Journal of Embedded Computing 1 (2005) 1–13
IOS Press

Tiny split data-caches make big performance impact for embedded applications

Afrin Naz, Krishna Kavi[*], Wentong Li and Philip Sweany
Department of Computer Science and Engineering, The University of North Texas, 3940 North Elm street, Denton, Texas 76206-1366, USA

Abstract. This paper shows that even very small data caches, when split to serve data streams exhibiting temporal and spatial localities, can improve performance of embedded applications without consuming excessive silicon real estate or power. It also shows that large block sizes or higher set-associativities are unnecessary with split cache organizations. We use benchmark programs from MiBench to show that our cache organization outperforms other organizations in terms of miss rates, access times, energy consumption and silicon area.

Keywords: Embedded systems, cache memories, victim cache, stream buffer, spatial locality, temporal locality

1. Introduction

Both experience and common sense tell us that careful design of memory systems is crucial to achieving good performance on any computer architecture. Many different memory organizational ideas have been evaluated in the literature. However, too often these studies are used to make generalizations as to the effectiveness of memory techniques across a very wide range of architectures, and not for specific application domains. Our experiments show that two cache memory techniques provide significant benefits for embedded systems. More specifically, our studies show that victim caches and cache line prefetching are effective in embedded systems when used in conjunction with split data caches.

Challenges to the design of processing elements for embedded applications are more stringent than those for desktop applications. Embedded applications place requirements along a number of dimensions including tighter constraints on functionality and implementation. Not only must the application's functionality be correct, it often must meet strict timing constraints and be designed to function within limited resources such as memory size, available power, and allowable weight. In embedded systems research, investigators have begun to focus on the use of cache memories for achieving ever increasing performance requirements. In this paper we explore how to design small caches that achieve high performance for embedded applications while remaining both energy and area efficient.

For two decades computer architects have proposed smart cache-control mechanisms and novel cache architectures that detect program access patterns and fine-tune cache policies to improve both overall cache use and data localities for desktop applications. In our previous research [29,30], we proposed split data-caches for scientific applications: an array cache and a scalar cache. Our results have shown that the split-cache organization achieves lower miss rates and shorter average access times even when the combined size of array and scalar caches is roughly one quarter the size of the unified data cache used in our comparisons. In this paper we show that split caches benefit even nonscientific applications. The profound benefits are evident when working with the small L-1 caches often found in embedded systems. This work makes several significant contributions. First, leveraging our previous studies of split data caches for scientific applications, we evaluate

[*]Corresponding author: Department of CSE, PO Box 311366, University of North Texas, Denton, 76203, USA. Tel.: +1 940 565 2767; E-mail: kavi@cse.unt.edu.

split data-caches for applications often encountered in embedded systems. Second we evaluate a cache architecture that uses split data caches with both a victim cache and a stream buffer to further reduce (silicon) area, access time, and dynamic power consumed by cache memories while retaining performance gains. Victim caches are based on the fact that reducing the cache misses due to line conflicts for data exhibiting temporal locality provides an effective way to improve cache performance. Stream buffers tend to eliminate cold misses and prefetch data exhibiting spatial locality. Victim cache nicely complements the direct mapped scalar cache in terms of minimizing conflict misses. Stream buffers add to our array cache with prefetching capabilities. When using our augmented split caches for embedded applications, our results show excellent reductions in both memory size and memory access time, translating into reduced power consumption. Our cache architecture (scalar cache 512 bytes, array cache 4096 bytes, victim cache 256 bytes and stream buffer 128 bytes) reduces the overall cache size by 80%, access time on average by 39%, and energy consumption on average by 26% when compared with an 8k unified direct-mapped cache with a 32 k level-2 cache.

We organize the remainder of this paper as follows. Section 2 provides a survey of cache optimization techniques, while Section 3 discusses limitations of caches in embedded systems. In Section 4, we describe our proposed cache organizations. Section 5 describes benchmarks and experimental set up used in our evaluation, while Section 6 presents the results. Section 7 provides a survey of related research. We present our conclusions in Section 8.

2. Cache and its optimization

Although caching dates back to Von Neumann's classic 1946 paper that laid the foundation for modern computing, cache memories have become critical to processor performance since the beginning of 1990's – as the gap between the processor cycle and memory latency times has increased dramatically. Cache memories reside between a large, relatively slow, inexpensive information source (main memory) and a much faster consumer of that information, the processor. The success of cache memories has been explained by the property of locality of reference [34], which all programs exhibit to a large degree. Programs exhibit two types of localities, temporal and spatial. Temporal locality implies that, once a location is referenced, a high probability exists that it will be soon referenced again, and less likely to do so as the time passes. Spatial locality implies that when an instruction or datum is accessed, nearby instructions or data will likely be accessed soon. Caches exploit locality to shorten the effective access time to data, thereby reducing the cost of accessing main memory. Numerous techniques have been proposed in the literature to further improve the efficiency of cache memories for desktop applications. Major cache optimization techniques (to improve either or both miss rate and miss penalty) are generally categorized as: (1) increasing block size and cache size, (2) increasing associativity, (3) complementing the regular cache with victim cache, (4) prefetching data, or (5) including additional cache hierarchy.

2.1. Increasing block size and cache size

The simplest way to reduce miss rate is to use large block sizes; large blocks reduce cold misses and aid in data prefetching. Unfortunately, larger blocks increase miss penalty, which may outweigh the benefits of reduced miss rates. Actually, increasing block size without increasing the total cache size increases other types of misses because increasing only block size (not cache size) reduces the number of lines, leading to an increase in conflict and capacity misses [13]. Current desktop computers have used larger caches with larger block sizes for off-chip caches. Some of these caches are as large as the main memories of a decade ago [13].

2.2. Increasing associativity and modifying associative caches

Another common technique for reducing miss rates is increased associativity. Caches with higher associativity (4- to 8-way associativity) have become common in both desktop and server systems. Unfortunately, the design of a first-level cache always involves fundamental tradeoffs between miss rates and access times. Direct-mapped caches have proven to be simpler, easier to design and require less silicon area than caches with higher associativities. The main disadvantage of a direct-mapped cache is the high conflict miss rate – conflict misses typically account for 40% of all direct-mapped cache misses [18]. Conversely for caches with higher associativity, the main advantage becomes lower miss rate, but such caches have higher access times as they require associative searches of sets and multiplexing of the appropriate data words to the processor.

