Revisiting Level-0 Caches in Embedded Processors

Nam Duong, Taesu Kim, Dali Zhao and Alexander V. Veidenbaum
Department of Computer Science
University of California, Irvine
{nlduong, tkim15, daliz, alexv}@ics.uci.edu

ABSTRACT

Level-0 (L0) caches have been proposed in the past as an inexpensive way to improve performance and reduce energy consumption in resource-constrained embedded processors. This paper proposes new L0 data cache organizations using the assumption that an L0 hit/miss determination can be completed prior to the L1 access. This is a realistic assumption for very small L0 caches that can nevertheless deliver significant miss rate and/or energy reduction. The key issue for such caches is how and when to move data between the L0 and L1 caches. The first new cache, a *flow cache*, targets a conflict miss reduction in a direct-mapped L1 cache. It offers a simpler hardware design and uses on average 10% less dynamic energy than the victim cache with nearly identical performance. The second new cache, a *hit cache*, reduces the dynamic energy consumption in a set-associative L1 cache by 30% without impacting performance. A variant of this policy reduces the dynamic energy consumption by up to 50%, with 5% performance degradation.

Categories and Subject Descriptors

B.3.2 [**Memory Structures**]: Design Styles—*Cache memories*

General Terms

Design, Performance

Keywords

cache design, level-0 cache, migration policy, conflict misses, dynamic energy

1. INTRODUCTION

The use of a small cache in conjunction with a level-1 data cache (L1 cache) has been proposed [11, 24, 13, 14, 21, 8]. They were used to reduce conflict misses in a direct-mapped L1 cache (the Miss cache, Victim cache [13]) or reduce accesses to the L1 cache (Filter cache [14], line buffers [21, 8], HotSpot cache [24]. These caches are typically very small compared to the L1 cache, often with only a few lines. With the exception of the victim cache, these are L0 caches. Given the small size of L0 caches, this L0/L1 cache hierarchy can benefit from a different management policy. This paper proposes two such policies and evaluates their performance and impact on energy consumption.

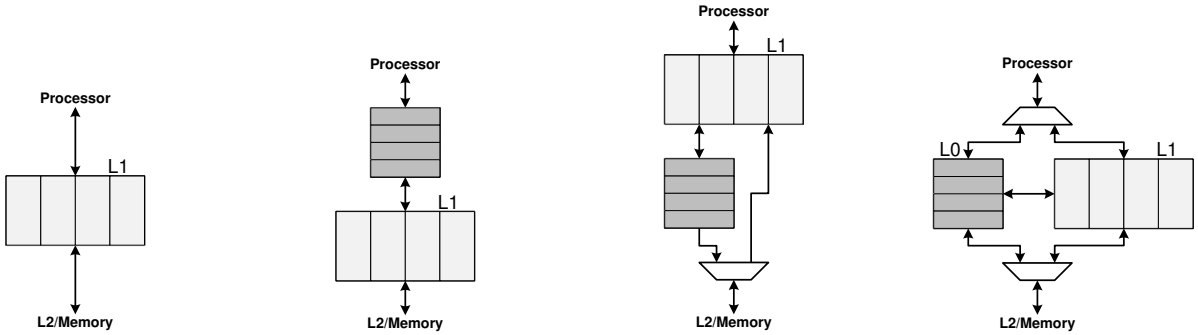
Prior work mostly assumed that the L0 cache is accessed prior to the L1, thus either increasing the L1 access time or requiring a predictor and a recovery mechanism to access the desired cache level directly. This papers assumes that the L0 cache hit/miss determination is known prior to processor access to the L1 cache. This is enabled by decoupling the tag array and data array of the L0 cache and performing tag comparison in the address computation stage of the processor pipeline. This is only possible due to the very small size of the L0 cache and relatively low clock rates.

Fig. 1 shows several possible L0 cache organizations. The baseline L1 cache is shown in Fig. 1a. In Fig. 1b, a small filter cache is placed between the processor and the L1 cache. In Fig. 1c, the victim cache is used to buffer lines evicted from the L1 cache before they are written back to the L2 cache or the memory. Fig. 1d shows a general L0/L1 organization we are are exploring. Here data can be moved between two caches and/or the L2 cache or the memory.

The L0/L1 organization introduces different opportunities and options. The L0 cache can be used to filter accesses to the L1 cache, as in the filter cache. Using the L0 cache, lines can be kept longer avoiding early eviction from L1. This is similar to the victim cache in a direct-mapped L1 cache. In fact, both benefits can be exploited at the same time in the new organization. The management policy needs to manage the different data paths and data movement between the two caches and the memory. The number of concurrent reads/writes in each cache, the number of read/write ports, cache coherence, and modifications to the existing L1 cache should all be considered in designing such policies.

This paper first systematically studies data movement between two caches and the memory. Specifically, it explores a number of *placement* and *migration* policies, which manage data transfer between the L0 and L1 caches, on a hit to either cache or a miss in both. The policies decide in which cache to insert a line on a miss or where a line is moved ("promoted") on a hit. In particular, we investigate 4 insertion policies and 4 promotion policies. As the hit and miss events are independent, their combination results in 16 different policies. However, not all of them are meaningful or useful. The selected policies are empirically analyzed

CASES'12, October 7-12, 2012, Tampere, Finland.



(a) The baseline L1 cache (b) The filter cache (c) The victim cache (d) Our target design

Figure 1: Different cache organizations.

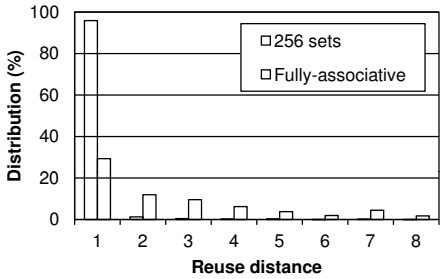


Figure 2: Reuse distance distribution.

and compared to prior policies. While prior designs are effective, they can be further optimized to reduce the energy consumption.

Two new policies/organizations are proposed in this paper. A *flow cache* reduces conflict misses for a direct-mapped L1 cache. Compared to the modified victim cache it uses simpler hardware and less energy. A *hit cache*, is effective in filtering accesses to a set-associative L1 cache.

This paper makes the following contributions:

1. It studies possible migration policies between the L0 and L1 caches.
2. It proposes two new migration policies to enhance operations of the L0/L1 hierarchy.
3. It describes a practical design to support these policies.

