavagno, H. Hsieh, A. Sangiovanni-Vincentelli. *A Formal Specification Model for Hardware/Software Codesign*. Handouts of International Workshop on Hardware-Software Codesign, Cambridge, Massachusetts, October, 1993.

[Cli94] Clifford Liem, Trevor May, Pierre Paulin. *Instruction-Set Matching and Selection for DSP and ASIP Code Generation*. EDAC'94, pp. 31-37.

[DAm94] J.G. D'Ambrosio, X. Hu. *Configuration-Level Hardware/Software Partitioning for Real-Time Embedded Systems*. Proceedings of the International Workshop on Hardware-Software Codesign, 1994, pp. 34-41.

[Deh95] D. Deharbe and D. Borrione. *A qualitative finite subset of VHDL and semantics*. In Correct Hardware Design and Verification Methods, Volume 987, Lecture Notes in Computer Science. Springer-Verlag, New York, 1995.

[DeM90] G. De Micheli, D.C. Ku, F. Mailhot, T. Truong. *The Olympus Synthesis System*. IEEE Design & Test of Computers, Vol. 7, N° 5, October 1990, pp. 37-53.

[Dij75] E.W. Dijkstra. *Guarded Commands, nondeterminancy and formal derivation of programs*. Communications of the ACM 18 (8), August 1975, pp. 453-457.

[Don91] C. Donnelly, R. Stallman. *BISON: The Yacc-compatible Parser Generator*. Free Software Foundation, 1991.

[Dou97] C. Dou. *Integration of SDL and VHDL for HW/SW Codesign of Communication Systems*. EUROMICRO'97, Budapest, Hungary.

[Dow95] K. A. Dowsland. *Modern Heuristic Techniques for Combinatorial Problems*. Ed. by C. R. Reeves, McGraw-Hill, 1995, ISBN 0-07-709239-2.

[Dru89] D. Drusinsky, D. Harel. *Using Statecharts for Hardware Description and Synthesis*. IEEE Trans. on Computer-Aided Design, Vol. 8, N° 7, July 1989, pp. 798-807.

[Eck93] W. Ecker. *Using VHDL for HW/SW Co-Specification*. Proceedings of the EURODAC'93, pp. 500-505.

[Eck97] W. Ecker. *Behavioral and System-Level VHDL*. Tutorial at CHDL'97/VHDL-FORUM'97, Toledo, Spain, April 1997.

[eda99a] http://www.eda.org/oovhdl/

[eda99b] http://www.eda.org/sddl/

[Eij91] J.T.J. van Eijndhoven, G.G. de Jong, L.Stok. *The ASCIS Data Flow Graph: Semantics and Textual Format*. Eindhoven University of Technology, 1991.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

[Eij92] J.T.J. van Eijndhoven, L. Stok. *A Data Flow Graph Exchange Standard*, EURODAC, 1992,pp. 193-199.

[eis99] http://eis.informatik.uni-oldenburg.de/publications/index.html

[Ele94] P. Eles, Z. Peng, A. Doboli. *VHDL System-Level Specification and Partitioning in a Hardware/Software Co-Synthesis Environment.* Proceedings of the International Workshop on Hardware-Software Codesign, 1994, pp. 49-55.

[Ell91] S.C. Ellis. *Power management in notebook PC's.* Proceedings of the Silicon Valley Personal Computing Conference, 1991, pp. 749-754.

[Ern93] R. Ernst, J. Henkel, T. Benner. *Hardware/Software Cosynthesis for Microcontrollers.* IEEE Design and Test of Computers, December 1993, pp. 64-75.

[Est87] International Standard, ESTELLE (Formal Description Technique Based on an Extended State Transition Model) ISO/DIS 9074, 1987.

[fdl98] http://c3iwww.epfl.ch/fdl98/

[Fig95] R.J.O. Figueiredo, I.S. Bonatti. *An Algorithm for the Translation of SDL into Synthesizable VHDL.* In High-Level System Modeling: Specification Languages (ed. J.M. Berge, O.Levia, J. Rouillard), Kluwer Academic Publishers, 1995, pp. 99-109.

[Gaj92] D. Gajski, N. Dutt, A. Wu, S. Lin. *High-Level Synthesis: Introduction to Chip and System Design.* Kluwer Academic Publishers, 1992.

[Gaj93] D. Gajski, F. Vahid, S. Narayan. *SpecCharts: A VHDL Front-End for Embedded Systems.* University of California, Irvine, TR 93-31, 1993.

[Gaj94] D. Gajski, F. Vahid, S. Narayan. *A System-Design Methodology: Executable-Specification Refinement.* Proceedings of EDAC'94, 1994, pp. 458-463.

[Gaj95] D. Gajski, F. Vahid. *Specification and Design of Embedded Hardware-Software Systems.* IEEE Design & Test of Computers, 1995, pp. 53-67.

[Gar79] M.R. Garey, D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman and Company, 1979.

[Goo95] Goossens, K.G.W. *Reasoning about VHDL using operational and observational semantics.* Advanced Research Workshop on Correct Hardware Design Methodologies, ESPRIT CHARME, Springer Verlag, 1995.

[Gos96] J. Gosling, B. Joy, G. Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[Gup92a] R.K Gupta, G. De Micheli. *System-level Synthesis Using Re-programmable Components.* Proceedings of the Third European Conference on Design Automation, 1992, pp. 2-7.

[Gup92b] R.K Gupta, C.N. Coelho, G. De Micheli. *Synthesis and Simulation of Digital Systems Containing Interactive Hardware and Software Components.* Proceedings of the 29th Design Automation Conference, 1992, pp. 225-230.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

[Gup93] R.K Gupta, G. De Micheli. *Hardware-Software Cosynthesis for Digital Systems*. IEEE Design & Test of Computers, September 1993, pp. 29-41.

[Gup94] R.K Gupta, C.N. Coelho, G. De Micheli. *Program Implementation Schemes for Hardware-Software Systems*. IEEE Computer, January 1994, pp. 48-55.

[Gup95] R.K Gupta. *Co-Synthesis of Hardware and Software for Digital Embedded Systems*. Kluwer Academic Publishers, 1995.

[Har87] D. Harel. *Statecharts: A Visual Formalism for Complex Systems*. Science of Computer Programming 8, 1987.

[Har88] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtul-Trauring. *Statemate: A Working Environment For The Development Of Complex Reactive Systems*. Proceedings of the International Conference on Software Engineering, IEEE Computer Society Press, 1988, pp. 396-406.

[Har90] D. Harel et al. *Statecharts: A Working Environment For The Development Of Complex Reactive Systems*. IEEE Trans. on Software Engineering, Vol. 16, April 1990.

[Har96] W. Hardt, W. Rosenstiel. *Speed-Up Estimation for Hw/Sw Systems*. Proceedings of the International Workshop on Hardware/Software Codesign, March 1996, pp. 36-43.

[Hen97] J. Henkel, R. Ernst. *A Hardware/Software Partitioner using dynamically determined Granularity*. Proceedings of the Design Automation Conference, June 1997, pp. 691-696.

[Her94] D. Herman, J. Henkel, R. Ernst. *An Approach to the Adaptation of Estimated Cost Parameters in the Cosyma System*. Proceedings of Third International Workshop on Hardware/Software Codesign, Grenoble, France, 1994, pp. 100-107.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[Hoc97] D.S. Hochbaum. *Approximation Algorithms for NP-hard Problems*. PWS Publishing Company, 1997.

[Hsi98] P.A. Hsiung, C.H. Chen, T.Y. Lee, S.J. Chen. *ICOS: An Intelligent Object-Oriented Synthesis Methodology for Multiprocessor Systems*. ACM Transactions on Design Automation of Electronic Systems, Vol. 3, N° 2, April 1998, pp. 109-135.

[Icl99] http://www.icl.com/da/svdocs/vhdlplus.htm

[Ism94] T.B. Ismail, K. O'Brien, A. Jerraya. *Interactive System-level Partitioning with PARTIF*. EDAC'94, 1994, pp. 464-468.

[Ism95] T.B. Ismail, A.A. Jerraya. *Synthesis Steps and Design Models for Codesign*. IEEE Computer Magazine, February 1995, pp. 44-52.

[Jan94] A. Jantsch, P. Ellervee, J. Öberg, A. Hermani, H.Tenhunen. *Hardware/Software partitioning and minimizing memory interface traffic*. Proceedings of the EURODAC, 1994.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

[Jer92] A. Jerraya, K. O'Brien. *SOLAR: An Intermediate Format for System-Level Design and Specification.* Workshop on Hardware/Software Co-Design, Germany, 1992.

[Jer95] A. Jerraya, K. O'Brien. *SOLAR: An Intermediate Format for System-Level Modeling and Synthesis.* Codesign: Computer-Aided Software/Hardware Engineering, J. Rozenblit, K. Buchenrieder (eds.), IEEE Press, 1995, pp. 145-175, ISBN: 0-7803-1049-7.

[Joh67] S.C. Johnson. *Hierarchical clustering schemes.* Psychometr., September 1967, pp. 241-254.

[Joh97] M.C. Johnson, K. Roy. *Datapath Scheduling with Multiple Supply Voltages and Level Converters.* ACM Transactions on Design Automation of Electronic Systems, vol. 2, nº 3, July 1997, pp. 227-248.

[Kal93] A. Kalavade, A. Lee. *A Hardware-Software Codesign Methodology for DSP Applications.* IEEE Design & Test of Computers, Vol. 10, Nº 3, 1993, pp.16-28.

[Kal94] A. Kalavade, E. Lee. *A Global Criticality/Local Phase Driven Algorithm for the Constrained Hardware/Software Partitioning Problem.* Proceedings of the International Workshop on Hardware/Software Codesign, 1994, pp. 42-48.

[Kal97] A. Kalavade, E. Lee. *The Extended Partitioning Problem: Hardware/ Software Mapping, Scheduling and Implementation-bin Selection.* Design Automation for Embedded Systems, Kluwer Academic Publishers, Vol. 2, Nº 2, March 1997, pp. 125-163.

[Kar98] I. Karkowski, H. Corporaal. *Design Space Exploration Algorithm for Heterogeneous Multi-processor Embedded System Design.* Proceedings of the Design Automation Conference (DAC), June 1998, pp. 82-87.

[Ker70] B. Kernighan, S. Lin. *An efficient heuristic procedure for partitioning graphs.* Bell System Technical Journal, February, 1970.

[Kir83] S. Kirkpatrick, C. D. Gelatt, M. P. Vecchi. *Optimization by Simulated Annealing.* Science, Vol. 220, Nº 4598, pp. 671-680, May 1983.

[Klo95] C.D. Kloos, A.M. López, T. M. Moro, T.R. Valladares. *From LOTOS to VHDL. In High-Level System Modeling: Specification Languages* (ed. J.M. Berge, O.Levia, J. Rouillard), Kluwer Academic Publishers, 1995, pp. 111-140.

[Knu96] P.V. Knudsen, J. Madsen. *PACE: A Dynamic Programming Algorithm for Hardware/Software Partitioning.* Proceedings of the International Workshop on Hardware/Software Codesign, March 1996, pp. 85-92.

[Ku88] D.C. Ku, G. De Micheli. *Hardware C: A Language For Hardware Design.* Tech. Report. CSL-TR-88-362, Computer Systems Laboratory, Stanford University, 1988.

[Ku90] D. Ku, G. de Micheli. *HardwareC: A Language for Hardware Design.* Technical report CSL-TR-90-419, Stanford University, 1990.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

[Kuh98] T. Kuhn, W. Rosenstiel, U. Kebschull. *Object Oriented Hardware Modeling and Simulation Based on Java.* International Workshop on IP Based Synthesis and System Design, Grenoble, France, December, 1998.

[Kum93] S. Kumar, J.H. Aylor, B.W. Johnson, A. Wulf. *A Framework for Hardware/ Software Codesign*, IEEE Computer, December 1993, pp. 39-45.

[Kum96] S. Kumar, J.H. Aylor, B.W. Johnson, A. Wulf. *The Codesign of Embedded Systems. A Unified Hardware/Software Representation.* Kluwer Academic Publishers, 1996.

[Lag91] E. Lagnese, D. Thomas. *Architectural Partitioning for System Level Synthesis of Integrated Circuits.* IEEE Transactions on Computer-Aided Design, July 1991.

[Lan93] P.E. Landman, J.M. Rabaey. *Power estimation for high-level synthesis.* Proceedings EDAC, February 1993, pp. 361-366.

[Lev92] J.R. Levine, T. Mason, D. Brown. *Lex & Yacc.* O'Reilly and Associates, 1992.

[Lia98] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, A. Wang. *Code Optimization Techniques in Embedded DSP Microprocessors.* Design Automation of Embedded Systems (Kluwer Academic Publishers), Vol. 3, N° 1, January 1998, pp. 59-73.

[Lin97] Y.R. Lin, C.T. Hwang, A.C.H. Wu. *Scheduling Techniques for Variable Voltage Low Power Designs.* ACM Transactions on Design Automation of Electronic Systems, vol. 2, n° 2, April 1997, pp. 81-97.

[Lip93] R. Lipsett, C. Schaefer, C. Ussery. *VHDL: Hardware Description and Design.* Kluwer Academic Publishers, 1993.

[Lop98] A. Lòpez, M. Veiga, M. Gonzalez Harbour, E. Villar. *Embedded System Specification and Design using ADA and VHDL.* Forum on Design Languages, FDL'98, Lausanne, Switzerland.

[Lot89] *LOTOS, a formal description technique based on the temporal ordering of observational behavior*, ISO, IS 8807, Feb. 1989.

[Mac97] E. Macii, M. Pedram, F. Somenzi. *High-Level Power Modeling, Estimation and Optimization.* In Proceedings of the 34th DAC, June 1997, pp. 504-511.

[Mad97] J. Madsen, J. Grode, P.V. Knudsen, M.E. Petersen, A. Haxthausen. *LYCOS: the Lyngby Co-Synthesis System.* Design Automation for Embedded Systems, Kluwer Academic Publishers, Vol. 2, N° 2, March 1997, pp. 195-235.

[Mar95] R.S. Martin, J.P. Knight. *Power-Profiler: Optimizing ASICs Power Consumption at the Behavioral Level.* Proceedings of the DAC, 1995, pp.42-47.

[Mar96] R.S. Martin, J.P. Knight. *Optimizing Power in ASIC Behavioral Synthesis.* IEEE Design & Test of Computers, Low Power Series, Summer 1996, pp. 58-70.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

[Mar98] M. Marrero, E. Carrión, Elsa M. López, Juan P. Castellano, David Sánchez, Onassis Cazorla, Juan C. Bordón. *Una herramienta para la especificación gráfica de sistemas complejos*. IV Jornadas de Informática, Las Palmas de G.C., 13-17 Julio 1998, pp. 191-200. ISBN 84-87526-61-6.

[Mat98] A. Mathur, A. Dasdan, R.K. Gupta. *Rate Analysis for Embedded Systems*. ACM Transactions on Design Automation of Electronic Systems, Vol. 3, N° 3, July 1998, pp. 408-436.

[McF90] M. McFarland, T. Kowalski. *Incorporating bottom-up design into hardware synthesis*. IEEE Transactions on Computer-Aided Design, September 1990.

[Meh94] R. Mehra, J. Rabaey. *Behavioral level power estimation and exploration*. Proceedings of the International Workshop on Low-Power Devices, April 1994, pp. 197-202.

[Met53] N. Metropolis, A.W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, E. Teller. *Equation of state calculation by fast computing machines*. Journal of Chemistry Physics, 21, pp. 1087-1091, 1953.

[Mic92] P. Michel, U. Lauther, P. Duzy. *The Synthesis Approach to Digital System Design*. Kluwer Academic Publishers, 1992.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[Mon96] J. Monteiro, S. Devadas, P. Ashar, A. Mauskar. *Scheduling Techniques to Enable Power Management*. Proceedings of the DAC, June 1996, pp. 349-352.

[Nar90] S. Narayan, F. Vahid, D. Gajski. *Translating System Specification to VHDL*.

[Nar91] S. Narayan, F. Vahid, D. Gajski. *System Specification and Synthesis with the SpecCharts Language*. Proceedings of the International Conference on Computer-Aided Design (ICCAD), 1991.

[Nar92a] S. Narayan. *A Survey of System-Level Specification Languages*. Technical report 92-100, Octuber, 1992, Department of Information and Computer Science. University of California, Irvine.

[Nar92b] S. Narayan, F. Vahid, D. Gajski. *System Specification with the SpecCharts Language*. IEEE Design & Test of Computers, December 1992.

[Nar92c] S. Narayan, D. Gajski. *Area and Performance Estimation from System-Level Specifications*. University of California, Irvine, TR 92-16, 1992.

[Nar93] S. Narayan, D. Gajski. *Features Supporting System-Level Specifications in HDLs*. EURODAC'93, pp. 540-545.

[Nar96] S. Narayan, D. Gajski. *Rapid Performance Estimation for System Design*. Proceedings of the EURO-DAC, September 1996, pp. 206-211.

[Nie96] R. Niemann, P. Marwedel. *Hardware/Software Partitioning using Integer Programming*. Proceedings of the European Design and Test Conference, 1996, pp. 473-479.

[Nie97] R. Niemann, P. Marwedel. *An Algorithm for Hardware/Software Partitioning Using Mixed Integer Linear Programming*. Design Automation for

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

Embedded Systems, Kluwer Academic Publishers, Vol. 2, N° 2, March 1997, pp. 165-193.

[Obr92] K. O'Brien, I. Park, A.A. Jerraya. *Communication Modelling for the System-Level Synthesis of Mixed Hardware/Software Designs.* Proceedings EDAC'92, Brussels, Belgium, March, 1992, pp.

[Obr93] K. O'Brien, T. Ben Ismail, A.A. Jerraya. *A Flexible Communication Modelling Paradigm for System-Level Synthesis.* Handouts of International Workshop on Hardware-Software Co-Design, Cambridge, Massachusetts, October, 1993.

[Pat90] D. Patterson, J. Hennessy. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 1990.

[Par92] I. Park, K. O'Brien, A.A. Jerraya. *AMICAL: Architectural Synthesis Based on VHDL.* IFIP Workshop on Control-Dominated Synthesis from RTL Descriptons, Grenoble, Sept. 1992.

[Ped96] M. Pedram. *Power minimization in IC design: principles and applications.* ACM Transactions on Design Automation of Electronic Systems, vol. 1, n° 1, January 1996, pp. 3-56.

[Pen93] Z. Peng, K. Kuchcinski. *An Algorithm for Partitioning of Application Specific Systems.* Proceedings of the EDAC, 1993, pp. 316-321.

[Pou87] D. Pountain, D. May. *A Tutorial Introduction to OCCAM Programming.* INMOS, 1987.

[Rab91] J.M. Rabaey, C. Chu, P. Hoang, M. Potkonjak. *Fast Prototyping of Datapath-Intensive Architectures.* IEEE Design and Test of Computers, June 1991, pp. 40-51.

[Rab96] J.M. Rabaey. *Digital Integrated Circuits: A Design Perspective.* Prentice-Hall, 1996.

[Rab97] J.M. Rabaey, M. Pedram. *Low-Power Design Methodologies.* Kluwer Academic Publishers, 1997.

[Rad98] M. Radetzki, W. Putzke Röming, W. Nebel. *Objective VHDL: Tools and Applications.* Forum on Design Languages, FDL'98, Lausanne, Switzerland, pp. 191-200.

[Rag94] A. Raghunathan, N.K. Jha. *Behavioral Synthesis for Low Power.* Proceedings of the International Conference on Computer Design: VLSI in Computers and Processors, 1994, pp. 318-322.

[Rag95] A. Raghunathan, N.K. Jha. *An iterative improvement algorithm for low power datapath synthesis.* Proceedings of the International Conference on Computer-Aided Design, 1995, pp. 597-602.

[San98] L. Sánchez, G. Koch, N. Martínez, M. López, C. D. Kloos, W. Rosenstiel. *Hardware-Software Prototyping from LOTOS.* Design Automation for Embedded Systems (Kluwer Academic Publishers), Vol. 3, N° 2/3, March 1998, pp. 117-148.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

[Sch93] P.J. Schoenmakers, *vc1: a VHDL to ASCIS DFG compiler.* Master's Thesis. Department of Electrical Engineering, Eindhoven University of Technology, 1993.

[Sen96] O. Sentieys, D. Chillet, J.P. Diguet, J.L. Philippe. *Memory Module Selection for High Level Synthesis.* IEEE 1996, pp. 273-282. http://www.enssat.fr/RECHERCHE/ARCHI.

[Spe93] *Special Report: Platforms. The New Contender*s. IEEE Spectrum, December 1993, pp. 20-23.

[Sri92] M.B. Srivastava, R. W. Brodersen. *Using VHDL for High-Level Mixed-Mode System Simulation.* IEEE Design & Test of Computers, September 1992, pp. 31-40.

[Sri96] M. Srivas, A. Camilleri. *Formal Methods in Computer-Aided Design.* Lecture Notes in Computer Science, Vol. 1166, Springer, 1996.

[Tas93] van Tassel, J.P. *Femto-VHDL: The Semantics of a Subset of VHDL and its Embedding in the HOL Proof Assistant.* Ph. D. Dissertation, University of Cambridge, 1993.

[The94] M. Theibinger, P. Stravers, H. Veit. *Castle: An Interactive Environment for HW/SW Co-Design.* Proceedings of the International Workshop on Hardware/Software Codesign, 1994, pp. 203-209.

[The97] G. Theodoropoulos, J. Woods. *Occam: an Asynchronous Hardware Description Language.* EUROMICRO'97, Budapest, Hungary.

[Tur93] K.J. Turner. *Using Formal Description Techniques.* John Wiley & Sons, 1993.

[Vah91] F. Vahid, S. Narayan, D. Gajski. *SpecCharts: A Language for System Level Synthesis.* Proceedings of the International Symposium on Computer Hardware Description Languages (CHDL'91) and their Applications, 1991, pp. 145-154.

[Vah92a] F. Vahid, S. Narayan, D. Gajski. *Constant-Time Cost Evaluation for Behavioral Partitioning.* University of California, Irvine, TR 92-29, 1992.

[Vah92b] F. Vahid, D. Gajski. *Specification Partitioning for System Design.* Proceedings of the Design Automation Conference (DAC'92), 1992, pp. 219-224.

[Vah95] F. Vahid, D. Gajski. *SLIF: A specification-level intermediate format for system design.* Proceedings of the European Design & Test Conference (EDTC'95), 1995, pp. 185-189.

[Vah95b] F. Vahid, D. Gajski. *Clustering for improved system-level functional partitioning.* Proceedings of the 8th International Symposium on System Synthesis, France, 1995, pp. 28-33.

[Vah96] F. Vahid, T. dm Le. *Towards a model for hardware and software functional partitioning.* Proceedings of the Fourth International Workshop on Hardware-Software Co-Design, March 1996, pp. 116-123.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

[Vah97] F. Vahid, T. dm Le. *Extending the Kernighan/Lin Heuristic for Hardware and Software Functional Partitioning*. Design Automation for Embedded Systems, Kluwer Academic Publishers, Vol. 2, N° 2, March 1997, pp. 237-261.

[Vec98] G. de Veciana, M. Jacome, J.H. Guo. *Hierarchical Algorithms for Assesing Probabilistic Constraints on System Performance*. Proceedings of the Design Automation Conference (DAC), June 1998, pp. 251-256.

[Vhd93] *IEEE Standard VHDL Language Reference Manual. ANSI/IEEE Std* 1076-1993.

[Whe96] D.A. Wheeler. *Ada 95: the lovelace tutorial*. Springer, 1996.

[Wil92] Wilsey, P.A. *Developing a formal semantic definition of VHDL*. In: Mermet, J., eds, VHDL for Simulation, Synthesis and Formal Proofs of Hardware, Kluwer Academic Publishers, 1992, pp. 243-256.

[Win93] P. Windirsch, P. Duzy. *The CALLAS Synthesis System and its Applications to Mechatronic ASIC Design Problems*. EURODAC'93, pp. 278-283.

[Wol96] W. Wolf, J. Hou. *Process Partitioning for Distributed Embedded Systems*. Proceedings of the Fourth International Workshop on Hardware-Software Co-Design, March 1996, pp. 70-76.

[Wol97] P. Wolkensdorfer, J. Harangozò. *The Application of the Formal Description Language LOTOS for the Design of Process Control Systems*. EUROMICRO'97, Budapest, Hungary.

[Wye93] W. Ye, R. Ernst, T. Benner, J. Henkel. *Fast Timing Analysis for Hardware-Software Co-synthesis*. Proceedings of ICCD, 1993, pp. 452-457.

[Wyt95] J. Wytrebowicz, S. Budkowski. *Communication Protocols Implemented in Hardware: VHDL Generation from Estelle*. In High-Level System Modeling: Specification Languages (ed. J.M. Berge, O.Levia, J. Rouillard), Kluwer Academic Publishers, 1995, pp. 77-98.

[You98] J. S. Young, J. MacDonald, M. Shilman, A. Tabbara, P. Hilfinger, A.R. Newton. *Design and Specification of Embedded Systems in Java Using Successive, Formal Refinement*. Proceedings of the Design Automation Conference (DAC), June 1998, pp. 70-75.

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

Apéndice A

Sintaxis del lenguaje VSS

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital. 2004

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

```
fichero_entrada:
        design_file

design_file:
        design_unit
      | design_file design_unit

design_unit:
        library_unit
      | context_clause library_unit

context_clause:
        context_clause context_item
      | context_item

context_item:
        library_clause
      | use_clause

library_clause:
        library identifier_list ";"

identifier_list:
        identifier
      | identifier_list "," identifier

use_clause:
        use selected_name_list ";"

selected_name_list:
        sel_id_or_funct
      | selected_name_list "," sel_id_or_funct

library_unit:
        primary_unit
      | secondary_unit

primary_unit:
        entity_declaration
      | configuration_declaration
      | package_declaration

subprogram_declaration:
        subprogram_specification ";"

subprogram_specification:
        procedure designator
      | procedure designator "(" interface_list ")"
      | function designator "(" interface_list ")" return actual_name
      | function designator return actual_name

designator:
        identifier
      | string_literal
```

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

```
subprogram_body:
        subprogram_specification is subprogram_declarative_items
                seq_stat_subnovss subprogram_end ";"
      | subprogram_specification is
                seq_stat_subnovss subprogram_end ";"

seq_stat_sub:
        begin
      | begin sequence_of_statements

seq_stat_subnovss:
        begin
      | begin sequence_of_statementsnovss

subprogram_declarative_items:
        subprogram_declarative_item
      | subprogram_declarative_items subprogram_declarative_item

subprogram_declarative_item:
        package_body_declarative_item
      | variable_declaration
      | attribute_declaration
      | attribute_specification

subprogram_end:
        end
      | end subprogram_kind designator
      | end subprogram_kind
      | end designator

subprogram_kind:
        procedure
      | function

variable_declaration:
        variable identifier_list ":"  interface_init ";"

attribute_declaration:
        attribute identifier ":" actual_name ";"

package_body_declarative_item:
        basic_declarative_item
      | subprogram_body

basic_declarative_item:
        subprogram_declaration
      | type_declaration
      | subtype_declaration
      | constant_declaration
      | file_declaration
      | alias_declaration
      | use_clause

type_declaration:
        full_type_declar
      | incomp_type_declar
```

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

full_type_declar:
type identifier **is** type_definition ";"

type_definition:
scalar_type_def
| composite_type_def
| file_type_def

scalar_type_def:
enumerat_type_def
| range_constraint

enumerat_type_def:
"(" enumeration_literal_list ")"

enumeration_literal_list:
enumeration_literal
| enumeration_literal_list "," enumeration_literal

enumeration_literal:
identifier
| **character_literal**

composite_type_def:
array_type_definition
| record_type_definition

array_type_definition:
array "(" definition_list ")" **of** subtype_indication
| **array** "(" definition_list ")" **of** actual_name

rangelist:
range
| rangelist "," range

definition_list:
un_list
| co_list

un_list:
actual_name **range lg**
| un_list "," actual_name **range lg**

co_list:
range
| subtype_y_attribute
| actual_name
| co_list "," range
| co_list "," subtype_y_attribute
| co_list "," actual_name

range:
simple_expression **to** simple_expression
| simple_expression **downto** simple_expression
| actual_name "'" **range**

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

```
subtype_y_attribute:
        subtype_indication

subtype_indication:
        actual_name range range
      | actual_name range attribute_name
      | actual_name actual_name range range
      | actual_name actual_name range attribute_name
      | actual_name actual_name

range_constraint:
        constraint1

constraint1:
        range range
      | range attribute_name

attribute_name:
        actual_name  "'" identifier
      | actual_name signature  "'" identifier

signature:
        "[" name_list return actual_name "]"
      | "[" return actual_name "]"
      | "[" "]"
      | "[" name_list "]"

record_type_definition:
        record element_declarations end record identifier
      | record element_declarations end record

element_declarations:
        element_declaration
      | element_declarations element_declaration

element_declaration:
        identifier_list ":" subtype_indication ";"
      | identifier_list ":" actual_name ";"

file_type_def:
        file of actual_name

incomp_type_declar:
        type identifier ";"

subtype_declaration:
        subtype identifier is subtype_indication ";"
      | subtype identifier is actual_name ";"

constant_declaration:
        constant identifier_list ":"  interface_init ";"

signal_declaration:
        signal identifier_list ":" subtype_indication signal_kind  ";"
      | signal identifier_list ":" subtype_indication signal_kind eq expression ";"
      | signal identifier_list ":" actual_name signal_kind  ";"
      |  signal identifier_list ":" actual_name signal_kind eq expression ";"
```

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

```
        | signal identifier_list ":" interface_init  ";"

signal_kind:
          register
        | bus

file_declaration:
          file identifier_list ":" subtype_indication file_open_information ";"
        | file identifier_list ":" subtype_indication ";"
        | file identifier_list ":" actual_name file_open_information ";"
        | file identifier_list ":" actual_name ";"

file_open_information:
          open expression is file_mode alias_designator
        | is file_mode alias_designator
        | open expression is   alias_designator
        | is  string_literal

file_mode:
          in
        | out

alias_declaration:
          alias alias_designator ":" subtype_indication is actual_name signature ";"
        | alias alias_designator ":" subtype_indication is actual_name ";"
        | alias alias_designator ":" actual_name is actual_name signature ";"
        | alias alias_designator ":" actual_name is actual_name ";"
        | alias alias_designator is actual_name signature ";"
        | alias alias_designator is actual_name  ";"

alias_designator:
          identifier
        | character_literal
        | string_literal

entity_declaration:
          entity identifier entity_header entity_declarative_items
                entity_statement_part entity_end ";"
        | entity identifier  entity_header entity_declarative_items entity_end ";"
        | entity identifier  entity_header   entity_statement_part entity_end ";"
        | entity identifier  entity_header   entity_end ";"

entity_end:
          end
        | end entity  identifier
        | end entity
        | end identifier

entity_declarative_items:
          entity_declarative_item
        | entity_declarative_items entity_declarative_item

entity_statement_part:
          begin
        | begin entity_statements
```

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

| entity_statements entity_statement

entity_statement:
c_assertion_s
| c_procedure_call_s
| process_statement

c_assertion_s:
identifier ":" **postponed** assertion ";"
| identifier ":" assertion ";"
| **postponed** assertion ";"
| assertion ";"

c_procedure_call_s:
identifier ":" **postponed** function_call ";"
| identifier ":" function_call ";"
| **postponed** function_call ";"
| function_call ";"

attribute_specification:
attribute identifier **of** entity_specification **is** expression ";"

entity_specification:
entity_name_list ":" entity_class

entity_name_list:
entity_designator_list
| **others**
| **all**

entity_designator_list:
entity_designator
| entity_designator_list "," entity_designator

entity_designator:
alias_designator signature
| alias_designator

entity_class:
entity
| **architecture**
| **configuration**
| **procedure**
| **function**
| **package**
| **type**
| **subtype**
| **constant**
| **signal**
| **variable**
| **component**
| **label**
| **literal**
| **units**
| **group**

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

| **file**

configuration_declaration:
configuration identifier **of** actual_name **is**
config_declar_items block_configuration configuration_end ";"
| **configuration** identifier **of** actual_name **is**
block_configuration configuration_end ";"

configuration_end:
end
| **end configuration** identifier
| **end configuration**
| **end** identifier

block_configuration:
for block_specification **end for** ";"
| **for** block_specification use_clauses **end for** ";"
| **for** block_specification configuration_items **end for** ";"
| **for** block_specification use_clauses configuration_items **end for** ";"

use_clauses:
use_clause
| use_clauses use_clause

configuration_items:
configuration_item
| configuration_items configuration_item

block_specification:
actual_name

configuration_item:
block_configuration
| comp_config

comp_config:
for comp_specifi binding_indication ";" **end for** ";"
| **for** comp_specifi block_configuration **end for** ";"
| **for** comp_specifi binding_indication ";" block_configuration **end for** ";"
| **for** comp_specifi **end for** ";"

config_declar_items:
config_declar_item
| config_declar_items config_declar_item

config_declar_item:
use_clause
| attribute_specification
| group_declaration

group_declaration:
group identifier ":" identifier "(" actual_name ")" ";"
| **group** identifier ":" identifier "(" **character_literal** ")" ";"
| **group** identifier ":" identifier "(" actual_name resto ")" ";"
| **group** identifier ":" identifier "(" **character_literal** resto ")" ";"
| **group** identifier **is** "(" entity_class_entry_list ")" ";"

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

```
entity_class_entry_list:
        entity_class_entry
      | entity_class_entry_list "," entity_class_entry

entity_class_entry:
        entity_class | entity_class lg

resto:
        "," character_literal
      | "," actual_name
      | resto "," character_literal
      | resto "," actual_name

comp_specifi:
        instantiation_list ":" actual_name

instantiation_list:
        identifier_list
      | others
      | all

binding_indication:
        use entity_aspect
      | use entity_aspect map_aspects
      | map_aspects

map_aspects:
        generic_map_aspect port_map_aspect
      | generic_map_aspect
      | port_map_aspect

entity_aspect:
        entity actual_name
      | configuration actual_name
      | open

generic_map_aspect:
        generic map "(" expression_list ")"
      | generic map "(" basic_assoc_list ")"
      | generic map "(" named_assoc_list ")"

port_map_aspect:
        port map "(" expression_list ")"
      | port map "(" basic_assoc_list ")"
      | port map "(" named_assoc_list ")"

package_declaration:
        package identifier pack_delclar_part package_end ";"

package_end:
        end
      | end package identifier
      | end package
      | end identifier
```