2.3. *Complementing cache with victim cache*

A Victim cache is a fully associative cache, with typically 4 to 16 cache lines that reside between a direct-mapped L1 cache and the next level of memory [18]. On a main cache miss, the victim cache is checked before going to the next level of memory. If the address hits in the victim cache the desired data is returned to the CPU and swapped (or promoted) with the data currently occupying the primary cache. Upon a miss in victim cache, the next level of memory is accessed and the arriving data is placed in the primary cache, moving the current data to victim cache. In this case an element from victim caches has to be removed (or written back to next level of memory) to make room for the newly victimized data.

While set-associative caches, with fewer conflict misses, offer lower miss rates than direct-mapped caches, they cost more and incur longer access times on a hit. Victim caches, in contrast, reduce conflict misses of direct-mapped caches without affecting its fast hit access times. Because victim caches are fully associative (albeit small), they can simultaneously hold many blocks that might conflict in direct-mapped cache. If most of the conflicting blocks can fit in victim cache, both the miss rates and the average access times improve.

2.4. *General prefetching*

As another technique to improve efficiency, prefetching or exploiting the overlap of processor computations with data access has proven to be effective in tolerating large memory latencies in desktop systems [3,26]. Although increasing line size presents the simplest way of prefetching, line sizes cannot be made arbitrarily large without both increasing miss rates and greatly increasing the amount of data transferred on cache misses (thus increasing miss penalties) [13]. Prefetching can be either hardware or software based [3,26]. Hardware-based prefetching requires additional hardware connected to the cache; prefetches are dynamic without compiler intervention. Software prefetching relies on compiler technology to insert explicit prefetch instructions. In desktop systems, both instruction and data prefetching frequently occur in hardware outside the cache. Typically, on a miss the processor fetches additional blocks along with the requested block. The processor places the requested block in the primary cache and places the prefetched blocks either in the primary cache or in an external buffer. On a future reference, if the processor locates the requested block in the buffer, the original cache request to next level of memory is cancelled, the processor reads the block from buffer and issues the next prefetch request. For data with spatial locality, prefetching is beneficial.

The stream buffer is a fully associative, FIFO buffer with 4 or 5 entries specially designed to support direct-mapped cache through hardware based prefetching [18]. A miss induces the fetching of the missed block along with successive blocks stored in the buffer rather than the cache. Our intent is to use the stream buffer for prefetched blocks and avoid cache pollution (premature data displacement).

2.5. *Multilevel caches*

Inclusion of additional cache hierarchy provides a common technique to improve performance of desktop applications. Adding a second level of cache between the original cache and the main memory improves access times. The first-level cache can remain small enough to match the clock cycle time of fast CPU, while the second-level cache can become large enough to capture most accesses, thereby lessening the access time. Some recent architectures contain a third level cache, whereby an even larger cache is placed between the second-level cache and the main memory.

3. Issues in embedded systems

Embedded systems present new challenges to computer architects. First, extreme cost sensitivity requires that designers pay more attention to optimization of physical size requirements. This may, in turn, require new approaches to the design of memory hierarchies and cache systems. Second, for almost all battery-operated systems, reducing total energy consumed is critical. Studies have shown that on-chip caches are responsible for 50% of an embedded processor's total power dissipation [7,22,38] and, thus, any savings in cache memory power can be significant in the overall power savings. For multimedia applications, the bandwidth and power requirements become more important issues in designing memory systems. Within embedded systems, hard real-time systems present additional problems. The strict deadlines of hard real-time systems require not only short *average* access times, but also narrow upper bounds on memory access time. Because caches generally lead to unpredictable execution times, real-time systems have traditionally not used caches [16]. At the same time, caches' performance benefits should not be ignored. Because of these often, conflicting demands, considerable interest in cache design has arisen within the real-time systems research community [9,17,19,22,31].

Caches with higher associativity remain unpopular for embedded systems mostly because higher associativity leads to high power consumption [31,38]. Thus most cache design efforts have concentrated on optimizing direct-mapped cache organizations. For desktop applications, the simplest way to improve performance is to increase the cache size and block size. Unfortunately, embedded systems cannot include large caches or arbitrarily large block sizes because of the physical size and power constraints. It should also be noted that victim caches commonly appear in desktop computing because energy consumption is generally less critical than higher performance requirements. Traditionally, designers have viewed complementing direct-mapped cache with a victim cache as an inappropriate choice for embedded systems [40], because fully associative victim caches require additional energy and silicon area. Similarly, although successful prefetching can reduce miss rates, any useless prefetching not only wastes the embedded system's valuable power but may also cause cache pollution. And cache pollution leads to additional misses and wasted energy. However, as we show in this paper, one can design embedded systems with victim caches and stream buffers, to improve cache performance without increasing energy requirements. While performing design space explorations, we have made another key observation. For desktop systems, designers typically emphasize speed or throughput, based on average-case behaviors. In contrast, designers of real-time embedded systems emphasize the system's accuracy, predictability, and reliability, all related to the system's worst-case behavior. When a real-time embedded system controls critical equipment, execution time variability is unacceptable. Access times of traditional hardware-managed caches are unpredictable and thus not suitable for embedded systems [16]. However, for the majority of (soft) real-time systems, the performance gains resulting from effective caches should not be ignored. Instead, analyses for bounding access times of cache memories should be explored.

4. Our proposed cache organization

We propose a split data cache organization for embedded systems. This split organization permits us to fine tune the two types of localities (viz., temporal and spatial) exhibited by data. Our evaluation methodology is discussed as a two-step process. We believe that cache splitting is a step in the right direction, since it will play a role in achieving cost/performance goals for embedded systems by exploiting temporal and spatial localities to improve performance, minimizes memory footprint, and lower energy consumption. To prove this claim, we first have performed a comprehensive evaluation of the split data cache, with separate array and scalar caches. We show the advantages of our split cache using the SPEC2000 benchmarks as well as embedded programs of the MiBench suite. Next, we have investigated the integration victim caches and stream buffers to further improve the slit cache design. In order to eliminate the potential data cache pollution caused by prefetching, a small stream buffer is used to supplement the array cache, while victim cache supplements the scalar cache. In our experimental evaluations of the integrated approach for embedded applications, we use benchmarks from the MiBench suite.

4.1. *Evaluation of split data caches*

We have simulated various cache techniques described in Section 2 for comparing our split data cache organization with these alternate organizations. For each of these cache configurations we have measured miss rates, access times and power consumptions.