The rest of the paper is organized as follows. In Sec. 2, by using distributions of reuse distances, we show design opportunities of a small L0 cache. Our exploration of migration policies is presented in Sec. 3. We also analyze potential applications of each design in this section. Evaluation framework is described in Sec. 4. Sec. 5 includes hardware designs and analysis of selected configurations. Sec. 6 differentiates our study to prior work, and finally Sec. 7 concludes.

2. BACKGROUND

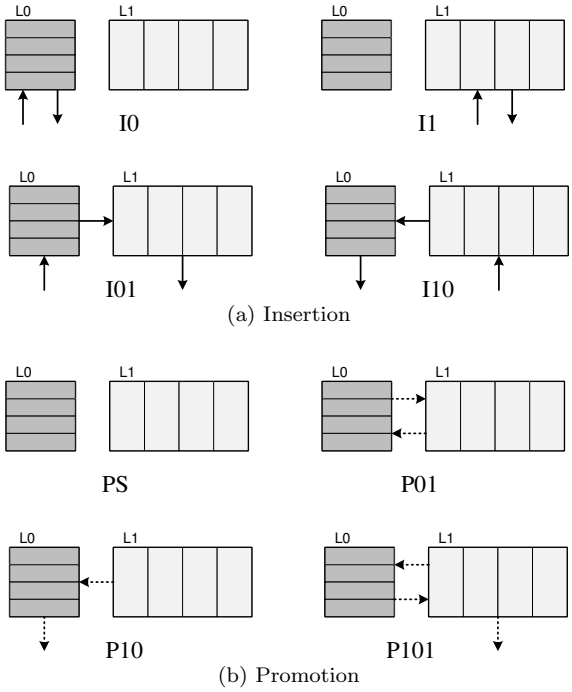
When designing a cache, we can use reuse distance and its distribution as a guide to select the best cache organization [17, 6]. It can predict the hit rate of a cache with LRU replacement algorithm. In this paper, the reuse distance to a cache line is defined as the number of references to the same cache set since its insertion or promotion until it is accessed again. We study the distributions for 2 different cache configurations: a fully associative cache and a 256-set cache to fully consider the effects of different cache mapping functions. The first configuration reflects the behavior of a fully associative L0 cache and the second corresponds to a direct-mapped or set-associative L1 cache. Using a simulator modeling a simple processor with the D-cache and I-cache, we collected memory traces of different benchmarks for each cache configuration for the D-cache. To simplify the calculation of reuse distance, we use a very large cache with very high associativity, so that no cache line will be evicted after its insertion. Distribution histograms are computed from the traces.

Fig. 2 shows the average distributions for different benchmarks using the methodology described in Sec. 4. As we can see, 96% of the accesses to a 256-set cache are immediate reuses. This implies high locality of accesses at the first level cache. L1 access latency is critical to processor performance, and a small increase in the miss rate can impact performance significantly. Therefore, a new L1 design should not increase the miss rate at this level.

Now let us consider the reuse distribution of the fully-associative cache. About 29% of the accesses are immediate reuses, while 41% and 57% of the accesses reuses lines that are 2 and 4 accesses ahead of time respectively. This means a small fully-associative L0 cache with 2 or 4 blocks would be beneficial if we store cache lines with small reuse distance in it. The benefits include filtering accesses to the L1 cache and keeping cache lines longer.

3. MIGRATION POLICY

As shown in Fig. 1d, the introduction of the L0 cache creates a non-trivial design space. There are several dimensions of this space as data can be moved among caches and/or the memory. Examples include which lines should be moved, when and how. This section explores this design space. Recall that a migration policy manages cache line movements between the L0 and the L1 cache. We focus on two independent events which happen during accesses from the processor: a miss in both caches or a hit in either cache. The miss results in an eviction and an insertion while the hit results in a promotion of the hit line. We first explore different individual migration policies during these two events. We do not aim to list all possibilities, but the potential policies only. We also leave the discussion of hardware design and optimizations to a later section. The only big constraint in this section is the L1 cache is set-associative, while the L0 cache is a fully-associative one. This implies that a line from



(a) Insertion

(b) Promotion

Figure 3: Individual migration policies.

the L1 cache can go to anywhere in the L0 cache, while an L0 line can be updated into a fixed set of the L1 cache only. Later in the section, we analyze selected combinations of policies and quantitatively compare them for their potential applications.

3.1 Migration During an Insertion

Fig. 3a shows 4 different migration policies due to a miss. During this event, the missed line is inserted into a cache and replaces another line, and the victim can be either written back to the other cache or other levels. The first two policies, I0 and I1, only insert the missed line into either the L0 or L1 cache, respectively. There is no migration between two caches. Because the missed lines are less likely to be reused than the reused lines [18, 12], by inserting them into a cache only, pollution of the other cache is reduced. In the other two policies, I10 and I01, after the missed line is inserted into one cache, the victim is written to the other cache, and victimizes a line from that cache. In contrast to the two former policies, the purpose of these two policies is to keep lines longer in the cache[1].

3.2 Migration During a Promotion

Fig. 3b shows different promotion policies. We classify them into 3 categories based on how the hit line is promoted and which cache should have higher priority.

Separated. The first and simplest policy is the separated promotion policy, PS in Fig. 3b. Upon a hit to each cache, the hit line is not promoted to the other cache, only state of the corresponding cache is updated.

L0 → L1. In this category, the L1 cache has a higher priority than the L0 cache, where a hit to a line in the L0 cache brings it to the L1 cache and swaps with a line there, while there is no data movement during a hit to the L1 cache. It is the P01 policy in Fig. 3b. The victim cache, which employs the L0 cache as a backup for victims from the L1 cache, uses this mechanism.

L1 → L0. This category emphasizes on the L0 cache, where the hit line in the L1 cache is *copied* to the L0 cache and replaces a line there[2]. A raising question at this point is how the evicted L0 line is treated. A simple solution, the P10 policy in Fig. 4d, is to write it back to other cache levels or the main memory. However, as the contents of lines in L0 may be modified by write hits, this policy must invalidate hit lines in the L1 cache once they are copied to the L0 cache to guarantee correctness.

While this policy is simple, as the L0 cache is very small, replacements happen frequently. Moreover, lines in L0 have short reuse distances as they are reused lines promoted from the L1 cache. This leads to degradation due to the eviction of reused lines in the L0 cache. The policy P101 in Fig. 4b is investigated as a solution for this issue. In this policy, an evicted line from the L0 cache is "saved" by writing back to the L1 cache. The closest study to this approach may be the filter cache [14] and the line buffers [21]. Later the multi-line buffer work [8] uses the L0 cache as a write-through cache, avoids this problem by writing back lines to the L1 cache which are modified during a write hit to the L0 cache.

