Evaluation of the Field-Programmable Cache: Performance and Energy Consumption *

Domingo Benitez
IUSIANI & DIS Department. University of Las Palmas G.C., 35017 Las Palmas, Spain.
dbenitez@dis.ulpgc.es

Juan C. Moure , Dolores I. Rexachs, Emilio Luque
Computer Architecture and Operating Systems Department
Universidad Autonoma de Barcelona, 08193 Barcelona, Spain.
{JuanCarlos.Moure, Dolores.Rexachs, Emilio.Luque}@uab.es

ABSTRACT

Many authors have proposed power management techniques for general-purpose processors at the cost of degraded performance such as lower IPC or longer delay. Some proposals have focused on cache memories because they consume a significant fraction of total microprocessor power. We propose a reconfigurable and adaptive cache microarchitecture based on field-programmable technology that is intended to deliver high performance at low energy consumption. In this paper, we evaluate the performance and energy consumption of a run-time algorithm when used to manage a field-programmable L1 data cache. The adaptation strategy is based on two techniques: a learning process provides the best cache configuration for each program phase, and a recognition process detects program phase changes by using data working-set signatures to activate a low-overhead reconfiguration mechanism. Our proposals achieve performance improvement and cache energy saving at the same time. Considering a design scenario driven by performance constraints, we show that processor execution time and cache energy consumption can be reduced on average by 15.2% and 9.9% compared to a non-adaptive high-performance microarchitecture. Alternatively, when energy saving is prioritized and considering a non-adaptive energy-efficient microarchitecture as baseline, cache energy and processor execution time are reduced on average by 46.7% and 9.4% respectively. In addition to comparing to conventional microarchitectures, we show that the proposed microarchitecture achieves better performance and more cache energy reduction than other configurable caches.

*This work was supported by the MCyT-Spain under contract TIN 2004-03388, the Gobierno de Canarias, the Generalitat de Catalunya, and the HiPEAC European Network.

CF'06 May 3–5, 2006, Ischia, Italy.

Categories and Subject Descriptors: B.3.2 Hardware: Memory Structures - Design Styles, Cache memories

General Terms

Design, Performance

Keywords: Reconfigurable cache memory, Run-time adaptation, Performance evaluation, Static and dynamic energy consumption, Adaptive processors.

1. INTRODUCTION

Programs and their execution phases exhibit different efficiencies when fixed processor hardware is adopted [21]. Adaptive processors can exploit this phenomenon to provide higher efficiency than fixed hardware systems [22, 24]. These processors activate the reconfiguration of its microarchitecture under architectural criteria: highest performance, lowest energy consumption, etc.

Hardware reconfigurability is a more general concept related to physical resources needed to modify the hardware organization after chip fabrication. Some reconfigurable cache memories have been proposed to improve the performance or power dissipation of general-purpose processors [2, 13, 14, 18]. However, improvements are limited by the level of reconfigurability, which has been forced to not degrade the operating frequency of processor. We have observed that if the clock speed of the reconfigurable system is allowed to be slightly slower, higher performance and reduced power dissipation can be achieved at the same time compared with non-reconfigurable hardware systems.

In this paper, we propose the microarchitecture of the *Field-Programmable Cache Array* (FPCA) and evaluate its impact on processor performance and cache energy consumption. FPCA is a specialized reconfigurable circuit for the cache memory of a general-purpose processor. In comparison with a conventional cache, the access time is only slightly longer. Furthermore, the temporal and power overheads required to control the reconfigurable microarchitecture are low because programs exhibit large execution phases and reconfigurations are only sporadically activated. The additional hardware is also small because the circuit is specialized in cache memory.

Our field-programmable data cache is similar to others reconfigurable caches in that they try to obtain the highest performance or the lowest energy consumption by selecting the best configuration at critical runtime points [2, 14, 18]. However, their range of configurations is fixed at design-time and is not scalable to different chip area budgets: from server processors to low-cost processors. In these cases, we have observed that it can be possible that a unique cache configuration is selected for the majority of applications. Furthermore, the efficiency of caches is not fully exploited because the same set of tuneable cache configurations is used for different goals: performance improvement, energy saving, etc.

The FPCA cache can be used in designs with different initial budgets for the processor development, each of them can adopt a distinct FPCA. Additionally, since the number of tuneable cache configurations can be large, they can be efficiently exploited not only by the range of applications and execution phases, but also by tuning a preferred metric (performance, power dissipation, etc.).

Conventional control algorithms for adaptive caches spend long times in the tuning process because they explore all the configurations for picking the best one [2, 6]. In these cases, the temporal and energy overheads are higher as the number of tuneable cache configurations increases. Then, the large number of cache configurations that can be implemented with a FPCA may prevent achieving improvements. We also propose an on-line control algorithm called *Cache Matching Algorithm* for instantaneously tuning the FPCA cache with accuracy, minimal hardware cost, and an overhead that is independent of the number of tuneable cache configurations.

The rest of the paper is organized as follows. Section 2 presents the internal organization of the FPCA cache, and a model that is used in our architectural evaluations. Section 3 describes the control methodology for run-time adaptation. In Section 4, we describe the simulation methodology employed to evaluate our proposals, the benchmark applications used, and additionally, a performance and cache energy analysis (static and dynamic) of several baseline configurations including three systems: conventional caches, the adaptive cache proposed by other authors, and an ideal adaptive mechanism with future knowledge. Section 5 evaluates our control methodology for high-performance processors with a FPCA L1 data cache and compares it with several baseline architectures. Sections 6 and 7 contain additional discussion of related work and concluding remarks.

2. A FIELD-PROGRAMMABLE CACHE

In this section, we describe a specialized reconfigurable circuit for cache memories called *Field-Programmable Cache Array* (FPCA), which is based on Field-Programmable Gate Array technology. The FPCA circuit can be integrated into conventional processors, from high-performance to low-cost processors. The FPCA circuit is organized into an array of reconfigurable cells called *Configurable Cache Blocks* (CCB), which are selectively connected by a power-on configuration bit (called *Vcc*). Fig. 1 shows a block diagram of the FPCA where four CCBs can be identified.

A CCB is based on the classical organization of CMOS memories, and its design was guided by results of the architectural study of the cache adaptation shown in Section 5. Each CCB consists of eight complete cache memories called *T-D*, each of them consists of 128 sets with 20 bits for tags and 8 bytes for data (*T* and *D* respectively in Fig. 1).

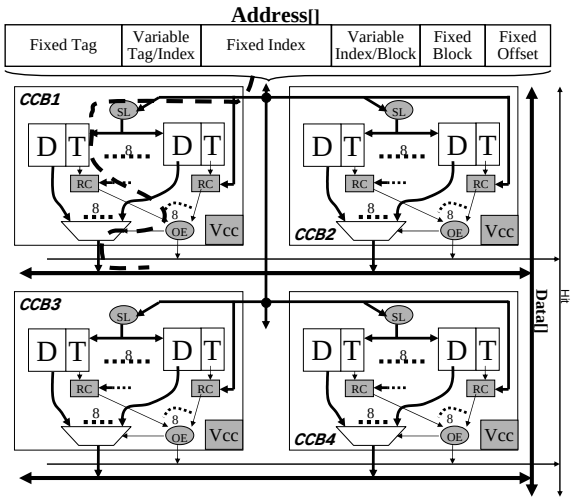

Figure 1: The Field-Programmable Cache Array

Some tag bits can be selectively activated, depending on the cache configuration. The overall capacity of each CCB is 8 Kbytes, and the reconfigurability of CCBs allows up to three degrees of set-associativity: 2-way, 4-way and 8-way, and up to four line sizes: 8, 16, 32, 64 bytes. A FPCA includes 10 additional configuration bits which are shared by all CCBs. When a different cache organization is required, the FPCA can implement it by changing the configuration bits.

Reconfigurability requires additional hardware resources: *SL*, *OE*, and *RC*. *SL* represents the selection logic that selects the index bits which are the same for all CCBs. This module uses four configuration bits to select the appropriate bits from the variable sections of the address called *Variable Tag/Index* and *Variable Index/Block* (see Fig. 1). *OE* represents the hardware module that selects the output data, which uses other four configuration bits and the block bits of the address (*Fixed Block* and *Variable Index/Block* in Fig. 1). *RC* represents the reconfigurable comparator where the information read from the tag array T is compared to the tag bits of the address (*Fixed Tag* and *Variable Tag/Index* in Fig. 1). This circuit uses two configuration bits.