Use of separate caches is not a new idea. Modern processors rely on split instruction and data caches, at least at the first cache level. However it is not common to see separate data caches based on the types of localities exhibited by different data items. Since not all data items exhibit both spatial and temporal localities, a unified treatment of references makes the data cache inefficient. Generally, caches exploit temporal locality by retaining recently referenced data for a long time, and spatial locality by fetching multiple neighboring words as a cache block. If a data item exhibits no temporal locality, bringing it into the cache is useless. Likewise bringing an entire cache block is needless if no spatial localities are exhibited by data. Thus traditional treatment of cache misses causes unnecessary movement of data among the levels of the memory hierarchy, causing significant interference between unrelated data inside the cache, leading to the removal of potentially useful data, causing cache pollution, higher miss ratios, longer memory access times and higher memory bandwidths.

In order to solve these problems, more recently, several split data cache architectures have been proposed, including Dual cache [10,33], Split Temporal/Spatial(STS) [27], Split Spatial/Non-Spatial cache (SS/NS) [28], array and scalar cache [35], HP-7200 Assist cache [23], Non-Temporal Streaming (NTS) [32] and Minimax cache [36]. In our prior work [29], we have evaluated a split cache architecture that grouped memory accesses as scalar or array references according to their inherent locality and each group has subsequently been mapped to a dedicated cache partition. The "array cache" is a direct mapped cache with larger block sizes to exploit spatial localities more aggressively by prefetching multiple neighboring small blocks on a cache miss. The "scalar cache" is a 2-way set associative cache with smaller block sizes to exploit temporal locality. In this system, since scalar references and streamed (or array) references have no longer negatively affect each other; and since significant numbers of compulsory and conflict misses are avoided, the size of each cache (and the combined cache capacity), as well as the power consumptions have greatly been reduced.

In this paper, we extend our case for split data caches to embedded applications. We use a simulation environment and models for estimating power consumption to compare the split caching with traditional cache designs when executing SPEC 2000 and MiBench benchmarks. We compare our cache designs with a direct mapped cache, a 2-way set associative cache, a cache supported by prefetching and a direct mapped cache augmented by a victim cache. For prefetching, we use a simple prefetching scheme where every miss induces prefetching of the next two blocks. The victim cache is an 8-line fully associative cache (with swapping of cache lines between the primary and victim caches on a miss).

The results illustrate the benefits of split data caches even for embedded systems. While considering only miss rates, for most of the applications split cache performs as well as or better than any of the other methods. Moreover, when we consider the overall performance with access times and power consumption, split cache delivers higher performance (improvements of more than 60%). Details of these comparisons are included in Section 6.1.

4.2. Generalization of split caches with victim cache and stream buffer

In the next set of experiments, we augmented our split cache design with a small victim cache and small stream buffers. Our goal is to demonstrate the applicability of victim caches and data prefetching to embedded systems. We compared our new designs with the alternate cache organizations using only MiBench suite of benchmarks. We compared our cache designs with set-associative caches, hardware prefetching where every miss induced prefetching of the next two blocks and victim cache augmenting a unified data cache. In this section we also compare our organization, which does not use any L-2 cache, with a direct-mapped cache supported by L-2 cache. It should be mentioned that other alternate cache organizations (2-way set associative cache, unified data cache augmented with victim cache or prefetch) did not include any L-2 cache. We compare the designs for average access times, silicon areas and energy consumptions. For all of the benchmarks except one, our split data caches deliver higher performance than any of the alternate organizations (with improvements of up to 69% in access times when compared with that of a direct mapped cache with L-2 cache and 62% reduction in power consumption when compared to a 2-way set associative cache). To our knowledge the benefits of split data caches that use victim caches and/or stream buffers for embedded applications have not been reported in the literature.

5. Experimental methods

In this section we describe the experimental environment used in this study. We also define performance metrics and power models used in our studies.

5.1. Benchmark

We use selected benchmark programs from both the SPEC 2000 [14] and MiBench suite [12], for the first set of comparisons shown in Section 6.1. For the second set of experiments when we augmented array caches with stream buffers and scalar caches with victim caches, we only used MiBench benchmark programs. MiBench includes benchmarks from several representative embedded application domains. For our purpose we included selected programs from these application domains: (1) Automotive and Industrial Control, (2) Office Automation, (3) Networking, (4) Security, (5) Consumer and (6) Telecommunications. The descriptions of the benchmarks used in our studies are listed in Table 1.

Table 1
Descriptions of benchmarks used in the experiment

Benchmark Family		Benchmark name	Description	Name in figure
Mibench	Network Suit	dijkstra	Shortest path problem	dk
Mibench	Telecommunication Suit	rawcaudio	Voice Encoding	rw
Mibench	Automotive Suit	bit counts	Test bit manipulation ability of a processor	bc
Mibench	Automotive Suit	qsort	Sorts a large array of strings using quick sort	qs
Mibench	Office Suit	ispell	Fast spelling checker is	
Mibench	Security Suit	blow fish	Encryption/decription	bf
Mibench	Consumer Suit	cjpeg	Image compression	cj
SPECfp2000		179.art	ImageRecognition/Neural networks	ar
SPECfp2000		188.ammp	Computational Chemistry	am
SPECfp2000		177.mesa	3-D Graphics Library	me

Table 2
Configuration of Memory hierarchies of different cache types

Scalar cache configuration	Size – 512 bytes, Block size – 32bytes, Associativity – Direct mapped
Array cache configuration	Size – 4 k, Block size – 32bytes, Associativity – Direct mapped
Scalar Victim configuration	Size – 256 bytes, Block size – 32bytes, Associativity – Fully associative, Replacement policy – LRU, non swapping
Stream buffer configuration	Total Size – 128 bytes, Block size – 32bytes, Number of Buffer – 2
Direct-mapped Level 1 cache	Size – 8 k, Block size – 32bytes, Associativity – Direct mapped
Direct-mapped Level 2 cache	Size – 32 k, Block size – 32bytes, Associativity – Direct mapped
Victim cache configuration	Main cache Size – 8k, Block size – 32bytes, Associativity – Direct mapped
Victim cache	Size – 256 bytes, Block size – 32bytes, Associativity – Fully associative, Replacement policy – LRU, swapping
Prefetching cache configuration	Size – 8k, Block size – 32bytes, Associativity – Direct mapped Prefetches 2 lines

5.2. Simulation

In order to compare the various cache design alternatives we developed a suite of simulators. We used trace-driven simulations where executables are instrumented using ATOM instrumentation and analysis functions; the traces collected by the instrumented programs are fed to the cache simulators to compute cache hits and misses [8]. We mark traces as array accesses and scalar accesses. For our purpose in this study, we identified array references by assuming that such references involve some form of indexing. Our ATOM analysis program track indexes so that any data item that uses an index will be marked as an array reference. While we cannot assure that all array data items are captured by our method, our analyses for selected sample programs show that most of the array data items (better than 99%) have been correctly identified. In an actual implementation of split caches, compile time analyses can be used to separate array and scalar data references. By using different instructions (e.g., Array_Load and Array_Store, Load and Store) data can be directed to array and scalar caches. A similar approach was used in dataflow architectures. Table 2 lists the various architectural parameters for each cache configuration used in our studies.