3.3 Combined Migration Policies

In the previous sections we have described individually 4 insertion policies and 4 promotion policies. There are a total of 16 combinations. However, not all of them are meaningful or useful. In this section we analyze a selected number of policies and their names, as shown in Fig. 4. In the figure, each migration policy is shown with their insertion and promotion policies. For example, I10P01 is the policy of the insertion policy I10 and promotion policy P01. The baseline caches are shown in Fig. 4a. I1PS is the configuration with the L1 cache only, and I0PS corresponds to only L0 cache.

The combined migration policies described in this section are independent of replacement policy. A replacement policy changes the state of lines within a set of L1 or within L0, and is orthogonal to the migration policy. In our design, replacement state of a line is updated based on if a line is inserted to a cache or promoted within the cache, regardless if it is a hit to the D-cache. For example, a hit line from L1 being promoted to L0 is treated as an inserted line whereas it is a hit to L1. In this paper, we assume the LRU policy for both L0 and L1 caches.

This section also compares these combined policies in terms of hits and misses for both the D-cache using the simulation framework described in Sec. 4. For the purpose of comparison, the D-cache consists of a 4-way L0 and an 8KB direct-mapped L1 cache. Both L0 and L1 caches have line size of 32B and use LRU replacement policy. Fig. 5 compares them for two aspects. The first is the misses per kilo instructions (MPKIs) of the D-cache normalized to the baseline cache with L1 cache only (I1PS). The second is the hit rates to the L0 cache of different policies, including the L0 cache only configuration (I0PS), are shown. Here the hit rate is computed by the number of hits to L0 divided by

[1]Another possibility is inserting a missed line into both caches, as in the miss cache [13]. That work showed that the miss cache is less effective than the victim cache.

[2]Due to the difference in organizations of two caches, two lines can not be swapped as in the previous category.

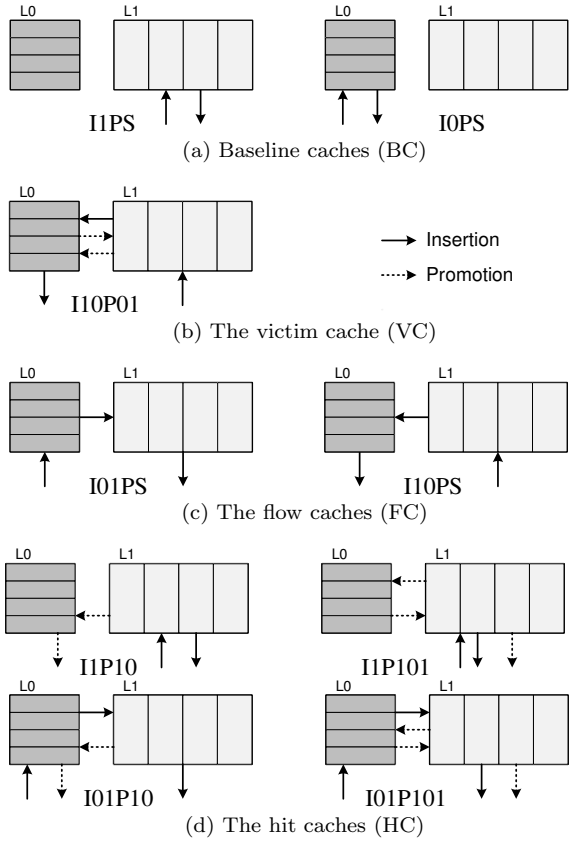


Figure 4: Selected migration policies.

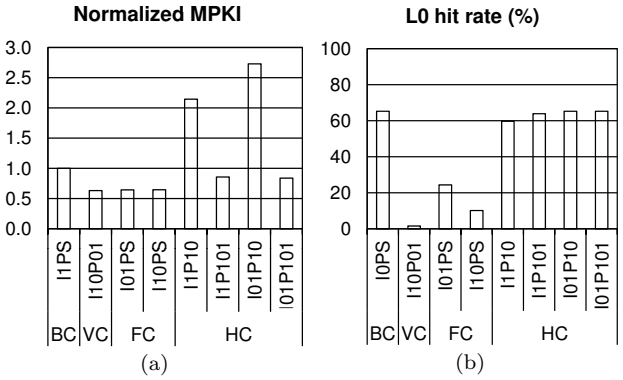


Figure 5: Comparing migration policies (See Fig. 4 for the abbreviations).

total number of *accesses to the D-cache*. This is also the bypass rate to the L1 cache.

3.3.1 The Victim Cache (VC)

Fig. 4b shows the policy I10P01 which is the configuration of the victim cache. In this policy the L0 cache is used as a backup cache for the L1 cache. A line evicted from L1 is stored in L0 for a few more accesses before is really evicted. In the case of a hit to L0, the hit line is swapped with a line in the L1 cache.

Fig. 5 shows that the victim cache is effective in reducing conflict misses for the direct-mapped L1 cache. Compared to the baseline L1, the victim cache removes up to 37% misses to the D-cache. This was also observed in the original work of the victim cache.

3.3.2 The Flow Caches (FC)

The flow caches are shown in Fig. 4c. Caches of this type have that name because a line inserted to the D-cache makes its way from a cache to another cache before is evicted to the main memory or other cache levels, hence they are kept longer before being evicted. Similar to the victim cache, the flow caches have potential application in a direct-mapped L1 cache. However, hardware design of the flow caches are simpler with no migration during a promotion. There are two caches of this type. In the first one, I01PS, missed lines are inserted into the L0 cache first, while in the other, I10PS, missed lines are inserted into the L1 cache first.

Now, let us compare them to the victim cache. In Fig. 5a, two flow caches have similar performance to the victim cache for the D-cache. From the figure, the difference between their normalized MPKIs is negligible. Let us compare the victim cache and the I10PS configuration. Recall that their insertion policies are the same, and the difference is that the victim cache swaps lines between caches on an L0 hit, while the I10PS policy does not. This implies that a line in the L0 cache can provide as many hits as the ones in the L1 cache, and swapping them does not make much impact. This also explains a similar hit rate between I10PS and I01PS. Note that the victim cache is proposed for the high performance domain, and a simpler design as the flow cache can be achieved for the embedded domain without negatively impacting performance.