The range of configurations that can be implemented with FPCA has the following constrains: the number of sets has to be higher than or equal to 128, 8-way set-associativity is the highest allowed associativity, the cache lines can store data from 8 to 64 bytes, and the biggest size depends on the number of CCBs, which is related with the chip area devoted to FPCA. Therefore, FPCA allows different cache configurations to be implemented, in which capacity, associativity and line size can be varied independently. The hit latency (in cycles) can also be variable and depends on the operating frequency. We have observed that better performance can be achieved when the operating frequency is reduced slightly in order to maintain a minimal hit latency in cycles of FPCA accesses. This is the reason why we do not fix the operating frequency, which is supposed to be limited by the frequency of the non-reconfigurable baseline system.

2.1 Architectural Model

An architectural model of the FPCA circuit is required for detailed cycle-by-cycle simulations of complete processors. We modified the analytical model used by CACTI tool [17],

which predicts the access time, power dissipation and chip area of conventional CMOS cache memories. Our modification was guided by PSPICE simulations of the FPCA circuit and provided the following average results.

Access Time. The conventional paths that are affected by the specialized reconfigurable architecture are the address decoder, comparator, multiplexer driver, and wire lines of the input address and output data (see dotted line in Fig. 1). The access time and cycle time of a FPCA cache configuration on average was 20% longer than provided by original CACTI for the same CMOS technology.

Power. Since a FPCA has more transistors than the equivalent non-reconfigurable cache memory, the power dissipation of a memory access is on average 5% higher.

Reconfiguration Time. This temporal overhead is proportional to the number of configuration bits. For the FPCA shown in Fig. 1, its four CCBs have 14 configuration bits (10 shared + 4 Vcc), which are loaded when a change of cache organization is activated. Supposing that these bits are serially loaded with a 100 MHz configuration clock signal, the reconfiguration time of FPCA is 0.14 μs. During this reconfiguration phase, we suppose that FPCA can not be accessed, and the previous content of the cache is discarded. The data cache uses write-back policy, and before its content is flushed out, L2 cache is updated. This additional overhead was accounted for in our architectural model.

Chip Area. The FPCA was 10% larger in chip area than the equivalent non-reconfigurable memory cache.

3. RUN-TIME ADAPTATION

In this section, we show how *Basic Block Vectors* (BBV) obtained from dynamic program traces can be used to determine changes in some characteristics of the data working-set accessed by an instruction interval. Then, we explain how a hardware algorithm called *Cache Matching Algorithm* collects BBV vectors during program run-time for instantaneously tuning an adaptive data cache. This algorithm is applied to the on-line control of an FPCA cache.

3.1 Predictor of the Data Working-Set

The size of a *Basic Block* (BB) is determined from the instruction count between branches. A *Basic Block Vector* (BBV) is gathered for each instruction interval during the program execution. Each component of a BBV vector collects the frequency of basic blocks with a determined size. BBV-based techniques for detecting recurring program phases have been shown to provide better sensitivity and lower performance variation in phases compared to other techniques [7]. These techniques have been used for accelerating architectural cycle-by-cycle simulations [20].

We propose using BBV vectors to feature the data working-set of an instruction interval and configure a FPCA cache. Working-set representations have the advantage that they can be used to estimate the working set size directly. They are useful in cases where performance of a functional unit is directly related to the working-set characteristics. Dhodapkar and Smith proposed another representation of instruction working-sets called *Working-Set Signature* [6], which has been shown to contain less information about the instruction interval than the BBV vector [7].

The usefulness of using BBV vectors as predictors of the properties of data working-set is shown with the following synthetic code, which consists of two nested loops.

```
//Nested loop #1, BB1= 12 instructions
   for (j=0;j<10;j++)
//BB2= 8 instructions
      for (i=0;i<1000;i++) x[i]=x[i]+x[i+1];
//Nested loop #2, BB3= 16 instructions
   for (j=0;j<10;j++)
//BB4= 11 instructions
      for (i=0;i<1000;i++) y[i]=x[i]+y[i]+y[i+1];
```

The first one uses one array (`int x[]`), and the second one additionally uses a second array (`int y[]`) with the same size. The sizes of dynamic basic blocks (BB_i) are shown in the comment lines of the example code above. Each basic block is determined by a branch instruction required to implement the loops. Now, suppose instruction intervals of 10,000 instructions. The first basic block executed is BB_1, which is executed in each iteration of the first loop `j`. After each execution of BB_1, basic block BB_2 is executed in 999 iterations. Taking intervals of 10,000 instructions, the most frequent basic block vector obtained from the first nested loop is the following,

$$BBV_1 = (0, \cdot\cdot, 1249BB_2 \times 8i/BB_2, 0, \cdot\cdot, 0, 1BB_1 \times 12i/BB_1, 0, \cdot\cdot, 0)$$

The data working-set, i.e. the amount of data accessed during each instruction interval in which we obtain BBV_1 is 4,004 Bytes. After that, the most frequent BBV obtained from the second nested loop is the following,

$$BBV_2 = (0, \cdot\cdot, 908BB_4 \times 11i/BB_4, 0, \cdot\cdot, 0, 1BB_3 \times 16i/BB_3, 0, \cdot\cdot, 0)$$

The number of different words accessed during the execution of the second nested loop is 7,276 Bytes. With a 4 KB data cache, the first nested loop would only exhibit cold misses. However, the second nested loop would exhibit cold and capacity misses. It is possible to use an 8 KB data cache for all the program and the capacity misses disappear. However, half of the cache is inefficiently activated during the first nested loop, i.e., superfluous energy is consumed and the access time is larger than required because the tag and data matrices can store more bits than accessed in the first nested loop.

BBV_1 is only collected from the first nested loop, and BBV_2 is only collected from the second nested loop. The detection of one of these BBVs can predict the volume of accessed data. So, a cache capacity accommodated to the data working-set can be activated when a BBV vector is recognized during program execution. Vectors very similar to BBV_1 or BBV_2 appear many times during run-time. Both groups of BBV vectors are clearly different from each other and represent two patterns or program phases. Traditional pattern-matching techniques can cluster these BBV vectors and partition the representation space [8]. We propose to match each partition with a distinct cache configuration by using the following algorithm.

3.2 Cache Matching Algorithm

The *Cache Matching Algorithm* (CMA) is proposed to dynamically detect changes of program phases and execute each phase with the maximum efficiency by reconfiguring an adaptive FPCA data cache. This hardware/software algorithm uses BBV vectors and is inspired in other approaches

used by human-like systems [8]. Three stages are required: *Learning*, *Recognition*, and *Actuation*.

3.2.1 Learning

The *Learning* stage is used to identify patterns/phases of program behaviour. Additionally, it associates each pattern with a configuration of the reconfigurable FPCA cache, which provides the highest performance and is selected from a set of many different configurations. This task is performed by software and is divided into three major steps.

In the *first learning step*, a BBV vector collects for each instruction interval the frequency and size of all executed basic blocks. Every BBV vector may have thousands of components. Since BBV vectors are processed by the runtime control system, they determine the major cost of the additional hardware. Then, it is necessary to find the lowest dimensional effective subspace for classification purposes. A feature extraction method called *Decision Boundary Feature Extraction* (DBFE) was used to calculate the optimal transformation to a lower dimensional space [15]. It uses training samples to determine discriminative informative features. Each feature is defined by an eigenfuncion and has associate an eigenvalue to characterize the usefulness of the corresponding feature. The DBFE was applied to BBV vectors collected from all SPEC benchmarks considering 5,000 intervals of 100,000 instructions/interval in each program. In all cases, the sum of eigenvalues is higher than 95% of the maximum value by using only three new features.

We propose to transform the original hyperspectral BBV vectors by accumulating the frequency of basic blocks whose sizes range in three non-superimposed intervals. These intervals are defined in the learning stage from the eigenfunctions. This step needs to be done only once for each program. For each instruction interval, three *BB Sensors* (counters) in the processor core hold the components of a three-dimensional BBV vector (called *3-D BBV*). Vectors that are close together in that space represent instruction intervals with similar behaviour, i.e. a program pattern/phase.