5.3. Evaluation framework

In this section we define the four performance metrics used in our comparisons: miss rate, silicon real-estate area, access time, and power consumption.

5.3.1. Miss rate

Miss rate is the percentage of cache accesses that are not found in the cache. Reducing miss rate can lead to improved access times, although miss penalties also affect access times. Energy consumed by an application is also affected by miss rates.

5.3.2. Access time

Memory access time is the average number of cycles required to successfully access a referenced address. We use CACTI [37] using a 0.8 micron technology to compute access times for cache hits. The equations for different cache configurations are included in Table 3. This metric proves useful in evaluating the performance of a cache scheme because, although a particular cache design may demonstrate lower miss rates, the lower miss rates may have been achieved at the expense of the hit access times. For example a cache with higher associativity can have lower miss rates than a direct-mapped cache, but an associative cache will have longer access times.

Table 3
Timing equations used to compare performance

Cache name	Equation to compute the Access Time
Direct-mapped cache with L2 cache	((Level 1 Hit*HAT) + (Level 2 Hit*(Level 1 FTM + (4*HAT))) + (Miss*(Level 1 FTM + Level 2 FTM + OAT)))
Direct-mapped and victim cache	((Hit*HAT) + ((Victim cache Hit*(FTM + Victim cache HAT + Victim cache SW)) + (Miss*(FTM + Victim cache FTM + MSW + OAT)))
Prefetching cache	((Hit * HAT) + (Miss*(FTM + OAT + EWP)))
Array cache with stream buffer	((Hit*HAT) + (Stream buffer Hit*(Array cache FTM + Stream buffer HAT)) + (Miss*(Array cache FTM + Stream buffer FTM + OAT + replacement cost + (2*EWP)[1]
Scalar cache with victim cache	((Hit * HAT) + ((Victim cache Hit*(Scalar cache FTM + Victim cache HAT) + (Miss * (Scalar cache FTM + Victim Cache FTM + MSW + OAT)))[2]

[1]In any miss writing in array cache is accompanied by writing two lines in a Stream Buffer.
[2]In our Victim cache we do not swap lines on hit.

Table 4
Power consumption equations used to compare performance

Cache name	Equation to compute the Power Consumption
Direct-mapped cache with level two cache	((Level 1 Hit*PCR) + (Level 2 Hit*(Level 1 FTM + PCR)) + (Miss * (Level 1 FTM + Level 2 FTM + OPC + PCW)))
Direct-mapped and victim cache	((Hit*PCR) + ((Victim cache Hit*(FTM + Victim cache PCR + Victim cache SW PC)) + (Miss *(FTM + Victim cache FTM + MSW PC+ OPC +PCW)))
Prefetching cache	((Hit * PCR) + (Miss * (FTM + OPC +PCW + EWP PC)))
Array cache with stream buffer	((Hit * PCR) + (Stream buffer Hit*(Array cache FTM + SB PCR)) + (Miss*(Array cache FTM + Stream buffer FTM + OPC+ PCW + (2*EWP PC))))
Scalar cache with victim cache	((Hit * PCR) + ((Victim cache Hit * (Scalar cache FTM + Victim cache PCR) + (Miss*(Scalar cache FTM + Victim cache FTM + OPC + PCW)))

5.3.3. Area consumption

Our performance evaluation also includes silicon area needed for cache systems, because embedded systems designers are interested not only in the performance but also in better use of silicon area. We use CACTI [37] for computing silicon areas need by caches.

5.3.4. Power consumption

We also use energy consumed by an application in our experimental evaluations. We include energy consumed due to misses and off-chip accesses. Our analyses have used the following general equations to compute the dynamic power consumption for a cache.

$$\text{power} = \text{Hit} * \text{power\_hit} + \text{Miss} * \text{power\_miss}$$

$$\text{power\_miss} = OPC + PCW + FTM$$

We obtain values for *hits* and *misses* for the various caches by executing benchmark programs. P*ower_hit is* the power consumed when accessing the cache (computed using CACTI [37]). Different cache structures show different *power_hit* values depending on the cache type, size, and hit type of each access. In Table 4, we describe the equations used to compute *power* for the different cache types. The *PCW is* the power consumed to write an entire line to the cache, which is computed using CACTI [37]. OPC is the power needed for off-chip access and calculated as $0.5 * Vdd^2 * (0.5 * W_{\mathrm{data}} + W_{\mathrm{addr}}) * 20pF$ [19–21,37], where W_{data} and W_{addr} are the number of bits for both data sent/returned and the addresses sent to the next lower level of memory on a miss request. On any miss, FTM includes the overhead for searching a cache. All the terms used in Tables 3 and 4 are defined in Table 5.

6. Results

The next two subsections present the results of our study. In the first section we show the results for our split data cache. The second section shows the results for our split caches that are augmented by a victim cache and stream buffers.

6.1. Evaluation of split data caches

Since our goal is to evaluate the benefits of a split cache, we do not include data that compares the silicon areas needed by the alternate cache organizations. For the study reported in this section we compare our split cache organization (with an array and a scalar cache) with a direct mapped cache, a conventional 2-

Table 5
Definition of terms used in timing and power consumption equations

Abbreviation	Term	Definition
HAT	Hit Access Time	Time to read a cache line in case of hit
FTM	First Time Miss	Time overhead to search in case of miss
OAT	Off-chip Access Time	Time to get data from next memory level
SW	Swap Time	Time to swap line for victim cache hit
MSW	Miss Swap	Time Time to swap line for victim cache miss
EWP	Extra Write for Prefetching	Time to prefetch for stream buffer or prefetching cache
PC	Power Consumption	Power Consumption for any activity
PCR	Power Consumption to Read	Power Consumption to read a cache line in case of hit
PCW	Power Consumption to Write	Power Consumption to write a cache line in case of miss
OPC	Off-chip Power Consumption	Power Consumption to get data from next memory level