Besides, the L0 cache also has potential in reducing accesses to the L1 cache in the D-cache, as seen in Fig. 5b. In the victim cache, the L0 cache has a low hit rate, with only 1.6%. Meanwhile, these numbers for two flow caches are 24% and 10%, respectively. These numbers are the results of the fact that in the first configuration, I01PS, missed lines are inserted into the L0 cache first, and in the second one, I10PS, the data evicted from the L1 to the L0 cache stay in the L0 cache before being evicted. Because accessing the L0 cache requires less dynamic energy than the L1 cache, the flow caches have higher potential of reducing energy consumption.

As a result, a simple hardware design combined with energy saving opportunities makes the flow caches attractive in a D-cache with a direct-mapped L1 cache. Between two flow caches, I01PS has higher L0 hits compared to I10PS, thus we decide to use I01PS in design a flow cache.

3.3.3 The Hit Caches (HC)

As analyzed in Sec. 2, the L0 cache can be used to filter accesses to the L1 cache. This is done by storing high locality lines in this cache. One class of such line is the reused lines. First, missed lines are inserted into the L1 cache and upon a hit they are promoted to the L0 cache. For this purpose, the L0 cache is called the hit cache, as illustrated in the first two configurations, I1P10 and I1P101 in Fig. 4d. The different between the two policies is that in I1P10, the evicted line from the L0 cache is written back to the memory while in I1P101, it is written back to the L1 cache. For comparison purpose, we also include two other policies, I01P10 and I01P101, which are similar to the described ones, except that a missed line is inserted into the L0 cache first. Note

Cache	Migration policy			Potential application	
	Name	Insertion	Promotion	Target L1 cache	Description
Flow cache (FC)	I01PS	L0 → L1	Separated	Direct-mapped	Reducing conflict misses
Hit cache (HC)	I1P101	L1 only	L1 → L0 → L1	Set-associative	Reducing energy consumption

Table 1: Summary of selected migration policies.

that I01P101 is very similar to the filter cache [14] or block buffering work [21].

Performance of different hit caches are compared at the last 4 bars in Fig. 5 for the D-cache. Due to the fact that I1P10 and I01P10 evict lines from the L0 cache while invalidating lines in the L1 cache, frequent L0 replacements result in overall degradation. I01P10 even has higher misses than I1P10.

This issue is not seen in the other two policies, I1P101 and I01P101. As described previously, they save the evicted lines from L0 by writing them back to the L1 cache. In fact, they even have less misses compared to the baseline L1 – I1P101 removes 14% misses while I01P101 removes 16%. So, where can they be applied? Let us consider the L0 hit rates (Fig. 5b). All 4 policies have high hit rates to the L0 cache, with at least 60%, and are similar to the baseline L0 cache only. This means that more than half of the accesses to the L1 cache are filtered, and I1P101 and I01P101 can be used to filter accesses to the L1 cache while not impacting performance. Compared to I01P101 which is similar to the filter cache, I1P101 will be more attractive because it has simpler hardware design. It can be concluded that the I1P101 policy is potential in filtering accesses to the L1 cache in the D-cache, especially when the L1 cache is large or has high associativities.

3.3.4 Summary

In this section, we have analyzed and compared different migration policies, and found two configurations which have potential enhancing L1 cache operations. Table 1 summaries their descriptions. Depending on the migration policy, each configuration has its own application for a specific target cache. The flow cache targets a direct-mapped data cache to reduce conflict misses. The hit caches are used to reduce energy due to the accesses to the L1 cache. We will describe in detail their designs and compare to prior designs in a later section.

3.4 Multi-core Environment

Until this point we have described different migration policies for the L0 cache. For the migration policies which do not have inclusiveness, this is achieved by also moving the state of a line state during the migration. In the other case, as in Fig. 4d, line state is also updated along with the update of a line's content. This helps make the design transparent to a multi-core configuration.

4. SIMULATION FRAMEWORK

We evaluate the parallel L0 cache and migration policies using the gem5 simulator [5]. For the purpose of comparing hit and miss rates of an in-order embedded processor, we model a simple single-stage in-order processor. The system has split D-cache and I-cache. There are no other levels of caches, but these two caches are connected directly to the main memory. Various configurations are described in Table 2.

Parameter	Configuration
Processor (in-order)	
Pipeline Depth	1
Issue Width	1 instruction/cycle
Width	32 bits
L1 cache (set-associative)	
Line size	32B
Cache size	4KB, 8KB, 16KB
Associativity	1 way, 2 ways, 4 ways
Interface ports	1 read, 1 write
L0 cache (fully-associative)	
Line size	32B
Associativity	2 ways, 4 ways, 8 ways
Interface ports	1 read, 1 write

Table 2: Processor configuration.

	Tag	Data	Total
64B (2-way)	1.12	1.77	2.89
128B (4-way)	1.92	2.33	4.25
256B (8-way)	3.55	3.46	7.01

(a) L0 cache

	1 way	2 ways	4 ways
4KB	4.14	7.24	12.46
8KB	5.17	10.83	19.44
16KB	7.36	18.90	27.33

(b) L1 cache

Table 3: Energy consumption (pJ) of an access to the L0 and L1 cache.

We study our design using benchmarks from the MiBench suite [10] (basicmath, qsort, susan, jpeg, lame, tiff, typeset, dijkstra, patricia, ghostscript, rsynch, stringsearch, blowfish, sha, adpcm, CRC32, FFT), the CommBench suite [23] (cast, drr, jpeg, reed, zip) and the MediaBench suite [15] (adpcm, epic, g721, gsm, jpeg, mpeg2). Some benchmarks are excluded due to compilation or runtime errors. Each benchmark was complied using a cross compiler for Alpha architecture and is run for 200M instructions or until completion. For each benchmark, all possible inputs are used. The resulting total number of executions is 46. We do not report results for individual benchmarks but their averages.

We use Cacti [22] to estimate energy consumption of accesses (read or write) to the L0 and L1 caches using the 65nm technology, as shown in Table 3. We do not use newer Cacti versions as they are not aimed to support small caches. For the L0 cache, because the tag and data arrays are decoupled, when checking for L0 hit, only the tags are accessed. The data array of the L0 cache only consumes energy during the real accesses to it. In Table 3a, energy for three L0 sizes are shown. Table 3b shows the energy consumption of the L1

cache with different sizes and associativities. The tag and data arrays of the L1 cache are accessed in parallel.