In the *second learning step*, the K-means clustering algorithm (used in the SimPoint Toolkit for another purpose [20]) runs iteratively on 3-D BBV vectors collected from the execution of a relatively large number of 100,000 instruction intervals. The number of iterations (K) is fixed previously, with values of K from 1 to M. Each run of K-means produces a clustering, which is a partition of the representation space into K clusters. Using a probabilistic measure of the goodness of fit of a clustering within a dataset, the smallest K is chosen. So, the 3-D BBV vectors are grouped into a set of K clusters called *SimPoint (SP) Classes*, where each SP class represents a different program pattern/phase. Finally, the 3-D representation space is partitioned into hypercubes; each of them encloses the 3-D BBV vectors assigned to an SP class. Fig. 2 shows an example of a bidimensional projection of the 3-D representation in which clusters of BBVs divide the representation space into SP classes.

In the *last learning stage*, associating each SP class with an optimal cache configuration involves executing several instructions intervals for every tuneable configuration, and monitoring the respective SP classes and execution times. After picking all cache configurations, each SP class is assigned to the cache configuration which exhibits the highest performance in most of the instruction intervals assigned to that SP class. This task and the periodic activation of

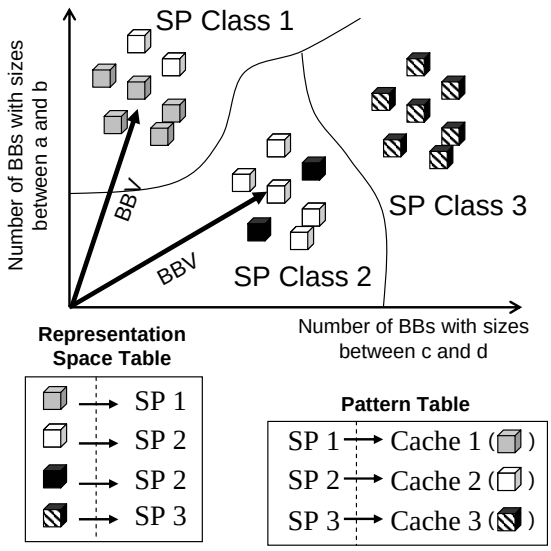

Figure 2: The Basic Block Vectors (BBV) are clustered and the Representation Space is divided into hypercubes that enclose the clusters. Each hypercube represents a SimPoint (SP) Class. The Representation Space Table associates a SP class with each BBV vector. The Pattern Table associates a SP class with the best cache configuration.

the cache reconfiguration are also implemented by software. Since we are assuming that the best cache configuration is the same for all the intervals belonging to the same SP class, it is not necessary to execute the full program for every available cache configuration. Since an SP class should be composed of many intervals, the learning time is negligible in comparison with the total execution time.

A *Representation Space Table* is used to store the SP class assigned to each 3-D BBV. Another *Pattern Table* contains the association between an SP class and the cache configuration with the highest performance (see Fig. 2). Both tables are implemented in hardware for a fast look-up, and are set-up by software after the last learning step has finished. Next, the reconfigurable cache can operate in one of two modes of operation called *Recognition* and *Actuation*.

3.2.2 Recognition and Actuation

The *Recognition* stage implements the cache tuning and is active on-line for the whole program execution. It detects if the current cache configuration does not provide the highest performance for the running program pattern/phase, and determines what different cache configuration should be used instead. A hardware coprocessor performs the recognition task by firstly reading the 3-D BBV vector from the BB sensors after each execution interval. Next, the vector position in the Representation Space Table allows the SP class of the interval to be recognized. If the currently activated cache configuration does not coincide with the configuration associated with this SP class, which is stored in the Pattern Table, it means that the instruction interval was not efficiently executed. The Recognition stage can be executed in parallel with the instruction flow and does not modify the critical execution path. So, this tuning stage provides no performance degradation.

The *Actuation* stage is activated by the coprocessor when the SP classes of three consecutive instruction intervals are assigned in the Pattern Table to the same cache configuration and this configuration is different from the current one. When an actuation is fired, the instruction flow is stalled and a *Configuration Table* is read to obtain the bitstream required for the reconfiguration process, including the operating frequency. After reconfiguring the hardware, the cache content is lost and the instruction flow and recognition process are restarted. Before running a different program, the Representation Space Table, Pattern Table and Configuration Table are loaded with the information derived from the respective learning stage.

In summary, the association of a cache configuration with a program phase/pattern is learned once, stored, and used each time the program phase/pattern is recognized. Therefore, the overhead at runtime is independent of the number of available configurations.

3.3 Cache Adaptation Driven By Other Preferred Metrics

The criterion used by the CMA algorithm in assigning the best cache configuration to each program phase was based on performance. Other preferred metrics such as energy dissipation, and time-energy product can be used instead. The CMA algorithm is extended to allow prioritizing other architectural metrics. For each SP class, by monitoring a preferred metric in the Learning stage, the cache configuration that achieves a minimal value can be determined. Then, the representation space table of a program can be associated with several pattern tables, which pursue different architectural goals. By loading one of the contents of the pattern table for the running program, the recognition stage can be also tuned to optimize a preferred metric. In this way, the same hardware can be architecturally tuned to achieve either the highest performance improvement, the highest energy saving, or the lowest time-energy product.

3.4 Hardware Controller

Supposing intervals of 100,000 instructions, three 17-bit counters measure the number of executed instructions in BBs with three different ranges of sizes. The three more significant bits from each counter builds one of the components of the 3-D BBV vectors. Six 12-bit registers load the upper and lower limits of the three ranges of BB sizes before running the program. Another 17-bit counter registers the clock cycles for each instruction interval, and a pair of 16-bit counters measures hits and misses in the L1 data cache.

The hardware coprocessor required for the Recognition and Actuation stages contains three small tables (see Fig. 3). The Representation Space Table provides for each 3-D BBV the respective SimPoint Class. Since each BBV component has three bits and 16 is an appropriate number of SP classes, its size is $2^9 \times 4$ bits. The Pattern Table contains the association between SP Class and cache configuration (identified by a Cache ID). Supposing that the maximum number of different configurations is 16, its size is $2^4 \times 4$ bits. The Configuration Table stores for each Cache ID the configuration bits needed for reconfiguring the FPCA data cache (14 bits), including the operating frequency and the hit and miss latencies. Its size is 50 bytes.

A FIFO memory is used to store the Cache ID of the last three instruction intervals, and a single register stores the current FPCA configuration ID. Another hardware module provides the activation signal of the Actuation Stage. Additionally, a small circuit is needed to read the Configuration Table and configure the FPCA in the Actuation Stage. Supposing eight instructions/cycle, 100,000 instructions are executed in 12,500 clock cycles. It is enough for reading the Representation Space and Pattern tables, and for the glue logic to activate the Actuation stage.

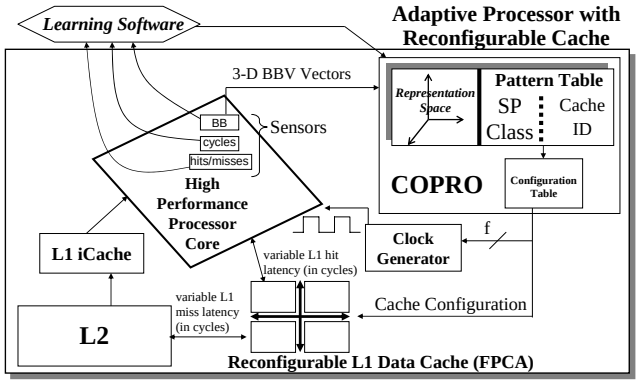

Figure 3: Microarchitecture of the adaptive processor with Field Programmable Data Cache and hardware support for the Cache Matching Algorithm.

Table 1: Selected SPEC benchmarks.

Program	Input	SPEC	Program	Input	SPEC
ammp	ref	fp00	applu	ref	fp00
apsi	ref	fp00	art	ref	fp00
bzip2	graphic	int00	crafty	ref	int00
eon	cook	int00	equake	ref	fp00
facerec	ref	fp00	fma3d	ref	fp00
galgel	ref	fp00	wupwise	ref	fp00
gcc	200, expr	int00	gzip	random, source	int00
ijpeg	ref	int95	lucas	ref	fp00
mcf	ref	int00	mesa	ref	fp00
mgrid	ref	fp00	parser	ref	int00
perlmbk	ref	int00	sixtrack	ref	fp00
swim	ref	fp00	twolf	ref	int00
vortex	ref	int00	vpr	route	int00

4. EXPERIMENTAL METHODOLOGY

We have used the Simplescalar-Alpha-3.0 tool set [5] to generate the dynamic instruction trace of the first 2 billion instructions for 26 programs of the SPEC benchmark suites. Table 1 shows the selected benchmarks and their inputs (compiled for the Alpha ISA, cc DEC 5.9, −O4). These programs were chosen to demonstrate how our proposed hardware/software methodology can outperform both highly efficient non-adaptive approaches and other adaptive systems on SPEC benchmarks, and additionally, because they represent different program domains (integer, floating-point, and multimedia).