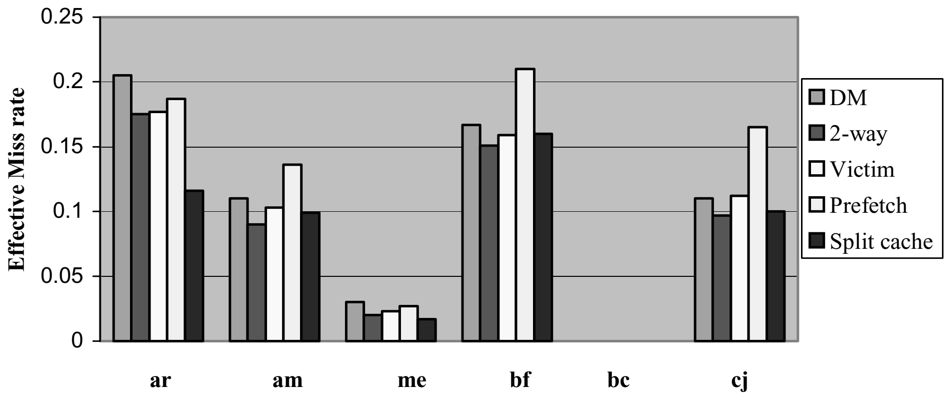

Fig. 1. Comparison of the Miss rate of split data cache with traditional cache architectures.

way set associative cache, a directed mapped cache that uses prefetching and a direct mapped cache that is augmented by a victim cache.

In Fig. 1, weighted effective miss rates for our method are compared with the miss rates of different conventional cache architectures. Since we use two separate caches (an array and a scalar), the effective miss rate is given by,

Effective miss rate $=$ *Array miss rate* * (*Number of Array references/Number of total references*) $+$ *Scalar miss rate* * (*Number of Scalarreferences/Number of total references*)

For the benchmark "bit counts" (bc) although our split data caches have lower miss rates than other cache configurations, the miss rates are for all cache organizations are very small (compared to other benchmarks) to be visible in the figure. In Figs 2 and 3 we compare our split data cache organization with other cache organizations in term of access times and power consumptions, respectively. In these figures we show the percentage improvement resulting from our approach when compared to other organizations. In other words, these figures show the relative advantage of using separate array and scalar cache. In each figure, we also include the average benefits accumulated over all the selected benchmarks. Both figures show that the split data cache organization has led to a significant reduction in access time and power consumption. For example, for benchmark "art" our split cache achieved more than 60% reduction in access time when compared with a direct mapped cache augmented with a victim cache. For benchmark "bf" our split data cache shows more than 60% reduction in power consumed when compared with a 2-way set associative cache. The average across all benchmarks shown in the figures indicate a 23% reduction in access times when compared to direct mapped cache, 21% when compared to 2-way set associative cache, 32% when compared to direct mapped cache with a victim cache and 31% when compared to direct mapped cache with prefetching. Likewise, in terms of power consumption our cache shows reductions of 27% when compared to direct mapped cache, 26% when compared to 2-way set associative cache, 23% when compared to direct mapped cache with a victim cache and 35% when compared to direct mapped cache with prefetching.

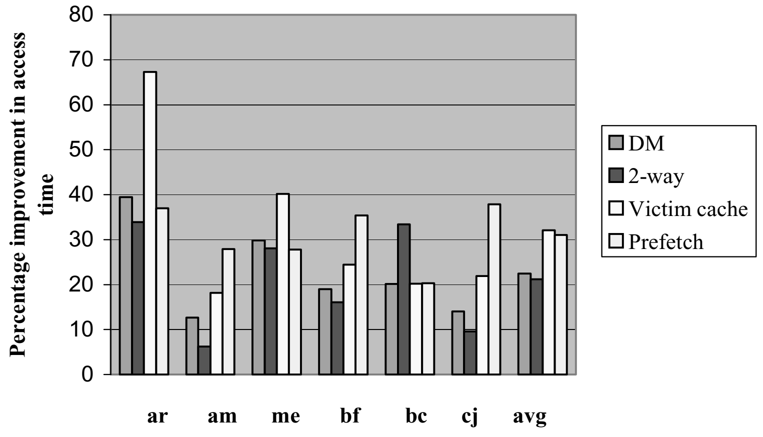

Fig. 2. Percentage improvement in Access time by using the split data cache.

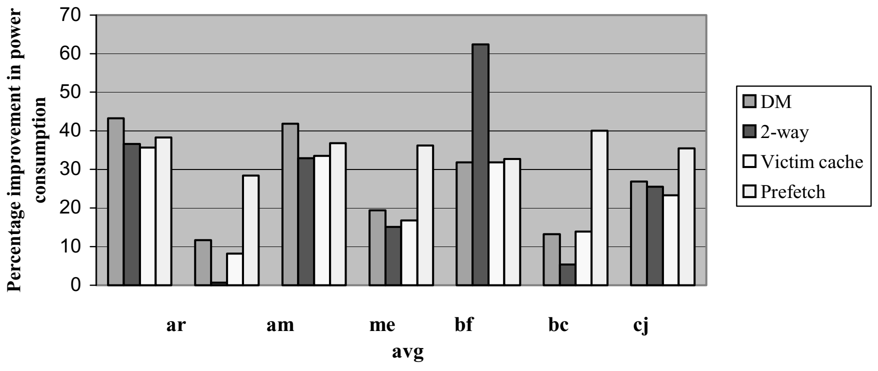

Fig. 3. Percentage improvement in Power Consumption by using the split data cache.

6.2. Generalization of Split caches with victim cache and stream buffer

In this section we present the experimental results of our split caches augmented by a victim cache and stream buffers. Since access time and power consumption are more important than the miss rate for embedded systems (and since we are actually using the miss rate to compute both access time and power consumption), in this section our evaluation compares cache designs for three metrics: cache area, access time, and power consumption.

In Fig. 4, we show the percentage improvement in silicon area (reduction in area) achieved by our cache organization when compared to the area needed by other cache organizations. It should be mentioned that the silicon area required depends only on the cache design and does not depend on the actual application. The Y-axis shows the percentage improvements (i.e., reduction in silicon area) exhibited by our cache design. For example, our cache organization shows an 80% reduction in the area when compared to a unified 8k direct-mapped cache with a 32 k L2 cache; 43% reduction when compared to a unified 8 k 2-way set-associative cache without an L2 cache. It should be noted that our cache organization does not include an L2 cache. Figure 4 clearly shows that our cache architecture consistently requires smaller silicon area when compared to the other cache organizations.