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

pack_delclar_part:
is
| **is** pack_delclar_items

pack_delclar_items:
pack_delclar_item
| pack_delclar_items pack_delclar_item

pack_delclar_item:
basic_declarative_item
| signal_declaration
| component_declaration
| attribute_declaration
| attribute_specification
| group_declaration

component_declaration:
component identifier **is** component_clauses component_end ";"
| **component** identifier component_clauses component_end ";"
| **component** identifier **is** component_end ";"
| **component** identifier component_end ";"

component_clauses:
generic_clause port_clause
| generic_clause
| port_clause

component_end:
end component identifier
| **end component**

secondary_unit:
architecture_body
| package_body

architecture_body:
architecture identifier **of** actual_name architecture_declarative_part
architecture_statement_part architecture_end ";"

architecture_end:
end architecture identifier
| **end architecture**
| **end** identifier
| **end**

architecture_declarative_part:
is
| **is** block_declarative_items

block_declarative_items:
block_declarative_item
| block_declarative_items block_declarative_item

block_declarative_item:
package_body_declarative_item
| signal_declaration

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

```
        | component_declaration
        | attribute_declaration
        | attribute_specification
        | config_specific
        | group_declaration
        | vss_specification

config_specific:
          for comp_specifi binding_indication ";"
        | for comp_specifi  ";"

architecture_statement_part:
          begin
        | begin concurrent_statements

concurrent_statements:
          concurrent_statement
        | concurrent_statements concurrent_statement

concurrent_statement:
          process_statement
        | c_procedure_call_s
        | c_assertion_s
        | c_signal_assign_s
        | comp_instant

c_signal_assign_s:
          identifier ":"  cond_signal_assign
        | identifier ":"  selec_signal_assign
        | cond_signal_assign
        | selec_signal_assign

cond_signal_assign:
          actual_name le  conditional_waveforms ";"
        | aggregate le  conditional_waveforms ";"

conditional_waveforms:
          waveform_head waveform when expression
        | waveform_head waveform
        | waveform when expression
        | waveform

waveform_head:
          waveform when expression else
        | waveform_head waveform when expression else

waveform:
          expression
        | unaffected_t

selec_signal_assign:
          with expression select actual_name le  selected_waveforms ";"
        | with expression select aggregate le  selected_waveforms ";"

selected_waveforms:
          waveform when choices
```

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

```
        | selected_waveforms "," waveform when choices

comp_instant:
          identifier ":" instantiated_unit generic_map_aspect port_map_aspect ";"
        | identifier ":" instantiated_unit generic_map_aspect ";"
        | identifier ":" instantiated_unit port_map_aspect ";"
        | identifier ":" actual_name generic_map_aspect port_map_aspect ";"
        | identifier ":" actual_name generic_map_aspect ";"
        | identifier ":" actual_name port_map_aspect ";"
        | identifier ":" instantiated_unit ";"

instantiated_unit:
          component actual_name
        | entity  actual_name
        | configuration actual_name

package_body:
          package body  identifier pack_body_declar_part package_body_end ";"

package_body_end:
          end
        | end package body identifier
        | end package body
        | end identifier

pack_body_declar_part:
          is
        | is package_body_declarative_items

package_body_declarative_items:
          package_body_declarative_item
        | package_body_declarative_items package_body_declarative_item

entity_header:
          is
        | is generic_clause port_clause
        | is generic_clause
        | is port_clause

generic_clause:
          generic "(" generic_list ")" ";"

generic_list:
          interface_list

interface_list:
          interface_declaration
        | interface_list ";" interface_declaration

interface_declaration:
          int_const_declarat
        | int_signal_declarat
        | int_variable_declarat
        | int_file_declarat
```

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

```
interface_init:
        subtype_indication
      | actual_name
      | subtype_indication eq expression
      | actual_name eq expression

mode:
        out
      | inout
      | buffer
      |  linkage

int_const_declarat:
        constant identifier_list ":" in  interface_init
      | constant identifier_list ":"  interface_init
      | identifier_list ":" interface_init

int_signal_declarat:
        signal identifier_list ":" mode  interface_init_bus
      | signal identifier_list ":" in  interface_init_bus
      | signal identifier_list  ":"  interface_init_bus
      | identifier_list  ":" mode   interface_init_bus
      | identifier_list  ":" in   interface_init_bus
      | identifier_list  ":" interface_init2

interface_init_bus:
        interface_init
      | interface_init2

interface_init2:
        subtype_indication bus
      | actual_name bus
      | subtype_indication bus eq expression
      | actual_name bus eq expression

int_variable_declarat :
        variable identifier_list ":" mode    interface_init
      | variable identifier_list ":" in    interface_init
      | variable identifier_list ":"  interface_init

int_file_declarat:
        file identifier_list ":" subtype_indication
      | file identifier_list ":" actual_name

port_clause:
        port "(" interface_list ")" ";"

sequential_statements:
        identifier ":" sequential_statement
      | sequential_statement

sequential_statementsnovss:
        identifier ":" sequential_statementnovss
      | sequential_statementnovss
```

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

sequence_of_statements:
sequential_statements
| sequence_of_statements sequential_statements

sequence_of_statementsnovss:
sequential_statementsnovss
| sequence_of_statementsnovss sequential_statementsnovss

sequence_of_statementsexcept:
vss_statementexcept
| sequence_of_statementsexcept sequential_statementsnovss
| sequence_of_statementsexcept vss_statementexcept
| sequential_statementsnovss

wait_statement:
wait sensitivity_clause condition_clause ";"
| **wait** sensitivity_clause ";"
| **wait** condition_clause ";"
| **wait** ";"

sensitivity_clause:
on name_list

condition_clause:
until expression

assertion_statement:
assertion ";"

assertion:
assert expression **report** expression **severity** expression
| **assert** expression **report** expression
| **assert** expression **severity** expression
| **assert** expression

report_statement:
report expression **severity** expression ";"
| **report** expression ";"

signal_assignament_statement:
actual_name **le** waveform ";"
| aggregate **le** waveform ";"

variable_assignament_statement:
actual_name **eq** expression ";"
| aggregate **eq** expression ";"

if_statement:
if expression seq_stat_then elseif_list seq_stat_else ifend ";"
| **if** expression seq_stat_then elseif_list ifend ";"
| **if** expression seq_stat_then seq_stat_else ifend ";"
| **if** expression seq_stat_then ifend ";"

seq_stat_then:
then
| **then** sequence_of_statements

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

```
seq_stat_else:
        else
      | else sequence_of_statements

elseif_list:
        elsif expression seq_stat_then
      | elseif_list elsif expression seq_stat_then

ifend:
        end if
      | end if identifier

case_statement:
        case expression is case_statement_alternative  end case identifier ";"
      | case expression is case_statement_alternative  end case ";"

case_statement_alternative:
        when choices seq_stat_case
      | case_statement_alternative when choices seq_stat_case

seq_stat_case:
        eg
      | eg sequence_of_statements

loop_statement:
        iteration_scheme sequence_of_statements end loop identifier ";"
      | iteration_scheme sequence_of_statements end loop ";"
      | iteration_scheme  end loop identifier ";"
      | iteration_scheme  end loop ";"

iteration_scheme:
        loop
      | while expression loop
      | for parameter_specification loop

parameter_specification:
        identifier in discrete_range

next_statement:
        next identifier when expression ";"
      | next identifier ";"
      | next when expression ";"
      | next ";"

exit_statement:
        exit identifier when expression ";"
      | exit identifier  ";"
      | exit when expression ";"
      | exit ";"

return_statement:
        return expression ";"
      | return  ";"

aggregate:
        "(" elem_assoc_list ")"
```

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

```
elem_assoc_list:
        choices eg expression
      | elem_assoc_list "," expression
      | elem_assoc_list "," choices eg expression
      | expression

choices:
        choice
      | choices "|" choice

choice:
        simple_expression
      | range
      | subtype_indication
      | others

decimal_literal:
        real
      | integer

based_literal:
        based_literal

identifier:
        basic_identifier
      | extended_identifier

simple_expression:
        character_literal
      | actual_name
      | qualified_expression
      | bit_string_literal
      | decimal_literal
      | based_literal
      | null
      | aggregate
      | abs simple_expression
      | not simple_expression
      | "-" simple_expression %prec uminus_t
      | "+" simple_expression
      | simple_expression  "+" simple_expression
      | simple_expression  "-" simple_expression
      | simple_expression  "&" simple_expression
      | simple_expression  "*" simple_expression
      | simple_expression  "/" simple_expression
      | simple_expression  mod simple_expression
      | simple_expression  rem simple_expression
      | simple_expression  pot simple_expression
      | stat_bufferfull
      | stat_bufferempty

shift_expression:
        simple_expression
      | simple_expression sll simple_expression
```

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

| simple_expression **srl** simple_expression
| simple_expression **sla** simple_expression
| simple_expression **sra** simple_expression
| simple_expression **rol** simple_expression
| simple_expression **ror** simple_expression

relation:
shift_expression "=" shift_expression
| shift_expression **ne** shift_expression
| shift_expression "<" shift_expression
| shift_expression **le** shift_expression
| shift_expression ">" shift_expression
| shift_expression **ge** shift_expression
| shift_expression

expression:
relation
| and_relation
| or_relation
| xor_relation
| xnor_relation
| relation **nand** relation
| relation **nor** relation

and_relation:
relation **and** relation
| and_relation **and** relation

or_relation:
relation **or** relation
| or_relation **or** relation

xor_relation:
relation **xor** relation
| xor_relation **xor** relation

xnor_relation:
relation **xnor** relation
| xnor_relation **xnor** relation

expression_list:
expression
| expression_list "," expression

sel_id_or_funct:
actual_name "." identifier
| actual_name "." **string_literal**
| actual_name "." **character_literal**
| actual_name "." **all**

basic_assoc_list:
expression_list "," **open**
| basic_assoc_list "," actual_part
| **open**

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

named_assoc_list:
actual_name **eg** actual_part
| expression_list "," actual_name **eg** actual_part
| basic_assoc_list "," actual_name **eg** actual_part
| named_assoc_list "," actual_name **eg** actual_part

actual_part:
expression
| **open**

function_call:
actual_name "(" expression_list ")"
| actual_name "(" basic_assoc_list ")"
| actual_name "(" named_assoc_list ")"
| sel_id_or_funct
| identifier

actual_name:
string_literal
| function_call
| actual_name "(" subtype_indication ")"
| actual_name "(" rangelist ")"
| attribute_name

discrete_range:
subtype_indication
| actual_name
| range

name_list:
actual_name
| name_list "," actual_name

qualified_expression:
actual_name "'" aggregate

process_statement:
process_beginsivss process_declarative_items seq_stat_sub process_end ";"
| process_beginsivss seq_stat_sub process_end ";"
| process_beginnovss process_declarative_items
seq_stat_subnovss process_end ";"
| process_beginnovss seq_stat_subnovss process_end ";"

process_beginnovss:
process "(" name_list ")" **is**
| **process** "(" name_list ")"
| **process is**
| **process**

process_beginsivss:
identifier ":" **process** "(" name_list ")" **is**
| identifier ":" **process** "(" name_list ")"
| identifier ":" **process is**
| identifier ":" **process**

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

process_end:
    end process
  | **end postponed process** identifier
  | **end postponed process**
  | **end process** identifier

process_declarative_items:
    process_declarative_item
  | process_declarative_items process_declarative_item

process_declarative_item:
    package_body_declarative_item
  | variable_declaration
  | attribute_declaration
  | attribute_specification
  | vss_declarative_item

process_declarative_itemsexcept:
    process_declarative_itemexcept
  | process_declarative_itemsexcept process_declarative_itemexcept

process_declarative_itemexcept:
    package_body_declarative_item
  | variable_declaration
  | attribute_declaration
  | attribute_specification

entity_declarative_item:
    package_body_declarative_item
  | signal_declaration
  | attribute_declaration
  | attribute_specification
  | group_declaration
  | vss_specification

sequential_statement:
    sequential_statementnovss
  | vss_statement
  | stat_get
  | stat_put

sequential_statementnovss:
    wait_statement
  | assertion_statement
  | report_statement
  | signal_assignament_statement
  | variable_assignament_statement
  | if_statement
  | case_statement
  | loop_statement
  | next_statement
  | exit_statement
  | return_statement
  | **null** ";"

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

```
        | function_call ";"

stat_get:
        get "(" shared_name eg actual_name ")" ";"

stat_put:
        put "(" shared_name le expression ")" ";"

shared_name:
        identifier
      | shared_name"(" sh_expressionlist ")"
      | shared_name "." identifier

sh_expressionlist:
        expression
      | sh_expressionlist "," expression

vss_specification:
        shared_spec
      | channel_spec
      | exception_spec
      | clock_spec
      | timer_spec

vss_declarative_item:
        exception_body
      | initialize_specification

vss_statementexcept:
        stat_join
      | stat_start
      | stat_kill
      | stat_initialize
      | stat_send
      | stat_receive
      | stat_channelfull
      | stat_exception
      | stat_enable
      | stat_disable
      | stat_check
      | stat_delay
      | stat_timeout
      | stat_resettime
      | stat_lock
      | stat_resetbuffer

stat_resetbuffer:
        resetbuffer identifier ";"

stat_bufferfull:
        bufferfull "(" identifier ")"

stat_bufferempty:
        bufferempty "(" identifier ")"
```

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

```
vss_statement:
        vss_statementexcept
      | stat_seq_fork
      | stat_par_fork

stat_seq_fork:
        seqfork expression process_list end fork ";"

stat_par_fork:
        parfork expression process_list_mode end fork  ";"

process_list:
        vss_process_statement
      | process_list "|" "|" vss_process_statement

process_list_mode:
        pr_mode vss_process_statement
      | process_list_mode "|" "|" pr_mode vss_process_statement

pr_mode:
        simple
      | iterative

vss_process_statement:
        vss_process_begin process_declarative_items seq_stat_sub  process_end ";"
      | vss_process_begin seq_stat_sub  process_end ";"

vss_process_begin:
        identifier ":" process is
      | identifier ":" process

stat_join:
        join ";"
      | join identifier ";"
      | join ":" identifier_list ";"
      | join identifier ":" identifier_list ";"

stat_start:
        start expression identifier ";"

stat_kill:
        kill identifier_list ";"

stat_initialize:
        initialize identifier_list ";"

stat_lock:
        lock "(" identifier ")" expression
              begin sequence_of_statements end lock ";"

stat_send:
        send "(" lista_mensajes ")" ";"

lista_mensajes:
        destinatario "," expression
      | lista_mensajes ";" destinatario "," expression
```

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

destinatario:
identifier
| identifier **le** identifier
| **broadcast le** identifier

stat_receive:
receive "(" lista_datos ")" ";"

lista_datos:
identifier "," identifier
| lista_datos ";" identifier "," identifier

stat_channelfull:
channelfull "(" identifier ")" ";"

stat_exception:
exception "(" identifier ")" expression ";"

stat_enable:
vssenable identifier_list ";"

stat_disable:
disable identifier_list ";"

stat_check:
check ";"
| **check on** name_list ";"
| **check on** name_list **until** expression ";"

stat_delay:
delay "(" identifier "," expression **time_cont** ")" ";"

stat_timeout:
timeout "(" identifier "," expression **time_cont** ")" ";"

stat_resettime:
resettimer "(" identifier ")" ";"

shared_begin:
shared identifier

shared_spec:
mainshared **end shared** ";"

mainshared:
shared_begin **is** shared_declarations

shared_declarations:
shared_item
| shared_declarations shared_item

shared_item:
shared_var_declaration

shared_var_declaration:
variable identifier_list ":" shared_type ";"

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

shared_type:
actual_name

variable_declarations:
variable_declaration
| variable_declarations variable_declaration

channel_spec:
channel identifier_list ":" identifier channel_mode ";"
| **channel** identifier_list ":" identifier **buffer integer** channel_mode ";"

channel_mode:
simple
| **broadcast**
| **incast**

exception_spec:
exception identifier_list **priority integer** ";"

clock_spec:
clocks "(" lista_relojes ")" ";"

lista_relojes:
identifier "," **integer time_cont**
| lista_relojes ";" identifier "," **integer time_cont**

timer_spec:
timer identifier_list ":" identifier **max_cont integer priority integer** ";"

exception_body:
exceptionbody identifier restoexcept

restoexcept:
is process_declarative_itemsexcept
begin sequence_of_statementsexcept **end** ";"
| **is begin** sequence_of_statementsexcept **end** ";"

initialize_specification:
initialize begin sequence_of_statementsnovss **end** ";"

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

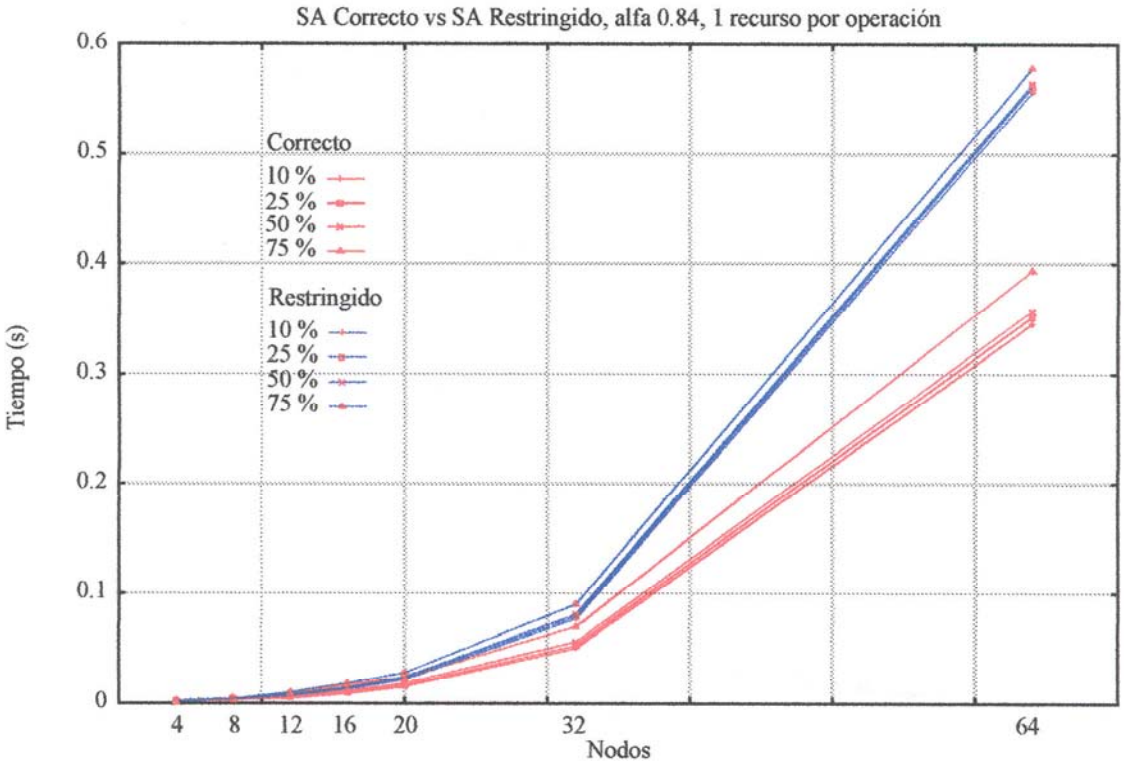
© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

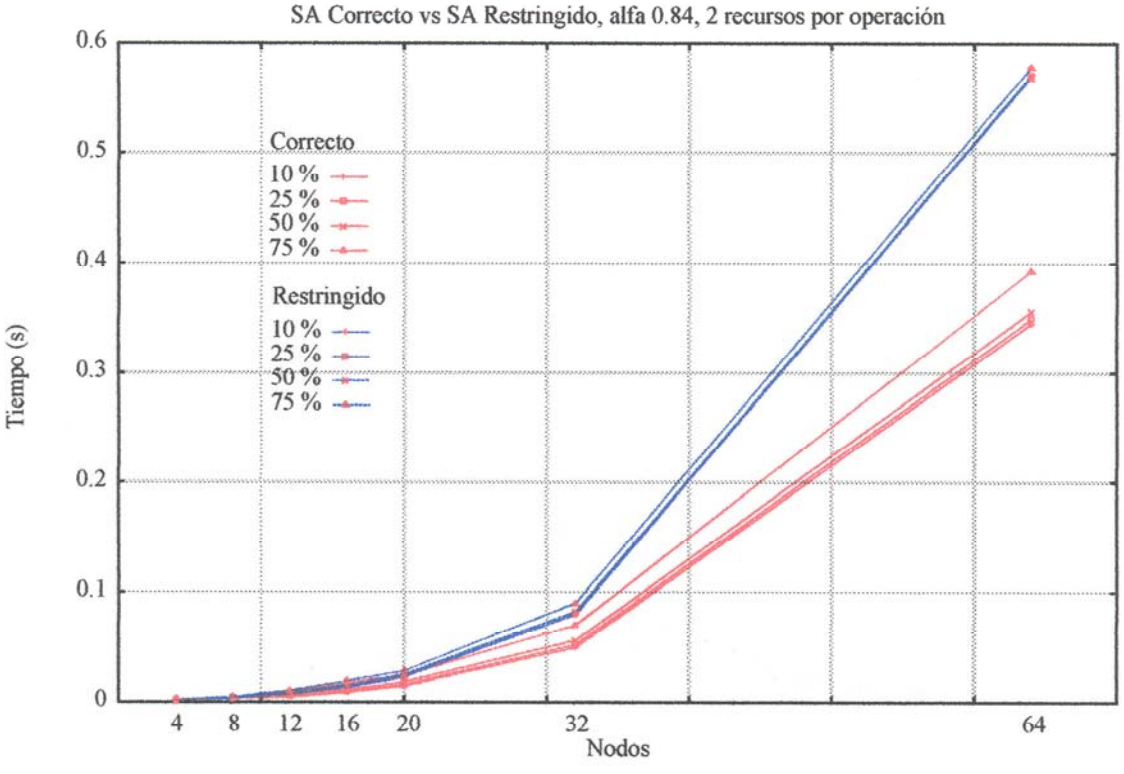
Apéndice B

Resultados de los algoritmos de planificación: simulated annealing

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

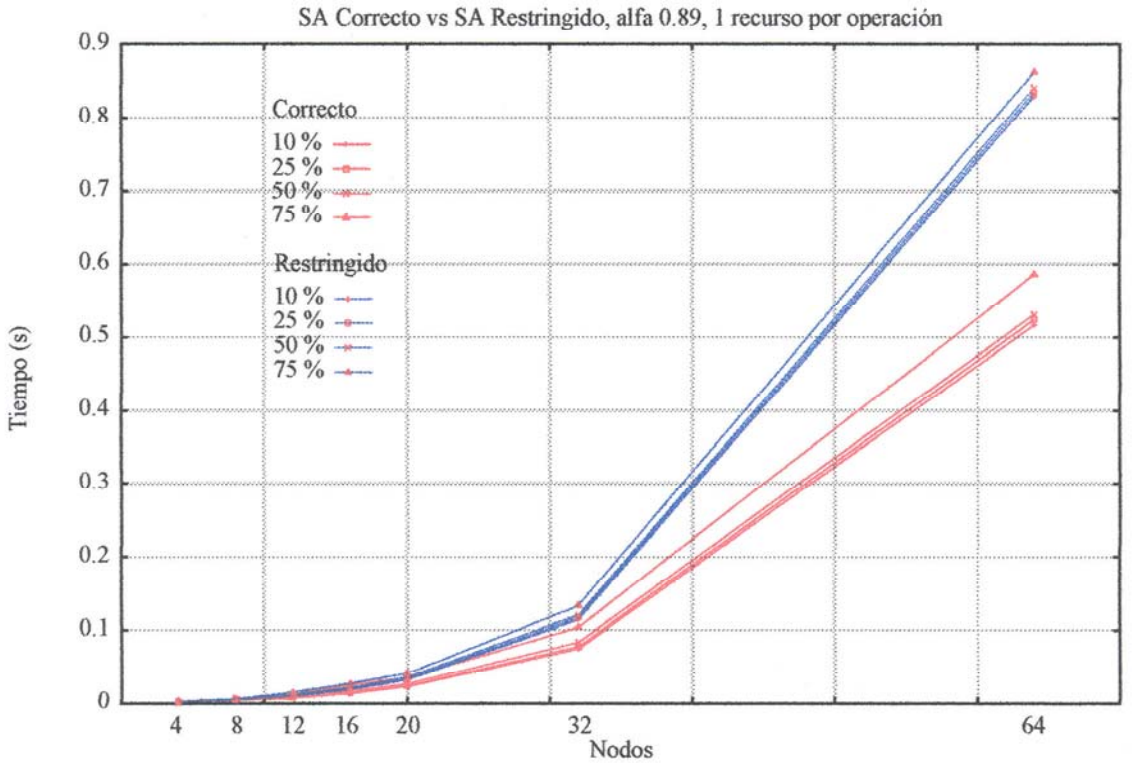
© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

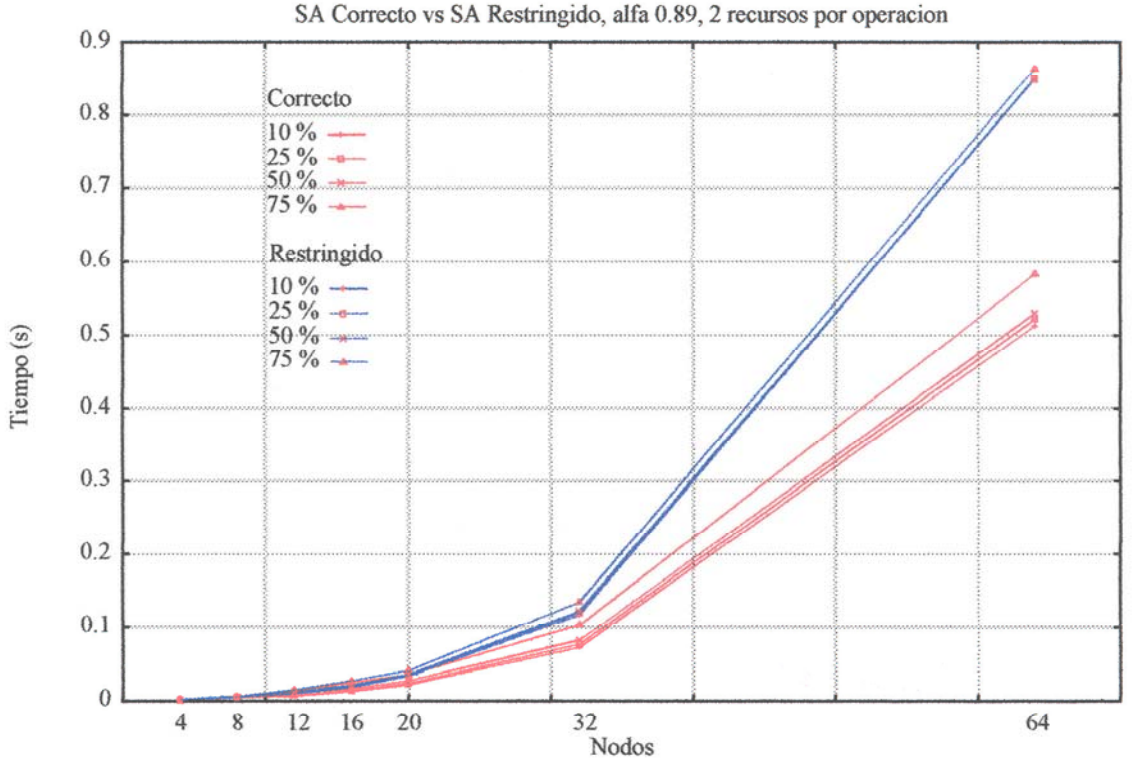




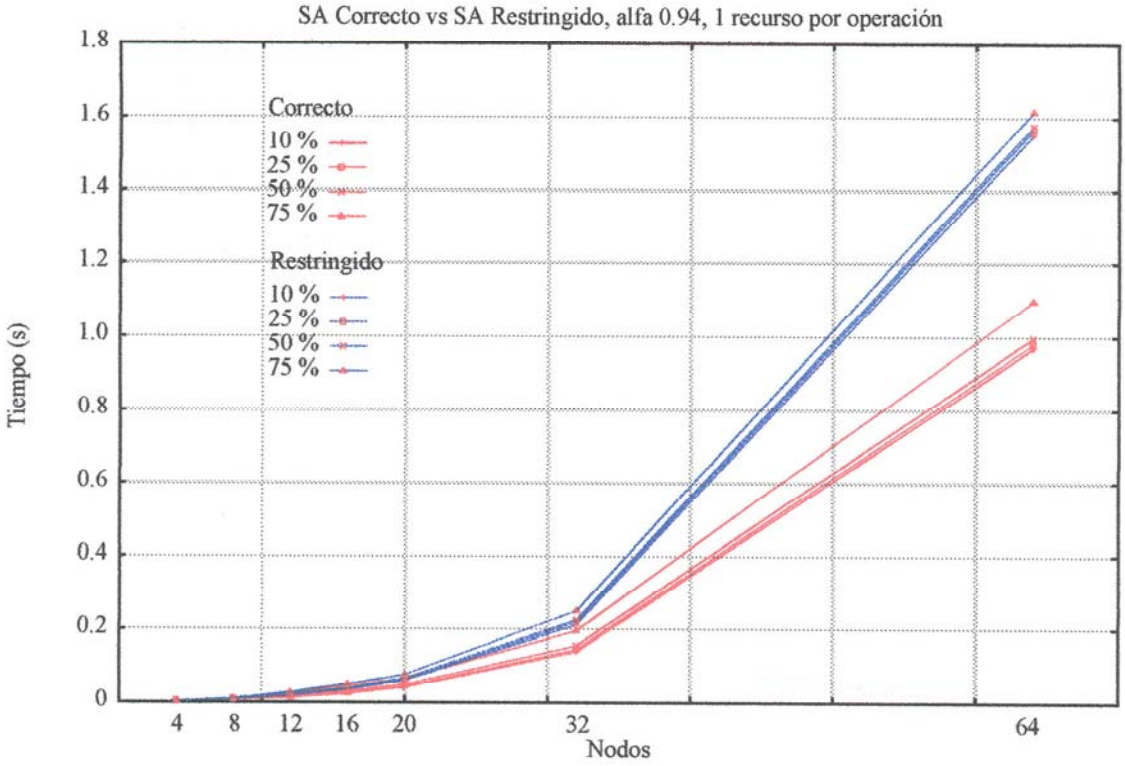
© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

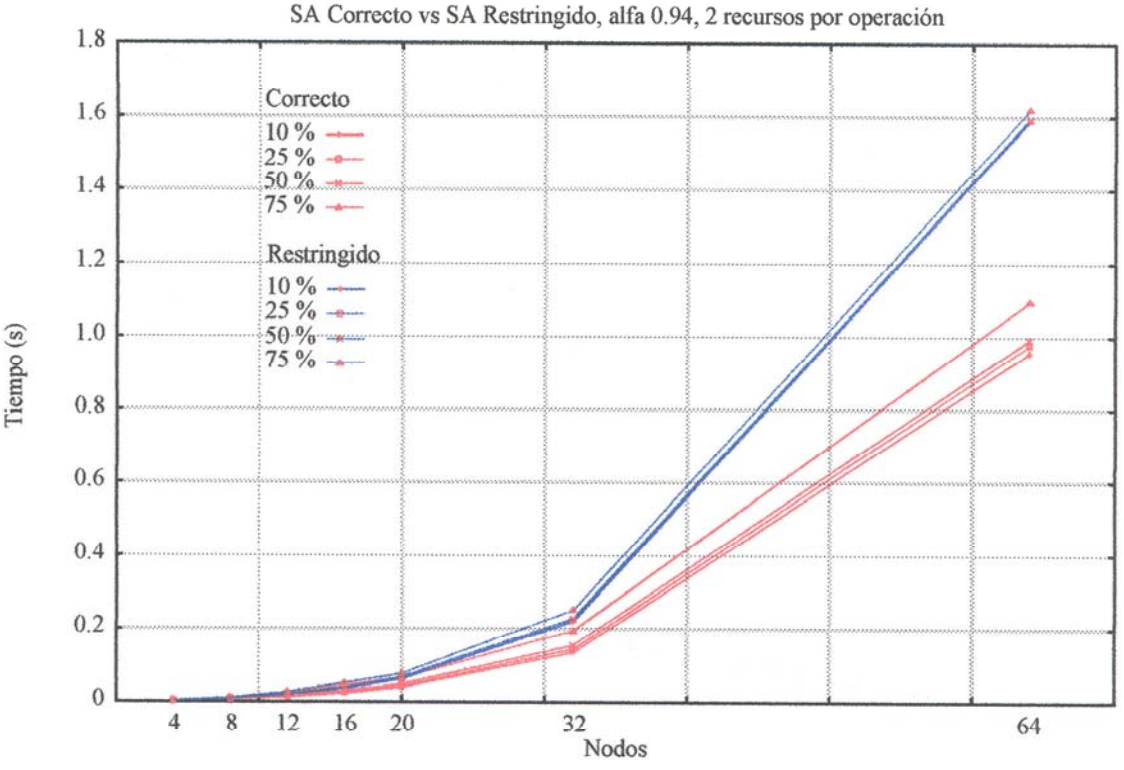
B.4



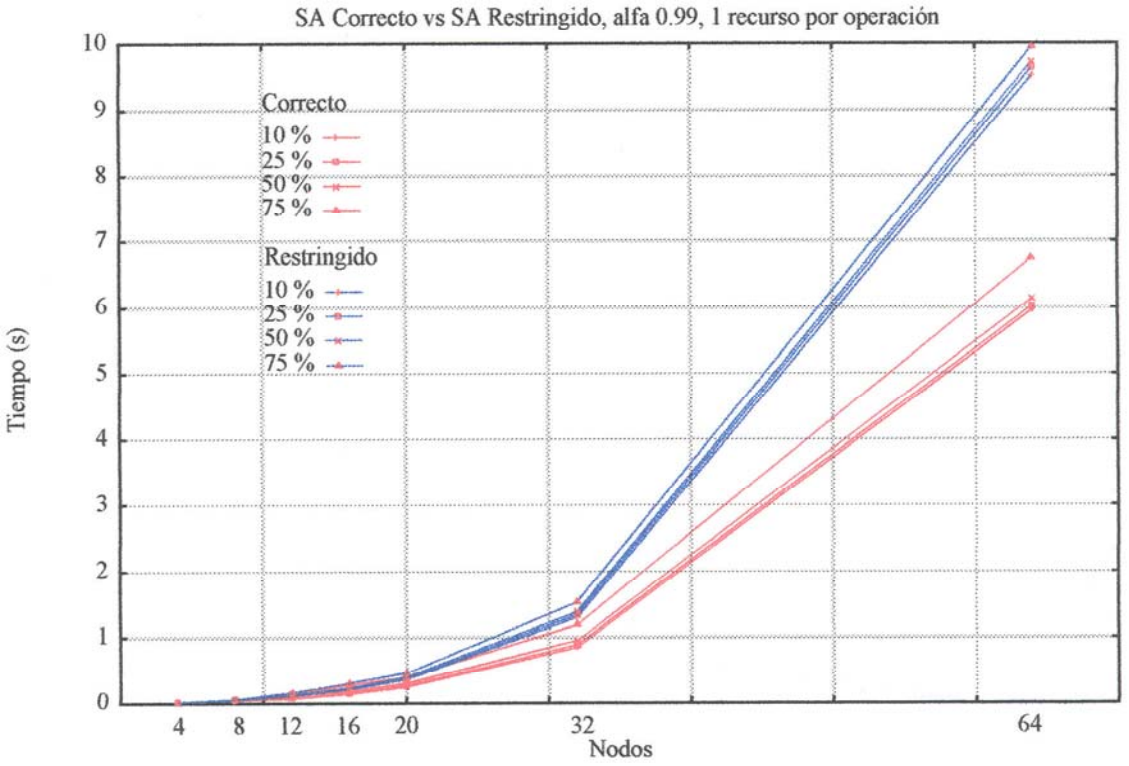


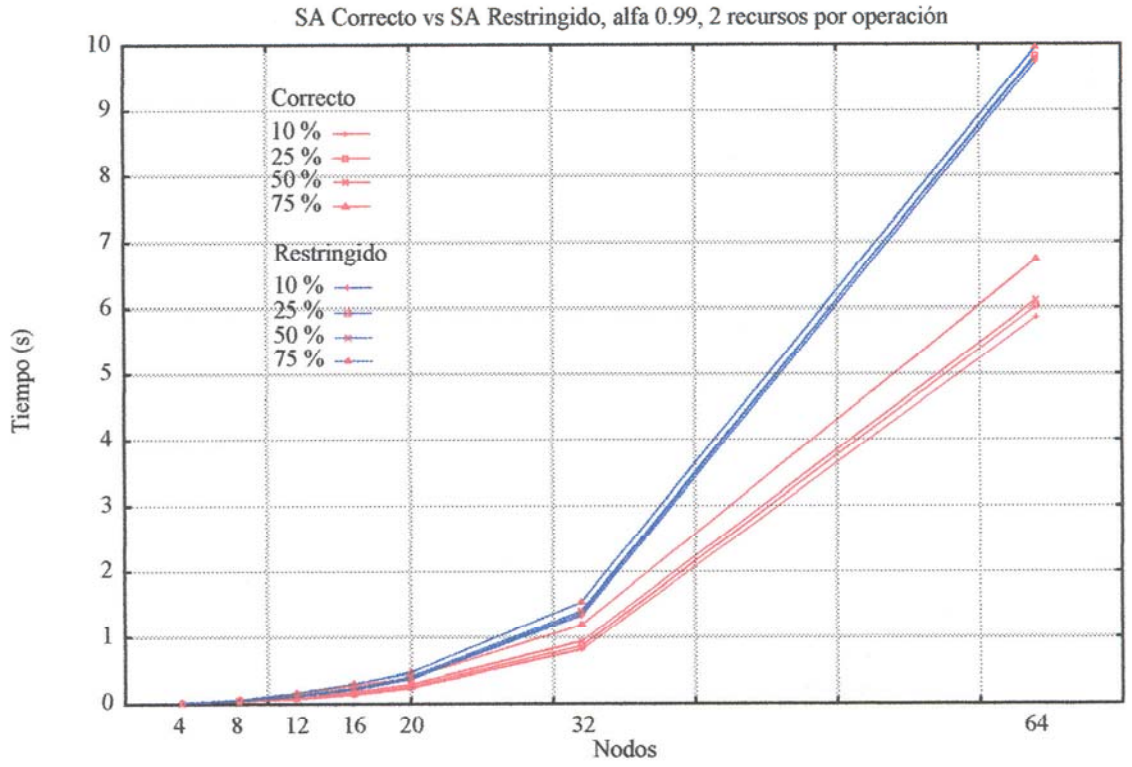
© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004