Accurate cycle-by-cycle simulation was performed using a superscalar CPU simulator based on Simplescalar [5], to

Table 2: Microarchitecture parameters for the cycle-accurate simulations.

General Out-of-Order Microarchitecture	L1 Field-Programmable Data Cache
Up to 8 instructions renamed, dispatched, issued and retired per cycle Fetch Queue: 16 instructions Branch Predictor: perfect Issue Queue: 48 instructions Reorder Buffer: 256 instructions Operation latencies like Pentium 4 Load/Store Queue: 64/32 instructions Perfect memory disambiguation Store to Load forwarding I-Cache: perfect, 2-cycle load-use latency L2-Cache: 4.6 ns access time (load-use latency depends on operating frequency, f), 1.83 nJ/access L1-L2 Interface Bandwidth: 16GB/s Main Memory: 70 ns access time, 5 nJ/access	Size: 1KB, 2KB, 4KB, ..., 32KB Set-Associativity: 2-way, 4-way, 8-way Line Size: 8, 16, 32, 64 Bytes Load-use Latency (n): 1, 2, 3, 4 clock cycles, (depends on cache organization and operating frequency) 2 read/write ports **Penalizations with respect to the same cache configuration built with fixed hardware:** CPU stall time during cache reconfiguration (0.14 μs and cache flush), energy consumption during cache reconfiguration (5 μJ), cache content is lost after reconfiguration.

subsequently calculate for each tuneable L1 data cache configuration the execution time, product time-energy, and energy consumption. Table 2 lists the parameters used for the simulated processor. Benchmarks were simulated using intervals of 100,000 instructions per 1 billion instructions executed, after a warming-up of 1 billion instructions. The whole analysis interval was divided into two equal intervals of 0.5 billion instructions. The first one was used for learning, and the second one for recognition and actuation.

The standard SimPoint-1.1 Toolkit [20] was used to extract basic block vectors (BBV) from the dynamic execution of benchmarks during the analysis interval. Each instruction interval provides one BBV vector, which was transformed to a three-dimensional vector (3-D BBV) as previously described in Section 3.2.1. The 3-D BBVs provided by the execution of the learning interval were analyzed by SimPoint for classifying instruction intervals into SimPoint (SP) classes. The resulting clusters partition the 3-D representation space and determine the content of the Representation Space Table. Given a preferred architectural metric and using the above described Cache Matching Algorithm, our CPU simulator takes 3-D BBV vectors obtained from the recognition interval and assign them to one of the learned SP classes.

We used a modified version of CACTI 3.2 [17] to estimate the access time, energy consumed in each memory access, and chip area of each L1 field-programmable data cache organization, for a CMOS technology with λ = 100 nm (see Section 2.1). Our experiments used the original CACTI tool to characterize the reference configurations, which are described below. With the history of the last three SP classes, the CMA algorithm also determines for the next instruction interval whether the L1 data cache must be reconfigured.

This paper reports results for three architectural metrics: execution time, static and dynamic energy consumption of the L1 data cache and unified L2 cache, and time-energy product. As CACTI only provides estimates of dynamic energy, we calculate static energy as described in [25], with k-static = 50%, i.e. static energy is 50% of the total energy. In the experiments, we have supposed that in each reconfiguration of the field-programmable cache, the instruction flow is stalled, the contents of the L1 data cache is discarded, and L2 is updated as described in Section 2.1. The energy consumed by sensors was not considered since they are physically very small. In each cache reconfiguration, the additional energy consumed by FPCA and coprocessor including tables is 5 μJ. When reconfiguration is not activated, the energy consumed by coprocessor is negligible because the activated hardware is also small: Representation Space Table (256 bytes), Pattern Table (8 bytes), Configuration Table (50 bytes), and glue logic (four 8-bit registers and three 8-bit comparators).

4.1 Reference Configurations

A group of conventional non-adaptive cache microarchitectures called *Configurations A, B and C* was used as baseline designs. Table 3 summarizes the parameters of these non-adaptive L1 data caches considering a 100 nm CMOS technology and the advantage of being fixed hardware. The processor core is as described in Table 2. The baseline Configuration A is similar to that used by Balasubramonian et al. to evaluate their proposal of adaptive memory hierarchy [2]. This large L1 data cache was used because we tried to repeat the experiments made in [2]. The baseline configuration B is adopted by the 90 nm version of the high-performance Intel Pentium 4 processor [4], and CACTI predicts a chip area of 1.5 mm^2 for λ_{CMOS}=100 nm. Taking a chip area of 1.5 mm^2 into account, Configuration C has a L1 data cache organization that is most similar to that of the low-power Intel PXA255 processor [12].

We also evaluated a second type of reference configuration called *Adaptive-Base*. It consists of an adaptive L1 data cache with fixed clock frequency (f=1.51 GHz), and varying cache latency and energy consumption that resemble the proposal of Balasubramonian et al. [2]. The differences are that we simulate inclusive L1 and L2 caches, instead of exclusive caches, that our design has an 8-way superscalar core instead of a 4-way core, and that we consider perfect branch prediction. CACTI predicts a chip area between 67.5 mm^2 and 8.4 mm^2 for the data cache (λ_{CMOS}= 100 nm), as its capacity can adapt from 2 MB to 256 KB respectively.

Our adaptive system was guided by results obtained from an oracle model called *Ideal Dynamic Adaptation*. This model is an ideal mechanism that for each instruction interval always knows in advance and selects the cache configuration that provides the best result for one of the following architectural measures: execution time, time-energy or energy consumption. In these cases, we say that the data cache adaptation is driven either by performance, time-energy, or energy consumption (static and dynamic). This mechanism is used to determine the upper bound of the real adaptive caches and allows evaluating the respective efficiencies.

Table 3: Non-Adaptive baseline configurations considering a 100 nm CMOS technology and the advantage of being fixed hardware

Name	Prioritized Metric	Capacity	Set Associativity	Bytes/ line	R/W Ports	Hit Latency(n)	Operating Frequency (f)	Energy (nJ/ access)	Chip Area (A)
A	All	256 KB	1-way	128	2	2 cycles	1.51 GHz	0.65	8.4 mm^2
B	Execution Time, Time-Energy	16 KB	8-way	64	2	4 cycles	4.20 GHz	0.70	1.5 mm^2
C	Energy	32 KB	8-way	32	2	1 cycle	1.00 GHz	0.72	1.5 mm^2

4.2 Maximum Improvements

Next, the potential of Ideal Dynamic Adaptation is estimated and compared with results obtained by baseline configurations A and Adaptive-Base. In a first experiment, we considered the tuneable cache configurations shown in Table 2, with different access times (T_{access}), different load-use latencies (n), and requiring a variable amount of chip area (A). We suppose that the access time determines the operating frequency of processor (f) in all cases: $f = n/T_{access}$. Benchmark traces were analyzed from simulations of 1 billion instructions after a warming-up of 1 billion instructions and using 10^5 instruction intervals. The maximum operating frequency of the Ideal Dynamic Adaptation was set to f_{limit}= 1.51 GHz, and the maximum chip area of the field-programmable data cache was limited to A_{limit}= *1.5 mm^2*, i.e. f=f_{limit} and A=A_{limit}. We supposed that there are no temporal and energy overheads in the tuning and reconfiguration stage of neither Ideal Dynamic Adaptation nor Adaptive-Base baseline system. For a selection of benchmarks, Fig. 4 compares the speedup over Configuration A, time-energy per instruction and energy per instruction of five systems: Configuration A, Adaptive-Base, and Ideal Dynamic Adaptation driven by each one of the three architectural metrics. As can be observed in Fig. 4a, Ideal Dynamic Adaptation driven by performance achieves an average speedup of 16% (range from 55.2% to 1%) with respect to Configuration A (84% reduction in data cache area), and 12.4% with respect to Adaptive-Base (85% reduction in data cache area). Additionally, the ideal adaptation reduces on average the time-energy per instruction by 84% (range from 93% to 67%; see Fig. 4b), and the energy consumption by 82% (range from 94% to 69%; see Fig. 4c) with respect to Configuration A, when time-energy and energy consumption are prioritized respectively. Therefore, there is a margin for proposing new adaptive cache memories, since Adaptive-Base, for the benchmarks above considered, only reaches 23% of the performance improvement due to Ideal Dynamic Adaptation and reduces energy and time-energy by 1%.