In Fig. 5 we compare our cache organization with other cache organizations in term of access times. In this figure we show the percentage reduction in access times resulting from our cache system when compared to the access times for alternate organizations. Our organization shows faster access times over all other cache designs across all of the benchmarks. For example, for benchmark “qsort” our split cache achieves more than 69% reduction in access times when compared with an 8 k direct mapped cache using 32 k L-2 cache. In this figure we also include the average reductions accumulated across all benchmarks. Our cache design improves in average by 39% when compared

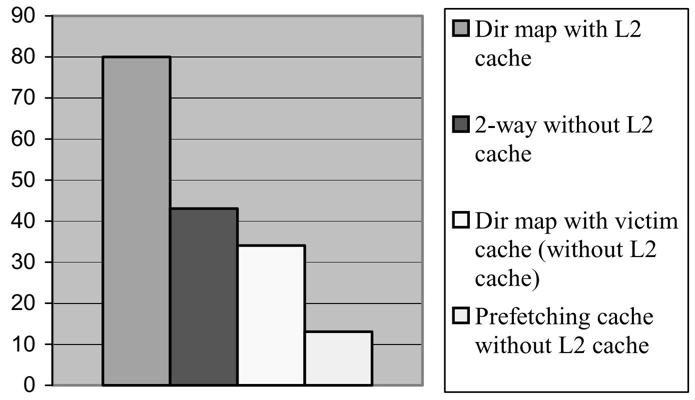

Fig. 4. Percentage improvement in area consumption by using the integrated approach.

to a direct mapped cache with L2 cache, 27% when compared to a 2-way set associative cache, 27% when compared to a direct mapped cache with a victim cache, and 43% when compared to a direct mapped cache with prefetching.

The percentage improvements in power consumption achieved by our design when compared to other cache organizations are plotted in Fig. 6. The figure shows that our split data caches result in a significant energy savings for all benchmarks except "ispell". This benchmark is a spelling checker and contains more scalar data items than array data. As a result, our tiny 512-byte scalar cache is too small for accommodating the needs of the benchmark, leading to more cache misses that must be satisfied by longer accesses paths to main memory, which in turns leads to higher energy consumption. Note that we do not have L-2 cache, thus all L-1 misses must be satisfied by access to main memory. It would have been beneficial if the space used by the array cache were reconfigured to support scalar data for this application. However, this will add to the complexity of the organization, in terms dynamic reconfiguration of the cache.

For streaming benchmarks "rawcaudio" and "bit counts" our approach achieves more than 60% energy reduction when compared with a conventional 8 k 2-way set associative cache configuration. The average reductions over all benchmarks are also included in the figure. The average across the benchmarks show that our cache reduces the power consumption by 25% when compared to a direct mapped cache with a level 2 cache, 24% when compared to a 2-way set associative cache, 21% when compared to a direct mapped cache with a victim cache and 38% when compared to a direct mapped cache with prefetching.

7. Related work

We group the related research into those that are related to the using a single data cache, and those that are related to split data caches.

8. Optimizations of unified data caches

The exploitation of various cache parameters such as associativity and block sizes offer the most common approaches to improve cache performance for desktop systems [34]. Following this approach, Givargis, et al. have explored the effects of cache size, block size and associativity on the cache performance in embedded systems [9]. Another common practice in desktop application includes additional cache hierarchies [4], which is becoming more common for even embedded systems [2,11]. For desktop applications, other cache performance improvement techniques include augmentation of a cache with additional structures and hardware prefetching. Jouppi [18] originally proposed complementing a direct mapped cache with an additional victim cache and stream buffers. Vahid, et al. [40] explore the role of victim caches in embedded systems. They conclude that because of its high energy required, victim caches do not offer a good option for embedded systems. Data prefetching, has long been known to significantly decrease cache miss latency and both hardware and software prefetching approaches have been studied extensively for desktop applications [3, 26]. However, they have not extensively been investigated for embedded systems because designers believe that they add to the embedded processors' energy requirements [13].

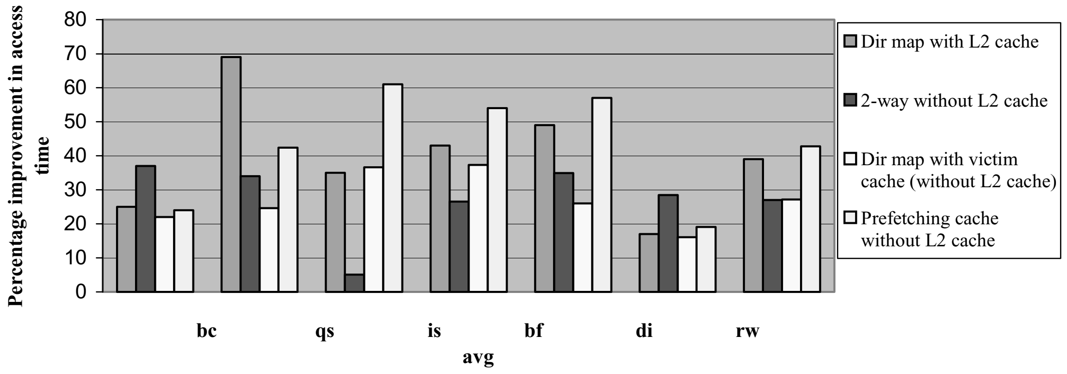

Fig. 5. Percentage improvement in Access time by using the integrated approach.

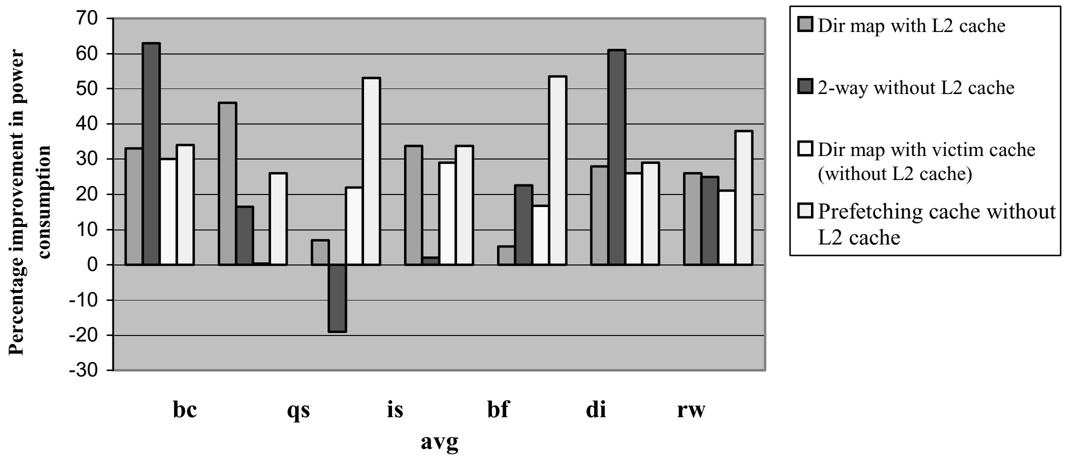

Fig. 6. Percentage improvement in Power Consumption by using the integrated approach.