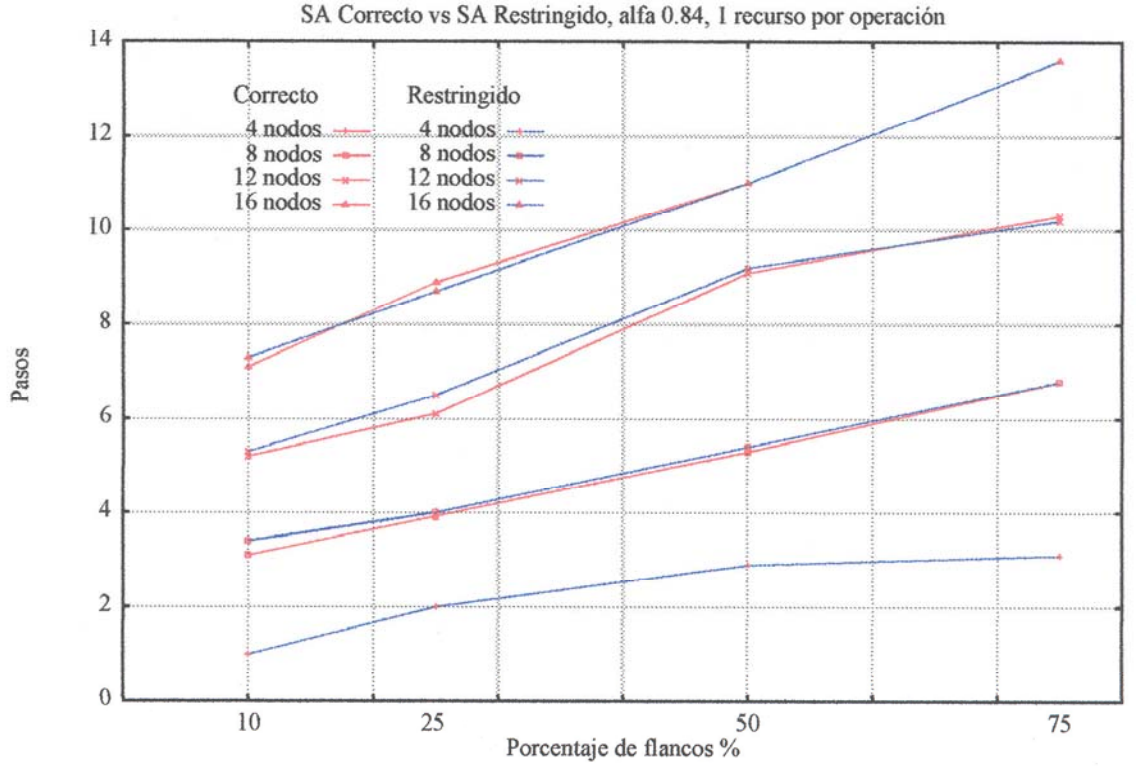


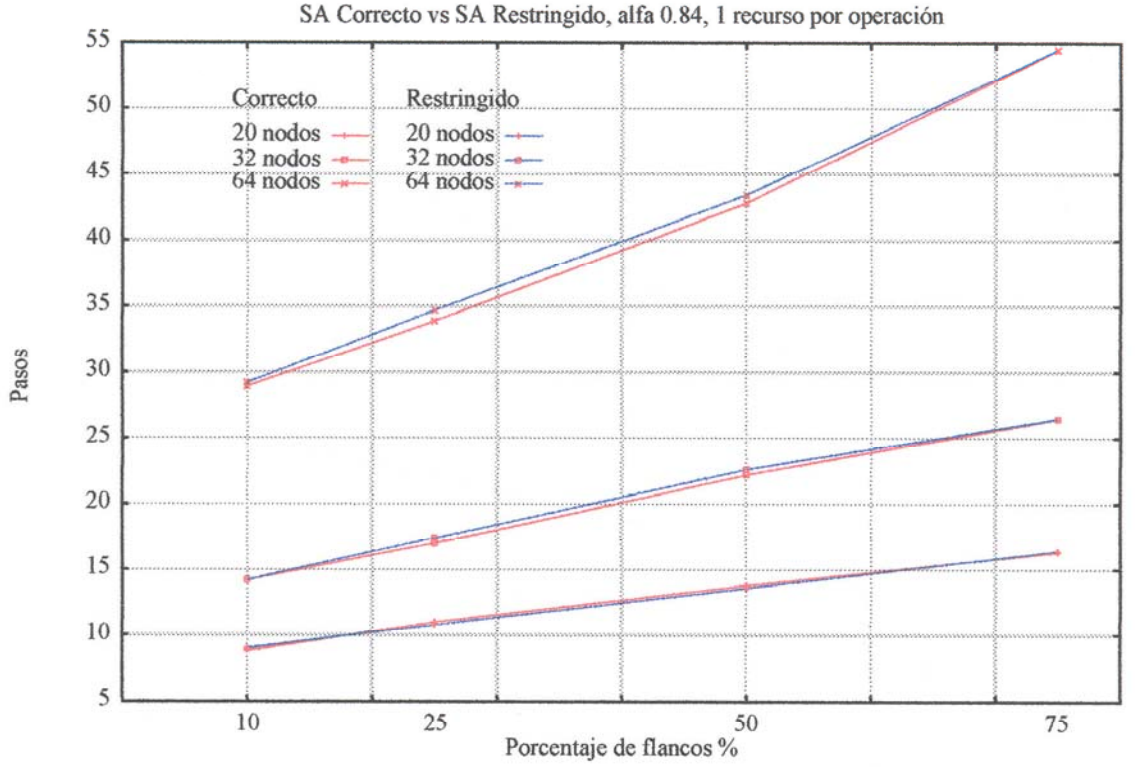
© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004



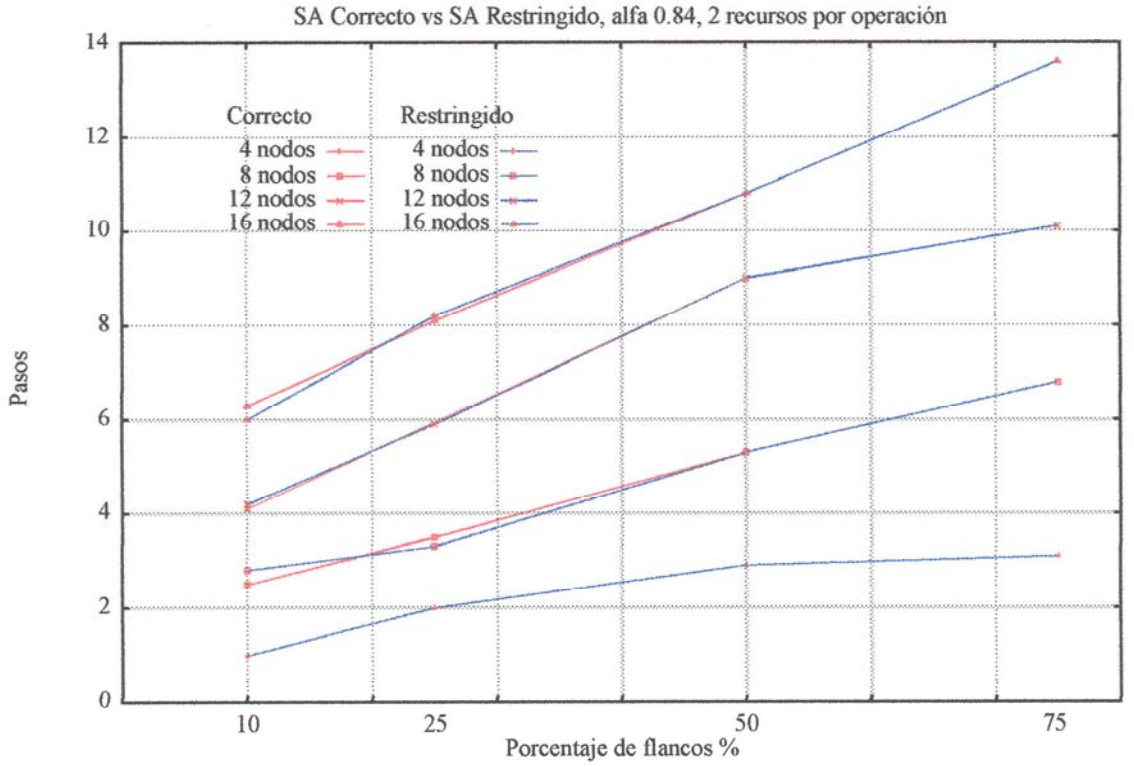


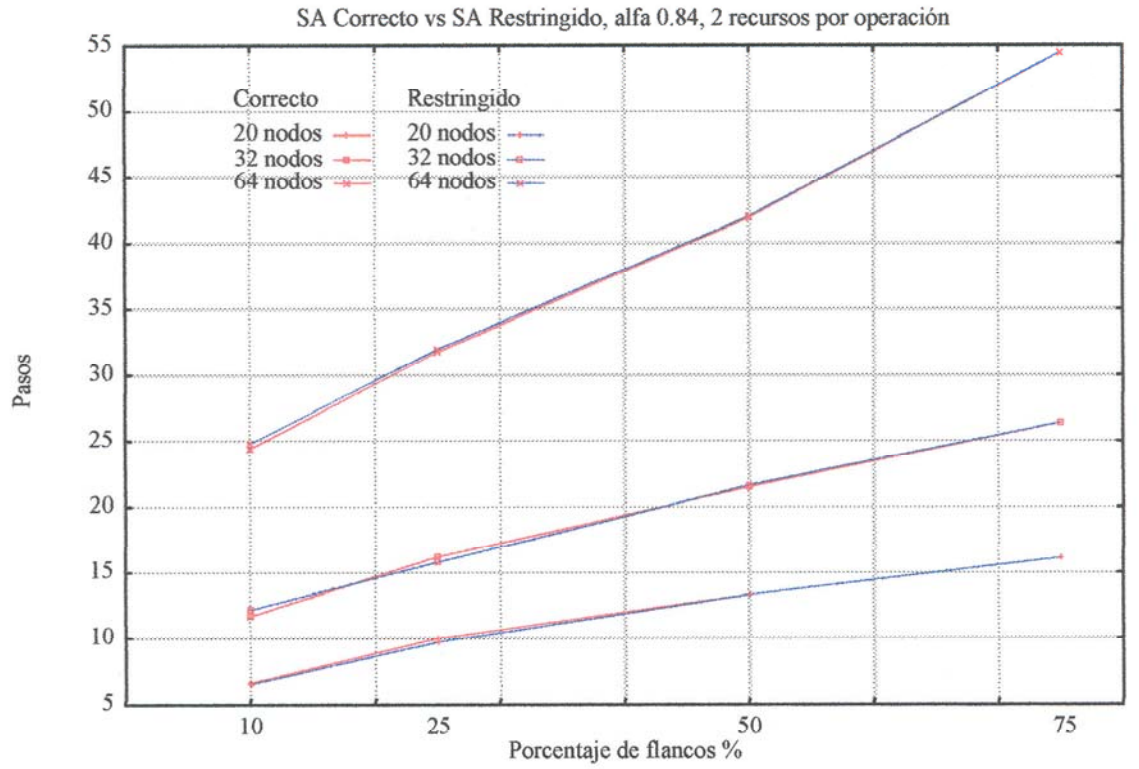
© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004



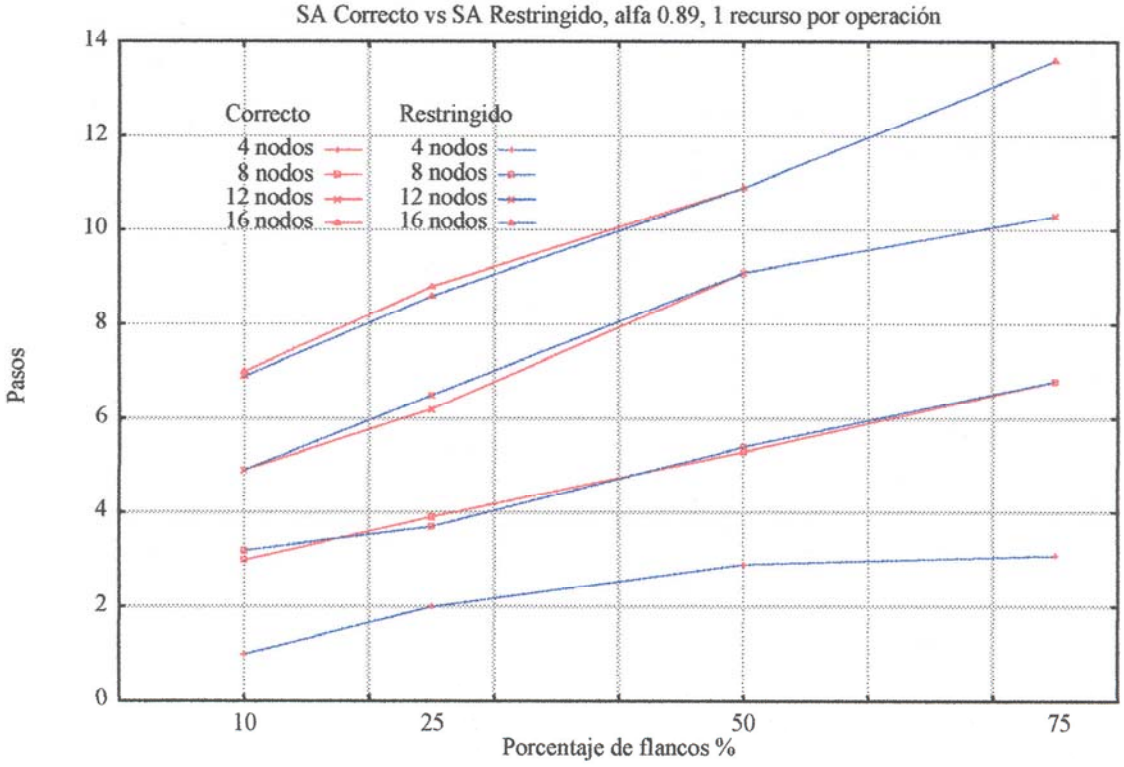


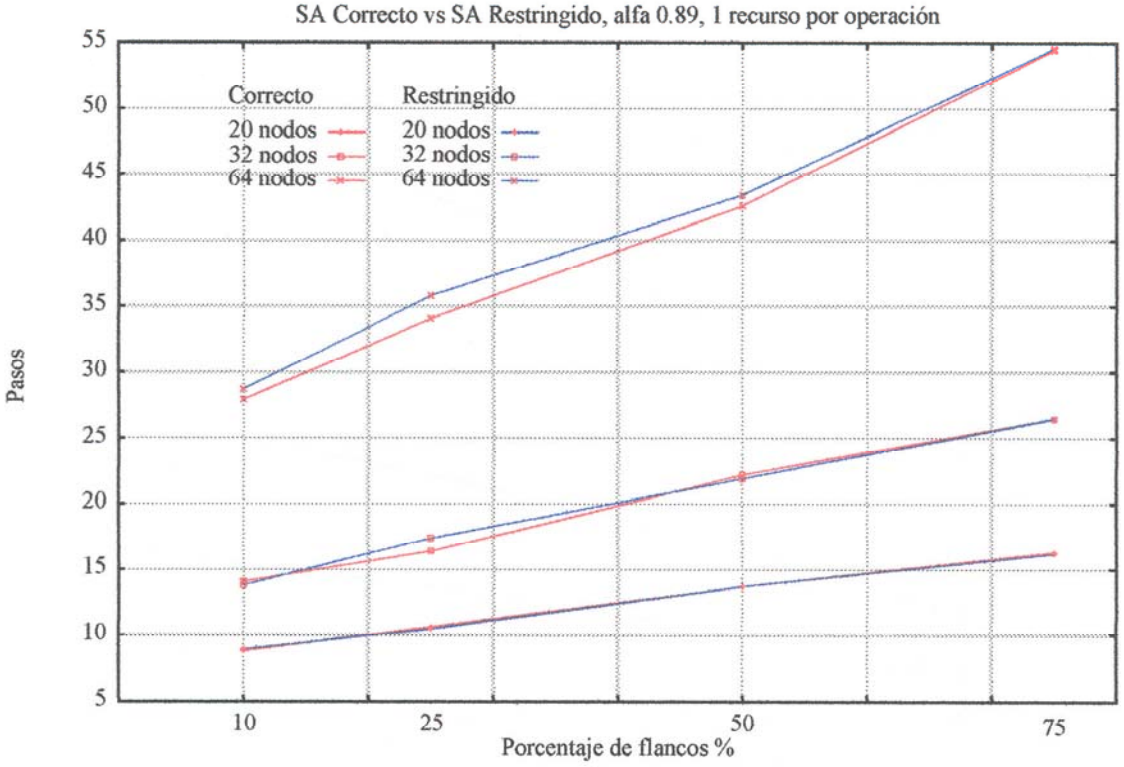
© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004



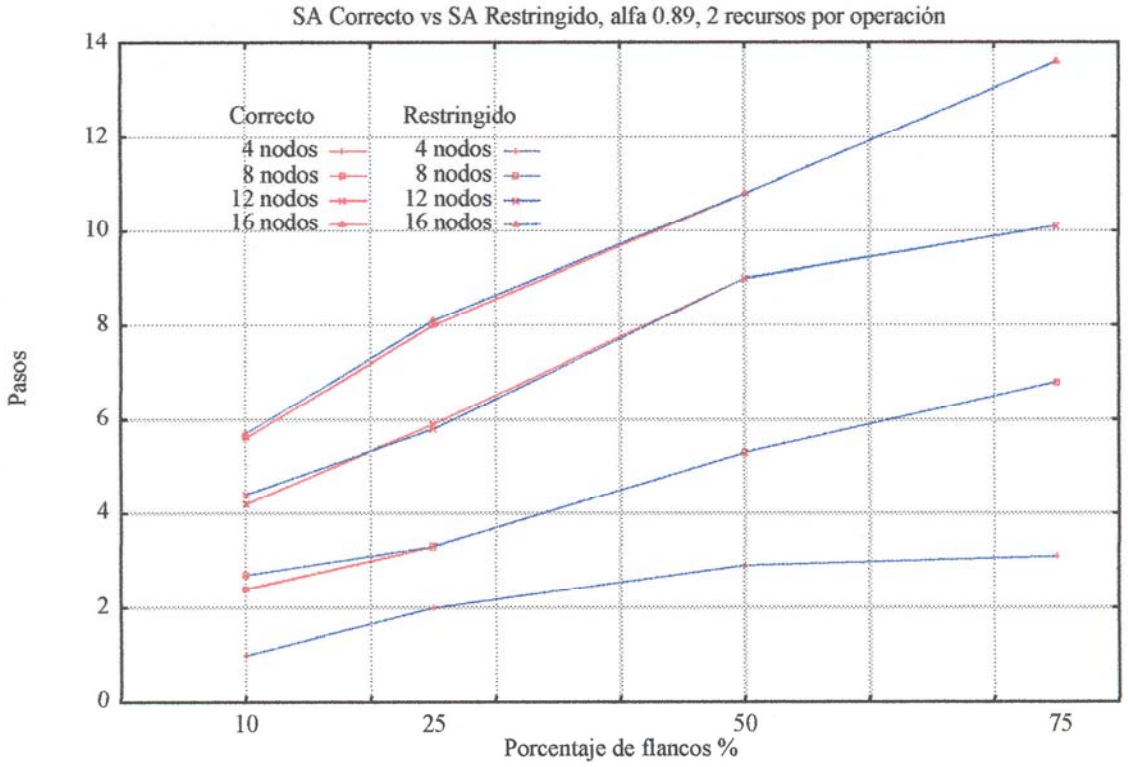


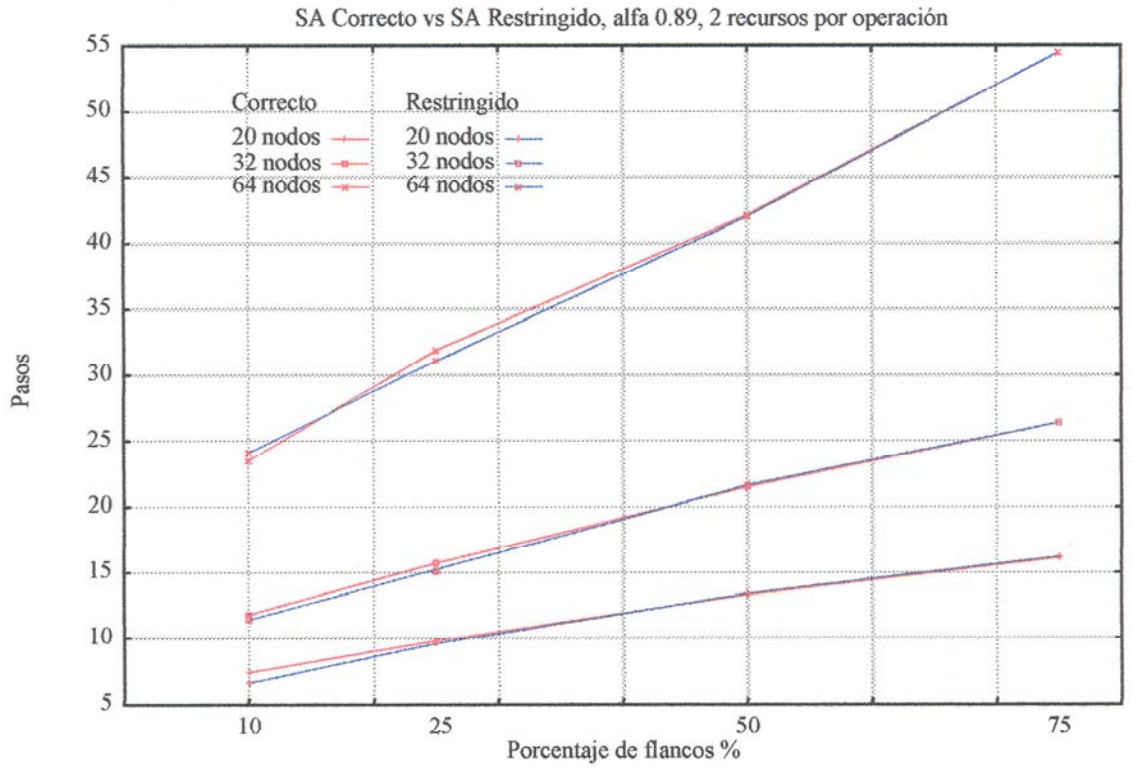
© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004



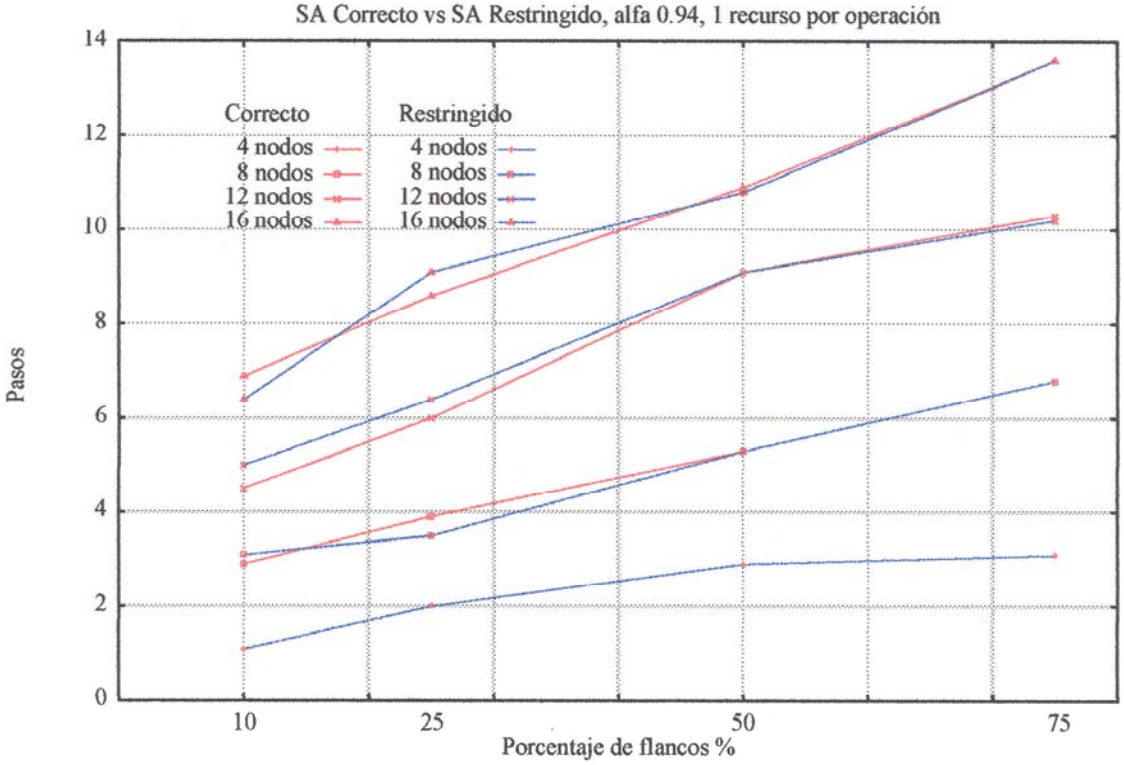


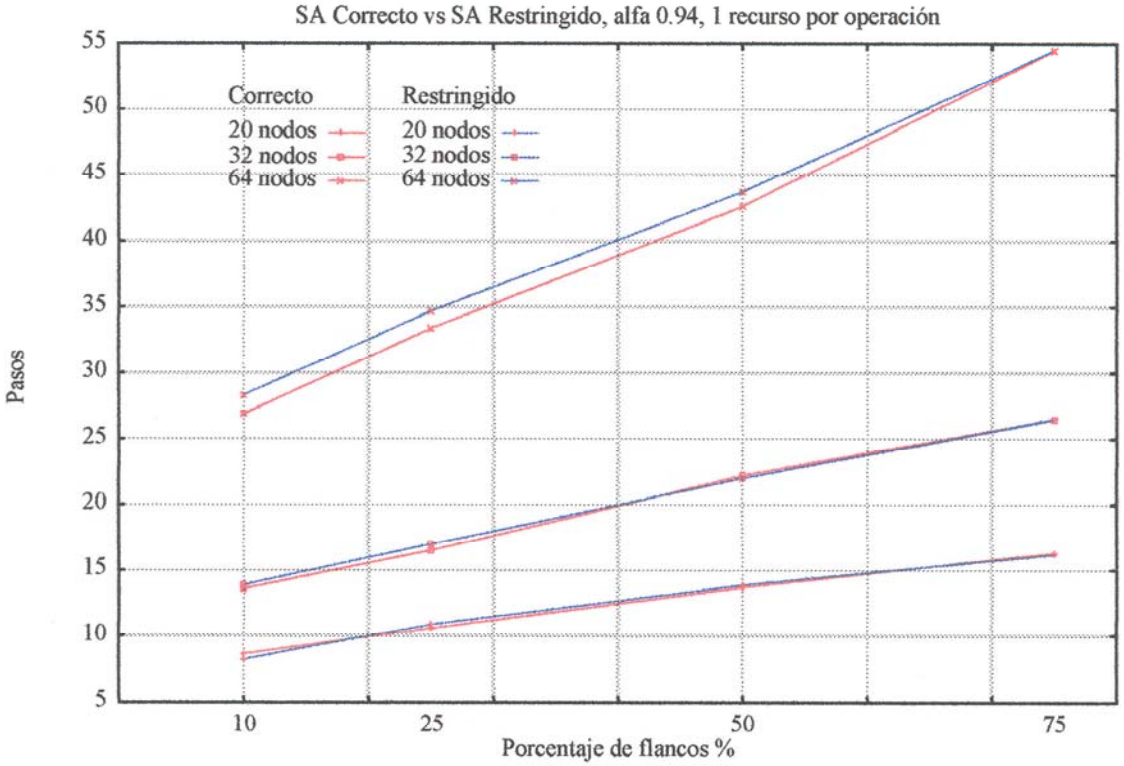
© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004



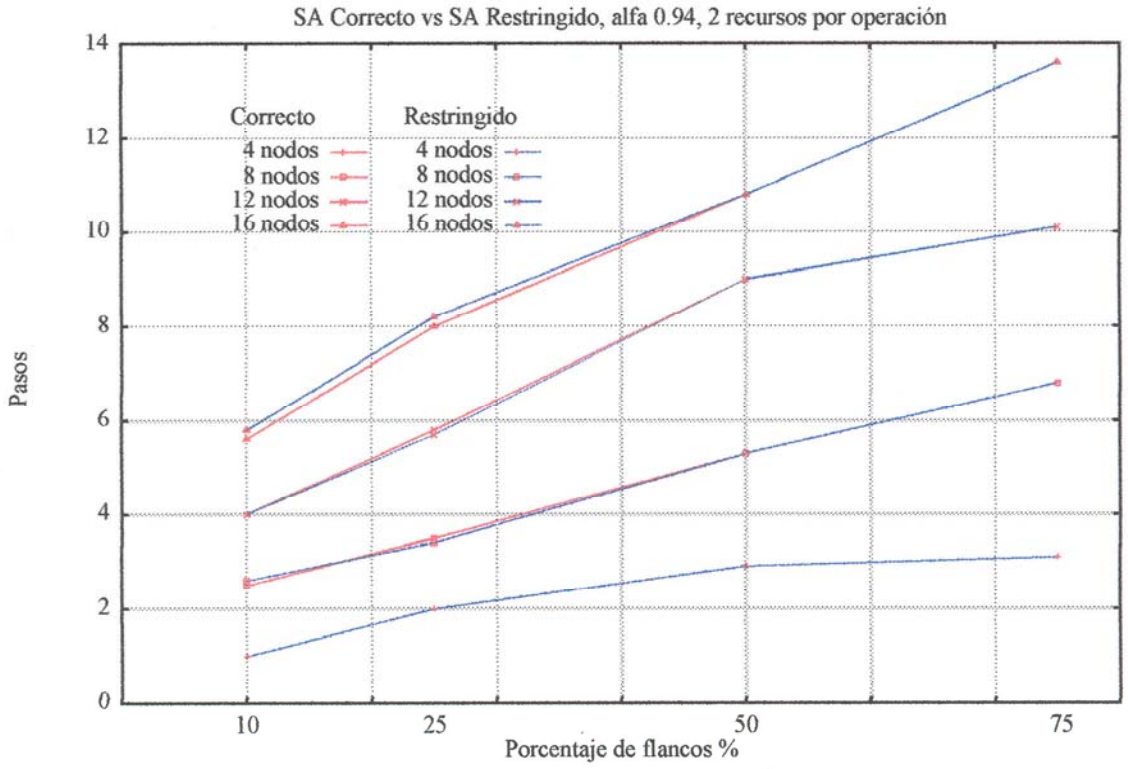


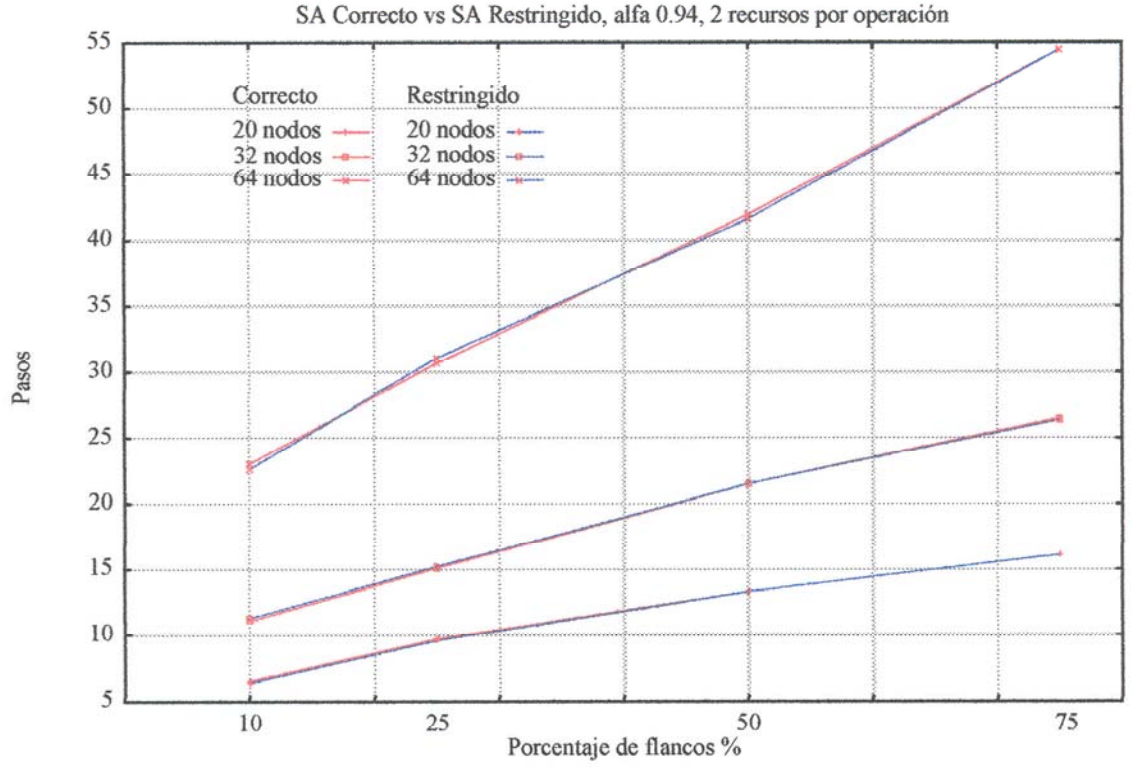
© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004