5. EVALUATION OF THE FIELD-PROGRAMMABLE DATA CACHE

This section evaluates the potential of our *Real Adaptive System* (i.e., the Field-Programmable Data Cache with runtime adaptation) in three different design scenarios, which prioritize performance, energy consumption, and time-energy respectively. The experiments consider the same tuneable cache configurations shown in Table 2. For the first and third design scenarios, the maximum operating frequency and maximum chip area of the tuneable cache configurations are f_{limit}= 4.2 GHz and A_{limit}= 1.5 mm^2 respectively. These initial conditions restrict the number of tuneable cache organizations to 188. In these two design scenarios, the high-performance non-adaptive baseline configuration is B. For the low-power design scenario, the maximum operating frequency and maximum chip area of the tuneable cache configurations are f_{limit}= 1.0 GHz and A_{limit}= 1.5 mm^2 respectively. With these restrictions, the number of tuneable cache configurations is 8. The low-power non-adaptive baseline configuration is C.

The pipeline of the processor with field-programmable L1 data cache is supposed to be designed for the maximum frequency flimit. A reduction of the clock speed determined by some cache configurations is practically possible with no change of pipeline design.

The evaluation of our adaptive data cache was made after applying the Cache Matching Algorithm to all benchmarks. Three algorithmic stages were established: Learning, Recognition, and Actuation. Each of them requires additional experimental conditions, which are described next, along with the evaluation results.

5.1 Results of the Learning Phase

In this stage, four tasks were performed by software in the following sequence: feature extraction, identification of the Representation Space, and setting of contents of the Pattern Table and Configuration Table. All of them were applied to each benchmark in simulations of 5,000 intervals of 100,000 instructions (0.5 billion instructions) just after warning-up.

A BBV vector was associated to each instruction interval. With the feature extraction, the upper and lower limits of the three ranges of BB sizes were determined. These ranges determine a linear transformation with equal coefficients, which allows a three-dimensional BBV to be created for each instruction interval. The representation space was determined after the 3-D BBV vectors had been clustered with the K-means algorithm, which was implemented by the SimPoint toolset. The Representation Space Table of each benchmark stores for every 3-D BBV a SP class code that identifies the cluster in which the BBV was classified. The number of SimPoint classes varied from 14 (`gzip`) to 1 (`crafty`, `equake`). On average, 6 classes per benchmark were observed.

Next, the Pattern Table was generated. In each table entry, a Cache-ID identifying the preferred cache configuration of a SP class is stored. For this task, the following steps were done: (a) the prioritized architectural metric (perfor-

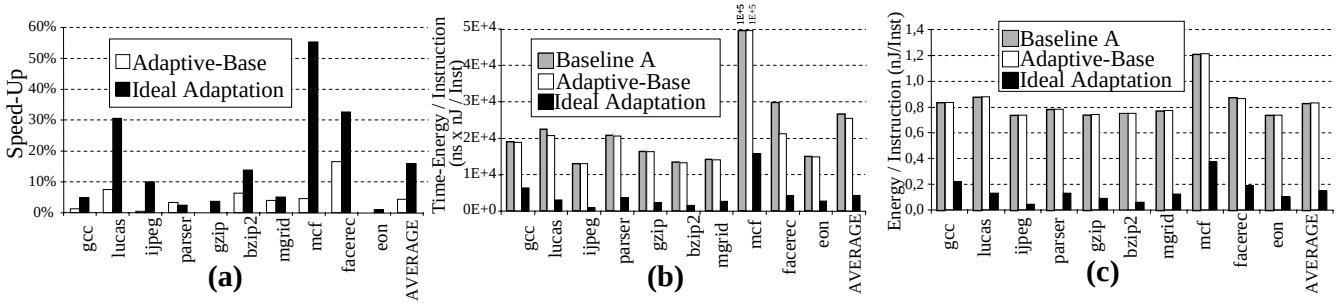

Figure 4: (a) Performance improvement over baseline Configuration A of Adaptive-Base and Ideal Dynamic Adaptation driven by Performance. (b) Time-Energy per Instruction for: baseline Configuration A, Adaptive-Base, and Ideal Dynamic Adaptation driven by Time-Energy. (c) Energy per Instruction for: baseline Configuration A, Adaptive-Base, and Ideal Dynamic Adaptation driven by Energy.

mance, energy, time-energy) was set, (b) the preferred metric was monitored during a complete learning interval after each cache configuration had been picked, and (c) the most frequent cache configuration that was selected with the best value of the preferred metric in the instruction intervals assigned to each SP class was identified. When adaptation was driven by performance, energy, or time-energy, the total number of cache configurations that were assigned to at least one SP class was 16, 3, and 20 respectively (see the first five columns in Table 4). As can be seen, the parameters of the selected cache configurations are very different from one preferred metric to another. This means that the adaptive cache must provide the possibility of picking at least these configurations in order to exploit program phases with high efficiency. When adaptation was driven either by performance, energy, or time-energy, the average number of cache configurations contained in the Pattern and Configuration tables of each benchmark was 3, 2 and 3 respectively (see Table 5). This means that for each program, not many cache configurations are required to provide an efficient use of the L1 data cache. Additionally, it can be observed that the most efficient configurations that are selected in Real Adaptation depend on the program.

We observed that the number of cache configurations in the Pattern Table for Real Dynamic Adaptation is significantly reduced with respect to the number of configurations picked in Ideal Adaptation (described in Section 4.1). This result is due to the existence of two effects associated to the lack of precision of the K-means clustering algorithm and lack of information of the BBV vectors. On the one hand, several SP classes are assigned to the same cache configuration. In this case, the SP classes are grouped together to form a single SP class. On the other hand, the cache configurations that exhibit the highest performance or lowest energy consumption in the instruction intervals of a same SP class are frequently different. Since an SP class must be matched with a single cache configuration, only one configuration is not discarded. Thus, the efficiency provided by Real Adaptation is lower than the ideal mechanism.

Finally, the Configuration Table should be loaded with the bitstreams of the cache configurations that appear in the Pattern Table. These bitstreams implement the change of cache configuration and are determined in design time, after establishing the internal organization of the field programmable cache.

5.2 Results of the Recognition and Actuation Phases

Once all the tables had been initialized, the behaviour of Real Adaptation was simulated in 5,000 intervals of 100,000 instructions (0.5 billion instructions) just after the learning interval. The execution time and energy consumption of the learning intervals are not accounted for because they are supposed to be amortized during the overall program execution. For benchmarks `gcc` and `gzip`, we simulated the recognition and actuation stages using two different inputs and a single learning stage with one of the respective inputs.

After the execution of each instruction interval, the three-dimensional BBV vector provided by the processor core and the contents of the Representation Space and Pattern tables are used to find the corresponding cache configuration. A reconfiguration is fired only when three consecutive instruction intervals are assigned to the same cache configuration, and this is different from the currently activated. We assume that each cache reconfiguration additionally introduces a 1.0 μs delay overhead and a 5.0 μJ energy overhead. These penalizations are necessary to take into account the reconfiguration process of the field-programmable data cache and the update of L2. When such reconfiguration occurs, the unused portion of the cache is power-down and the cache content is lost.

When performance, energy, or time-energy was prioritized, the average number of reconfigurations performed during the recognition and actuation phase was 52, 24 and 65 respectively, i.e. less than 1.5% of all instruction intervals simulated for the recognition stage. This is equivalent to observe average phase lengths of 9.6, 20.8 and 7.7 million instructions respectively. These results indicate that the program patterns/phases exhibit high temporal locality, which reduces the overheads due to hardware reconfiguration. Note that, after finalizing the learning phase, the runtime selection of a cache configuration does not require previous tuning of all available microarchitectures before the selection of one with the highest performance, as proposed in other adaptive methodologies [2, 19]. Therefore, our on-line control method requires lower overhead for the determination of the stable state of the microarchitecture than prior reported studies.

5.3 Performance and Energy Consumption

Fig. 5 shows the performance results for Real Adaptation driven by the three metrics and Ideal Adaptation driven by

Table 4: Cache Configurations selected in the learning stage of the Real Dynamic Adaptation when is driven either by performance, time-energy, or energy. The column *% Total Intervals* shows the percentage of instruction intervals with respect to the overall simulated instruction intervals (26 benchmarks × 5,000 intervals) that a cache configuration was selected in the recognition phase. Legends: K= KBytes of capacity, w= ways of set-associativity, B=bytes/line, c= hit latency (cycles).