We use the following model to estimate energy consumption. Let us denote H_C and A_C as the number of hits and accesses in the cache C, where C is *L0*, *L1* or D (D-cache); and E_{L0T}, E_{L0D} and E_{L1} as the energy of one access to the L0 tag, L0 data and L1, respectively (Table 3). We have the following observations. First, the L0 tag array is accessed on every reference from the processor. Second, an L0 hit does not access the L1 cache and only the L0 data array is accessed. Third, an L0 miss results in an access to the L1 cache. Finally, let M_{L0} and M_{L1} denote the number of L0 and L1 accesses due to the migration between two caches. Energy accessing the D-cache is computed as:

$$E = A_D * E_{L0T} + H_{L0} * E_{L0D} + (A_D - H_{L0}) * E_{L1} + M_{L0} * (E_{L0T} + E_{L0D}) + M_{L1} * E_{L1}$$

Dividing by A_D we have the average energy of an access:

$$E_A = E_{L0T} + HR_{L0} * E_{L0D} + (1 - HR_{L0}) * E_{L1} + WB_{L0} * (E_{L0T} + E_{L0D}) + WB_{L1} * E_{L1}$$

Where HR_{L0} is the L0 hit rate; WB_{L0} and WB_{L1} are the fractions of writebacks to the destination L0 or L1 cache from the other cache, respectively. These factors are normalized to the number of D-cache accesses. In this model we do not include energy consumption of the control logic as it is negligible.

5. CACHE DESIGN AND APPLICATIONS

In this section we describe in detail hardware designs of each migration policy. This includes the interface between the processor and the caches, and the hardware design of each migration policy.

5.1 Processor-Cache Interface

In this section, we describe hardware design for the interface between the processor and the caches. This includes the accesses from the processor and the writeback of read data to the processor. For the first aspect, prior studies have shown that the addition of a L0 cache in between the L1 cache and the processor can degrade performance significantly if the hit rate at L0 is not enough to compensate the increased distance between L1 and the processor [14]. Various solutions were proposed to avoid such negative impact. In [24], mechanism to decide access to L0 or L1 at the instruction cache is integrated into the BTB. The block buffering work [21] implemented a write-through L0 cache, and decision to access L0 or L1 is known after the set IDs are decoded. In [16], the decision of which way to access is done in parallel with the access to the LSQ.

Our design is based on three observations. First, the number of L0 line tags to compare is very small and they are known early. Second, we target embedded domain, where the high-end processors are clocked at around 1GHz. And third, we observe that the address generation unit (AGU) is often assigned a stage as in ARM processors [1]. This stage takes place after the register read and before the memory access[3]. The AGU is just shift/addition of a base address with an offset, and is fast. The data cache organization with L0 and L1 caches is shown in Fig. 6.

[3]In ARM Cortex-M3 [2], while the AGU does not take a whole stage, its clock speed is low, around 100MHz [20]).

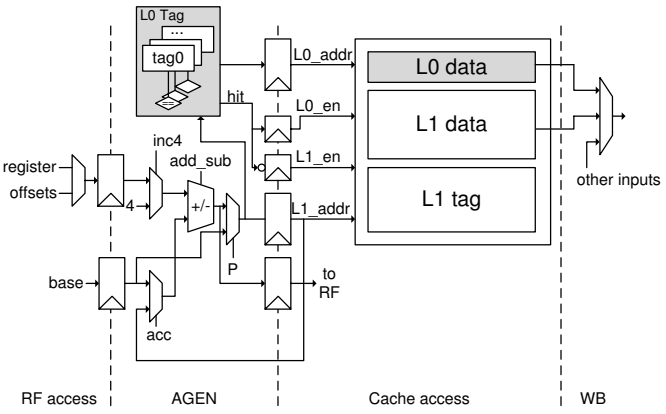


Figure 6: Processor-cache interface.

In this figure, the target architecture is the ARM11 processor [1]. Note that the cache access takes one or more cycles depending on the implementation. All memory instructions such as coprocessor load/store or load/store instructions use this AGU logic. In our design, the output of generated address is compared with all tags stored in L0 tag memory. The AGU was modeled using Verilog-HDL and synthesized using the TSMC 65nm TC library. The maximum clock speed for this block is 1GHz with 13% timing margin for place and route. This can be further reduced by overlapping the AGU and the tag comparison.

The outputs from the L0 and L1 caches are multiplexed with other sources, such as from coprocessors, at the writeback stage.

5.2 The Flow Cache – Reducing Conflict Misses

5.2.1 *Hardware Design*

Recall that the flow cache inserts a missed line into the L0 cache first, and upon eviction the line is migrated into the L1 cache. On a hit to either cache, no data movement happens, but the state of the cache which is hit is updated. In a conventional design, upon a miss to the L1 cache, a line is evicted and the missed line is inserted through the refill path. This can happen in one or more cycles depending on line size and bus width. With the introduction of the L0 cache, victim selection at the L0 and L1 cache are done in parallel, and the migration from L0 to L1 happens in parallel with the refill to L0 and writeback from L1. This implies that the L0 cache does not create any extra latency due to data migration, and the simplicity of hardware design makes it suitable in a low-end embedded processor.

5.2.2 *Analysis*

Fig. 7 compares different techniques to reduce conflict misses in different direct-mapped L1 caches. The techniques include increasing associativity, using modified victim caches and using flow caches. In the figure, three metrics are compared: misses per kilo instruction (MPKI) of the D-cache, the L0 hit rate of the victim caches and the flow caches, and energy consumption. The first and the last metrics are normalized to the baseline direct-mapped L1 cache. The second is the real hit rate computed by the number of L0 hits divided by the number of D-cache accesses. Note that this is also the L1 bypass rate.

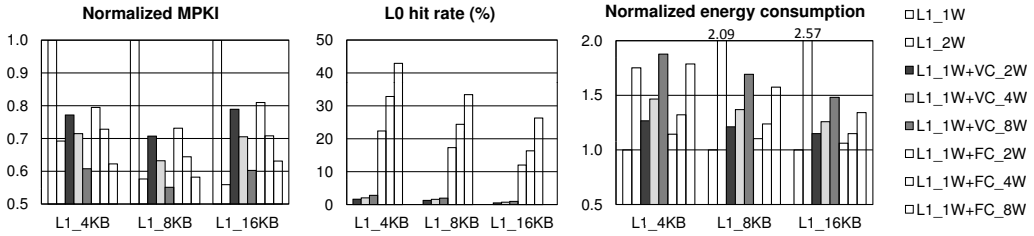


Figure 7: Comparing techniques to reduce conflict misses (VC – victim cache, FC – flow cache).