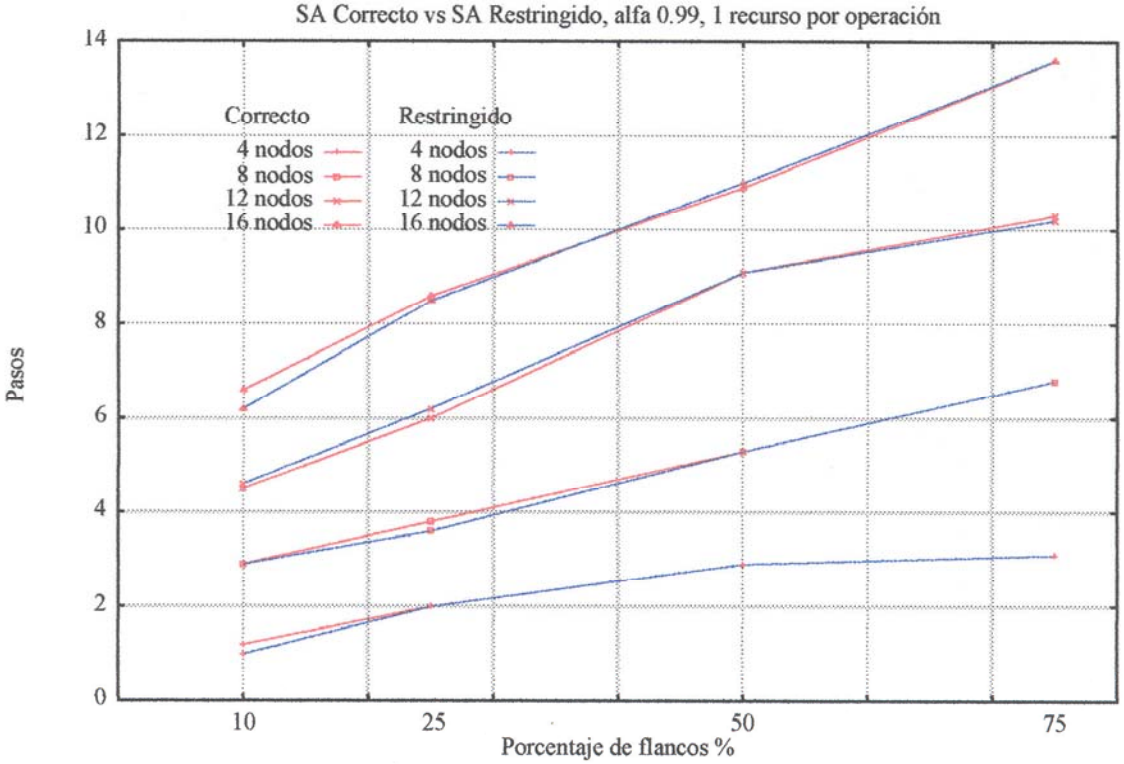


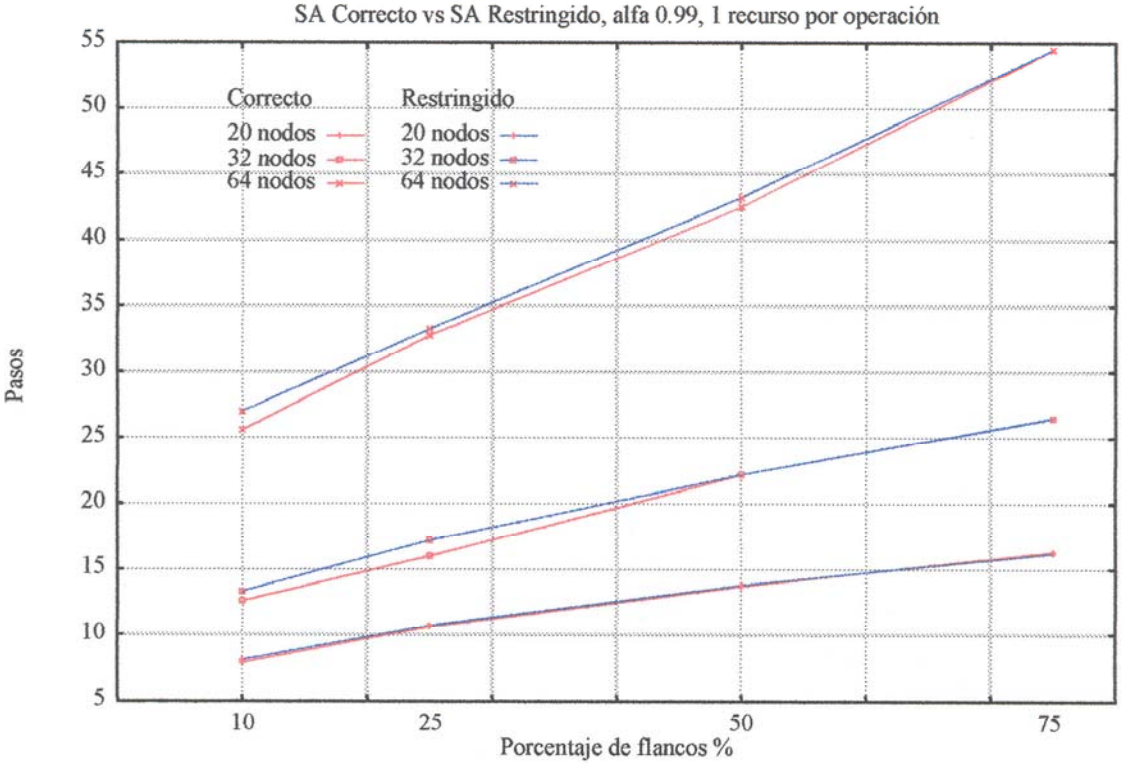
© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004



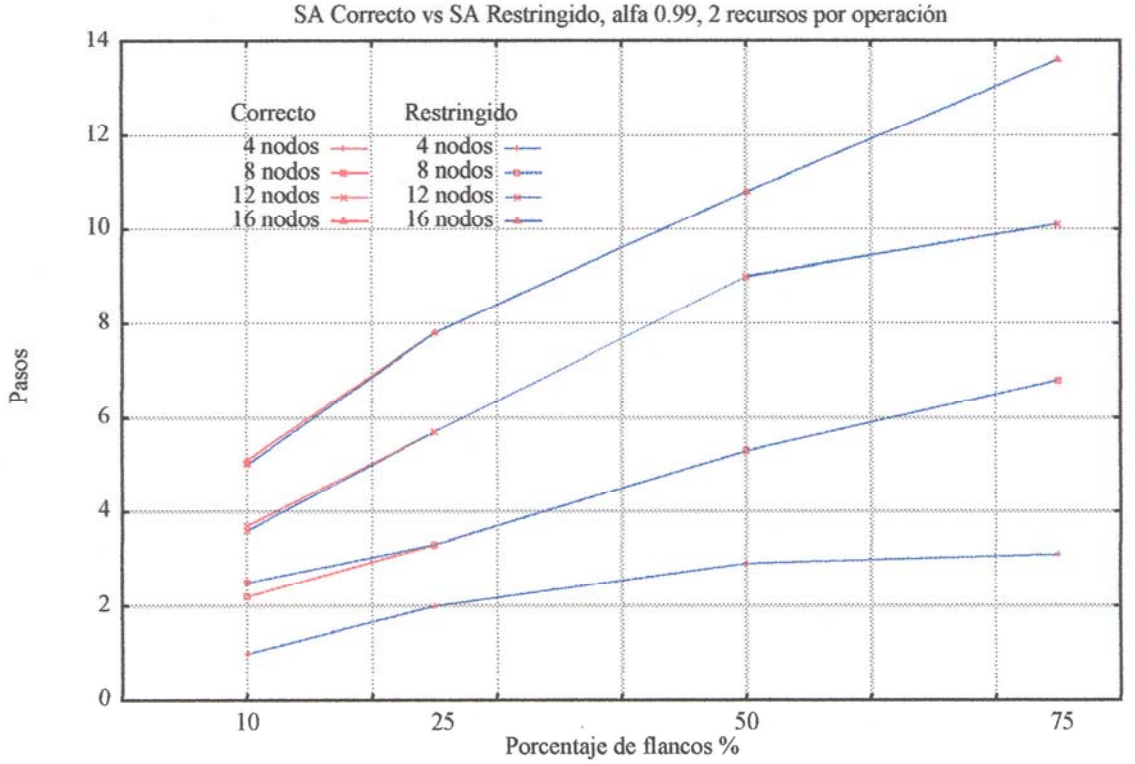


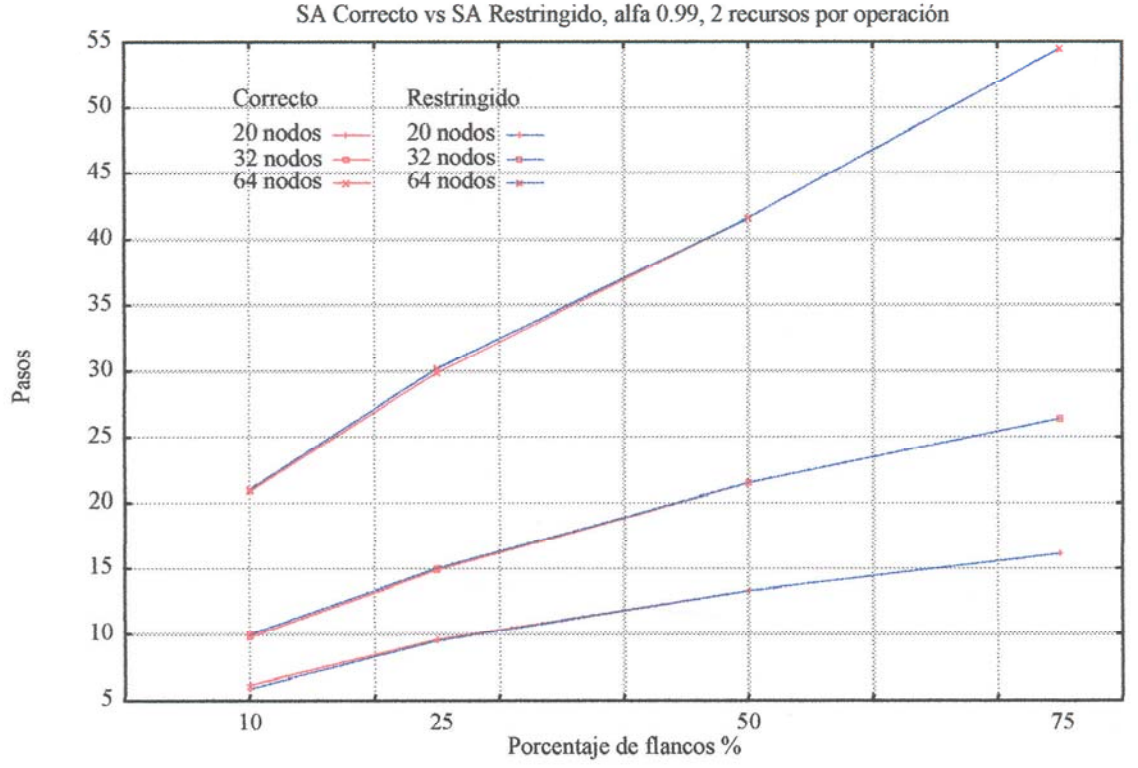
© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004





© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004





© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

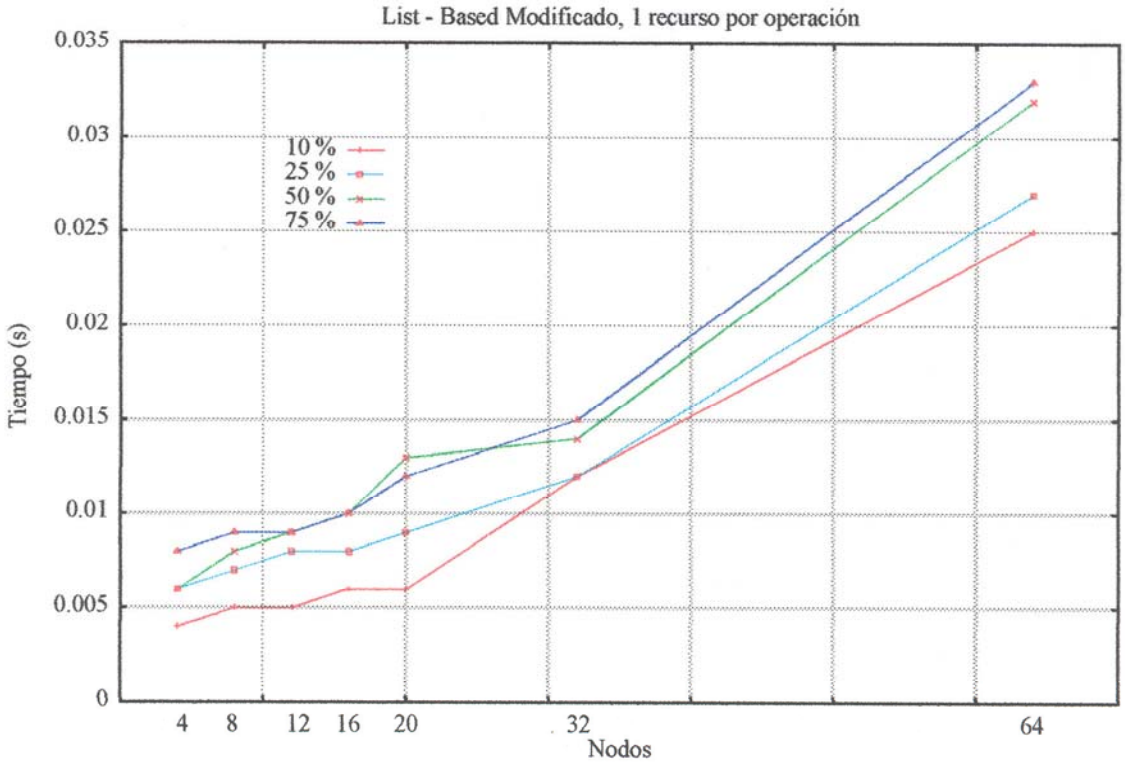
© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

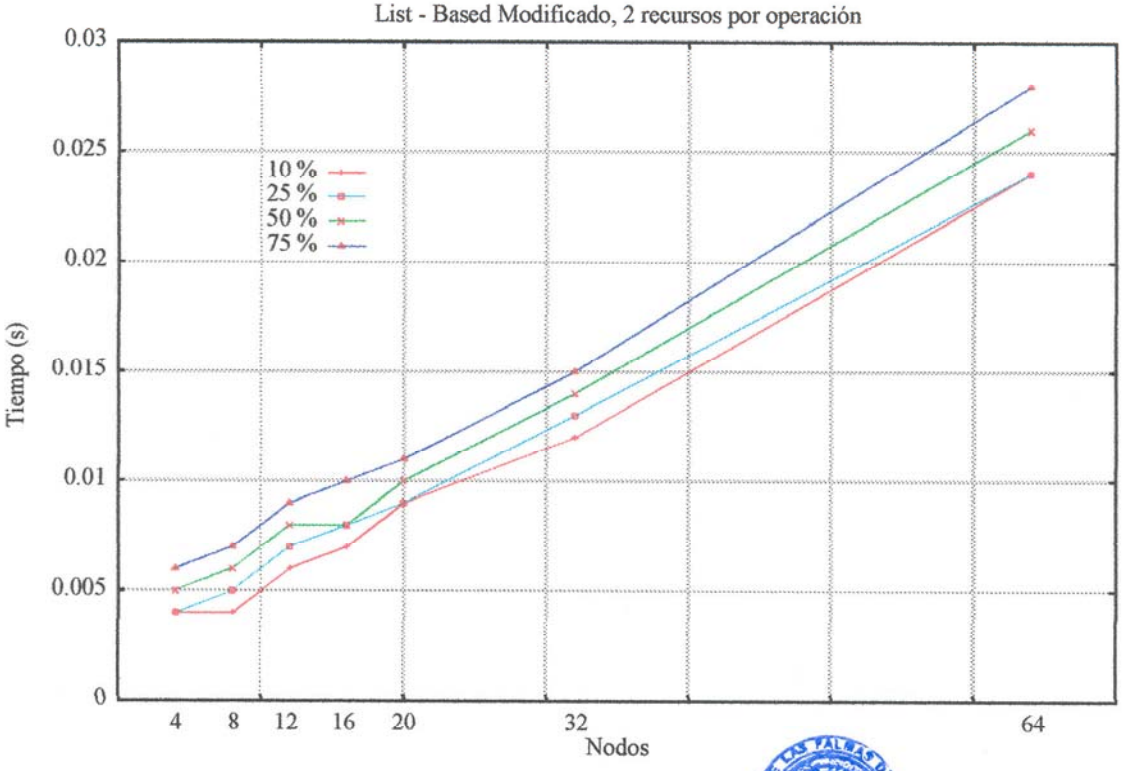
Apéndice C

Resultados de los algoritmos de planificación: list-based

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

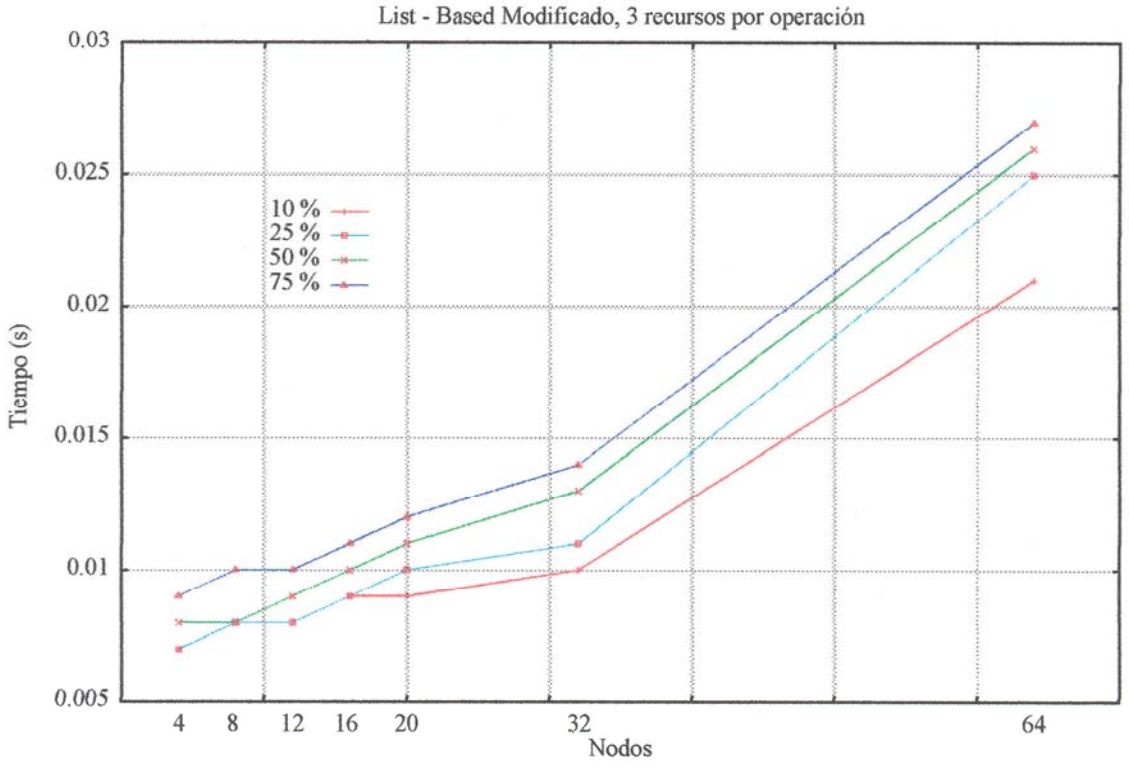
© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

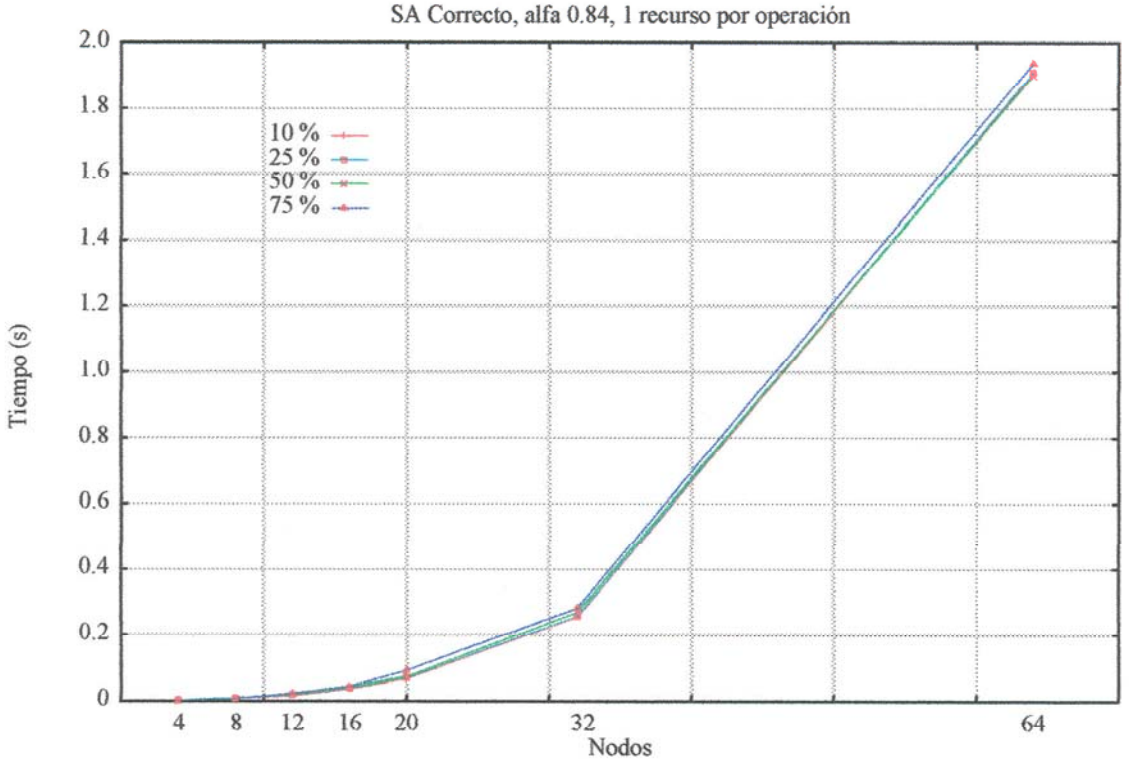




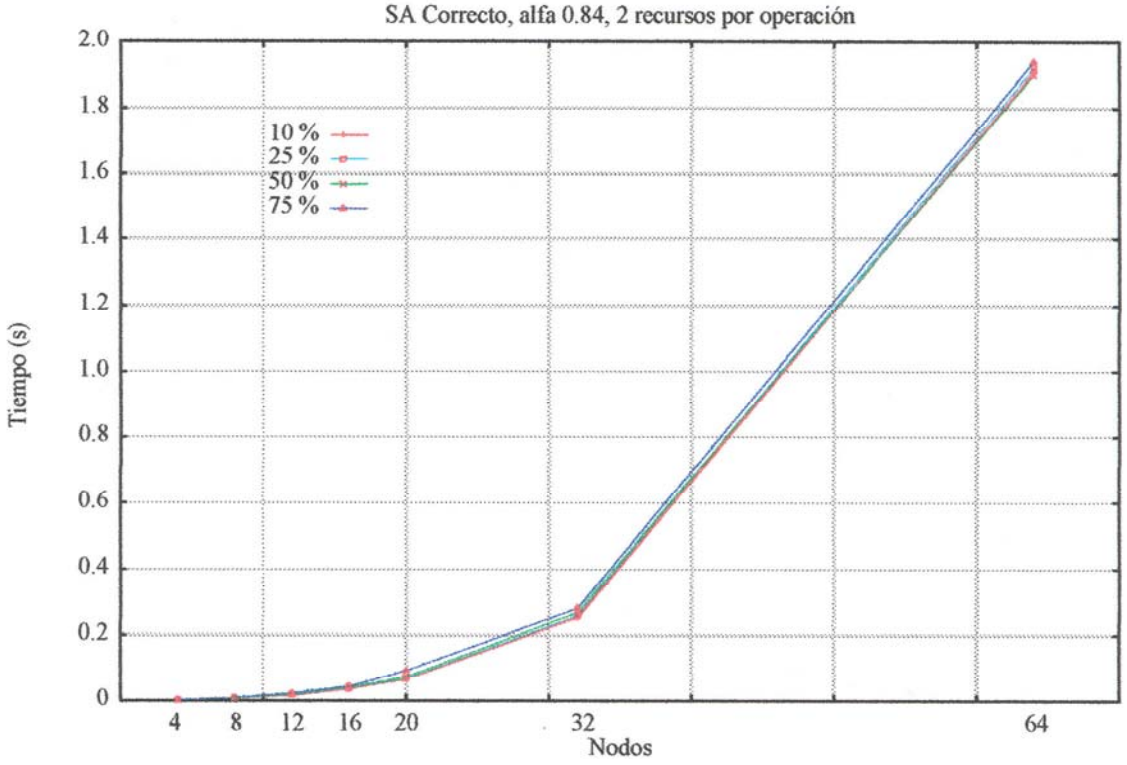
BIBLIOTECA GENERAL

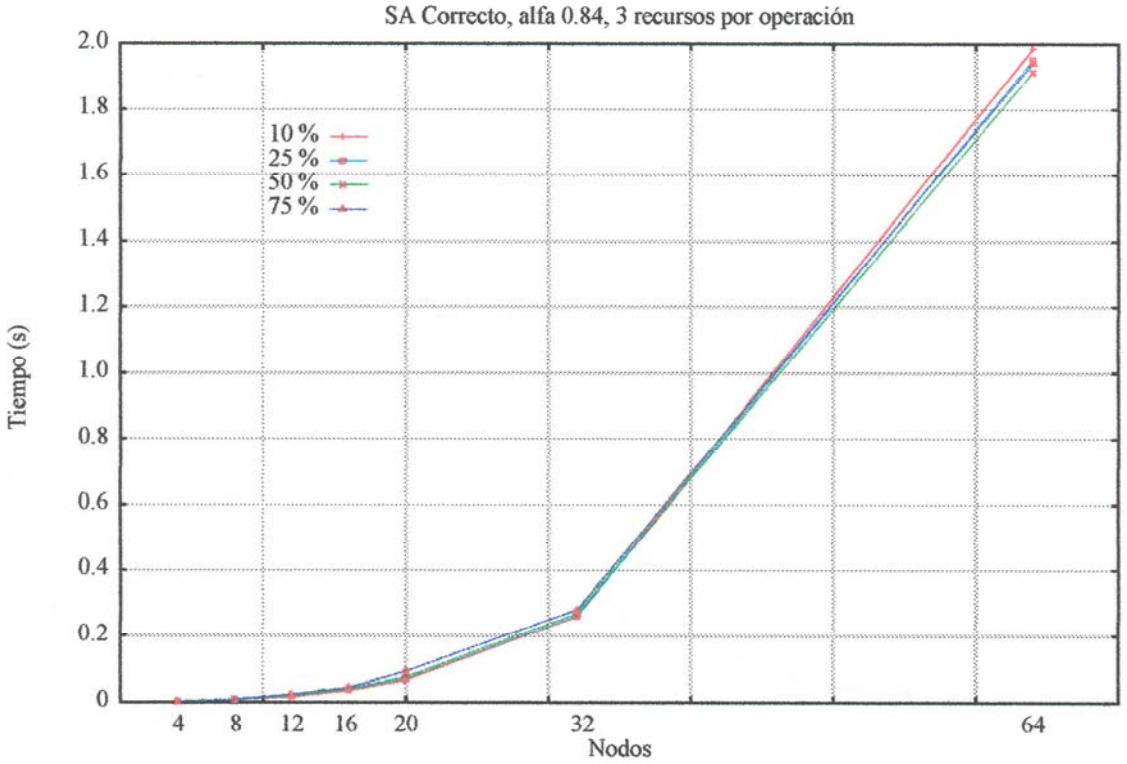
© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004





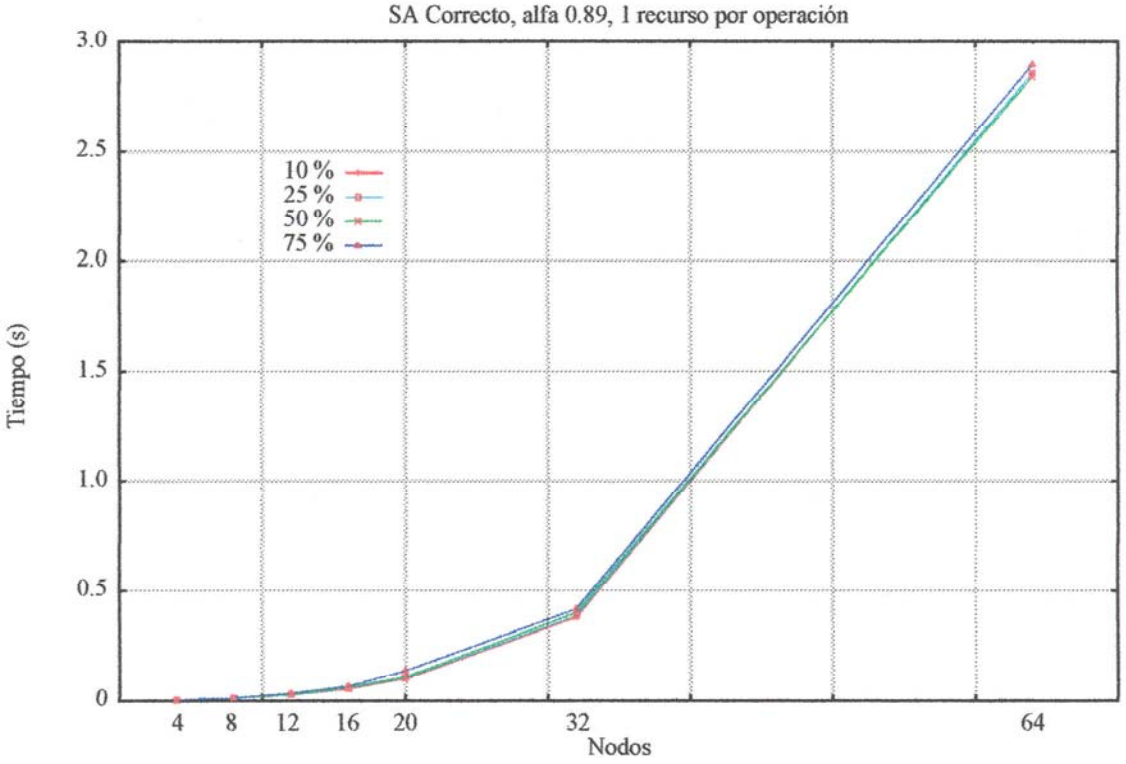
© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