Cache ID	Configuration of the L1 Data Cache				% Total Intervals
Adaptation driven by Performance					
158	32K	2w	64B	3c	1.40%
174	32K	4w	64B	3c	9.50%
178	32K	8w	8B	3c	0.81%
186	32K	8w	32B	3c	7.69%
226	16K	4w	8B	3c	0.31%
230	16K	4w	16B	3c	1.08%
242	16K	8w	8B	3c	0.77%
246	16K	8w	16B	3c	22.13%
250	16K	8w	32B	3c	11.13%
254	16K	8w	64B	3c	3.23%
261	8K	1w	16B	2c	0.38%
265	8K	1w	32B	2c	0.88%
269	8K	1w	64B	2c	0.46%
314	8K	8w	32B	3c	31.00%
318	8K	8w	64B	3c	1.56%
321	4K	1w	8B	2c	7.65%
Adaptation driven by Time-Energy					
129	32K	1w	8B	2c	0.81%
133	32K	1w	16B	2c	0.12%
137	32K	1w	32B	2c	0.58%
141	32K	1w	64B	2c	0.35%
154	32K	2w	32B	3c	9.00%
158	32K	2w	64B	3c	20.54%
174	32K	4w	64B	3c	3.73%
193	16K	1w	8B	2c	0.08%
197	16K	1w	16B	2c	12.77%
201	16K	1w	32B	2c	6.19%
205	16K	1w	64B	2c	4.06%
210	16K	2w	8B	3c	4.04%
214	16K	2w	16B	3c	10.35%
246	16K	8w	16B	3c	0.58%
261	8K	1w	16B	2c	2.98%
265	8K	1w	32B	2c	7.02%
269	8K	1w	64B	2c	7.44%
318	8K	8w	64B	3c	0.18%
321	4K	1w	8B	2c	9.17%
409	2K	2w	32B	2c	0.04%
Adaptation driven by Energy					
152	32K	2w	32B	1c	62.94%
208	16K	2w	8B	1c	36.71%
252	16K	8w	64B	1c	0.35%

Table 5: For every benchmark, cache configurations that are selected in the learning phase of each type of real adaptation (driven either by performance, time-energy or energy). Every configuration is identified by its Cache ID, which is shown in Table 4. The percentage of instruction intervals executed with each cache configuration and type of dynamic adaptation during the recognition stage (5,000 intervals/benchmark for each adaptation type) is shown in brackets.

Bench	Performance	Time-Energy	Energy
ammp	314(12%), 321(88%)	321(100%)	208(100%)
applu	314(100%)	158(100%)	152(100%)
apsi	246(98%), 318(2%)	154(18%), 197(80%), 318(2%)	152(98%), 252(2%)
art	242(7%), 246(2%), 250(45%), 321(46%)	261(11%), 265(38%), 269(2%), 321(49%)	152(98%), 208(2%)
bzip	174(15%), 246(1%), 246(28%), 314(56%)	197(1%), 205(15%), 214(34%),261(7%), 265(42%)	152(20%), 208(80%)
crafty	246(100%)	210(100%)	208(100%)
eon	250(91%), 314(3%), 321(6%)	214(91%), 265(9%)	152(9%), 208(91%)
equake	314(100%)	201(100%)	208(100%)
facerec	158(10%), 226(8%), 230(28%), 246(20%), 250(26%), 314(9%)	158(71%), 193(2%), 269(24%), 321(3%)	152(80%), 208(14%), 252(5%)
fma3d	314(100%)	197(62%), 201(38%)	208(100%)
galgel	186(97%), 246(3%)	133(3%), 174(97%)	152(100%)
gcc	158(1%), 186(33%), 250(41%), 314(27%)	137(8%), 154(49%), 158(26%), 205(1%), 261(6%), 265(11%)	152(96%), 208(5%)
gzip	174(13%), 186(24%), 246(42%), 265(5%), 269(11%), 314(2%), 318(2%), 321(2%)	154(23%), 261(13%), 265(5%), 269(20%), 321(41%)	152(87%), 208(13%)
ijpeg	158(13%), 174(54%), 186(23%), 314(9%)	141(4%), 158(10%), 201(23%), 205(45%), 265(9%), 269(9%)	152(100%)
lucas	158(13%), 242(13%), 246(15%), 261(10%), 265(18%), 314(21%), 318(10%)	158(22%), 246(15%), 269(63%)	152(85%), 208(15%)
mcf	178(21%), 314(36%), 321(43%)	129(21%), 261(36%), 321(43%)	152(58%), 208(42%)
mesa	186(9%), 314(91%)	141(5%), 158(5%), 197(89%), 261(1%)	152(11%), 208(89%)
mgrid	250(10%), 254(84%), 318(5%)	158(95%), 205(5%)	152(100%)
parser	174(20%), 186(11%), 246(45%), 250(20%), 314(4%)	154(45%), 158(40%), 205(11%), 261(1%), 321(3%)	152(96%), 208(4%)
perlmbk	314(100%)	210(5%), 214(95%)	208(100%)
sixtrack	174(99%), 246(1%)	158(100%)	152(100%)
swim	250(29%), 314(71%)	137(7%), 205(29%), 61(3%), 265(19%), 269(41%)	152(100%)
twolf	246(100%)	154(100%)	152(100%)
vortex	246(100%)	197(100%)	208(100%)
vpr	174(46%), 186(4%), 246(49%), 269(1%)	158(36%), 214(49%), 265(14%), 269(1%)	152(99%), 252(1%)
wupwise	314(66%), 318(20%), 321(14%)	158(29%), 265(36%), 269(34%), 409(1%)	152(100%)

performance. This figure depicts the speedup over the execution time for the respective baseline configurations (B when adaptation is driven by performance and time-energy, and C when adaptation is driven by energy). On average, the performance improvement achieved by our adaptive cache memory with on-line control is 15.2% and 3.9% with respect to the high-performance baseline configuration B when adaptation is driven by performance and time-energy respectively (depicted in Fig. 5 as *Real(performance)* and *Real(time-energy)*). The maximum improvement was 64.6% and 61.2% for `galgel`.

In spite of the limitations on the operating frequency and energy consumption of the field-programmable cache, the efficiency achieved by the Cache Matching Algorithm and the reconfiguration overhead, the Real Adaptation driven by performance can achieve 78.8% of the average improvement provided by Ideal Adaptation driven by performance (depicted in Fig. 5 as *Ideal(performance)*). Note that the Real Adaptation driven by energy provides 9.4% performance improvement with respect to the low-power baseline configuration C. This means that it is not necessary to reduce performance in order to save energy. This is due to the selection of tuneable cache configurations that provide lower execution times and consume lower energy than the baseline configuration C.

Fig. 6 shows the time-energy results for Real Adaptation driven by the three metrics and Ideal Adaptation driven by time-energy. As can be observed, Real Adaptation driven by time-energy demonstrated a mean reduction of time-energy by 53.9% with respect to baseline B. This is equivalent to achieve 96% of the maximum reduction achieved by Ideal Adaptation driven by time-energy (depicted in Fig. 6 as *Ideal(time-energy)*). Fig. 7 shows the energy consumption results for Real Adaptation driven by the three metrics and Ideal Adaptation driven by energy. When Real Adaptation is driven by energy (depicted in Fig. 7 as *Real(energy)*), this metric is reduced on average by 46.7% with respect to baseline configuration C, which corresponds to 99% of the maximum reduction achieved by ideal adaptation driven by energy (depicted in Fig. 7 as *Ideal(energy)*).

Table 6 summarizes the average performance improvements and reductions of time-energy and energy consumption when each one of the three architectural metrics is prioritized. As can be observed, there is no degradation of performance and energy consumption when each architectural metric is prioritized. This phenomenon is mainly due to the frequent selection of caches with smaller sizes and lower energy cost per memory access than the non-adaptive baseline configurations. Real Adaptation enables us to increase frequency when the size is reduced, while not increasing both the number of hit cycles and CPI. Some benchmarks exhibit a reduction in the execution time and an increase in the energy consumption. See for example the results for Real Adaptation driven by performance of `applu`, `apsi` or `gcc` in Fig. 5 and 7 (depicted as *Real(performance)*). In these cases, the selection of larger caches causes the access time and energy per memory access to increase. The number of hit cycles is allowed to increase provided that the operating frequency (f) is lower than the maximum f_{limit}. Then, the operating frequency can be higher, but the CPI and energy consumption are also higher. However, these are not common cases.