8.1. Different cache splitting techniques

The use of split caches has been investigated for desktop applications. Gonzalez, et al. [10,33] propose a dual data cache consisting of both a fully associative buffer called a temporal module along with a direct-mapped cache called a spatial module. At compile time, memory references are tagged as bypass, spatial, or temporal, and directed to appropriate modules. Tomasko, et al propose a separate array and a scalar cache and explore the effects of cache parameters [35]. In STS (Split Temporal/Spatial) cache, proposed by Milutinovic, et al. [27] the temporal part is organized as a two-level hierarchy with one-word block size while the spatial part is organized as a one-level cache with four word blocks (128 bit blocks), assisted by a hardware prefetching mechanism. In a later study, Milutinovic, et al. [28] propose a split cache design called the Split Spatial/Non-Spatial cache (SS/NS) which uses a flag-based method for detecting data references to appropriate caches. The NTS (Non-Temporal Streaming) cache proposed by Rivers and Davidson [32] dynamically detects temporal (T) and non-temporal (NT) data references and caches them separately. Their NTS cache system includes a non-temporal detection unit (NTDU) to monitor the reuse behavior of the blocks. There have been other studies of split caches [23–25]. Unsal et al. [36] propose minimax cache which has a 2-way set associative cache for non scalar data while the scalar data is directed to a 512 byte fully associative mini-cache. Intel's StrongARM SA-1110 [2, 15], a low-power processor for embedded system, has a 8k data cache with 32-way set associativity and a 512 byte fully associative mini-data-cache to enhance performance when dealing with temporal references.

Our proposed integrated cache differs from these split caches in three ways. First, our proposed cache is augmented by a victim cache and stream buffers while the others are not. Second, unlike the other reported studies (except minimax cache), we performed analyses not only with miss rates but also on power consumption and access times. Finally, all of these caches except minimax cache (they used only selective multimedia benchmarks) and StrongARM, were investigated for desktop applications and not embedded applications.

9. Conclusion

In this paper we have shown that when carefully designed, embedded systems can benefit significantly from small split caches. Our proposed integrated method permits a systematic trade-off between memory size, power and performance, which has up to now, not been feasible for embedded systems. Our approach can significantly reduce the power consumed as well as the memory size while providing better performance (by reducing execution time) – compared to the same applications executing with conventional direct-mapped or set associative caches. We have shown that by separating data accesses into scalar and array (or stream) references, one can eliminate conflicts between these competing locality types. This, in turn, allows us to reduce total cache size. A smaller (combined) cache leads to smaller footprints and reduced power requirements.

The most significant achievement is the ability to include prefetching and victim caching into embedded systems. As mentioned earlier, because of their high energy requirements, victim caches and prefetching are seldom used in embedded systems. Our experimental data using a unified direct-mapped data cache with a victim cache and a unified direct-mapped data cache with stream buffers do support this assertion. However in this paper we show that a split-data organization with very small scalar and array caches can benefit significantly from victim caches and stream buffers. While traditional prefetching techniques have been explored [3], (premature) prefetching can adversely affect performance if it leads to cache pollution by displacing needed data in an untimely manner. This is the primary reason for not using pre-fetching in embedded systems. However we show that a carefully designed cache system not only solves the deficiencies of general prefetching, it also solves the problems of stream buffers. Jouppi's analysis [18] included a stream buffer for a unified data cache, and the buffer was flushed every time a scalar data is accessed (since stream buffers assume contiguous data items). In our study because we are removing the contaminating scalar data from array caches, stream buffer associated with the array cache are flushed less frequently. Likewise victim caches are not popular in embedded system because fully associative victim cache consumes significant amounts of energy. Again, we show that carefully designed cache systems can benefit from victim caches while maintaining low energy requirements. In a split cache organization, as the array references are removed from the scalar cache, the victim cache only has to deal with fewer scalar references. In our study the reduced costs of using non-swapping victim cache that augments our tiny scalar cache (of 512 bytes) allow us to achieve up to 30% reduction in power consumption when compared with a traditional unified caches with victim caches. Our proposed integrated cache performs better than larger unified caches using additional levels of cache hierarchy (such a large L-2 cache). Ideally, the split cache organization should be dynamically reconfigurable to meet the application requirements. For example for applications that have very little array or stream references, the array cache should be used to supplement the scalar data cache. However, such dynamic reconfigurations require additional hardware. We will investigate the trade-offs and reconfiguration options in our future work.

Acknowledgement

We would like to thank Dr. Mehran Rezaei for helping us to set up some of the experiments.

References

[1] A. Agarwal, J. Hennessy and M. Horowitz, Cache performance of operating systems and multiprogramming, *ACM Transactions on Computer Systems* **6**(4) (Nov. 1988), 39343 1.
[2] ARM, www.arm.com.
[3] J.L. Baer and T.F. Chen, *An effective on–chip preloading scheme to reduce data access penalty*, in Proceedings of the Supercomputing, 1991, 176–186.
[4] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu and S. Dwarkadas, Memory hierarchy reconfiguration for energy and performance in general-purpose processor architecture, *33rd International Symposium on Microarchitecture* (December, 2000).
[5] B. Calder, D. Grunwald and J. Emer, *Predictive sequential associative cache*, in Proceedings of the Second IEEE Symposium on High-Performance Computer Architecture, Feb. 1996, 244–253.
[6] J. H. Change, H. Chao and K. So, *Cache design of a submicron CMOS Systerd370*, in Proceedings of the 14th Annual International Symposium on Computer Architecture, June 1987, 208–213.

[7] S. Cotterell and F. Vahid, *Synthesis of customized loop caches for core-based embedded systems*, International Conference on Computer Aided Design (ICCAD), November 2002, 655–662.

[8] A. Eustance and A. Srivastava, ATOM: A flexible interface for building high performance program analysis tools, *Western Research Laboratory* **TN-44** (1994).