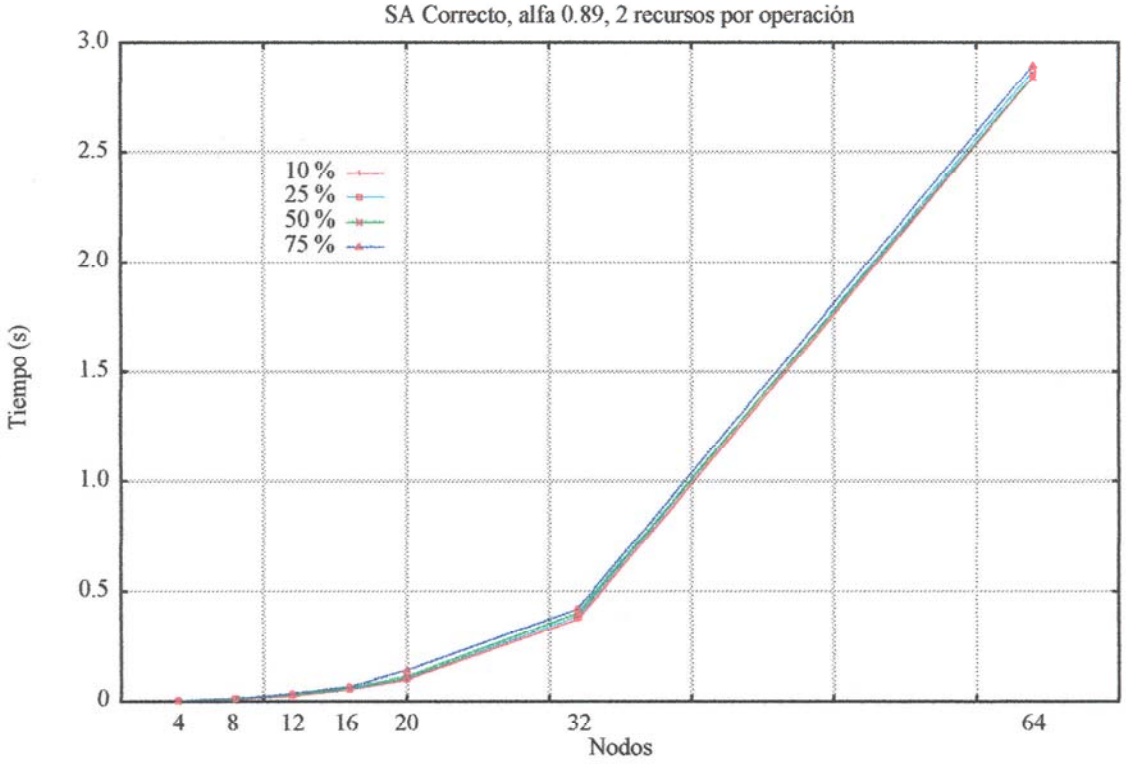




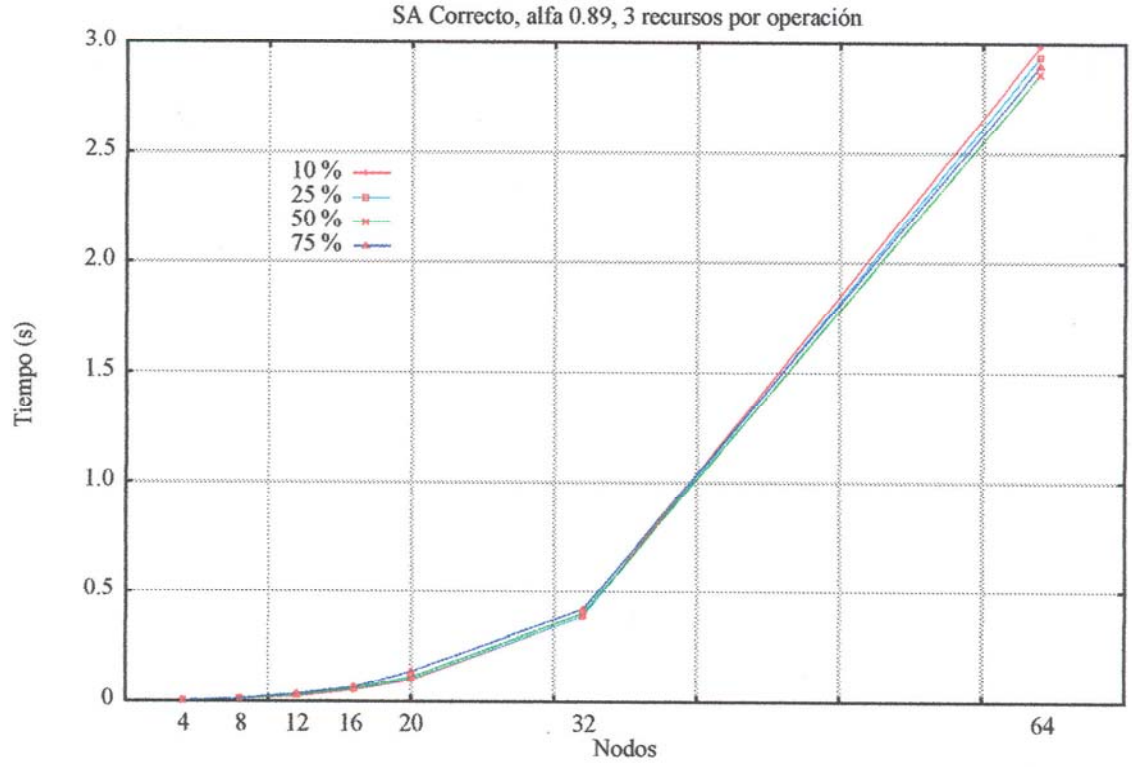
© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

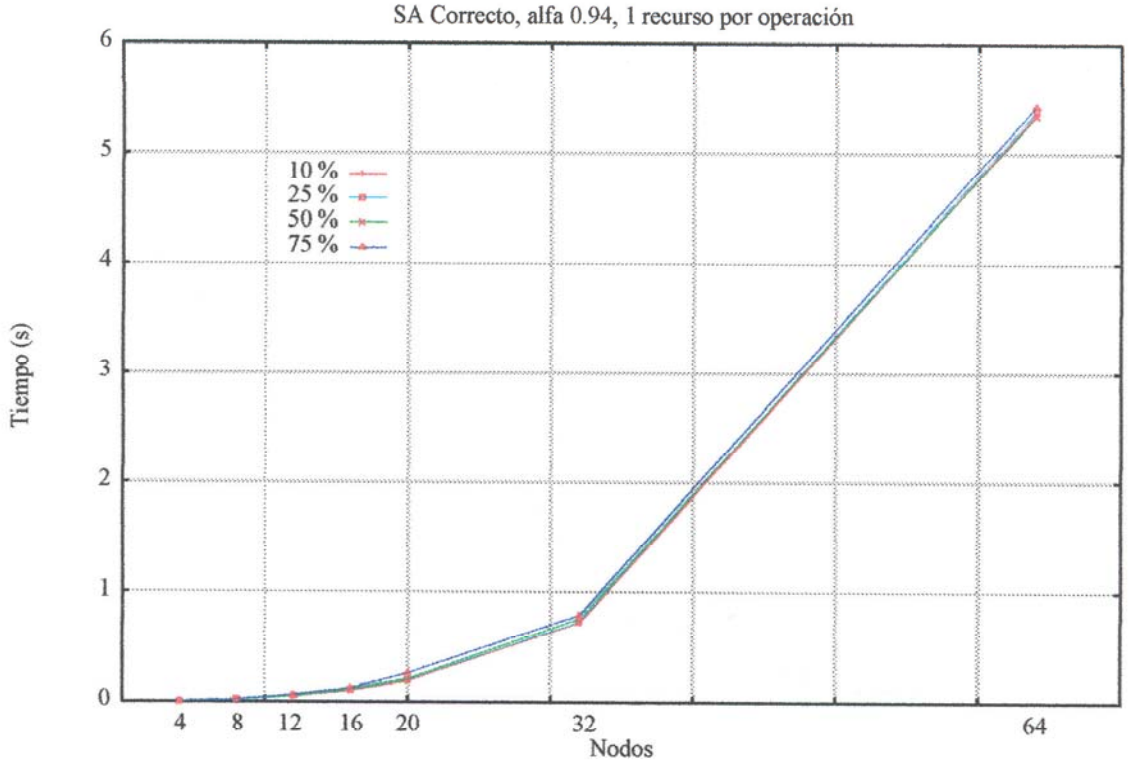
C.6



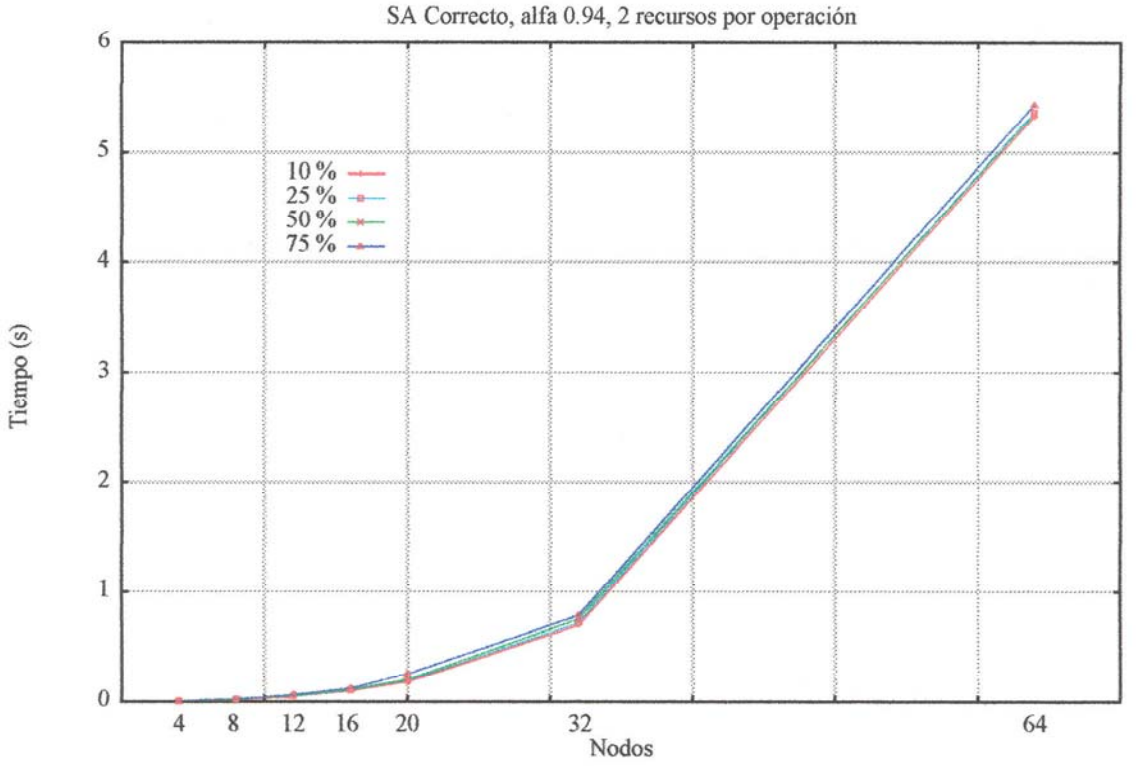


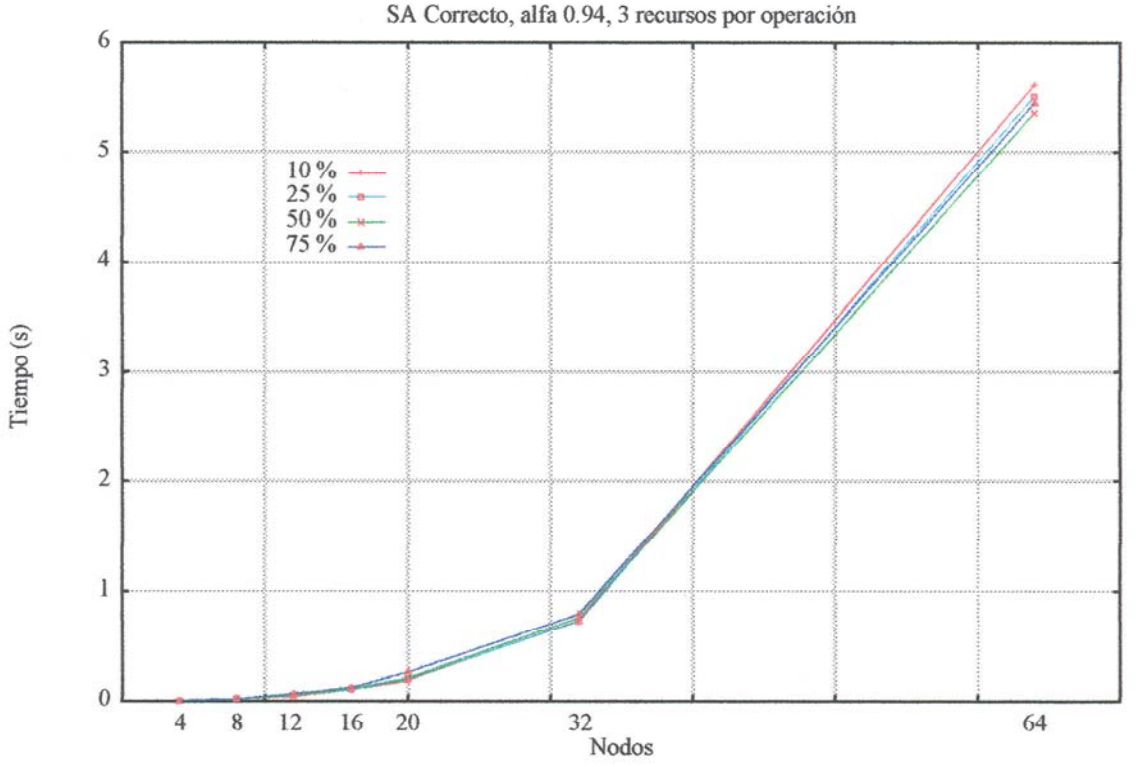
© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004



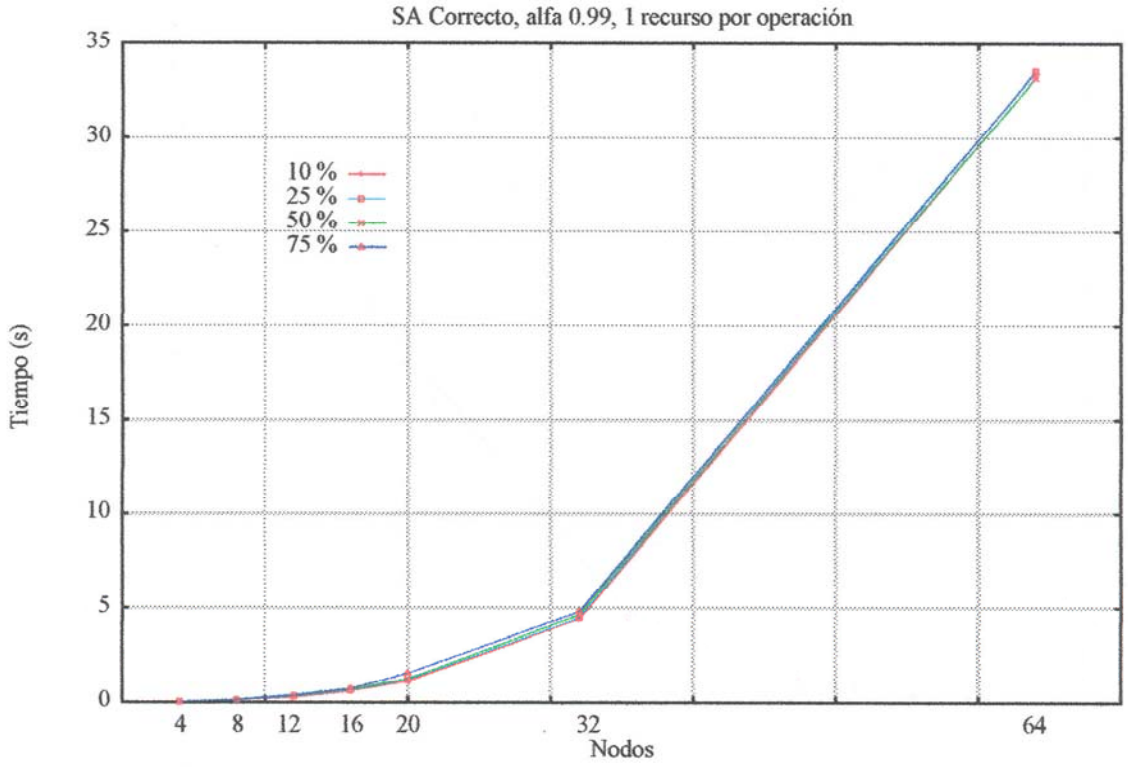


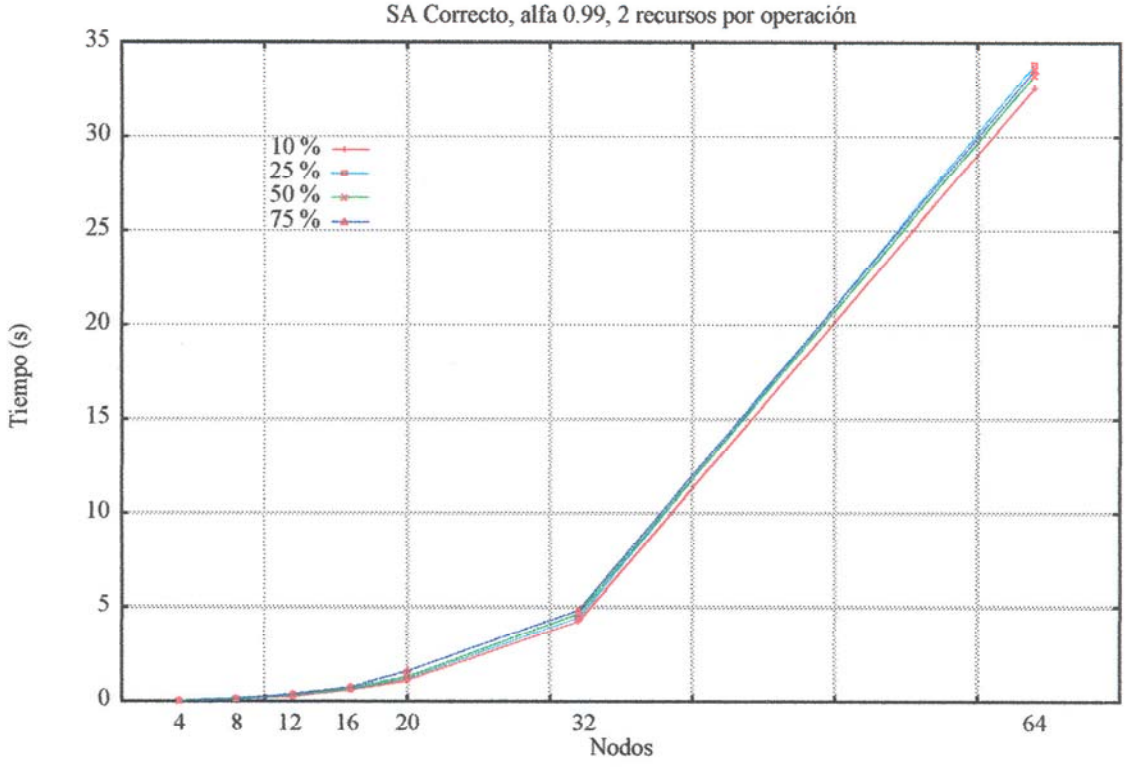
© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004





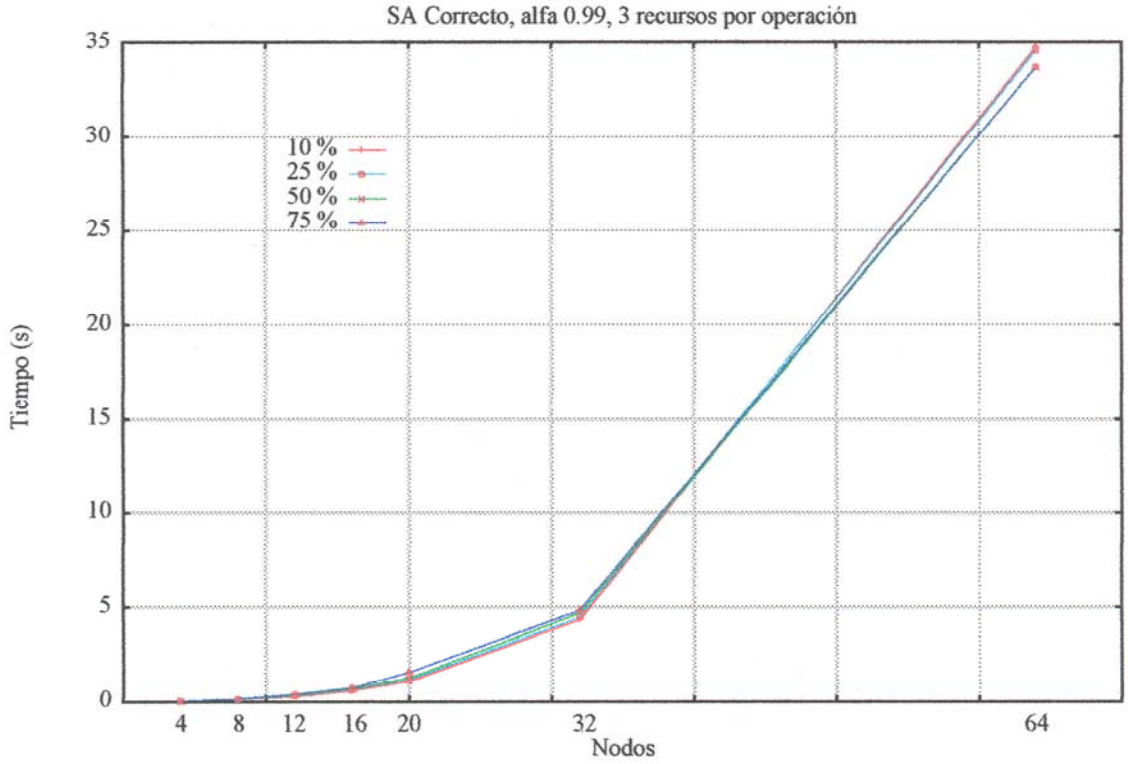
© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004



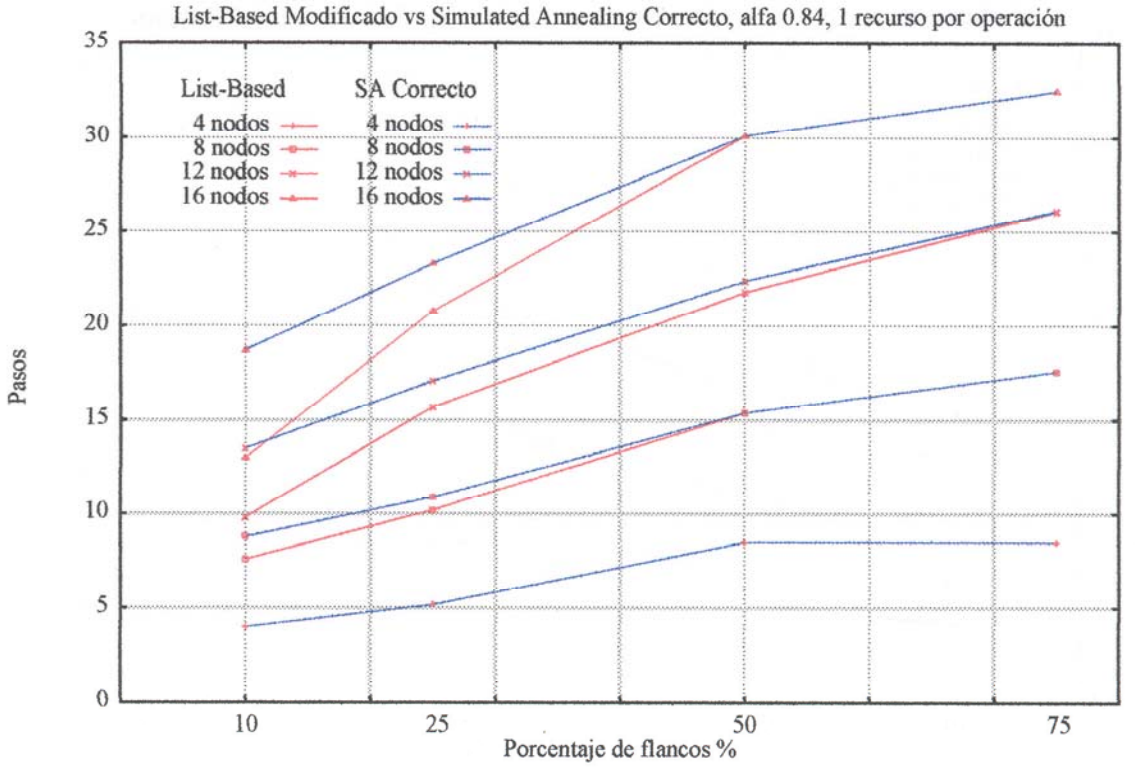


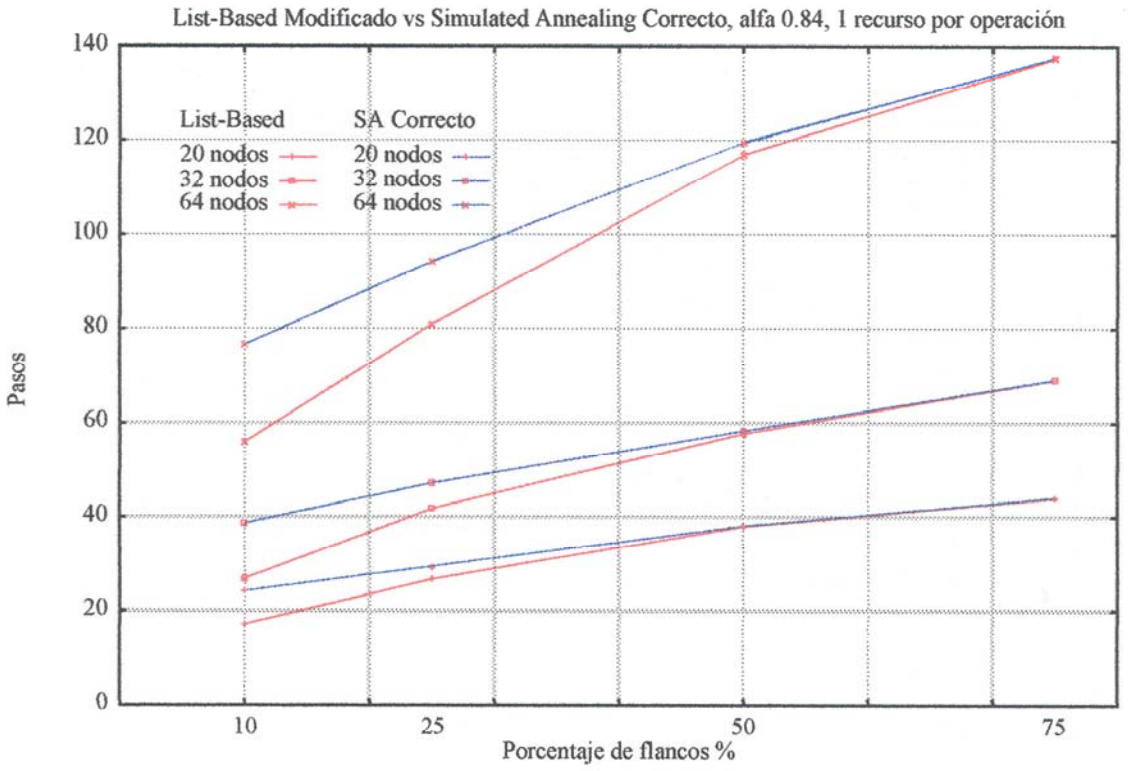
© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

C.10

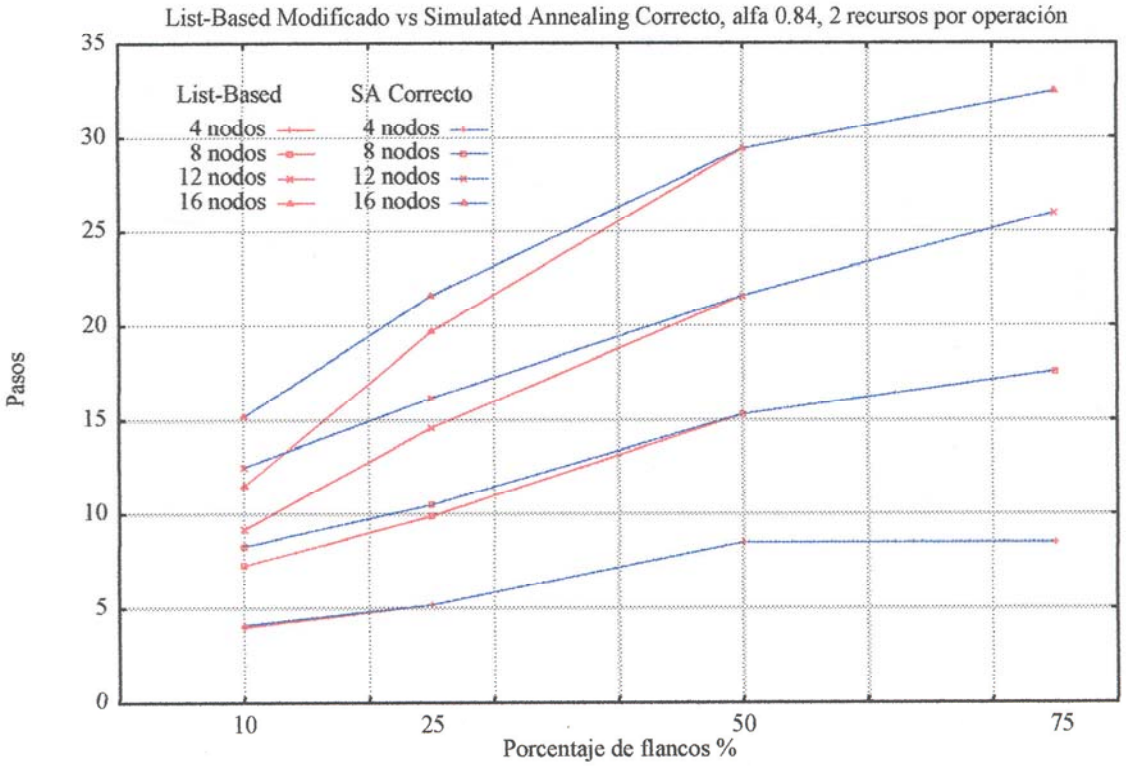


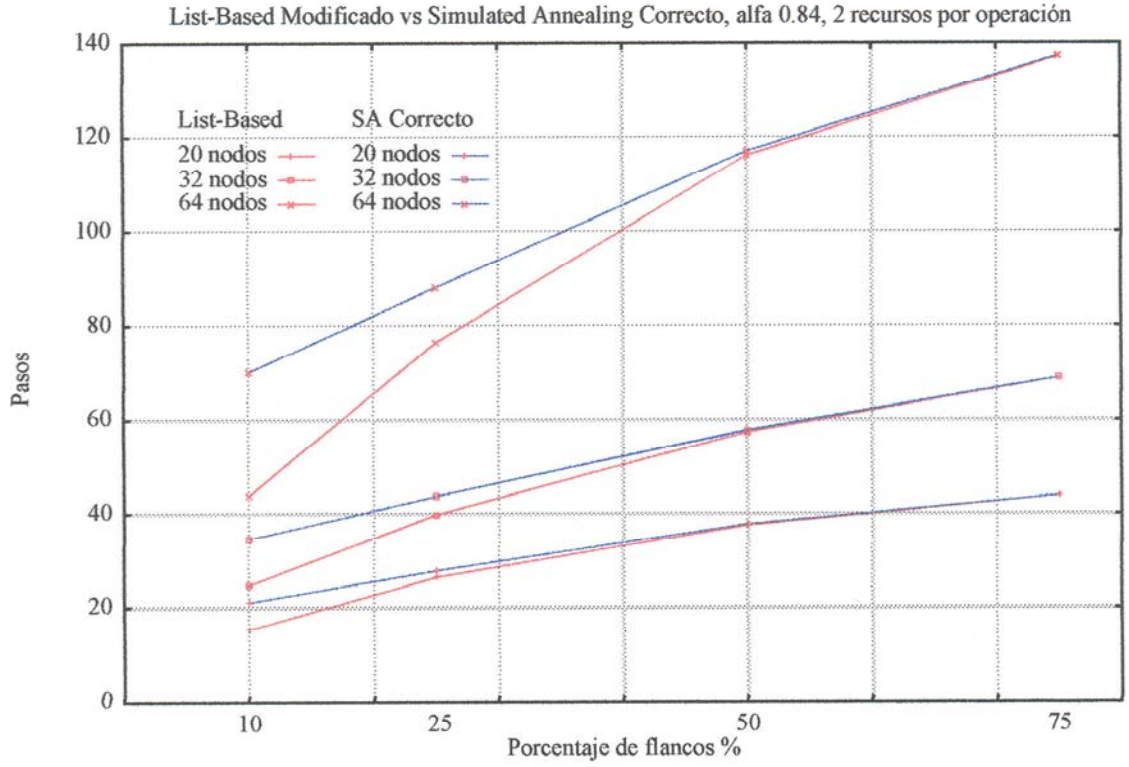
© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004



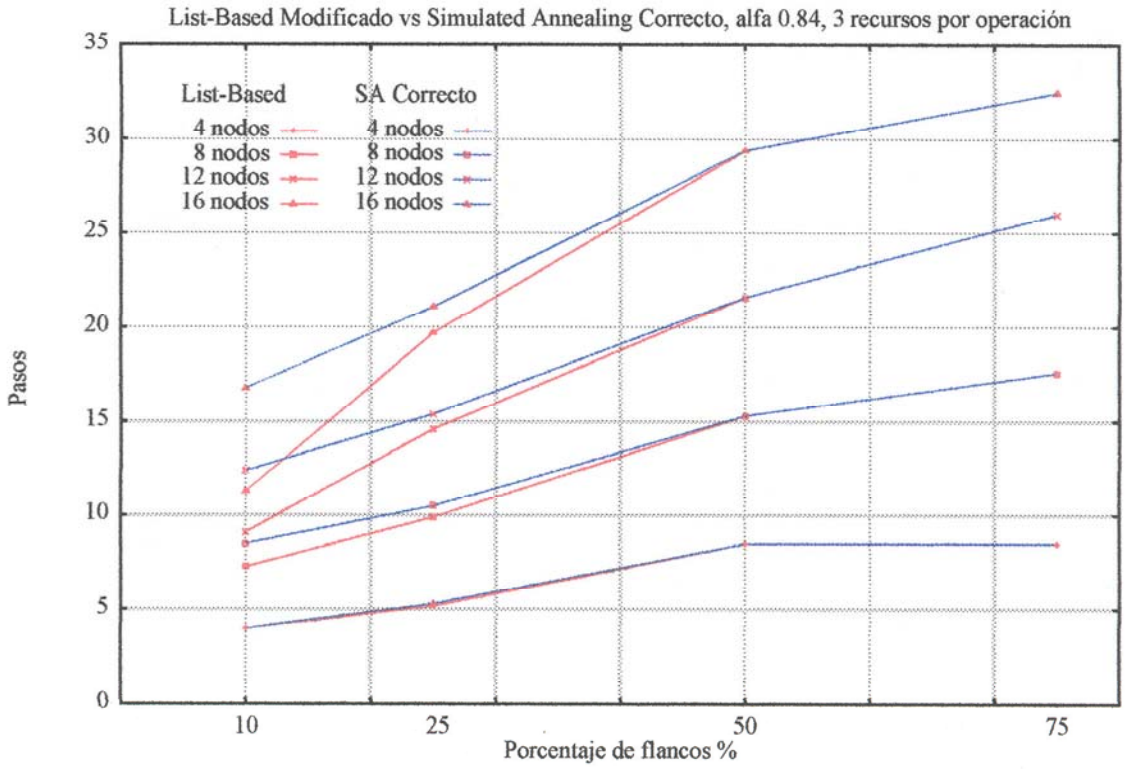


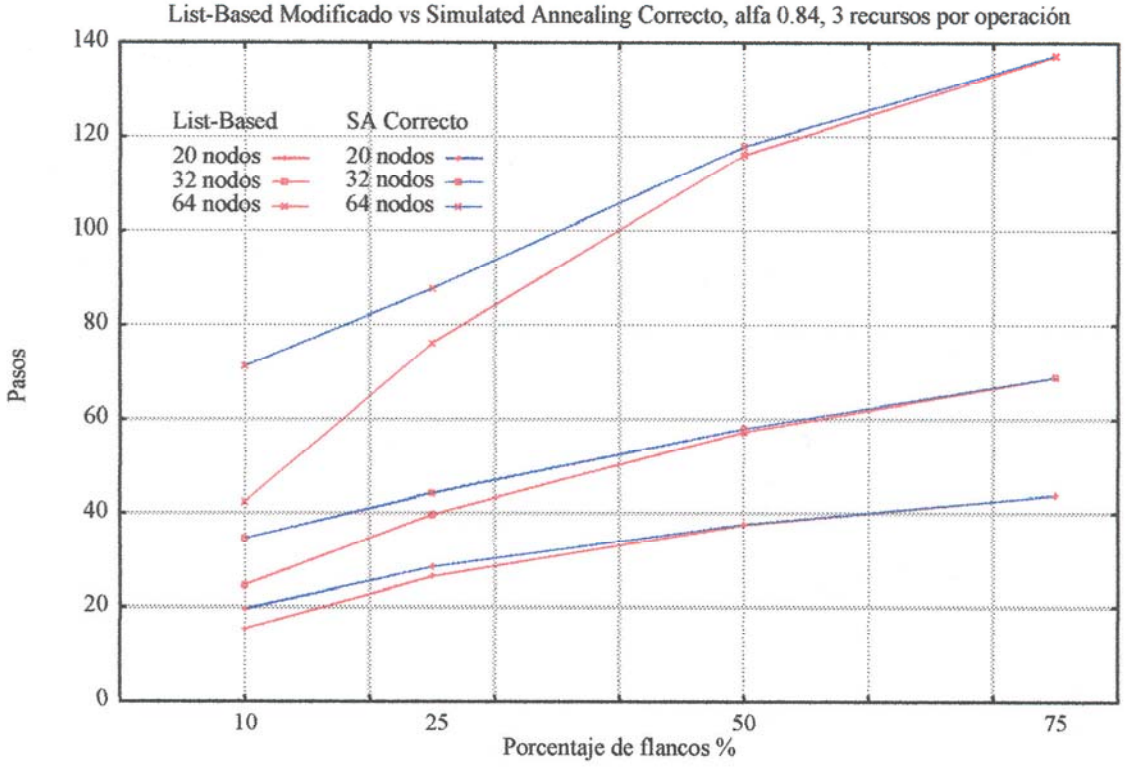
© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004