First, increasing the associativity is known to be effective in reducing conflict misses. For example a 2-way 16KB L1 cache has around 40% fewer misses compared to the direct-mapped L1 cache of the same size. However, the performance improvement results in the increase of dynamic energy consumption. Doubling the associativity increases the dynamic energy by 1.7 to 2.5 times, meaning that increasing the associativity is not always a good choice.

Second, the victim caches are also effective in reducing conflict misses. A 2-way victim cache can remove around 20% while an 8-way one around 40%. Using victim cache requires less energy consumption compared to increasing associativity. For example, a 2-way victim cache increases the dynamic energy by 15% to 28% compared to the baseline. The numbers are higher for a 4- or 8-way victim cache. The reason of low energy consumption is a small L0 cache, which consumes much less energy compared to a large L1 cache.

While the victim cache has been more effective than increasing the associativity, our results also show that using a flow cache is a competitive choice to the two other methods due to two reasons. The first is performance. The flow cache, with simpler hardware design, achieves almost as much miss reduction as the victim cache of the same size. The second reason is dynamic energy consumption. Results show that a flow cache consumes less energy compared to a same-size victim cache. For a 4KB L1 cache, a 2-way L0 cache implemented as a flow cache consumes 14% more energy while 20% misses are removed, while a same size victim cache consumes 27% energy to remove a similar number of misses. Energy reduction is even more effective when the cache size is increased. A 2-way L0 cache increases only 6% more energy for a 16KB L1 cache to remove 20% misses. In general, the flow cache further optimizes energy consumption while still achieves similar performance. Note that the benefit does not simply come from having more data arrays. Our results (not shown here) confirm that a 4KB direct-mapped L1 plus an 8-way L0 perform as well as a 8KB direct-mapped L1 cache, while the size is only 53%.

To understand the effectiveness of the flow cache, let us compare the hit rates of the L0 cache. Fig. 7 also shows the L0 hit rates of different victim caches and flow caches. It is shown that the L0 cache used as a victim cache has a hit rate of less than 3% for all configurations, while for the flow caches it is much higher, from 10% to 40%. The L0 hit rate in the flow caches is high in a small L1 cache. By filtering accesses to the L1 cache, energy consumption is smaller than the victim cache.

In summary, the effectiveness in reducing conflict misses combined with simple hardware design and low energy consumption makes the flow cache attractive in enhancing operations of a direct-mapped L1 cache, especially in a low-end embedded processor.

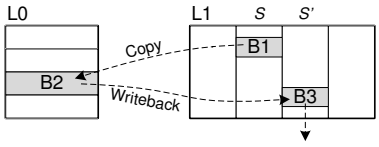


Figure 8: Migration during hit to the L1 cache in the hit cache.

5.3 The Hit Cache – Reducing Energy Consumption

5.3.1 Hardware Design

As shown in the previous section, the hit cache is used to reduce accesses to the L1 cache, hence is effective in reducing energy consumption for a set-associative cache. Unlike the filter cache, on a miss to both caches, it inserts a missed line into the L1 cache only. The L1 cache is bypassed on a hit to the L0 cache. During a hit to the L1 cache, the hit line is promoted to the L0 cache and replaces a line there. The evicted line is written back to the L1 cache, as illustrated in Fig. 8. In this figure, an access results in a hit to line B1 of a set S in the L1 cache, and victim selection finds a victim B2 in the L0 cache. B2 is written back to the L1 cache and replaces a line B3 in a set S' there. B1 is copied to the L0 cache in the place of B2. Now B1 and B2 become identical. It can be observed that if the access to B1 is a read, then the writeback of B3 can not be happen in the same cycle with reading B1 because we only allow 1 read and/or 1 write at any cycle. In this section, we describe a hardware implementation to support this policy. The hardware design must satisfy that the number of accesses to the L1 cache is minimized to save energy, while performance is not significantly impacted.

In [21, 8] the block buffers are used to store a cache line which are likely to be reused. This is enabled by placing the buffers close to the data array of the L1 cache, before the multiplexing to choose desired word. We also implement a similar mechanism to enable fast migration between the L0 and L1 cache, by moving the L0 data array close to the L1 data array, as depicted in Fig. 6. Note that the caches are also close to the AGEN unit, therefore while the tag and data arrays of the L0 cache are decoupled, the distance is short, allowing fast updates of the L0 cache. An alternative is to use the latches for buffering lines when migrating data between two caches. This is similar to the write buffers which are popular in multi-cycle pipelined caches.

We investigate two hardware implementations for the hit

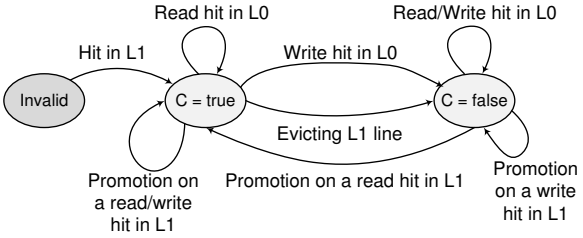


Figure 9: State transition of L0 lines.

cache, depending on when and how to write B2 back to the L1 cache, the *eager* and the *lazy* mechanisms, described as follows.

Eager update. In this design, each valid line in the L0 cache always has a copy in the L1 cache, and a write hit into the L0 cache also updates line content in the L1 cache. There is no need to search for the victim B3 and a line B2 when evicted from the L0 cache is simply discarded. Also, the miss rate of this implementation does not change compared to the baseline L1 cache. In this design, the L0 cache operates as a write-through cache, and an eviction of the L1 line also invalidates its L0 copy, if existing. Note that the works in [14, 21, 8] also use this approach. Because this policy updates the L1 copy content of B2 every write hit, we call it the eager design.

Lazy update. An alternative design is the lazy design, which delays the update until B2 is evicted. Compared to the eager design, this design has more potential to reduce accesses to the L1 cache, especially when B2 is written multiple times before it is evicted from the L0 cache. In this design, it is not required to force B2 to have a copy. Instead, when the B2 line is modified, its copy in the L1 cache is *invalidated*, by updating the corresponding valid bit. B2 can only be written back to the L1 cache if and only if B3 is an invalid line. We also allow the replacement of the L1 copy without invalidating the L0 line as in the write-through cache.