[9] T. Givargis, J. Henkel and F. Vahid, *Interface and cache power exploration for core-based embedded system design*, International Conference on Computer-Aided Design (ICCAD), San Jose, November 1999.

[10] C. Gonzalez, A. Aliagas and M. Valero, *Data cache with multiple caching strategies tuned to different types of locality*, in Proceedings of International Conference on Supercomputing '95, July 1995, 338–347.

[11] A. Gordon-Ross, F. Vahid and N. Dutt, *Automatic tuning of two-level caches to embedded applications*, Design Automation and Test in Europe Conference (DATE), February 2004, 208–213.

[12] M. Guthaus, J. Ringenberg, T. Austin, T. Mudge and R. Brown, *MiBench: A free, commercially representative embedded benchmark suite*, in Proceedings of the IEEE 4th Annual Workshop on Workload Characterization, Austin, TX, December 2001.

[13] J. L. Hennessy and D.A. Patterson, *Computer architecture a quantitative approach*, Morgan Kaufmann Publishers, Third Edition 2003, 423–430.

[14] L. Henning, SPEC CPU2000: Measuring CPU performance in the new millennium, *IEEE Computer* **33**(7) (July 2000), 28–35.

[15] Intel StrongARM SA-1110 Microprocessor Brief Datasheet, April 2000.

[16] B, Jacob, *Cache design for embedded real-time systems*, Embedded Systems Conference, June 30, 1999.

[17] P. Jain, S. Devadas, D.W. Engels and L. Rudolph, *Software-assisted cache replacement mechanisms for embedded systems*, ICCAD, 2001, 119–126.

[18] N.P. Jouppi, *Improving direct-mapped cache performance by the Addition of a small fully associative cache and prefetch buffers*, in Proceedings of the 17th ISCA, May 1990, 364–373.

[19] P. Jung-Wook, K. Cheong-Ghil, L. Jung-Hoon and K. Shin-Dug, *An energy efficient cache memory architecture for embedded systems*, in Proceedings of the 2004 ACM symposium on Applied computing, March 2004.

[20] M.B. Kamble and K. Ghose, *Energy-efficiency of VLSI caches: a comparative study*, in Proceedings of Tenth International Conference on VLSI Design, Jan. 1997, 261–267.

[21] M.B. Kamble and K. Ghosse, *Analytical energy dissipation models for low power caches*, in Proceedings of International Symposium on Low Power Electronics and Design, Aug. 1997, 143–148.

[22] C. Kulkarni, C. Ghez, M. Miranda, F. Catthoor and H.D. Man, Cache conscious data layout organization for conflict miss reduction in embedded multimedia applications, *IEEE Transactions on computers* **54**(1) (January 2005).

[23] G. Kurpanek, K. Chan, J. Zheng, E. DeLano and W. Bryg, PA7200: a PA-RISC processor with integrated high performance MP bus interface, *COMPCON Digest of Papers* (Feb 1994), 375–382.

[24] J.H. Lee, J.S. Lee and S.D. Kim, A new cache architecture based on temporal and spatial locality, *Journal of Systems Architecture* **46** (Sep. 2000), 1451–1467.

[25] J.H. Lee, S.D. Kim and C. Weems, Application adaptive intelligent cache memory system, *ACM Transactions on Embedded Computing Systems* **1**(1) (Dec. 2002), 56–78.

[26] C.K. Luk and T. Mowry, *Compiler based prefetching for recursive data structures*, in Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems, Oct. 1996, 222–233.

[27] V. Milutinovic, M. Tomasevic, B. Markovic and M. Tremblay, *The split temporal/spatial cache: initial performance analysis*, SCIzzL-5, Mar. 1996.

[28] V. Milutinovic, M. Prvulovic, D. Marinov and Z. Dimitrijevic, The splits spatial/non-spatial cache: a performance and complexity evaluation, in Newsletter of Technical Committee on Computer Architecture, *IEEE Computer Society* (July 1999).

[29] A. Naz, K.M. Kavi, P.H. Sweany and M. Rezaei, *A study of separate array and scalar caches*, in Proceedings of the 18th International Symposium on High Performance Computing Systems and Applications (HPCS 2004), Winnipeg, Manitoba, Canada, May 16–19, 2004, 157–164.

[30] A. Naz, M. Rezaei, K. Kavi and P. Sweany, *Improving data cache performance with integrated use of split caches, victim cache and stream buffers*, in Proceedings of the Workshop on Memory performance dealing with applications, systems and architecture (MEDEA-2004), held in conjunction with Parallel Architectures and Compiler Technology (PACT-2004) Conference, Sept. 29–Oct. 3, 2004, Antibes Juan-Les-Pins, France.

[31] P. Ranganathan, S.V. Adve and N.P. Jouppi, Reconfigurable caches and their application to media processing, *in Proceedings of the 27th International symposium on Computer Architecture* (June 2000), 214–224.

[32] J.A. Rivers and E.S. Davidson, *Reducing conflicts in direct-mapped caches with a temporality based design*, in Proceedings of the International Conference on Parallel Processing, August 1996.

[33] F.J. Sanchez, A. Gonzalez and M. Valero, Software management of selective and dual data caches, *IEEE TCCA NEWSLETTERS* (March 97), 3–10.

[34] A.J. Smith, Cache memories, *ACM Computing Surveys* **14** (1982), 473–530.

[35] M. Tomasko, S. Hadjiyiannis and W.A. Najjar, Experimental evaluation of array and scalar caches, *IEEE TCCA Newslatters* (March 97), 11–16.

[36] O.S. Unsal, I. Koren, C.M. Krishna and C.A. Moritz, *The minimax cache: an energy-efficient framework for media processors*, 8th International Symposium on High-Performance Computer Architecture, HPCA8, Cambridge, MA, February 2002, 131–140.

[37] S.J.E. Wilton and N.P. Jouppi, CACTI: an enhanced cache access and cycle time model, *IEEE Journal of Solid-State Circuits* **31**(5) (May 1996), 677–688.

[38] C. Zhang, F. Vahid and W. Najjar, Energy benefits of a configurable line size cache for embedded systems, *IEEE International Symposium on VLSI Design, Tampa, Florida* (February 2003).

[39] C. Zhang, F. Vahid and W. Najjar, A highly configurable cache architecture for embedded systems, *in Proceedings of 30th Annual International Symposium on Computer Architecture* (June 2003), 136–146.

[40] C. Zhang and F. Vahid, *Using a victim buffer in an application-specific memory hierarchy*, Design Automation and Test in Europe Conference (DATE), February 2004, 220–225.