6. RELATED WORK

Two aspects of the reconfigurable systems that are integrated into a processor are needed to be determined in the design process: the hardware organization, and its control methodology. Both aspects are reviewed next.

Some sections of a general-purpose processor have been proposed to be reconfigurable: the ALU functional unit [9, 22, 24], the clock generator and power supply [19], the cache memory [2, 13, 18, 23, 25]. Our proposal of adaptive cache memory differentiates from these systems in the following ways: (1) The number of possible cache configurations is two orders of magnitude higher than other adaptive caches, with approximately a 10% increase in chip area. Depending on each application, some configurations are used to provide the highest performance and other configurations are used to provide the lowest energy consumption. (2) The operating frequency of the processor/cache system and cache hit latency can be independently varied. A wide range of frequencies and chip areas can be used in different scenarios, from low-cost processors to high-performance processors. (3) The configuration bitstream is sufficiently small to not significantly impact performance improvement and energy consumption.

The control methodology is used to determine the hardware configuration that best suits the characteristics of a given program or execution phase. Depending on the dynamic or static system behaviour, two groups of control methodologies can be identified: on-line and off-line control. Methodologies for on-line control take clues from the processor hardware to infer characteristics of programs. Different hardware events have been used: branch frequency [2], number of cache misses [13, 23], number of cache hits [13], utilization of the issue queues [19], and the invocation of the major subroutines of the applications [11]. Work on this subject has explored three basic properties of algorithms [2, 6]: (a) efficiency on detecting a phase boundary during execution of a process, (b) the tuning overhead, and (c) the reconfiguration overhead. Additionally, we have observed that a fourth property, the set of tuneable configurations, is required to be analyzed.

Our on-line control methodology differentiates from the above mentioned in the following ways: (1) We do not use the same tuneable cache configurations for all programs. This is one of the keys of our work. (2) We propose a methodology based on basic block vectors to know the most efficient configurations for each program. (3) The tuning and reconfiguration overheads are relatively low and independent of the number of tuneable configurations, as we do not prove all the possible cache configurations each time a program phase change is detected. (4) Most of the adaptive techniques that have been proposed for energy saving in cache memory reduce the energy or the time-energy product but also reduce performance [1, 2, 10, 14]. On the other hand, prior adaptive systems proposed for performance improvement increase energy consumption [2, 19]. Our proposal improves processor performance and reduces energy consumption at the same time. (5) And finally, we show that using the same reconfigurable hardware, cache adaptation can be driven either by performance, energy or time-energy. So, cache efficiency can be also tuned to one of these prioritized architectural metrics in run-time. This was pointed out in [11] but neither analysed nor applied to any processor hardware.

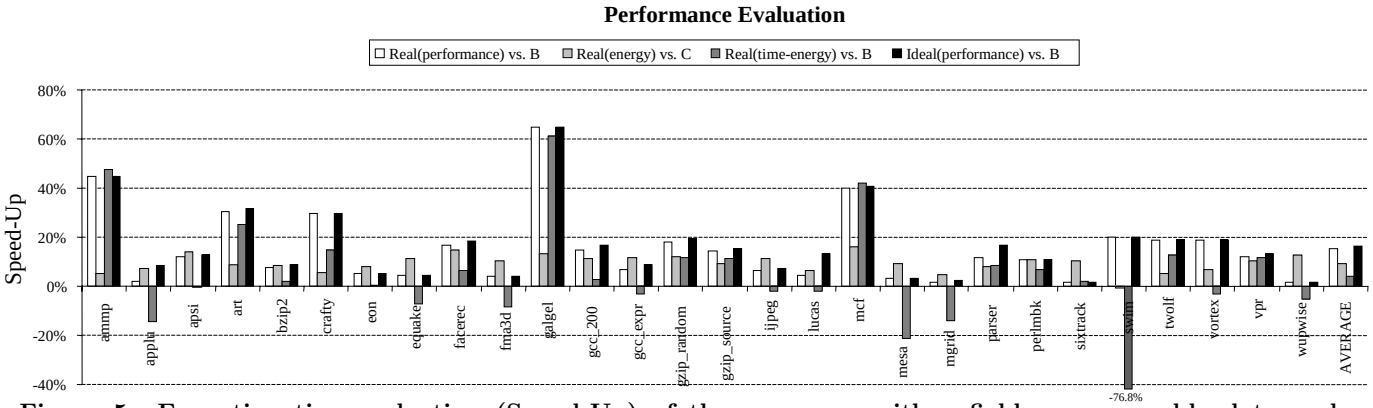

Figure 5: Execution time reduction (Speed-Up) of the processor with a field-programmable data cache managed by either our proposal of on-line control (Real) or with future knowledge (Ideal). Legends: *x(y) vs. z*, *x* represents the type of cache adaptation (Real,Ideal), *y* represents the prioritized metric that drives the cache adaptation (performance,energy,time-energy), and *z* represents the baseline configuration (B,C).

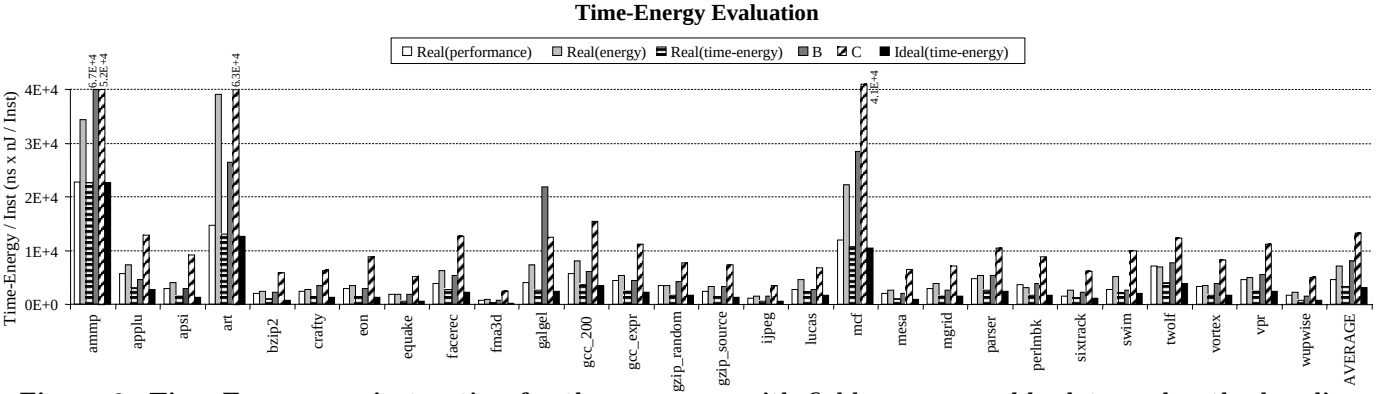

Figure 6: Time-Energy per instruction for the processor with field-programmable data cache, the baseline configurations B and C, and Ideal Adaptation driven by time-energy. The legends are the same as used in Fig. 5.

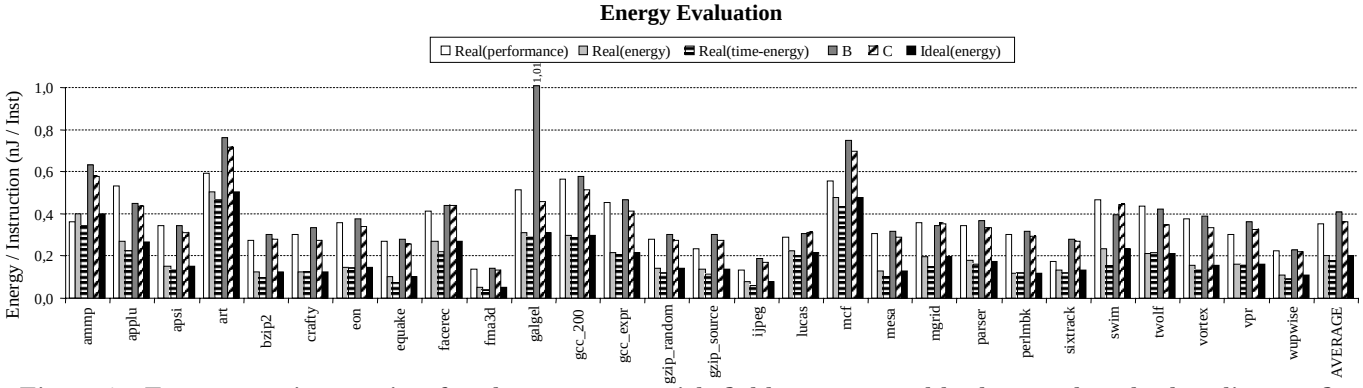

Figure 7: Energy per instruction for the processor with field-programmable data cache, the baseline configurations B and C, and Ideal Adaptation driven by energy consumption. The legends are the same as used in Fig. 5 and 6.