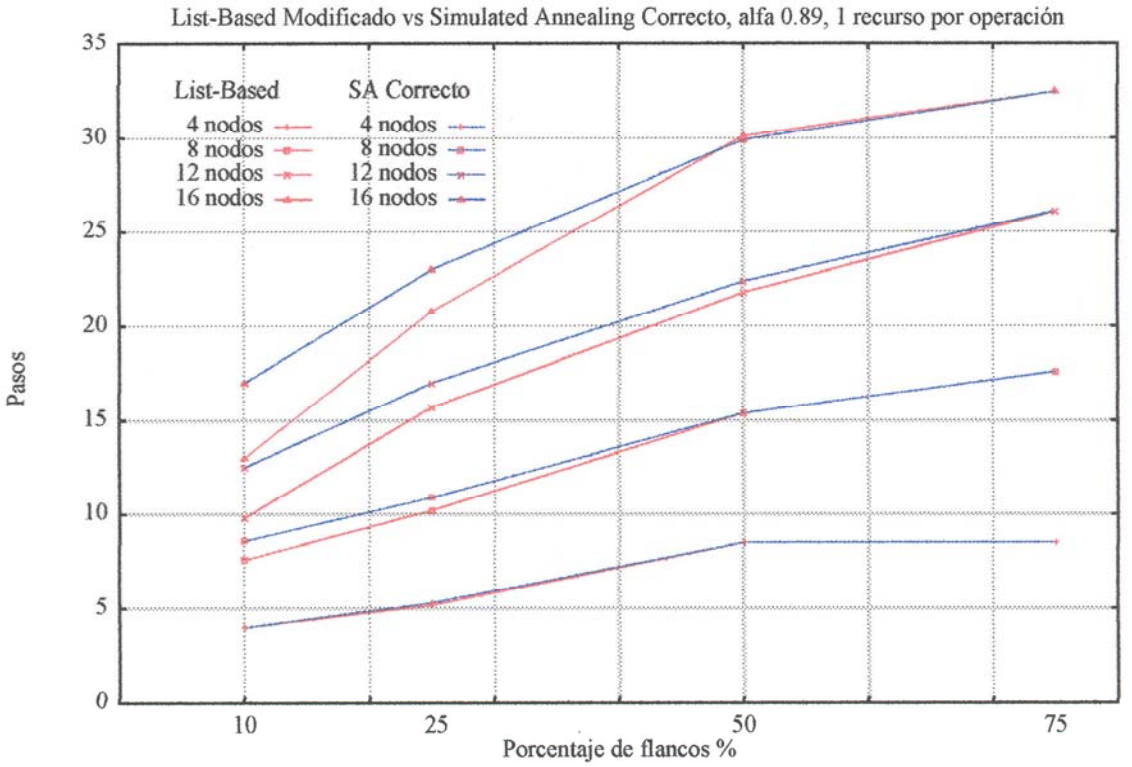


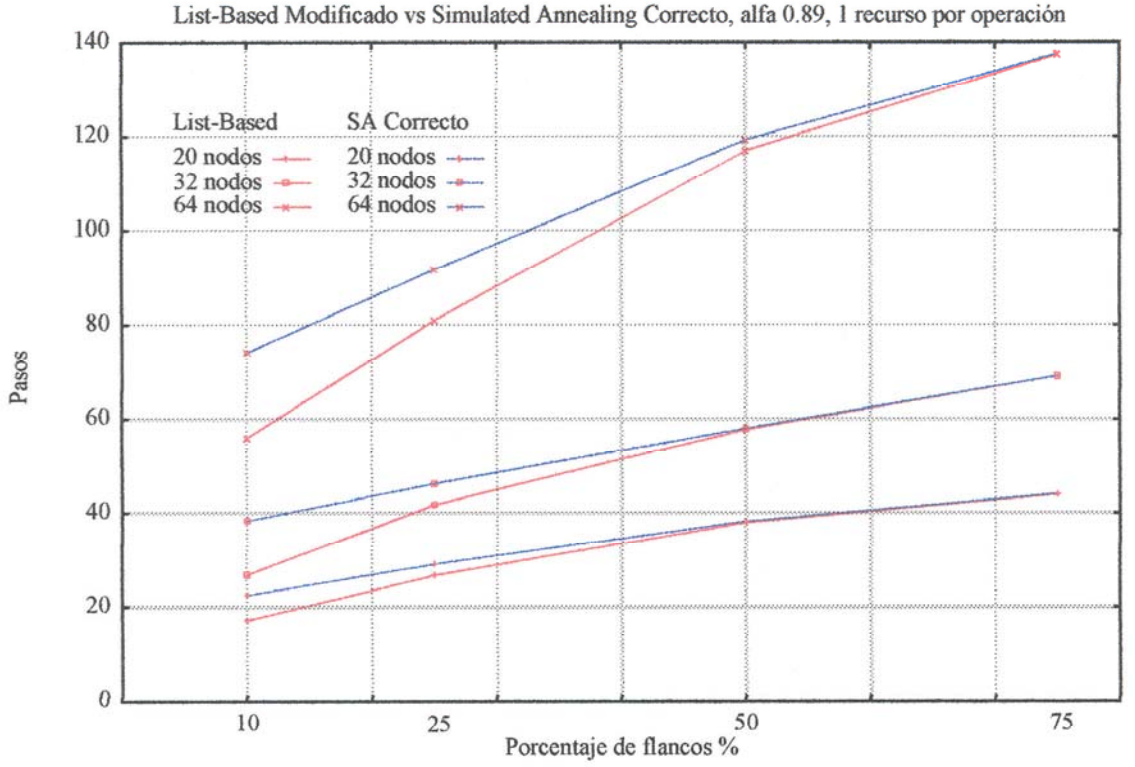
© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004



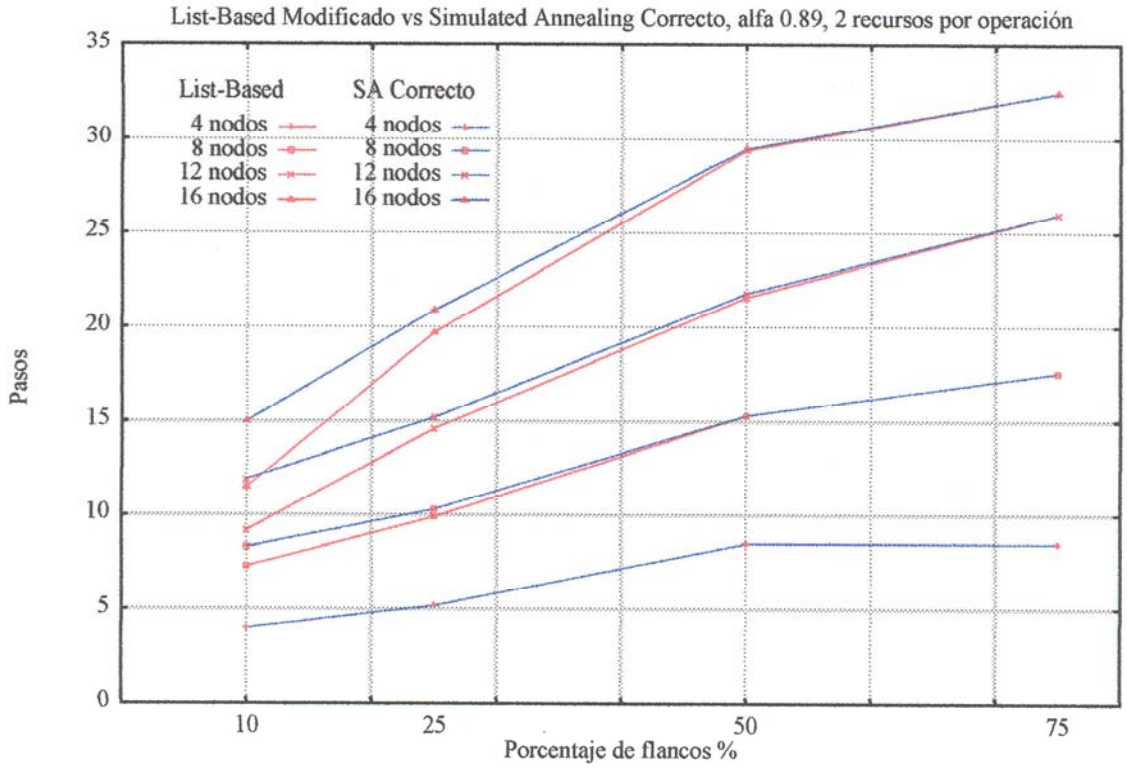


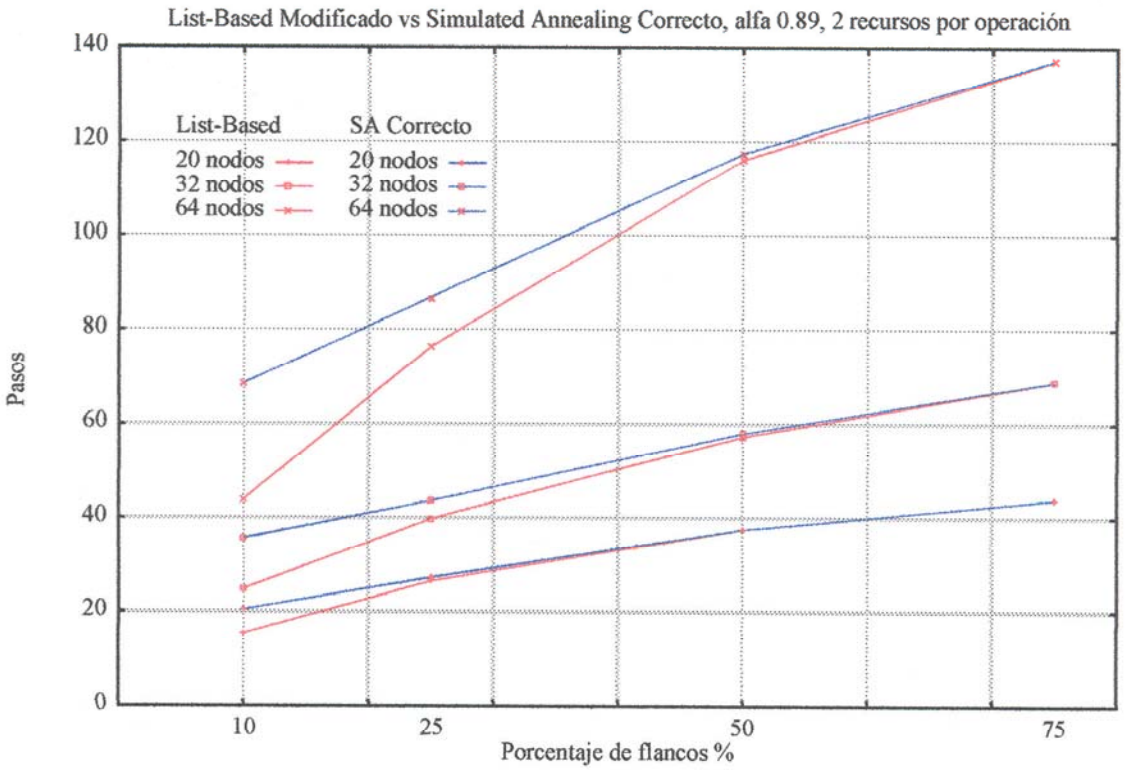
© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004



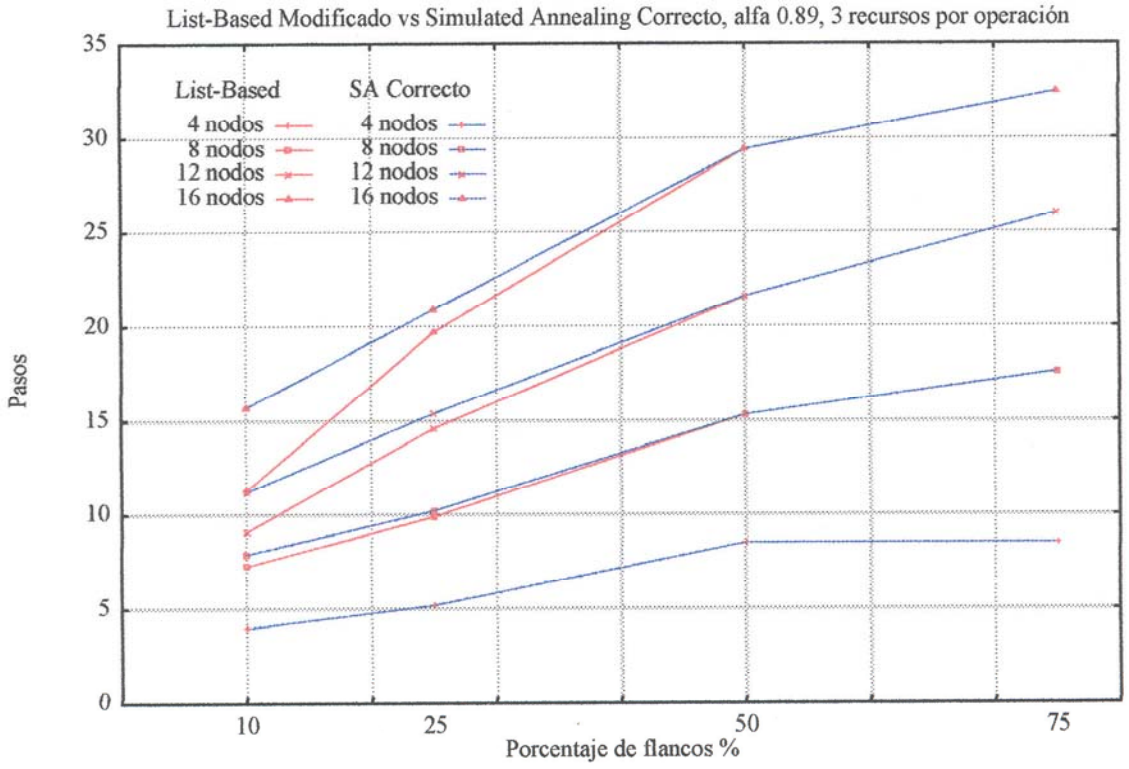


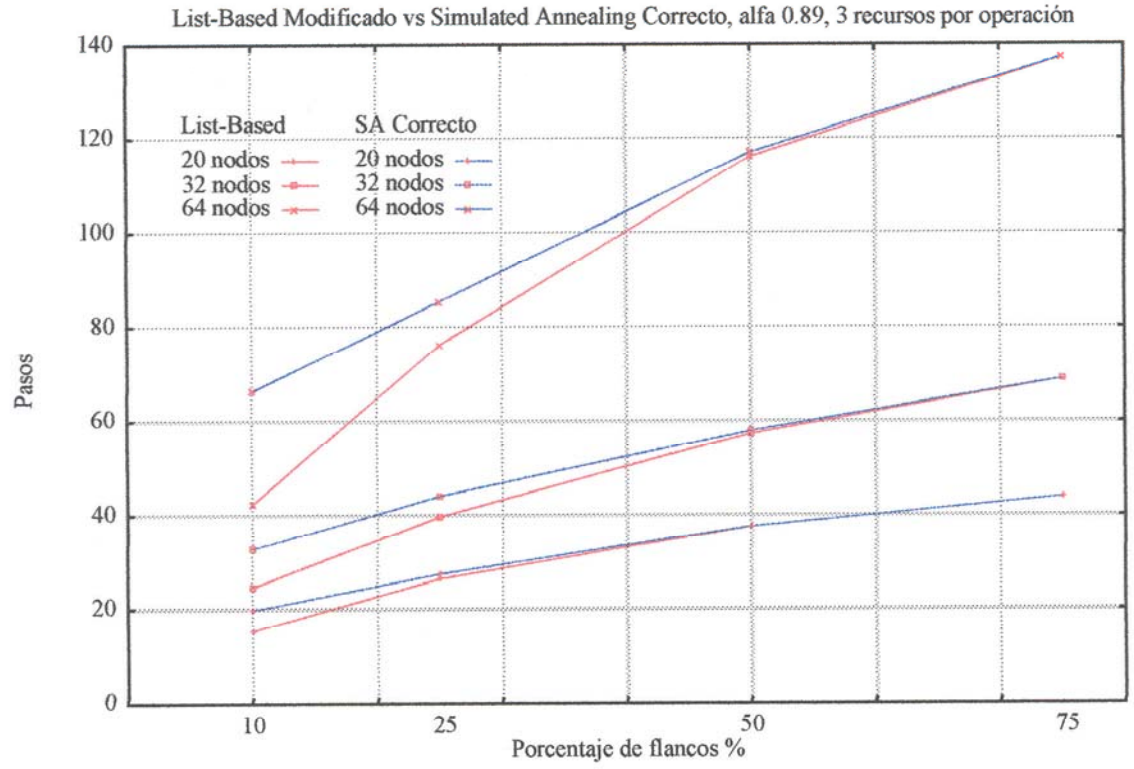
© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004



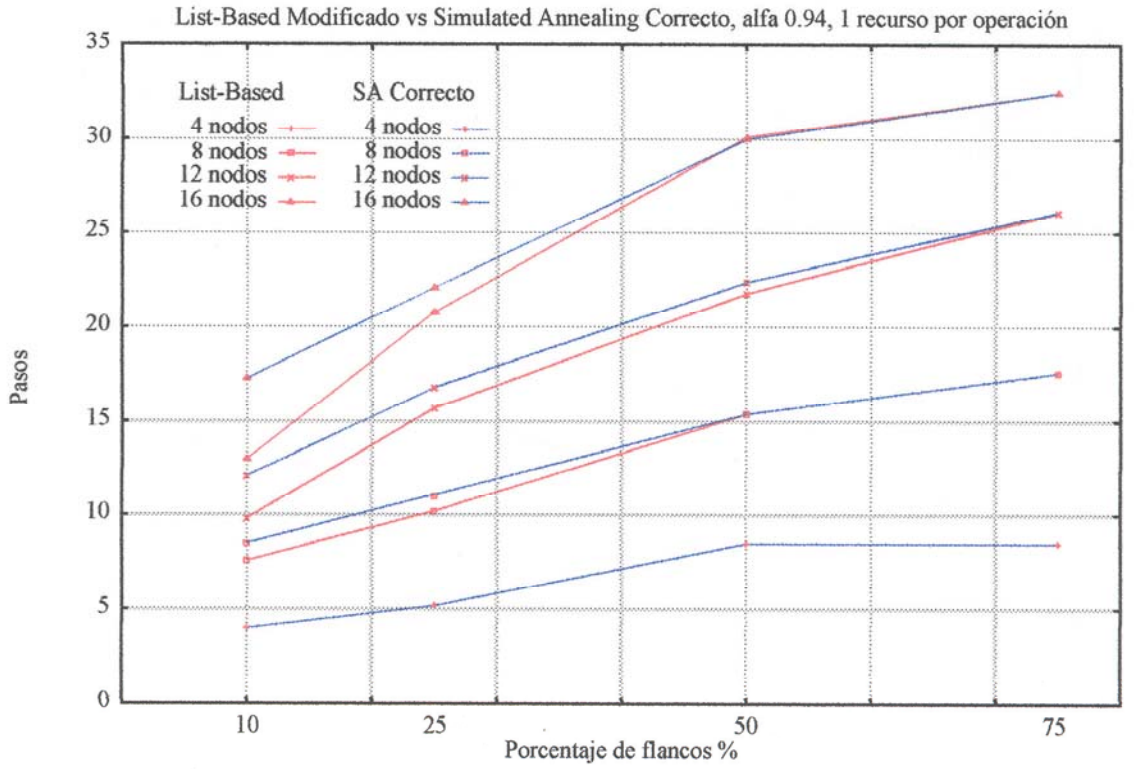


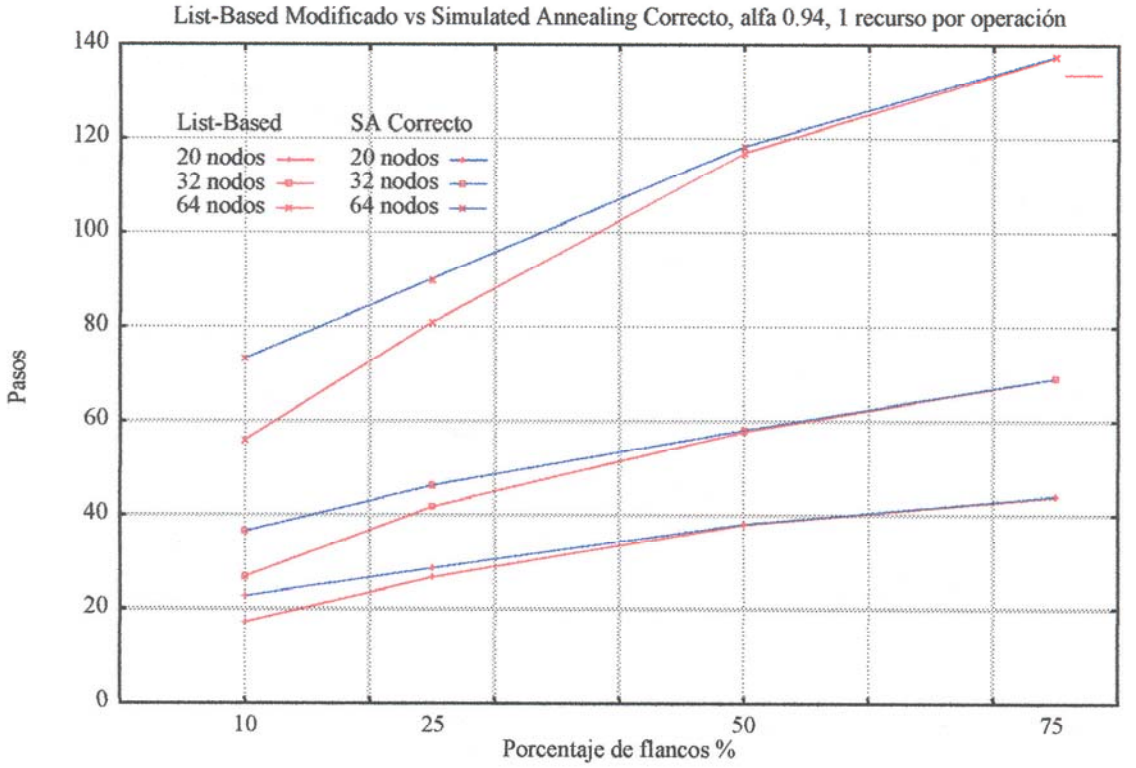
© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004



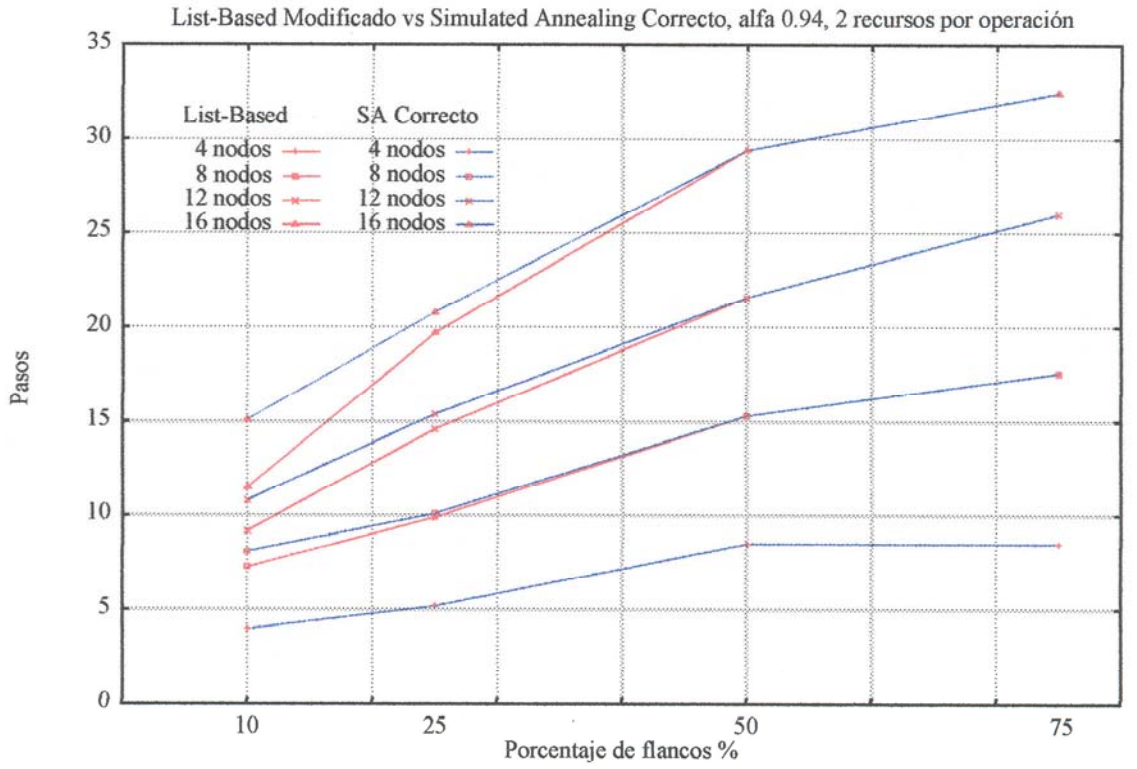


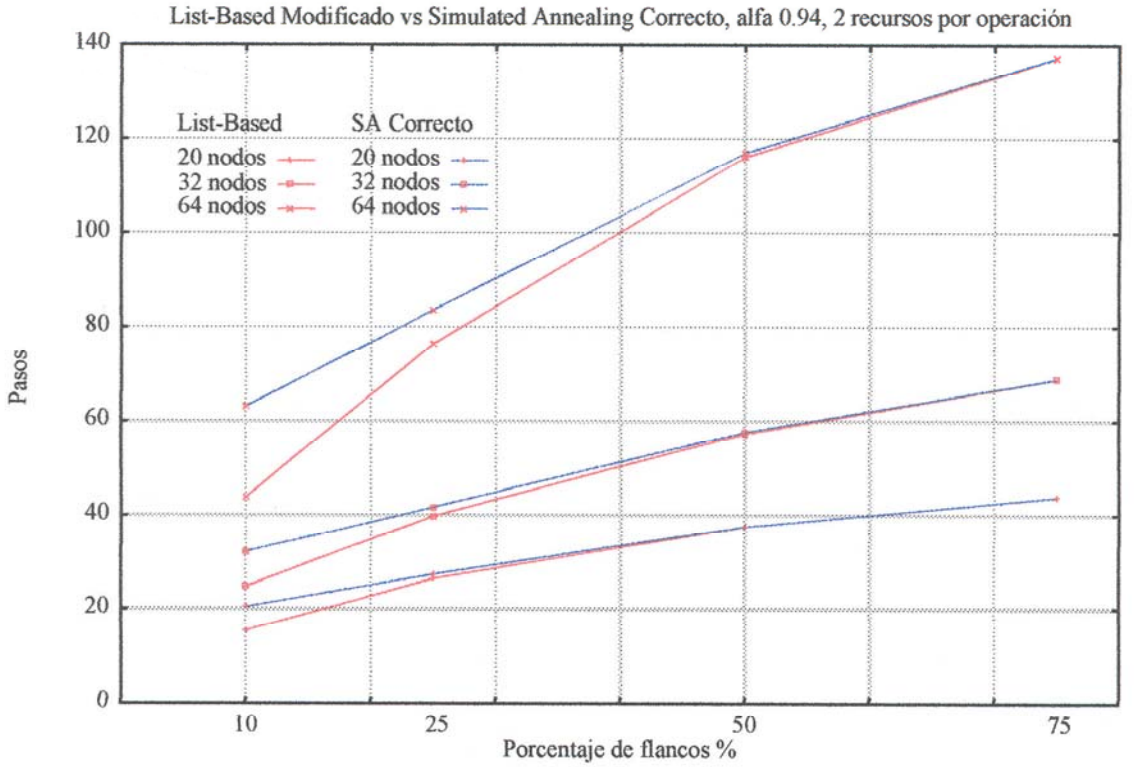
© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004



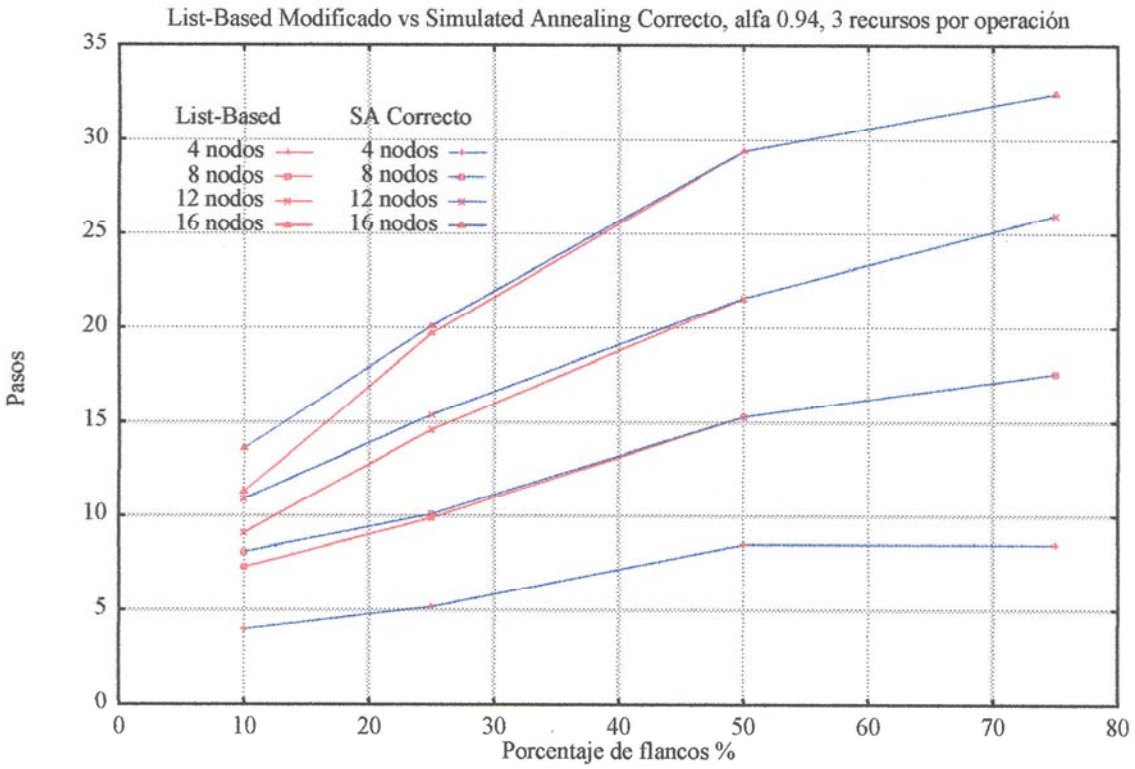


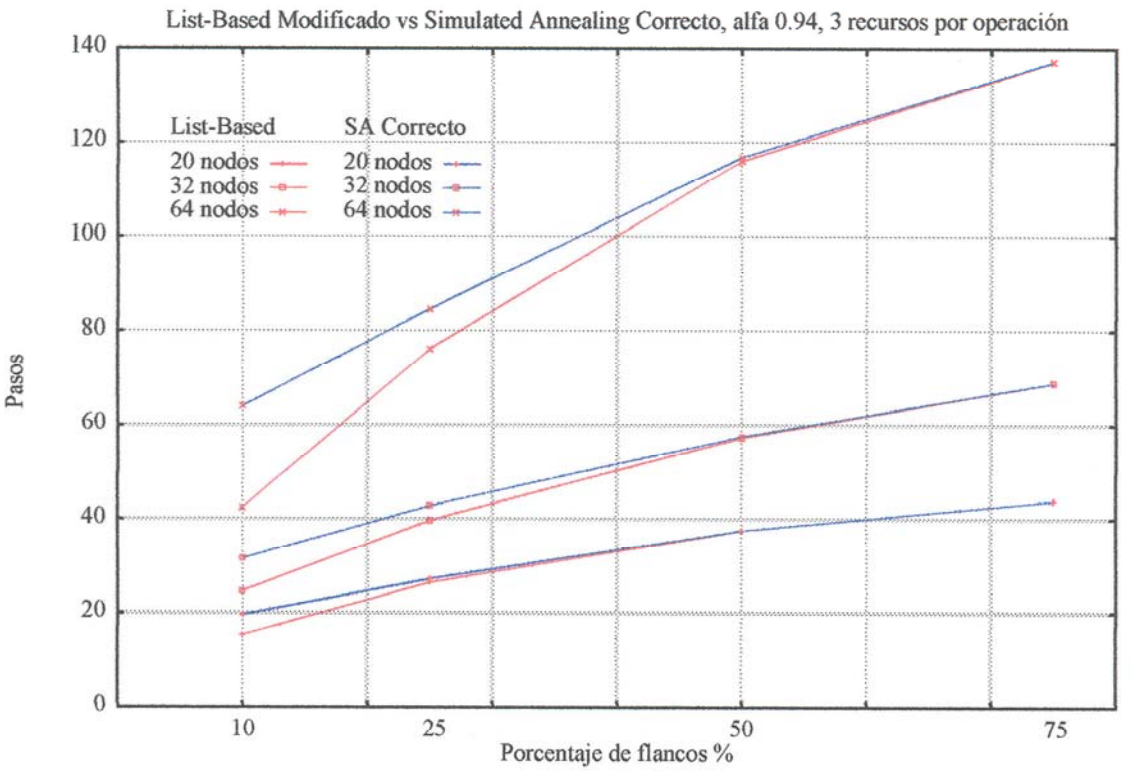
© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004