To support this mechanism, each L0 cache line is instrumented with an extra *copy* bit, the **C** bit, to indicate whether it has a valid copy in the L1 cache. The state transition is shown in Fig. 9. When a line is hit in the L1 cache, it is copied to the L0 cache, and the **C** bit is set. This bit is unset if the L0 line is modified due to a write hit or the L1 copy is evicted (which does not happen frequently). Note that in a write hit the L1 line is invalidated. The state does not change if the access is a read hit in the L0 cache. When the line is evicted due to a promotion from the L1 cache, the **C** bit is updated for the *new* line and cache migration is performed. The migration is done with the awareness of hardware constraints. If the previous line has **C** set, then it is simply discarded and the new line has a copy in both caches (**C** is set). If the evicted line has **C** unset, a read hit in the L1 cache does the writeback of the line to the L1 cache, and the new line is read from the L1 cache and written into the L0 cache. In this case, there is 1 read and 1 write happening in each cache. Otherwise, if it is a write hit in the L1 cache, there are two writes happen in the L1 cache, one is the write access and the other is the writeback to the L1 cache. Due to the constraints, only the writeback to the L1 cache is done. The hit line is read from the L1, modified and written into the L0 cache without having a copy in the L1 cache. In the figure, the promotion due to a write in the L1 cache results in the unset **C** bit.

Algorithm 1 Promotion during a hit to B1 in L1.
Note – *set(Bi)*: set ID in L1 of Bi, Bi.C: bit **C** of Bi.

```
1: Find victim B2 in L0
2: if B2 is invalid then
3:    Copy (B1 → B2)
4: else if B2.C = true then
5:    Discard B2
6:    Copy (B1 → B2)
7: else if set(B1) = set(B2) then
8:    Swap B1 and B2
9: else
10:   Find victim B3 from set(B2)
11:   if B3 is invalid then
12:      Move B2 to the position of B3
13:   else
14:      Write back B2 to main memory
15:   end if
16:   Copy (B1 → B2)
17: end if
```

In Fig. 9 we have shown the state transition, including a promotion from the L1 cache. Let us describe how the migration is performed in this case. Using the notations in Fig. 8, the algorithm is shown in Alg. 1. There are 3 possibilities after B2 is already valid. First, if B2 has an L1 copy (bit **C** is set), it is simply discarded and B1 is copied to B2 (in the case of a write, data is written into both lines). Second, if B1 and B2 are in the same set, they are simply swapped. In these first two cases, because at most 1 read and 1 write happen to each cache, the read and write can be done in one cycle. In the last case, because L1 has only one read port then there is no available port to write back B3, hence B2 is written back to L1 only if B3 is invalid, and is written back to the main memory otherwise. This might degrade performance because of the discarded B2. Fortunately, the case of discarding B2 because B3 is valid is very rare.

Now, we describe hardware support for the case where B2 must be written back to the memory (other cases are similar or simpler). With the target L1 as a set-associative cache, then process is as follow:

Step 1: Calculate set S of L1 to access; find victim B2 from L0.

Step 2: Calculate way ID in S to access data (block B1); start writing back B2 to the memory.

Step 3: Copy hit data from B1 to B2; if the access is a write then update the content of B1 and B2.

In the case where the L1 lines are invalidated due to different reasons, such as due to coherence protocols, the invalid lines can be exploited to minimize writing back B2 to the memory. It can also be observed that this lazy mechanism lays between two promotion policies, P10 and P101. Recall that P10 never saves the evicted line in L0 while P101 always does, our real design saves lines which have a copy or can be written back to the L1 cache given the hardware constraints.

5.3.2 Analysis

In this section we analyze and compare two design described in the previous section. Fig. 10 shows the operations of the hit cache, including the hit rate in the L0 cache, misses per kilo instructions (MPKI) of the D-cache, fraction of writeback from L0 to L1, and energy consumption. The

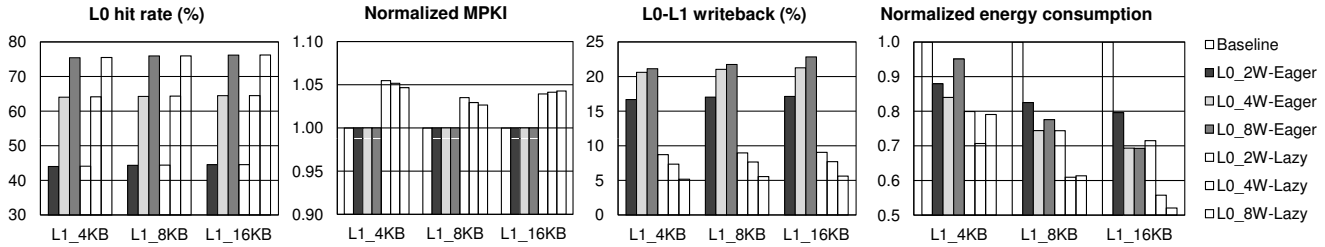


Figure 10: Hit cache operations.

L0 hit rate and the writeback fraction are computed by dividing the number of L0 hits and writebacks by the total number of accesses to the D-cache, respectively. MPKI and energy consumption are normalized to the baseline L1 cache. For each eager and lazy design, we show the results for the L0 cache of 2, 4 and 8 ways, and the 4-way L1 cache of 4KB, 8KB, and 16KB.

First, the L0 hit rate is very similar with different L1 sizes and designs, and is similar to that in Fig. 5, with the difference less than 1%. For a 2-way L0 cache, the hit rate is about 45% while in a 8-way L0 cache it is around 75%. This confirms the importance of the L0 cache in reducing accesses to the L1 cache. Second, the MPKI of the eager design is the same with the baseline L1 design, while that of the lazy design is around 5% higher. It is interesting that for the L1 size of 4BK and 8KB, the misses decrease when the associativity of the L0 increases, while they increase for the 16KB L1 cache. Third, let us compare the fraction of L0-L1 writebacks. In general, with the same L0 and L1 sizes, the lazy design offers around 2 to 3 times less writebacks than the eager design. This confirms that a line can be updated multiple times while in the L0 cache.

For the eager design, a high L0 associativity creates more updates to the L1 cache, while in the lazy design, it saves more writebacks. For example, an 8-way L0 has around 20% writebacks with eager design, while it is only 5% with the lazy design.

Finally, let us analyze the saving of energy consumption. It is obvious that high L0 hit rate and low writeback fraction combined help save more energy due to less accesses to the L1 cache, especially when the L1 access energy is high. Due to this fact both the eager and lazy designs see the best energy savings with a 4-way L0 cache, but not a 2-way nor 8-way one with the L1 cache of 4KB and 8KB. For the 16KB L1 cache, the 8-way L0 cache saves more compared to the 4-way L0 cache. Compared to the eager design, the lazy design saves more energy, with around 10% more. For the target 16KB L1, the 8-way L0 energy saving of the eager design is30% while that of the lazy design is up to 50%.