Table 6: Results of Real Adaptation. Reconfiguration overhead, clock speed penalization and energy consumption overhead for the field-programmable data cache were taken into account.

Real Adaptation driven by...	f_{limit}	Execution Time Reduction	Time-Energy Reduction	Energy Reduction	Baseline Configuration (operating frequency)
Performance	4.0 GHz	15.2%	14.8%	9.9%	B: High-Performance (4.2 GHz)
Time-Energy	4.0 GHz	3.9%	53.9%	53.8%	B: High-Performance (4.2 GHz)
Energy	1.25 GHz	9.4%	51.9%	46.7%	C: Low-Power (1.0 GHz)

Off-line compiling, profiling and instrumentation of the application can be used to alternatively implement the adaptation control [11, 24]. It can provide a more global view of the program than with a hardware solution and in some cases achieves better results [16]. Our FPCA cache can be exclusively managed by a software procedure. We have observed that a high percentage of the performance improvement and energy saving demonstrated by Real Adaptation is achieved (more than 65%). This approach decreases the chip area by avoiding the use of the hardware coprocessor and adaptive tables [3].

7. CONCLUSIONS AND FUTURE WORK

We have proposed and evaluated the performance and energy consumption of an adaptive L1 data cache, which is based on field-programmable technology and managed by a human-like control system. With this proposal, a high efficiency of use of the data cache can be achieved by using a specialized reconfigurable circuit.

The main contributions of the paper are the followings. The field-programmable data cache (1) provides a change mechanism with low reconfiguration overhead, (2) is characterized by access times only slightly larger (20%) than similar non-adaptive circuits, (3) uses the SimPoint learning mechanism, but applied to a different goal: to reduce tuning overhead, (4) can be tuned to performance improvement or energy saving using the same hardware, (5) improves performance and energy consumption at the same time, (6) achieves a high percentage of the performance improvement that a ideally adaptive cache with future knowledge would achieve, with independence of the design-time constraints on operating frequency and chip area, and (7) is superior to previous and similar approaches. We additionally (8) proposed a predictor mechanism of the data working-set of a program, and discovered that (9) higher performance and energy saving can be achieved when for each benchmark and prioritized metric (execution time, energy consumption, etc.), the set of preferred cache configurations is accurately determined, which justifies the existence of a field-programmable cache. One of the goals of the proposed control methodology consists in efficiently doing the cache matching operation.

The efficiency provided by the field-programmable cache that is presented in this paper could be exploited in other cache levels. This is one of our research goals in the near future.

8. REFERENCES

[1] R. Bahar, G. Albera, and S. Manne. Power and performance tradeoffs using various caching strategies. In *ISLPED '98: Proc. of the 1998 Intl. Symposium on Low Power Electronics and Design*, pages 64–69. ACM Press, 1998.

[2] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. A dynamically tunable memory hierarchy. *IEEE Tran. Computers*, 52(10):1243–1257, 2003.

[3] D. Benitez, J. Moure, D. Rexachs, and E. Luque. Performance and power evaluation of an intelligently adaptive data cache. *Lecture Notes in Computer Science*, 3769:363–375, December 2005.

[4] D. Boggs, A. Baktha, J. Hawkins, D. Marr, J. Miller, P. Roussel, R. Singhal, B. Toll, and K. Venkatraman. The microarchitecture of the intel pentium 4 processor on 90nm technology. *Intel Technology Journal*, http://developer.intel.com/technology/itj, February 2004.

[5] D. Burger, T. M. Austin, and S. Bennett. Evaluating future microprocessors: The simplescalar tool set. Technical Report CS-TR-1996-1308, University of Wisconsin-Madison. Computer Sciencies Department, 1996.

[6] A. S. Dhodapkar and J. E. Smith. Managing multi-configuration hardware via dynamic working set analysis. In *ISCA '02: Proc. of the 29th Intl. Symposium on Computer Architecture*, pages 233–244. IEEE Computer Society, 2002.

[7] A. S. Dhodapkar and J. E. Smith. Comparing program phase detection techniques. In *MICRO '03: Proc. of the 36th IEEE/ACM Intl. Symposium on Microarchitecture*, pages 217–227. IEEE Computer Society, 2003.

[8] R. Duda, P. Hart, and D. Stork. *Pattern Classification*. Wiley Interscience, 2nd edition, 2000.

[9] M. Epalza, P. Ienne, and D. Mlynek. Adding limited reconfigurability to superscalar processors. In *PACT '04: Proc. of the 13th Intl. Conference on Parallel Architectures and Compilation Techniques*, pages 53–62. IEEE Computer Society, 2004.

[10] Z. Hu, S. Kaxiras, and M. Martonosi. Timekeeping in the memory system: predicting and optimizing memory behavior. *SIGARCH Comput. Archit. News*, 30(2):209–220, 2002.

[11] M. C. Huang, J. Renau, and J. Torrellas. Positional adaptation of processors: application to energy reduction. In *ISCA '03: Proc. of the 30th Intl. Symposium on Computer Architecture*, pages 157–168. ACM Press, 2003.

[12] Intel. *Intel XScale Microarchitecture for the PXA255 Processor. User Manual*. Intel, 2003.

[13] C. Kim, D. Burger, and S. W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *ASPLOS-X: Proc. of the 10th Intl. Conference on Architectural Support for Programming Languages and Operating Systems*, pages 211–222. ACM Press, 2002.

[14] J. Kin, M. Gupta, and W. H. Mangione-Smith. The filter cache: an energy efficient memory structure. In *MICRO '97: Proc. of the 30th ACM/IEEE Intl. Symposium on Microarchitecture*, pages 184–193. IEEE Computer Society, 1997.

[15] C. Lee and D. A. Landgrebe. Feature extraction based on decision boundaries. *IEEE Trans. Pattern Anal. Mach. Intell.*, 15(4):388–400, 1993.

[16] G. Magklis, M. L. Scott, G. Semeraro, D. H. Albonesi, and S. Dropsho. Profile-based dynamic voltage and frequency scaling for a multiple clock domain microprocessor. In *ISCA '03: Proc. of the 30th Intl. Symposium on Computer Architecture*, pages 14–27. ACM Press, 2003.

[17] P.Shivakumar and N. Jouppi. Cacti 3.0: An integrated cache timing, power, and area model. Technical Report 2001/2, Compact WRL, 2001.

[18] P. Ranganathan, S. Adve, and N. P. Jouppi. Reconfigurable caches and their application to media processing. In *ISCA '00: Proc. of the 27th Intl. Symposium on Computer Architecture*, pages 214–224. ACM Press, 2000.

[19] G. Semeraro, G. Magklis, R. Balasubramonian, D. H. Albonesi, S. Dwarkadas, and M. L. Scott. Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling. In *HPCA '02: Proc. of the 8th Intl. Symposium on High-Performance Computer Architecture*, pages 29–42. IEEE Computer Society, 2002.

[20] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *ASPLOS-X: Proc. of the 10th Intl. Conference on Architectural Support for Programming Languages and Operating Systems*, pages 45–57. ACM Press, 2002.

[21] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *Proc. of the 30th Intl. Symposium on Computer Architecture*, pages 336–349. ACM Press, 2003.

[22] S. Vassiliadis, S. Wong, and S. Cotofana. The MOLEN $\rho\mu$-coded processor. *Lecture Notes in Computer Science*, 2147:275–285, 2001.

[23] S. H. Yang, B. Falsafi, M. D. Powell, and T. N. Vijaykumar. Exploiting choice in resizable cache design to optimize deep-submicron processor energy-delay. In *HPCA '02: Proc. of the 8th Intl. Symposium on High-Performance Computer Architecture*, page 151. IEEE Computer Society, 2002.

[24] A. Ye, A. Moshovos, S. Hauck, and P. Banerjee. Chimaera: A high-performance architecture with a tightly-coupled reconfigurable functional unit. In *Proc. of the 27th Intl. Symposium on Computer Architecture*, pages 225–235, June 2000.

[25] C. Zhang, F. Vahid, and W. Najjar. A highly configurable cache architecture for embedded systems. In *ISCA '03: Proc. of the 30th Intl. Symposium on Computer Architecture*, pages 136–146. ACM Press, 2003.