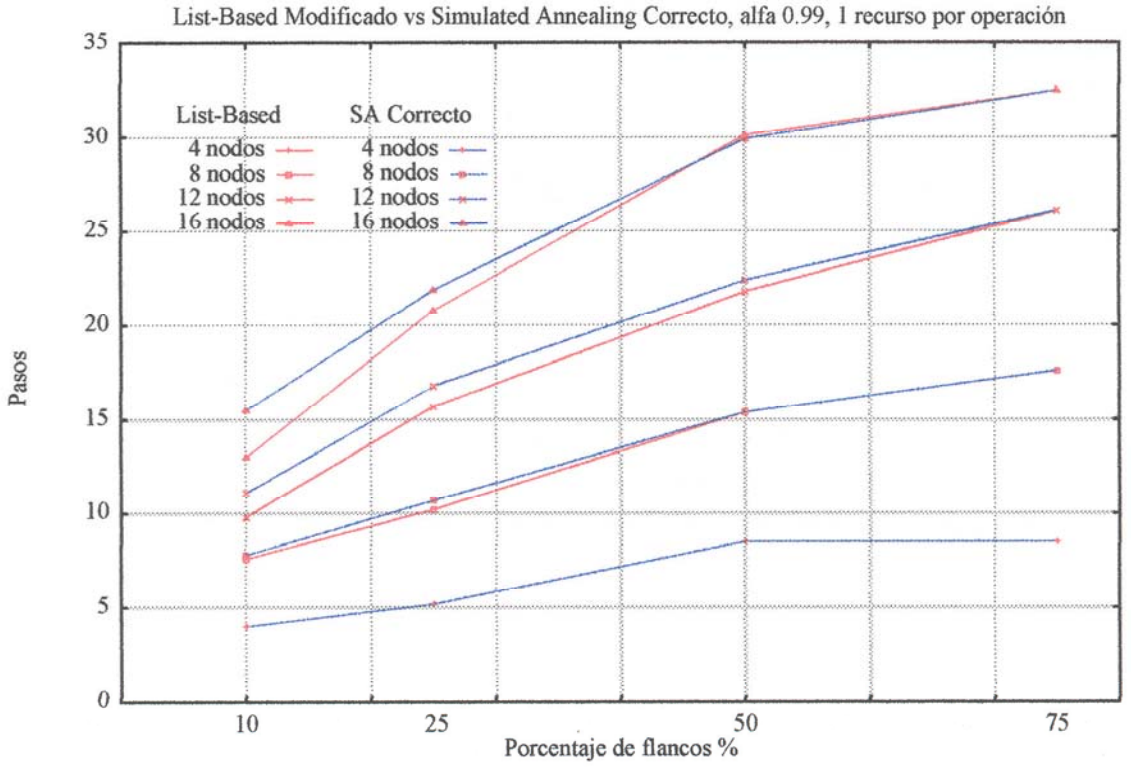


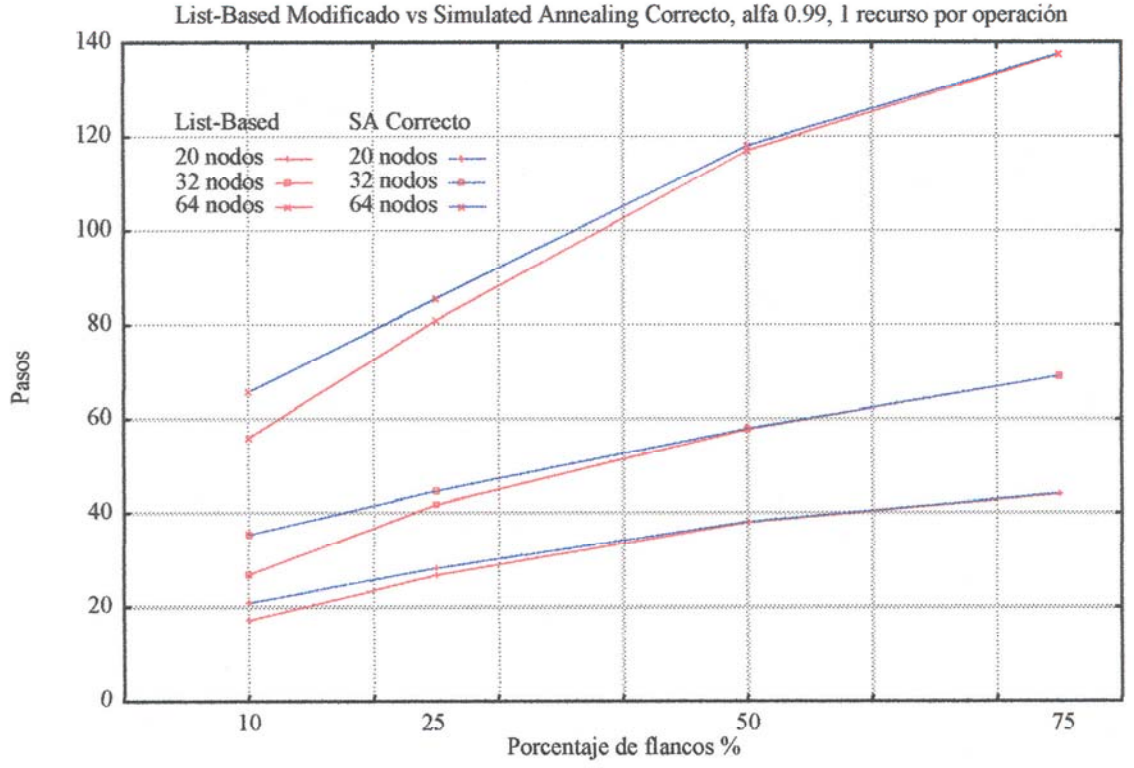
© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004



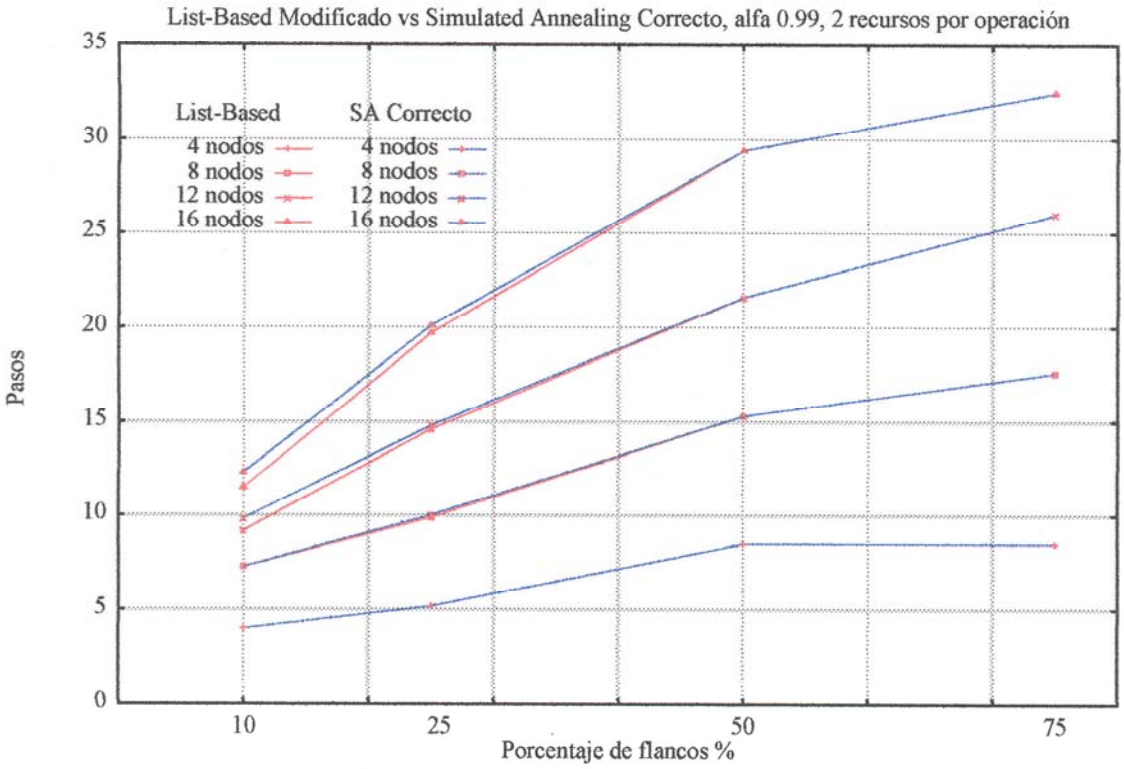


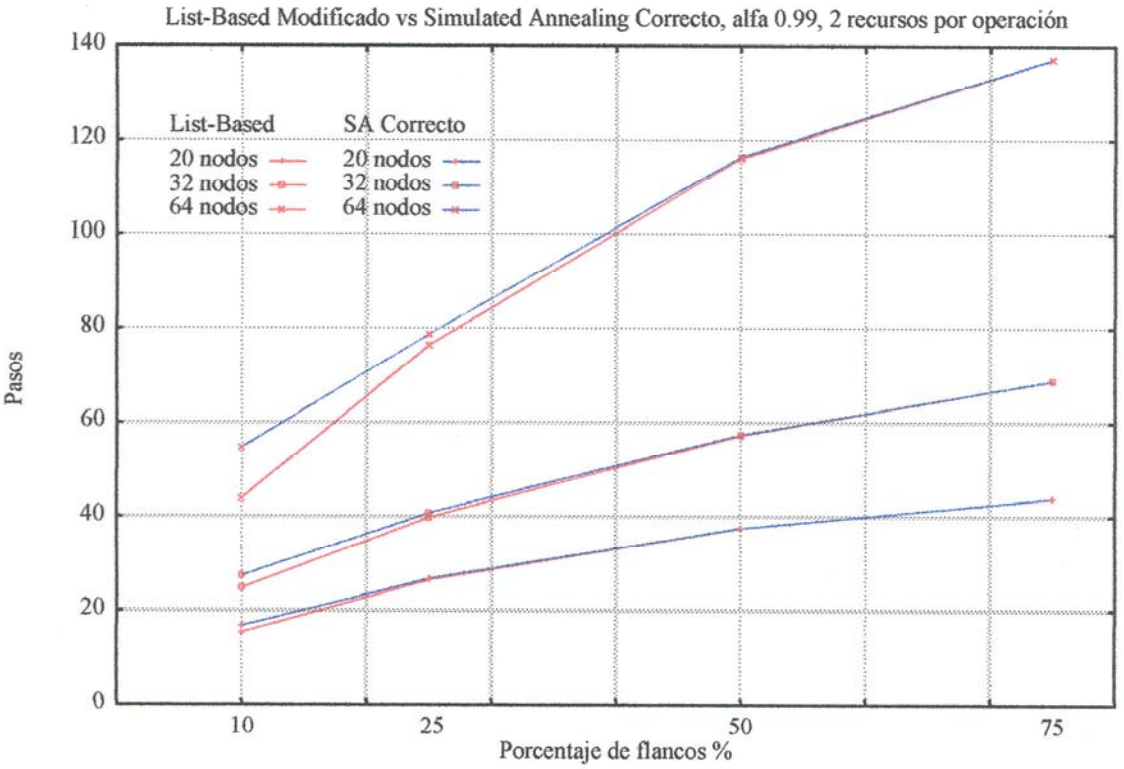
© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004



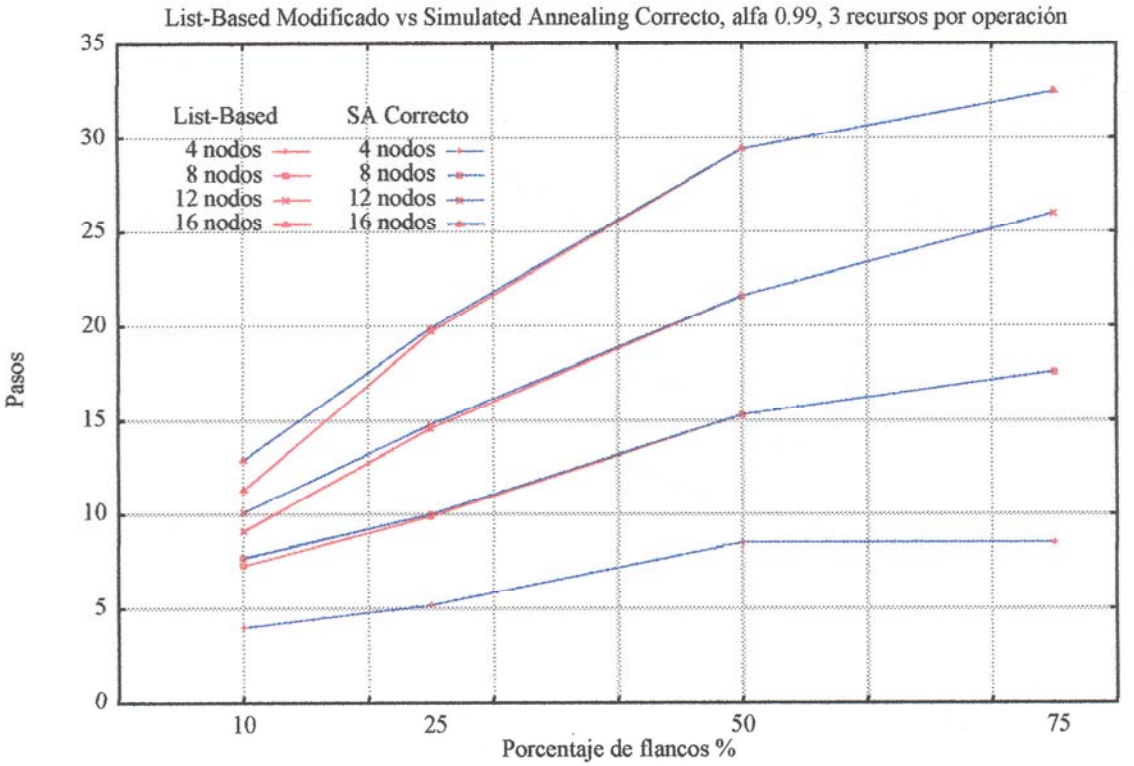


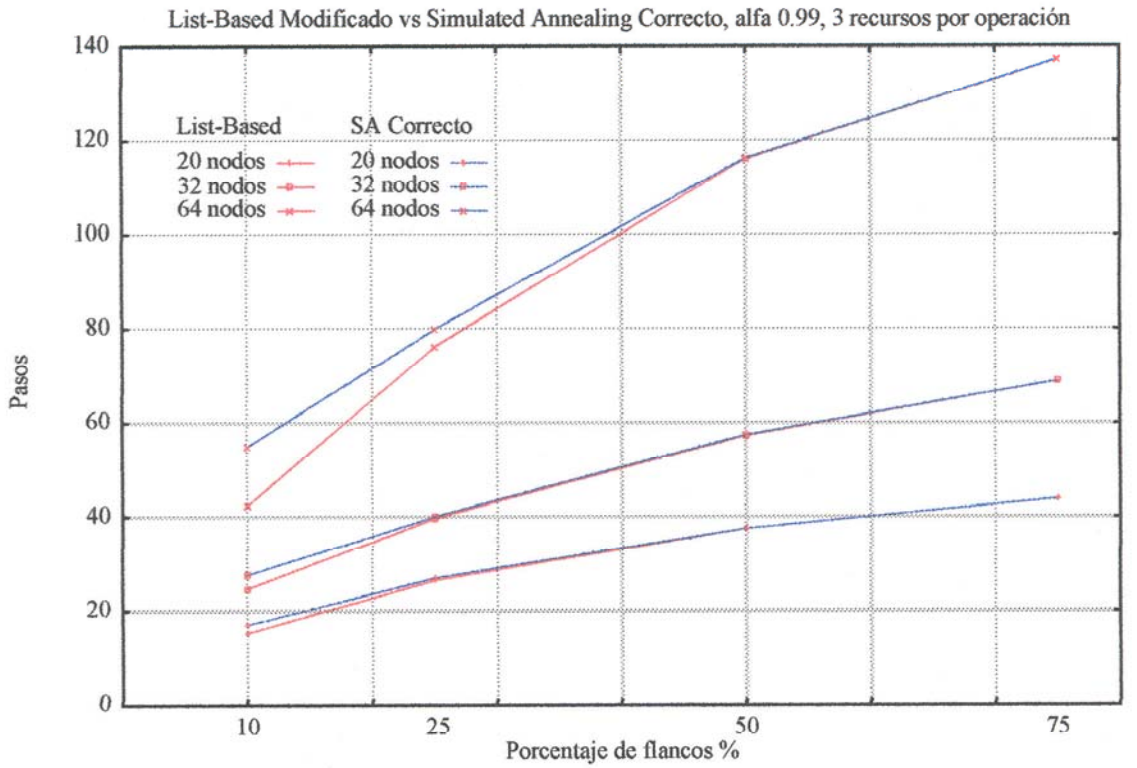
© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004





© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004





© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

Apéndice D

Especificación VSS: Ejemplo

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital. 2004

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

```
entity prub_voz is end;
architecture behav of prub_voz is

   type vector_real_one_dim is
      array(integer range<>) of real;
    type complejo is
       record
          rl:real;
          ig:real;
       end record;

   type vector_complejo_one_dim is
      array(integer range<>) of complejo;
    constant pi:real:=3.141592654;

   signal ceps_err:real;

   constant lowestFrequency:real:= 133.3333;
   constant linearSpacing:real:= 66.66666666;
   constant logSpacing:real:= 1.0711703;
   constant cepstralCoefficients:integer:= 13;
   constant totalFilters:integer:=40;

   type int is range -1000000 to 1000000;
   type int_exp is range -1000 to 1000;
   type intid is range -64000064 to 64000063;
   type int_expid is range -64064 to 64063;
   type int_vector is array(integer range<>) of integer;

   channel ch1_mant:int broadcast;
   channel ch1_expo:int_exp broadcast;
   channel ch2_mant:intid incast;
   channel ch2_expo:int_expid incast;
   clocks(test,1 ns);


begin

padre:process
   file entrada_file:text open read_mode is "trama.ddd";

   variable mant,expo,iden_man,iden_exp:integer;
   variable man,exp:int_vector(0 to 39);
   variable mantisa:intid;
   variable exponente:int_expid;
   variable r:real;
   variable entrada_test:vector_real_one_dim(0 to 127);
   variable l:line;
   variable acumulador,mem,auxpot:real;
   variable freqs:vector_real_one_dim(0 to 42);
   variable hamwindow:vector_real_one_dim(0 to 127);
   variable preemphasid:vector_real_one_dim(0 to 127);
   variable auxiliar:vector_complejo_one_dim(0 to 127);
   variable fftdata,fftmag:vector_real_one_dim(0 to 127);
   variable i,j:integer;

function "+"(a,b:complejo) return complejo is
   variable salida:complejo;
 begin
   salida.rl:=a.rl+b.rl; salida.ig:=a.ig+b.ig;
   return salida;
end;

function "-"(a,b:complejo) return complejo is
   variable salida:complejo;
begin
   salida.rl:=a.rl-b.rl; salida.ig:=a.ig-b.ig;
   return salida;
 end;

 function "*"(a,b:complejo) return complejo is
    variable salida:complejo;
 begin
    salida.rl:=a.rl*b.rl-a.ig*b.ig;
    salida.ig:=a.ig*b.rl+a.rl*b.ig;
    return salida;
end;

function "abs"(a:real) return real is
   variable salida:real;
begin
   salida:=a;
   if salida<0.0 then salida:=-salida;end if;
   return salida;
 end;

function sqr(a:real) return real is
   constant error:real:=1.0e-3;
   variable auxant,acu,aux,min,max:real;
   variable bucle:boolean;
begin
   min:=0.0;max:=a;aux:=(max-min)/2.0;
    auxant:=0.0;
    if a<0.0 then aux:=0.0;
    else
       bucle:=true;
       while bucle loop
          acu:=(a-aux*aux);
          if abs(acu)/a<error or (max-min)/max<error then
             bucle:=false;
          else
             if acu<0.0 then
                max:=aux;
             else
                min:=aux;
             end if;
             aux:=min+(max-min)/2.0;
          end if;
       end loop;
    end if;
    return aux;
end;
```

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

```
function "abs"(a:complejo) return real is
  variable salida:real;
begin
  salida:=sqr(a.rl*a.rl+a.ig*a.ig);
  return salida;
end;

function sin(alfa:in real) return real is
  variable auxalfa:real;
  variable signo:integer;
  variable s,c,x:vector_real_one_dim(0 to 4);
  variable x1,x2,x3,x4,salida:real;
  variable x0:integer;
begin
  c(0):=1.0;c(1):=0.866025403;c(2):=0.707106781;
  c(3):=0.5;c(4):=0.0;
  x(0):=0.0;x(1):=pi/6.0;x(2):=pi/4.0;x(3):=pi/3.0;
  x(4):=pi/2.0;
  for i in 0 to 4 loop
    s(i):=c(4-i);
  end loop;
  auxalfa:=alfa; signo:=1;
  if auxalfa<0.0 then
    signo:=-signo;
    auxalfa:=-auxalfa;
  end if;
  while auxalfa>2.0*pi loop auxalfa:=auxalfa-2.0*pi;
  end loop;
  if auxalfa>3.0*pi/2.0 then
    signo:=-signo;
    auxalfa:=2.0*pi-auxalfa;
  else
    if auxalfa>pi then
      signo:=-signo;
      auxalfa:=auxalfa-pi;
    else
      if auxalfa>pi/2.0 then auxalfa:=pi-auxalfa; end if;
    end if;
  end if;
  x0:=0;
  if auxalfa>5.0*pi/12.0 then x0:=4;
  else
    if auxalfa>7.0*pi/24.0 then x0:=3;
    else
      if auxalfa>5.0*pi/24.0 then x0:=2;
      else
        if auxalfa>pi/12.0 then x0:=1;end if;
      end if;
    end if;
  end if;
  x1:=auxalfa-x(x0);x2:=x1*x1;x3:=x2*x1;x4:=x3*x1;
  salida:=s(x0)+c(x0)*x1-s(x0)*x2/2.0
          -c(x0)*x3/6.0+s(x0)*x4/24.0;
  return salida*real(signo);
end;

function cos(alfa:in real) return real is
  begin
    return sin(alfa+pi/2.0);
end;

function inicializacion_exp_comp
  return vector_complejo_one_dim is
  variable arg:real;
  variable wrecur,waux:complejo;
  variable w:vector_complejo_one_dim(0 to 63);
begin
  arg:=(180.0/64.0)*2.0*pi/360.0;
  wrecur.rl:=cos(arg); wrecur.ig:=-sin(arg);
  waux:=wrecur;
  for j in 1 to 63 loop
    w(j-1):=wrecur;
    wrecur:=wrecur*waux;
  end loop;
  return w;
end;

procedure fft_128(x:inout
   vector_complejo_one_dim (0 to 127)) is
      variable w:vector_complejo_one_dim(0 to 63):=
      inicializacion_exp_comp;
  variable temp,tmp:complejo;
  variable wptr,windex,n,le,l,i,j,k:integer:=0;
begin
  le:=128; windex:=1;
  for l in 0 to 6 loop
    le:=le/2; i:=0;
    while i<128 loop
      temp:=x(i)+x(i+le); x(i+le):=x(i)-x(i+le);
      x(i):=temp; i:=i+2*le;
    end loop;
    wptr:=windex-1;
    for j in 1 to le-1 loop
      i:=j;
      while i<128 loop
        temp:=x(i)+x(i+le); mp:=x(i)-x(i+le);
        x(i+le):=tmp*w(wptr); x(i):=temp;i:=i+2*le;
      end loop;
      wptr:=wptr+windex;
    end loop;
    windex:=2*windex;
  end loop;
  j:=0;
  for i in 1 to 126 loop
    k:=64;
    while k<=j loop j:=j-k; k:=k/2; end loop;
    j:=j+k;
    if i<j then
      temp:=x(j); x(j):=x(i); x(i):=temp;
    end if;
  end loop;
end;
```

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

```
    -- Esta función extrae la mantisa ( -1e6 y 1e6)
    -- y el exponente del número real "valor"

  procedure real_to_mantisa_expo(valor:in real;
     mantisa,exponente:inout integer) is
     variable factor_escala,auxval,signo:real:=1.0;
  begin
     exponente:=0;
     if valor<0.0 then
       signo:=-signo;
       auxval:=-valor;
     else
       auxval:=valor;
     end if;
     if auxval=0.0 then
       mantisa:=0;
       exponente:=0;
     else
       if auxval<1000000.0 then
          while(auxval*factor_escala*10.0)<1.0e6 loop
             factor_escala:=factor_escala*10.0;
             exponente:=exponente-1;
          end loop;
          mantisa:=integer(signo*auxval*factor_escala);
       else
          while (auxval/factor_escala)>1000000.0 loop
             factor_escala:=factor_escala*10.0;
             exponente:=exponente+1;
          end loop;
          mantisa:=integer(signo*auxval/factor_escala);
       end if;
     end if;
  end;


begin

  fork1:par_fork true

    simple son00:process
      variable mant,expo:integer;
      variable mantisa:int;
      variable exponente:int_exp;
      variable frecbase,acumulador,auxpot:real;
      variable freqs:vector_real_one_dim(0 to 8);
      variable filterweight:
        vector_real_one_dim(0 to 7*128-1);
      variable triangleheight:
        vector_real_one_dim(0 to 6);
      variable fftfreqs:vector_real_one_dim(0 to 127);
      variable earmag:vector_real_one_dim(0 to 39);
      variable fftmag:vector_real_one_dim(0 to 127);

    -- procedure real_to_mantisa, idéntico al del proceso
    -- padre

begin
    --Cálculo de la respuesta en frecuencia
    -- de los filtros 0 a 6

 for i in 0 to 8 loop
    freqs(i):=lowestfrequency + real(i)*linearspacing;
 end loop;

 -- Se calcula la ganancia de cada uno de los filtros

 for k in 0 to 6 loop
    triangleheight(k):=2.0/(freqs(k+2)-freqs(k));
 end loop;

 for i in 0 to 127 loop
    fftfreqs(i):=real(i)*8000.0/128.0;
 end loop;

 -- Se construye la respuesta en frecuencia de los
 -- filtros

 for chan in 0 to 6 loop
    for j in 0 to 127 loop
      if fftfreqs(j)>freqs(chan) and
       fftfreqs(j)<=freqs(chan+1) then
         filterweight(chan*128+j):=
            triangleheight(chan)*
            (fftfreqs(j)-freqs(chan))/
            (freqs(chan+1)-freqs(chan) );
      else
        if fftfreqs(j)>freqs(chan+1) and
          fftfreqs(j)<freqs(chan+2) then
          filterweight(chan*128+j):=
            triangleheight( chan)*
            (freqs(chan+2)-fftfreqs(j))/
            (freqs(chan+2)-freqs( chan+1));
        else
          filterweight(chan*128+j):=0.0;
        end if;
      end if;
    end loop;
 end loop;

 -- Ahora se calcula la energía a la salida de los filtros

 for i in 0 to 127 loop
    receive(ch1_mant,mantisa;ch1_expo,exponente);
   fftmag(i):=real(integer(mantisa))*
          real(10.0** integer( exponente));
 end loop;
 for i in 0 to 6 loop
    acumulador:=0.0;
    for j in 0 to 127 loop
      acumulador:=acumulador+fftMag(j)*
          filterweight i*128+j);
    end loop;
```

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

```
      earmag(i):=acumulador;
      real_to_mantisa_expo(earmag(i),mant,expo);
      if mant>0 then
        mant:=(mant*64)+i;
      else
        mant:=(mant*64)-i;
      end if;
      if expo>0 then
        expo:=(expo*64)+i;
      else
        expo:=(expo*64)-i;
      end if;
      send(ch2_mant,mant;ch2_expo,expo);
    end loop;
  end process;

||

  simple son01:process
    -- variables y funciones idénticas a son00
  begin
    -- Cálculo de la respuesta en frecuencia de
    -- los filtros 7 a 13

    for i in 7 to 12 loop
      freqs(i-7):=lowestfrequency + real(i)*linearspacing;
    end loop;
    auxpot:=logspacing;
    for i in 0 to 2 loop
      freqs(i+6):=freqs(5)*auxpot;
      auxpot:=auxpot*logspacing;
    end loop;

    -- El cálculo de la ganancia de cada uno de los filtros
    -- en su frecuencia central es idéntico a son00

    -- Ahora se calcula la energía a la salida de los filtros

    for i in 0 to 6 loop
      acumulador:=0.0;
      for j in 0 to 127 loop
        acumulador:=acumulador+
          fftMag(j)*filterweight(i*128+j);
      end loop;
      earmag(i):=acumulador;
      real_to_mantisa_expo(earmag(i),mant,expo);
      if mant>0 then
        mant:=(mant*64)+i+7;
      else
        mant:=(mant*64)-i-7;
      end if;
      if expo>0 then
        expo:=(expo*64)+i+7;
      else
        expo:=(expo*64)-i-7;
      end if;
      send(ch2_mant,mant;ch2_expo,expo);
    end loop;
  end process;

||

  simple son02:process
    variable mant,expo:integer;
    variable mantisa:int;
    variable exponente:int_exp;
    variable frecbase,acumulador,auxpot:real;
    variable freqs:vector_real_one_dim(0 to 7);
    variable filterweight:
    vector_real_one_dim(0 to 6*128-1);
    variable triangleheight:vector_real_one_dim(0 to 5);
    variable fftfreqs:vector_real_one_dim(0 to 127);
    variable earmag:vector_real_one_dim(0 to 39);
    variable fftmag:vector_real_one_dim(0 to 127);

    --procedure real_to_mantisa_expo
    --Idéntico al proceso son00

  begin

    --Cálculo de la respuesta en frecuencia de los filtros
    -- 14 a 19

    auxpot:=logspacing*logspacing;
    freqs(7):=lowestfrequency+12.0*linearspacing;
    for i in 1 to 8 loop
      freqs(i-1):=freqs(7)*auxpot;
      auxpot:=auxpot*logspacing;
    end loop;

    -- Se calcula la ganancia de cada uno de los filtros

    for k in 0 to 5 loop
      triangleheight(k):=2.0/(freqs(k+2)-freqs(k));
    end loop;
    for i in 0 to 127 loop
      fftfreqs(i):=real(i)*8000.0/128.0;
    end loop;

    -- Se construye la respuesta en frecuencia de los filtros

    for chan in 0 to 5 loop
      for j in 0 to 127 loop
        if fftfreqs(j)>freqs(chan) and
            fftfreqs(j)<=freqs(chan+1) then
          filterweight(chan*128+j):=
          triangleheight( chan)*(fftfreqs(j)-freqs(chan))/
          (freqs(chan+1)-freqs(chan ));
        else
          if fftfreqs(j)>freqs(chan+1) and
              fftfreqs(j)< freqs(chan+2) then
            filterweight(chan*128+j):=
```

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

```
            triangleheight(chan)*(freqs(chan+2)
            -fftfreqs(j))/(freqs(chan+2)-freqs( chan+1));
          else
            filterweight(chan*128+j):=0.0;
          end if;
        end if;
      end loop;
    end loop;

    -- Ahora se calcula la energía a la salida de los filtros

    for i in 0 to 127 loop
      receive(ch1_mant,mantisa;ch1_expo,exponente);
      fftmag(i):=real(integer(mantisa))*
            real(10.0** integer(exponente));
    end loop;
    for i in 0 to 5 loop
      acumulador:=0.0;
      for j in 0 to 127 loop
        acumulador:=acumulador+fftMag(j)
              *filterweight( i*128+j);
      end loop;
      earmag(i):=acumulador;
      real_to_mantisa_expo(earmag(i),mant,expo);
      if mant>0 then
        mant:=(mant*64)+i+14;
      else
        mant:=(mant*64)-i-14;
      end if;
      if expo>0 then expo:=(expo*64)+i+14;
        else expo:=(expo*64)-i-14;
      end if;
      send(ch2_mant,mant;ch2_expo,expo);
    end loop;
  end process;

||

  simple son20:process

  -- variables y funciones idénticas a son00

  begin

    --Cálculo de la respuesta en frecuencia de los filtros

    auxpot:=logspacing**8;
    frecbase:=lowestfrequency + 12.0*linearspacing;
    for i in 7 to 15 loop
      freqs(i-7):=frecbase*auxpot;
      auxpot:=auxpot*logspacing;
    end loop;

    -- El cálculo de la ganancia de cada uno de los filtros
    -- es idéntico a son00

    -- Ahora se calcula la energía a la salida de los filtros

    for i in 0 to 127 loop
      receive(ch1_mant,mantisa;
      ch1_expo,exponente);
      fftmag(i):=real(mantisa)*10.0**integer(exponente);
    end loop;
    for i in 0 to 6 loop
      acumulador:=0.0;
        for j in 0 to 127 loop
          acumulador:=acumulador+fftMag(j)*
            filterweight(i*128+j);
        end loop;
      earmag(i):=acumulador;
      real_to_mantisa_expo(earmag(i),mant,expo);
      if mant>0 then
        mant:=mant*64+i+20;
      else
        mant:=mant*64-i-20;
      end if;
      if expo>0 then
        expo:=expo*64+i+20;
      else
        expo:=expo*64-i-20;
      end if;
      send(ch2_mant,mant;ch2_expo,expo);
    end loop;
  end process;

||

  simple son21:process

    -- variables y funciones idénticas a son00

  begin

    -- Cálculo de la respuesta en frecuencia de
    -- los filtros 27a 33

    auxpot:=logspacing**15;
    frecbase:=lowestfrequency + 12.0*linearspacing;
    for i in 14 to 22 loop
      freqs(i-14):=frecbase*auxpot;
      auxpot:=auxpot*logspacing;
    end loop;

    -- El cálculo de la ganancia de cada uno de los filtros
    -- es idéntico a son00

    for i in 0 to 127 loop
      receive(ch1_mant,mantisa;ch1_expo,exponente);
      fftmag(i):=real(mantisa)*10.0**integer(exponente );
    end loop;
```