In summary, the eager design saves about 30% access energy without impacting performance, while the lazy design saves 50% but with 5% more misses.

6. RELATED WORK

In the previous sections we have compared our study with several prior studies. This section summarizes them as well as discusses more related work. In the best or our knowledge, our work is the first to study data migration in a systematic way. From the exploration we find new opportunities to optimize the L0 cache design. Compared to the victim cache [13], the flow cache design is simpler and uses the L0 cache to filter accesses to the L1 cache, hence achieve similar conflict miss reduction, while requires less energy. We achieve our goals due to the victim cache targets high performance processors, while ours is the embedded processors. The hit cache may be closest to the filter cache [14] and the block buffering work [8, 21]. In [14], the filter cache is placed in between the processor and the L1 cache. In fact the filter cache can be considered another version of the buffering blocks in [21]. In these works, the L0 hit rate must be high enough to compensate the increased latency between the processor and the L1 cache. Later the work in [8] designs the L0 cache as a write-through cache, and it do not impact the system's performance. In our design, we further study the tradeoffs between performance and energy, and from that propose two configurations addressing the tradeoffs. Note that the filter cache is proposed for the embedded processors, and the block buffering work is proposed for high performance ones.

Besides these designs, other designs also exploit a small L0 cache [24, 3, 11, 4, 7, 9]. In the HotSpot cache [24] and its variant [3], the L0 cache is used in the I-cache to store "hot" loops which have high execution fraction, to reduce accesses to the L1 cache. It does so by using the BTB to identify fetch addresses which belong to the hot loops. A similar approach [7] stores basic instruction blocks in a Tagless Instruction Cache. It also uses the BTB to store block information. Another approach [11] is similar to our design in that the L0 cache is known to be hit or missed before real accesses from the processor. That work is targeted for the instruction cache. The loop cache [9] focuses on designing cache to store loops for special purposes. The Non-Temporal Streaming cache [19] improves the direct-mapped L1 cache by using a small cache to hold non-temporal lines and targeted numerical programs. The scratchpad memory [4] was proposed to store critical data to avoid long accesses to the memory. In terms for access determination from the processor, the study in [16] proposes to do the set determination in parallel with the load store queue access, hence saving time comparing set IDs. Our study is proposed for the data cache, where the L0 tag comparison is moved to the processor pipeline.

7. CONCLUSIONS

In this paper we show that a migration policy between an L0 cache and an L1 data cache is crucial in optimizing cache performance in an embedded processor. We systematically investigated different policies during an insertion and a promotion, and found two configurations which have potential applications. The first is the *flow cache*, which reduces conflict misses in a direct-mapped L1 cache. By comparing to the victim cache, we show that the flow cache is a suitable

choice for the embedded system domain. The second is the *hit cache*, which reduces accesses to the set-associative L1 cache. Two hardware mechanisms were proposed for the hit cache. One does not impact performance, while the other has a slight degradation but further reduces energy consumption.

8. ACKNOWLEDGMENTS

This work is supported in part by NSF grant CISE-SHF 1118047. Nam Duong is also supported by the Dean's Fellowship, Donald Bren School of Information and Computer Sciences, UC Irvine. The authors would like to thank the anonymous reviewers for their useful feedback.

9. REFERENCES

[1] *ARM1156T2-S Technical Reference Manual*, 2005-2007.

[2] *Cortex-M3 Technical Reference Manual*, 2005-2008.

[3] K. Ali, M. Aboelaze, and S. Datta. Modified hotspot cache architecture: A low energy fast cache for embedded processors. In *Embedded Computer Systems: Architectures, Modeling and Simulation, International Conference on*, 2006.

[4] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *Proceedings of the tenth international symposium on Hardware/software codesign*, 2002.

[5] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 2011.

[6] C. Cascaval and D. A. Padua. Estimating cache misses and locality using stack distances. In *Proceedings of the 17th annual international conference on Supercomputing*, 2003.

[7] C.-W. Chen and C.-J. Ku. A tagless cache design for power saving in embedded systems. *The Journal of Supercomputing*, 2011.

[8] K. Ghose and M. B. Kamble. Reducing power in superscalar processor caches using subbanking, multiple line buffers and bit-line segmentation. In *Proceedings of the international symposium on Low power electronics and design*, 1999.

[9] A. Gordon-Ross, S. Cotterell, and F. Vahid. Exploiting fixed programs in embedded systems: A loop cache example. *IEEE Computer Architecture Letters*, 2002.

[10] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Workload Characterization*, 2001.

[11] S. Hines, D. Whalley, and G. Tyson. Guaranteeing hits to improve the efficiency of a small instruction cache. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, 2007.

[12] A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer. High performance cache replacement using re-reference interval prediction (RRIP). In *Proceedings of the 37th annual international symposium on Computer architecture*, 2010.

[13] 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 annual international symposium on Computer Architecture*, 1990.

[14] J. Kin, M. Gupta, and W. H. Mangione-Smith. The filter cache: an energy efficient memory structure. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, 1997.

[15] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, 1997.

[16] D. Nicolaescu, A. Veidenbaum, and A. Nicolau. Reducing power consumption for high-associativity data caches in embedded processors. In *Proceedings of the conference on Design, Automation and Test in Europe*, 2003.

[17] P. Petoumenos, G. Keramidas, and S. Kaxiras. Instruction-based reuse-distance prediction for effective cache management. In *Systems, Architectures, Modeling, and Simulation, International Symposium on*, 2009.

[18] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer. Adaptive insertion policies for high performance caching. In *Proceedings of the 34th annual international symposium on Computer architecture*, 2007.

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

[20] S. Sadasivan. *An Introduction to the ARM Cortex-M3 Processor*. ARM, 2006.

[21] C.-L. Su and A. M. Despain. Cache design trade-offs for power and performance optimization: a case study. In *Proceedings of the international symposium on Low power design*, 1995.

[22] D. Tarjan, S. Thoziyoor, and N. Jouppi. CACTI 4.0. Technical Report HPL-2006-86, HP Labs, 2006.

[23] T. Wolf and M. Franklin. CommBench-a telecommunications benchmark for network processors. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, 2000.

[24] C.-L. Yang and C.-H. Lee. HotSpot cache: joint temporal and spatial locality exploitation for i-cache energy reduction. In *Proceedings of the international symposium on Low power electronics and design*, 2004.