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

```
  for i in 0 to 6 loop
    acumulador:=0.0;
    for j in 0 to 127 loop
      acumulador:=acumulador+fftMag(j)*
        filterweight i*128+j);
    end loop;
    earmag(i):=acumulador;
    real_to_mantisa_expo(earmag(i),mant,expo);
    if mant>0 then
      mant:=mant*64+i+27;
    else
      mant:=mant*64-i-27;
    end if;
    if expo>0 then
      expo:=expo*64+i+27;
    else
      expo:=expo*64-i-27;
    end if;
    send(ch2_mant,mant;ch2_expo,expo);
  end loop;

end process;

||

simple son22:process

  -- variables y funciones idénticas a son02

begin

  -- Cálculo de la respuesta en frecuencia de los
  -- filtros 34 a 39

  auxpot:=logspacing**22;
  frecbase:=lowestfrequency + 12.0*linearspacing;
  for i in 21 to 28 loop
    freqs(i-21):=frecbase*auxpot;
    auxpot:=auxpot*logspacing;
  end loop;

  -- El cálculo de la ganancia de cada uno de los filtros
  -- es idéntico a son02

  -- Ahora se calcula la energía a la salida de filtros

  for i in 0 to 127 loop
    receive(ch1_mant,mantisa;
    ch1_expo,exponente);
    fftmag(i):=real(mantisa)*10.0**integer(exponente);
  end loop;

  for i in 0 to 5 loop
    acumulador:=0.0;
    for j in 0 to 127 loop
      acumulador:=acumulador+fftMag(j)*
        filterweight(i*128+j);
    end loop;
    earmag(i):=acumulador;
    real_to_mantisa_expo(earmag(i),mant,expo);
    if mant>0 then
      mant:=mant*64+i+34;
    else
      mant:=mant*64-i-34;
    end if;
    if expo>0 then
      expo:=expo*64+i+34;
    else
      expo:=expo*64-i-34;
    end if;
    send(ch2_mant,mant;ch2_expo,expo);
  end loop;

end process;

||

simple son3:process
  file ceps_file:text open read_mode is "ceps.ddd";
  variable mant,expo,iden_man,iden_exp:integer;
  variable man,exp:int_vector(0 to 39);
  variable mantisa:intid;
  variable exponente:int_expid;
  variable r:real;
  variable ceps_test:vector_real_one_dim(0 to 12);
  variable l:line;
  variable acumulador,mem,auxpot:real;
  variable dctmatrix:vector_real_one_dim(0 to 13*40-1);
  variable ceps:vector_real_one_dim(0 to 12);
  variable earmag:vector_real_one_dim(0 to 39);
  variable i,j:integer;

  --procedure real_to_mantisa_expo
  -- idéntico al proceso son00

  --function sin, idéntica a la del proceso padre
  --function cos, idéntica a la del proceso padre

begin

  for i in 0 to 12 loop
    readline(ceps_file,l);
    read(l,ceps_test(i));
  end loop;
  for j in 0 to cepstralcoefficients-1 loop
    for i in 0 to totalfilters-1 loop
      dctmatrix(j*40+i):=cos(real(i)/real(totalfilters)*
          (real(j)+0.5)*pi);
    end loop;
  end loop;
```

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

```
   for i in 0 to 39 loop
     receive(ch2_mant,mantisa;ch2_expo,exponente);
     iden_man:=integer(mantisa) rem 64;
     iden_exp:=integer(exponente) rem 64;
     if iden_man<0 then
       iden_man:=-iden_man;
     end if;
     if iden_exp<0 then
       iden_exp:=-iden_exp;
     end if;
     man(iden_man):=integer(mantisa)/64;
     exp(iden_exp):=integer(exponente)/64;
   end loop;
   for i in 0 to 39 loop
      earMag(i):=real(man(i))*10.0**exp(i);
   end loop;

   -- Por último se calcula los coeficientes a partir
   -- de la transformada coseno de las energías
   -- obtenidas mediante los filtros

   for i in 0 to 12 loop
     acumulador:=0.0;
      for j in 0 to 39 loop
        acumulador:=acumulador+earMag(j)*
          dctmatrix( i*40+j);
     end loop;
     ceps(i):=acumulador;
     ceps_err<=(ceps(i)-ceps_test(i))/ceps(i);
     delay(test,1 ns);
   end loop;
end process;
end fork;

for i in 0 to 127 loop
   readline(entrada_file,l);
   read(l,entrada_test(i));
end loop;

for i in 0 to 127 loop
   hamwindow(i):=0.54 - 0.46*cos(2.0*pi*real(i)/127.0);
end loop;

--Escalado de la señal de entrada

   for i in 0 to 127 loop
      entrada_test(i):=entrada_test(i)*100000.0;
   end loop;

--Preenfasis de la señal de entrada

mem:=0.0;
for i in 0 to 127 loop
   preemphasid(i):=entrada_test(i)-0.97*mem;
    mem:=entrada_test(i);
end loop;

   -- Enventanado de la señal de entrada preenfatizada

   for i in 0 to 127 loop
     fftdata(i):=preemphasid(i)*hamwindow(i);
   end loop;

   --Desplazamos circularmente la entrada enventanada

   for i in 0 to 63 loop
     auxiliar(i).rl:=fftdata(i+64);
     auxiliar(i).ig:=0.0;
   end loop;
   for i in 64 to 127 loop
     auxiliar(i).rl:=fftdata(i-64);
     auxiliar(i).ig:=0.0;
   end loop;

   --Se calcula el modulo de la FFT de la entrada
   -- preenfatizada y enventanda.

   FFT_128(auxiliar);

   for i in 0 to 127 loop
     fftmag(i):=abs(auxiliar(i));
     real_to_mantisa_expo(fftmag(i),mant,expo);
     send(broadcast<=ch1_mant,mant;
     broadcast<=ch1_expo,expo);
   end loop;
   wait;
end process;
end;
```

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

Apéndice E

Especificación VHDL: Ejemplo

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital. 2004

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

```
library ieee;
library std;
library vss;
use std.textio.all;
use ieee.std_logic_1164.all;
use vss.paramemorias.all;

architecture behav of prub_voz is
   type vector_real_one_dim is array ( integer range <> ) of real ;
   type vector_real_two_dim is array ( integer range <> , integer range <> ) of real ;
   type complejo is record
       rl : real ;
       ig : real ;
   end record ;
   type vector_complejo_one_dim is array ( integer range <> ) of complejo ;
   constant pi : real := 3.141592654 ;
   signal ceps_err : real ;
   constant lowestfrequency : real := 133.3333 ;
   constant linearfilters : integer := 13 ;
   constant linearspacing : real := 66.66666666 ;
   constant logfilters : integer := 27 ;
   constant logspacing : real := 1.0711703 ;
   constant cepstralcoefficients : integer := 13 ;
   constant totalfilters : integer := 40 ;
   type int is range - 1000000 to 1000000 ;
   type int_exp is range - 1000 to 1000 ;
   type intid is range - 64000064 to 64000063 ;
   type int_expid is range - 64064 to 64063 ;
   type int_vector is array ( integer range <> ) of integer ;
   signal vss_je_son00_start,vss_je_son00_start_ack:std_logic:='z';
   signal vss_je_son01_start,vss_je_son01_start_ack:std_logic:='z';
   signal vss_je_son02_start,vss_je_son02_start_ack:std_logic:='z';
   signal vss_je_son20_start,vss_je_son20_start_ack:std_logic:='z';
   signal vss_je_son21_start,vss_je_son21_start_ack:std_logic:='z';
   signal vss_je_son22_start,vss_je_son22_start_ack:std_logic:='z';
   signal vss_je_son3_start,vss_je_son3_start_ack:std_logic:='z';
   constant vss_co_ch1_mant_id_z:unsigned(2 downto 0):=(others=>'z');
   constant vss_co_ch1_mant_ack_z:std_logic_vector(5 downto 0):=(others=>'z');
   constant vss_co_ch1_mant_ack_0:std_logic_vector(5 downto 0):=(others=>'0');
   constant vss_co_ch1_mant_son00_id:unsigned(2 downto 0):="001";
   constant vss_co_ch1_mant_son00_id_nat:natural:=0;
   constant vss_co_ch1_mant_son01_id:unsigned(2 downto 0):="010";
   constant vss_co_ch1_mant_son01_id_nat:natural:=1;
   constant vss_co_ch1_mant_son02_id:unsigned(2 downto 0):="011";
   constant vss_co_ch1_mant_son02_id_nat:natural:=2;
   constant vss_co_ch1_mant_son20_id:unsigned(2 downto 0):="100";
   constant vss_co_ch1_mant_son20_id_nat:natural:=3;
   constant vss_co_ch1_mant_son21_id:unsigned(2 downto 0):="101";
   constant vss_co_ch1_mant_son21_id_nat:natural:=4;
   constant vss_co_ch1_mant_son22_id:unsigned(2 downto 0):="110";
   constant vss_co_ch1_mant_son22_id_nat:natural:=5;
   constant vss_co_ch1_mant_broadcast:unsigned(2 downto 0):="000";
   signal vss_co_ch1_mant_dato:signed(20 downto 0):=(others=>'z');
   signal vss_co_ch1_mant_rdy:std_logic:='z';
   signal vss_co_ch1_mant_ack:std_logic_vector(5 downto 0):=(others=>'z');
   signal vss_co_ch1_mant_id:unsigned(2 downto 0):=(others=>'z');
```

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

```
constant vss_co_ch1_expo_id_z:unsigned(2 downto 0):=(others=>'z');
constant vss_co_ch1_expo_ack_z:std_logic_vector(5 downto 0):=(others=>'z');
constant vss_co_ch1_expo_ack_0:std_logic_vector(5 downto 0):=(others=>'0');
constant vss_co_ch1_expo_son00_id:unsigned(2 downto 0):="001";
constant vss_co_ch1_expo_son00_id_nat:natural:=0;
constant vss_co_ch1_expo_son01_id:unsigned(2 downto 0):="010";
constant vss_co_ch1_expo_son01_id_nat:natural:=1;
constant vss_co_ch1_expo_son02_id:unsigned(2 downto 0):="011";
constant vss_co_ch1_expo_son02_id_nat:natural:=2;
constant vss_co_ch1_expo_son20_id:unsigned(2 downto 0):="100";
constant vss_co_ch1_expo_son20_id_nat:natural:=3;
constant vss_co_ch1_expo_son21_id:unsigned(2 downto 0):="101";
constant vss_co_ch1_expo_son21_id_nat:natural:=4;
constant vss_co_ch1_expo_son22_id:unsigned(2 downto 0):="110";
constant vss_co_ch1_expo_son22_id_nat:natural:=5;
constant vss_co_ch1_expo_broadcast:unsigned(2 downto 0):="000";
signal vss_co_ch1_expo_dato:signed(10 downto 0):=(others=>'z');
signal vss_co_ch1_expo_rdy:std_logic:='z';
signal vss_co_ch1_expo_ack:std_logic_vector(5 downto 0):=(others=>'z');
signal vss_co_ch1_expo_id:unsigned(2 downto 0):=(others=>'z');
constant vss_co_ch2_mant_son00_id_nat:natural:=0;
constant vss_co_ch2_mant_son01_id_nat:natural:=1;
constant vss_co_ch2_mant_son02_id_nat:natural:=2;
constant vss_co_ch2_mant_son20_id_nat:natural:=3;
constant vss_co_ch2_mant_son21_id_nat:natural:=4;
constant vss_co_ch2_mant_son22_id_nat:natural:=5;
signal vss_co_ch2_mant_lock,vss_co_ch2_mant_locked:std_logic_vector(0 to 5):=(others=>'z');
signal vss_co_ch2_mant_dato:signed(26 downto 0):=(others=>'z');
signal vss_co_ch2_mant_ack,vss_co_ch2_mant_rdy:std_logic:='z';
constant vss_co_ch2_expo_son00_id_nat:natural:=0;
constant vss_co_ch2_expo_son01_id_nat:natural:=1;
constant vss_co_ch2_expo_son02_id_nat:natural:=2;
constant vss_co_ch2_expo_son20_id_nat:natural:=3;
constant vss_co_ch2_expo_son21_id_nat:natural:=4;
constant vss_co_ch2_expo_son22_id_nat:natural:=5;
signal vss_co_ch2_expo_lock,vss_co_ch2_expo_locked:std_logic_vector(0 to 5):=(others=>'z');
signal vss_co_ch2_expo_dato:signed(16 downto 0):=(others=>'z');
signal vss_co_ch2_expo_ack,vss_co_ch2_expo_rdy:std_logic:='z';
signal test:bit:='0';


component control_shared
    generic (transmisores:positive);
    port(lock:in std_logic_vector(0 to transmisores-1);
    locked:out std_logic_vector(0 to transmisores-1));
end component;


component generador_reloj
    generic (semiperiodo:real);
    port(clock:inout bit);
end component;
```

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

```
begin
padre:process
        file entrada_file : text open read_mode is "trama.ddd";
        variable mant , expo , iden_man , iden_exp : integer ;
        variable man , exp : int_vector ( 0 to 39 ) ;
        variable mantisa : intid ;
        variable exponente : int_expid ;
        variable r : real ;
        variable entrada_test : vector_real_one_dim ( 0 to 127 ) ;
        variable l : line ;
        variable acumulador , mem , auxpot : real ;
        variable freqs : vector_real_one_dim ( 0 to 42 ) ;
        variable hamwindow : vector_real_one_dim ( 0 to 127 ) ;
        variable preemphasid : vector_real_one_dim ( 0 to 127 ) ;
        variable auxiliar : vector_complejo_one_dim ( 0 to 127 ) ;
        variable fftdata , fftmag : vector_real_one_dim ( 0 to 127 ) ;
        variable i , j : integer ;

        function "+" ( a , b : complejo ) return complejo is
            variable salida : complejo ;
        begin
            salida . rl := a . rl + b . rl ;
            salida . ig := a . ig + b . ig ;
            return salida ;
        end ;

        function "-" ( a , b : complejo ) return complejo is
            variable salida : complejo ;
        begin
            salida . rl := a . rl - b . rl ;
            salida . ig := a . ig - b . ig ;
            return salida ;
        end ;

        function "*" ( a , b : complejo ) return complejo is
            variable salida : complejo ;
        begin
            salida . rl := a . rl * b . rl - a . ig * b . ig ;
            salida . ig := a . ig * b . rl + a . rl * b . ig ;
            return salida ;
        end ;

        function "abs" ( a : real ) return real is
            variable salida : real ;
        begin
            salida := a ;
            if salida < 0.0 then
                salida := - salida ;
            end if ;
            return salida ;
        end ;

        function sqr ( a : real ) return real is
            constant error : real := 1.0e-3 ;
            variable auxant , acu , aux , min , max : real ;
            variable bucle : boolean ;
```

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

E.6

```
begin
   min := 0.0 ;
   max := a ;
   aux := ( max - min ) / 2.0 ;
   auxant := 0.0 ;
   if a < 0.0 then
      aux := 0.0 ;
   else
      bucle := true ;
      while bucle loop acu := ( a - aux * aux ) ;
         if abs ( acu ) / a < error or ( max - min ) / max < error then
            bucle := false ;
         else
            if acu < 0.0 then
               max := aux ;
            else
               min := aux ;
            end if ;
            aux := min + ( max - min ) / 2.0 ;
         end if ;
      end loop ;
   end if ;
   return aux ;
end ;

function "abs" ( a : complejo ) return real is
   variable salida : real ;
begin
   salida := sqr ( a . rl * a . rl + a . ig * a . ig ) ;
   return salida ;
end ;

function sin ( alfa : in real ) return real is
   variable auxalfa : real ;
   variable signo : integer ;
   variable s , c , x : vector_real_one_dim ( 0 to 4 ) ;
   variable x1 , x2 , x3 , x4 , salida : real ;
   variable x0 : integer ;
begin
   c ( 0 ) := 1.0 ;
   c ( 1 ) := 0.866025403 ;
   c ( 2 ) := 0.707106781 ;
   c ( 3 ) := 0.5 ;
   c ( 4 ) := 0.0 ;
   x ( 0 ) := 0.0 ;
   x ( 1 ) := pi / 6.0 ;
   x ( 2 ) := pi / 4.0 ;
   x ( 3 ) := pi / 3.0 ;
   x ( 4 ) := pi / 2.0 ;
   for i in 0 to 4 loop s ( i ) := c ( 4 - i ) ; end loop ;
   auxalfa := alfa ;
   signo := 1 ;
   if auxalfa < 0.0 then
      signo := - signo ;
      auxalfa := - auxalfa ;
   end if ;
```

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

```
    while auxalfa > 2.0 * pi loop
        auxalfa := auxalfa - 2.0 * pi ;
    end loop ;
    if auxalfa > 3.0 * pi / 2.0 then
        signo := - signo ;
        auxalfa := 2.0 * pi - auxalfa ;
    else
        if auxalfa > pi then
            signo := - signo ;
            auxalfa := auxalfa - pi ;
        else
            if auxalfa > pi / 2.0 then
                auxalfa := pi - auxalfa ;
            end if ;
        end if ;
    end if ;
    x0 := 0 ;
    if auxalfa > 5.0 * pi / 12.0 then
        x0 := 4 ;
    else
        if auxalfa > 7.0 * pi / 24.0 then
            x0 := 3 ;
        else
            if auxalfa > 5.0 * pi / 24.0 then
                x0 := 2 ;
            else
                if auxalfa > pi / 12.0 then
                    x0 := 1 ;
                end if ;
            end if ;
        end if ;
    end if ;
    x1 := auxalfa - x ( x0 ) ;
    x2 := x1 * x1 ;
    x3 := x2 * x1 ;
    x4 := x3 * x1 ;
    salida := s ( x0 ) + c ( x0 ) * x1 - s ( x0 ) * x2 / 2.0 - c ( x0 ) * x3 / 6.0 + s ( x0 ) * x4 / 24.0 ;
    return salida * real ( signo ) ;
end ;


function cos ( alfa : in real ) return real is
begin
    return sin ( alfa + pi / 2.0 ) ;
end ;


function inicializacion_exp_comp return vector_complejo_one_dim is
    variable arg : real ;
    variable wrecur , waux : complejo ;
    variable w : vector_complejo_one_dim ( 0 to 63 ) ;
begin
    arg := ( 180.0 / 64.0 ) * 2.0 * pi / 360.0 ;
    wrecur . rl := cos ( arg ) ;
    wrecur . ig := - sin ( arg ) ;
    waux := wrecur ;
```

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

```
    for j in 1 to 63 loop
        w ( j - 1 ) := wrecur ;
        wrecur := wrecur * waux ;
    end loop ;
    return w ;
end ;


procedure fft_128 ( x : inout vector_complejo_one_dim ( 0 to 127 ) ) is
    variable w : vector_complejo_one_dim ( 0 to 63 ) := inicializacion_exp_comp ;
    variable temp , tmp : complejo ;
    variable wptr , windex , n , le , l , i , j , k : integer := 0 ;
begin
    le := 128 ;
    windex := 1 ;
    for l in 0 to 6 loop le := le / 2 ;
        i := 0 ;
        while i < 128 loop
            temp := x ( i ) + x ( i + le ) ;
            x ( i + le ) := x ( i ) - x ( i + le ) ;
            x ( i ) := temp ;
            i := i + 2 * le ;
        end loop ;
        wptr := windex - 1 ;
        for j in 1 to le - 1 loop
            i := j ;
            while i < 128 loop
                temp := x ( i ) + x ( i + le ) ;
                tmp := x ( i ) - x ( i + le ) ;
                x ( i + le ) := tmp * w ( wptr ) ;
                x ( i ) := temp ;
                i := i + 2 * le ;
            end loop ;
            wptr := wptr + windex ;
        end loop ;
        windex := 2 * windex ;
    end loop ;
    j := 0 ;
    for i in 1 to 126 loop
        k := 64 ;
        while k <= j loop
            j := j - k ;
            k := k / 2 ;
        end loop ;
        j := j + k ;
        if i < j then
            temp := x ( j ) ;
            x ( j ) := x ( i ) ;
            x ( i ) := temp ;
        end if ;
    end loop ;
end ;

procedure real_to_mantisa_expo ( valor : in real ;
    mantisa , exponente : inout integer ) is
    variable factor_escala , auxval , signo : real := 1.0 ;
```

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

```
        begin
            exponente := 0 ;
            if valor < 0.0 then
                signo := - signo ;
                auxval := - valor ;
            else
                auxval := valor ;
            end if ;
            if auxval = 0.0 then
                mantisa := 0 ;
                exponente := 0 ;
            else
                if auxval < 1000000.0 then
                    while ( auxval * factor_escala * 10.0 ) < 1.0e6 loop
                        factor_escala := factor_escala * 10.0 ;
                        exponente := exponente - 1 ;
                    end loop ;
                    mantisa := integer ( signo * auxval * factor_escala ) ;
                else
                    while ( auxval / factor_escala ) > 1000000.0 loop
                        factor_escala := factor_escala * 10.0 ;
                        exponente := exponente + 1 ;
                    end loop ;
                    mantisa := integer ( signo * auxval / factor_escala ) ;
                end if ;
            end if ;
        end ;
begin

    --****traducción del par_fork****

    vss_je_son00_start<='0';
    vss_je_son01_start<='0';
    vss_je_son02_start<='0';
    vss_je_son20_start<='0';
    vss_je_son21_start<='0';
    vss_je_son22_start<='0';
    vss_je_son3_start<='0';
    wait until vss_je_son00_start_ack='0' and vss_je_son01_start_ack='0' and vss_je_son02_start_ack='0'
                and vss_je_son20_start_ack='0' and vss_je_son21_start_ack='0'
                and vss_je_son22_start_ack='0' and vss_je_son3_start_ack='0' ;
    vss_je_son00_start <='z';
    vss_je_son01_start <='z';
    vss_je_son02_start <='z';
    vss_je_son20_start <='z';
    vss_je_son21_start <='z';
    vss_je_son22_start <='z';
    vss_je_son3_start <='z';
    wait for 1 ns;

    --******fin de traduccion del par_fork ******

    for i in 0 to 127 loop
        readline ( entrada_file , l ) ;
        read ( l , entrada_test ( i ) ) ;
    end loop ;
```

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

```
for i in 0 to 127 loop
    hamwindow ( i ) := 0.54 - 0.46 * cos ( 2.0 * pi * real ( i ) / 127.0 ) ;
end loop ;

 --escalado de la señal de entrada

for i in 0 to 127 loop
    entrada_test ( i ) := entrada_test ( i ) * 100000.0 ;
end loop ;

--preenfatizado de la señal de entrada

mem := 0.0 ;
for i in 0 to 127 loop
    preemphasid ( i ) := entrada_test ( i ) - 0.97 * mem ;
    mem := entrada_test ( i ) ;
end loop ;

-- enventanado de la señal de entrada preenfatizada

for i in 0 to 127 loop
    fftdata ( i ) := preemphasid ( i ) * hamwindow ( i ) ;
end loop ;

--desplazamos circularmente la entrada enventanada 64 muestras

for i in 0 to 63 loop
    auxiliar ( i ) . rl := fftdata ( i + 64 ) ;
    auxiliar ( i ) . ig := 0.0 ;
end loop ;
for i in 64 to 127 loop
    auxiliar ( i ) . rl := fftdata ( i - 64 ) ;
    auxiliar ( i ) . ig := 0.0 ;
end loop ;

--se calcula el módulo de la fft de la entrada preenfatizada y enventanda

fft_128 ( auxiliar ) ;
for i in 0 to 127 loop
    fftmag ( i ) := abs ( auxiliar ( i ) ) ;
    real_to_mantisa_expo ( fftmag ( i ) , mant , expo ) ;

    --begin_comu br,br send(broadcast <= ch1_mant , mant; broadcast <= ch1_expo , expo);

    vss_co_ch1_mant_dato<=to_signed(integer(mant),21);
    vss_co_ch1_mant_rdy<='0';
    vss_co_ch1_mant_id<=vss_co_ch1_mant_broadcast;
    vss_co_ch1_expo_dato<=to_signed(integer(expo),11);
    vss_co_ch1_expo_rdy<='0';
    vss_co_ch1_expo_id<=vss_co_ch1_expo_broadcast;
    wait for 1 ns;

    while  vss_co_ch1_mant_rdy='0' or vss_co_ch1_mant_ack /=vss_co_ch1_mant_ack_z
              or  vss_co_ch1_expo_rdy='0' or vss_co_ch1_expo_ack /=vss_co_ch1_expo_ack_z loop
```

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

```
            if vss_co_ch1_mant_ack=vss_co_ch1_mant_ack_0 then
                vss_co_ch1_mant_rdy<='z';
                vss_co_ch1_mant_id<=vss_co_ch1_mant_id_z;
                vss_co_ch1_mant_dato<="zzzzzzzzzzzzzzzzzzzzz";
            end if;
            if vss_co_ch1_expo_ack=vss_co_ch1_expo_ack_0 then
                vss_co_ch1_expo_rdy<='z';
                vss_co_ch1_expo_id<=vss_co_ch1_expo_id_z;
                vss_co_ch1_expo_dato<="zzzzzzzzzzz";
            end if;
            wait on vss_co_ch1_mant_ack,vss_co_ch1_mant_rdy,vss_co_ch1_expo_ack,vss_co_ch1_expo_rdy;
        end loop;

        --end_comu

    end loop ;
    wait ;
end process;


son00:process
    variable vss_co_ch1_mant_stop:boolean:=false;
    variable vss_co_ch1_expo_stop:boolean:=false;
    variable mant , expo : integer ;
    variable mantisa : int ;
    variable exponente : int_exp ;
    variable frecbase , acumulador , auxpot : real ;
    variable freqs : vector_real_one_dim ( 0 to 8 ) ;
    variable filterweight : vector_real_one_dim ( 0 to 7 * 128 - 1 ) ;
    variable triangleheight : vector_real_one_dim ( 0 to 6 ) ;
    variable fftfreqs : vector_real_one_dim ( 0 to 127 ) ;
    variable earmag : vector_real_one_dim ( 0 to 39 ) ;
    variable fftmag : vector_real_one_dim ( 0 to 127 ) ;

    procedure real_to_mantisa_expo ( valor : in real ; mantisa , exponente : inout integer ) is
        variable factor_escala , auxval , signo : real := 1.0 ;
    begin
        exponente := 0 ;
        if valor < 0.0 then
            signo := - signo ;
            auxval := - valor ;
        else
            auxval := valor ;
        end if ;
        if auxval = 0.0 then
            mantisa := 0 ;
            exponente := 0 ;
        else
            if auxval < 1000000.0 then
                while ( auxval * factor_escala * 10.0 ) < 1.0e6 loop
                    factor_escala := factor_escala * 10.0 ;
                    exponente := exponente - 1 ;
                end loop ;
                mantisa := integer ( signo * auxval * factor_escala ) ;
            else
```

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

```
            while ( auxval / factor_escala ) > 1000000.0 loop
                factor_escala := factor_escala * 10.0 ;
                exponente := exponente + 1 ;
            end loop ;
            mantisa := integer ( signo * auxval / factor_escala ) ;
        end if ;
    end if ;
end ;


begin
  if vss_je_son00_start='z' then
     wait until vss_je_son00_start='0';
  end if;
  vss_je_son00_start_ack<='0';
  wait until vss_je_son00_start='z';

  --*****inicio cuerpo principal****

  --cálculo de la respuesta en frecuencia de los filtros 0 a 6

  for i in 0 to 8 loop
     freqs ( i ) := lowestfrequency + real ( i ) * linearspacing ;
  end loop ;

   -- se calcula la ganancia de cada uno de los filtros en su frecuencia central

  for k in 0 to 6 loop
     triangleheight ( k ) := 2.0 / ( freqs ( k + 2 ) - freqs ( k ) ) ;
  end loop ;
  for i in 0 to 127 loop
     fftfreqs ( i ) := real ( i ) * 8000.0 / 128.0 ;
  end loop ;

  -- se construye la respuesta en frecuencia de los 7 filtros

  for chan in 0 to 6 loop
     for j in 0 to 127 loop
        if fftfreqs ( j ) > freqs ( chan ) and fftfreqs ( j ) <= freqs ( chan + 1 ) then
           filterweight (chan * 128 + j) := triangleheight (chan) * (fftfreqs (j) - freqs (chan)) /
                                            (freqs(chan + 1) - freqs ( chan ) ) ;
        else
           if fftfreqs ( j ) > freqs ( chan + 1 ) and fftfreqs ( j ) < freqs ( chan + 2 ) then
              filterweight ( chan * 128 + j ) := triangleheight ( chan ) * ( freqs ( chan + 2 ) -
                                                 fftfreqs ( j ) ) / ( freqs ( chan + 2 ) - freqs ( chan + 1 ) ) ;
           else
              filterweight ( chan * 128 + j ) := 0.0 ;
           end if ;
        end if ;
     end loop ;
  end loop ;

  -- ahora se calcula la energía a la salida de los filtros
```

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

```
for i in 0 to 127 loop

    --begin_comu br,br receive(ch1_mant , mantisa;ch1_expo , exponente);

    vss_co_ch1_mant_stop:=true;
    vss_co_ch1_expo_stop:=true;
    while  vss_co_ch1_mant_stop or vss_co_ch1_expo_stop loop
        if vss_co_ch1_mant_stop and vss_co_ch1_mant_rdy='0' and
                    (vss_co_ch1_mant_id=vss_co_ch1_mant_son00_id
                    or vss_co_ch1_mant_id=vss_co_ch1_mant_broadcast) then
            mantisa:=int(to_integer(vss_co_ch1_mant_dato));
            vss_co_ch1_mant_ack(vss_co_ch1_mant_son00_id_nat)<='0';
        else
            if vss_co_ch1_mant_stop and vss_co_ch1_mant_ack(vss_co_ch1_mant_son00_id_nat)='0' then
                vss_co_ch1_mant_ack(vss_co_ch1_mant_son00_id_nat)<='z';
                vss_co_ch1_mant_stop:=false;
            end if;
        end if;

        if vss_co_ch1_expo_stop and vss_co_ch1_expo_rdy='0' and
                        (vss_co_ch1_expo_id=vss_co_ch1_expo_son00_id
                        or vss_co_ch1_expo_id=vss_co_ch1_expo_broadcast) then
            exponente:=int_exp(to_integer(vss_co_ch1_expo_dato));
            vss_co_ch1_expo_ack(vss_co_ch1_expo_son00_id_nat)<='0';
        else
            if vss_co_ch1_expo_stop and vss_co_ch1_expo_ack(vss_co_ch1_expo_son00_id_nat)='0' then
                vss_co_ch1_expo_ack(vss_co_ch1_expo_son00_id_nat)<='z';
                vss_co_ch1_expo_stop:=false;
            end if;
        end if;
        wait on vss_co_ch1_mant_ack,vss_co_ch1_mant_rdy,vss_co_ch1_expo_ack,vss_co_ch1_expo_rdy;
    end loop;

    --end_comu

    fftmag ( i ) := real ( integer ( mantisa ) ) * real ( 10.0 ** integer ( exponente ) ) ;
end loop ;

for i in 0 to 6 loop
    acumulador := 0.0 ;
    for j in 0 to 127 loop
        acumulador := acumulador + fftmag ( j ) * filterweight ( i * 128 + j ) ;
    end loop ;
    earmag ( i ) := acumulador ;
    real_to_mantisa_expo ( earmag ( i ) , mant , expo ) ;
    if mant > 0 then
        mant := ( mant * 64 ) + i ;
    else
        mant := ( mant * 64 ) - i ;
    end if ;
    if expo > 0 then
        expo := ( expo * 64 ) + i ;
    else
        expo := ( expo * 64 ) - i ;
    end if ;
```

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004

E.14

```
      --begin_comu in,in send(ch2_mant , mant;ch2_expo , expo);

      vss_co_ch2_mant_lock(vss_co_ch2_mant_son00_id_nat)<='0';
      vss_co_ch2_expo_lock(vss_co_ch2_expo_son00_id_nat)<='0';
      wait for 0 ns;
      while vss_co_ch2_mant_lock(vss_co_ch2_mant_son00_id_nat)='0' or
                  vss_co_ch2_mant_locked(vss_co_ch2_mant_son00_id_nat)='0' or
                  vss_co_ch2_mant_ack='0' or vss_co_ch2_expo_lock(vss_co_ch2_expo_son00_id_nat)='0' or
                  vss_co_ch2_expo_locked(vss_co_ch2_expo_son00_id_nat)='0' or vss_co_ch2_expo_ack='0'
      loop
          if vss_co_ch2_mant_locked(vss_co_ch2_mant_son00_id_nat)='0'
                        and vss_co_ch2_mant_rdy='z' and vss_co_ch2_mant_ack='z' then
              vss_co_ch2_mant_dato<=to_signed(integer(mant),27);
              vss_co_ch2_mant_rdy<='0';
          elsif
              vss_co_ch2_mant_locked(vss_co_ch2_mant_son00_id_nat)='0'
              and vss_co_ch2_mant_rdy='0' and vss_co_ch2_mant_ack='0' then
                  vss_co_ch2_mant_rdy<='z' ;
                  vss_co_ch2_mant_dato<="zzzzzzzzzzzzzzzzzzzzzzzzzz";
                  vss_co_ch2_mant_lock(vss_co_ch2_mant_son00_id_nat)<='z';
          end if;

          if vss_co_ch2_expo_locked(vss_co_ch2_expo_son00_id_nat)='0' and
                        vss_co_ch2_expo_rdy='z' and vss_co_ch2_expo_ack='z' then
              vss_co_ch2_expo_dato<=to_signed(integer(expo),17);
              vss_co_ch2_expo_rdy<='0';
          elsif
              vss_co_ch2_expo_locked(vss_co_ch2_expo_son00_id_nat)='0' and
                      vss_co_ch2_expo_rdy='0' and vss_co_ch2_expo_ack='0' then
              vss_co_ch2_expo_rdy<='z' ;
              vss_co_ch2_expo_dato<="zzzzzzzzzzzzzzzzz";
              vss_co_ch2_expo_lock(vss_co_ch2_expo_son00_id_nat)<='z';
          end if;

          wait on vss_co_ch2_mant_ack, vss_co_ch2_mant_rdy, vss_co_ch2_mant_lock,
                  vss_co_ch2_mant_locked, vss_co_ch2_expo_ack, vss_co_ch2_expo_rdy,
                  vss_co_ch2_expo_lock, vss_co_ch2_expo_locked;
      end loop;

      --end_comu

   end loop ;

--******fin cuerpo principal *******

   vss_je_son00_start_ack<='z';
   wait for 0 ns;
end process;

-- RESTO DE HIJOS (no se muestra el código por motivos de espacio)
```

© Universidad de Las Palmas de Gran Canaria. Biblioteca Digital, 2004