# Miss Reduction in Embedded Processors Through Dynamic, Power-Friendly Cache Design

Garo Bournoutian
University of California, San Diego
9500 Gilman Dr. #0404
La Jolla, CA 92093-0404
garo@cs.ucsd.edu

Alex Orailoglu
University of California, San Diego
9500 Gilman Dr. #0404
La Jolla, CA 92093-0404
alex@cs.ucsd.edu

## ABSTRACT

*Today, embedded processors are expected to be able to run complex, algorithm-heavy applications that were originally designed and coded for general-purpose processors. As a result, traditional methods for addressing performance and determinism become inadequate. This paper explores a new data cache design for use in modern high-performance embedded processors that will dynamically improve execution time, power efficiency, and determinism within the system. The simulation results show significant improvement in cache miss ratios and reduction in power consumption of approximately 30% and 15%, respectively.*

## Categories and Subject Descriptors

B.3.2 [**Memory Structures**]: Design Styles—*cache memories*; C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems—*real-time and embedded systems*

## General Terms

Design, Performance

## Keywords

embedded processors, data cache, multi-core, dynamic associativity

## 1. INTRODUCTION

The prevalence of embedded processors in modern computational systems has grown significantly over the last few years. At the current rate, embedded processors will become increasingly ubiquitous throughout our society, resulting in a broader range of applications that will be expanded to run on these devices. Even today, embedded processors are expected to be able to run complex, algorithm-heavy applications that were originally designed and coded for general-purpose processors. Furthermore, embedded processors are becoming increasingly complex to respond to this more diverse application base. Many embedded processors have begun to include features such as multi-level data caches, as well as expanding into the arena of multi-core.

With the constraints embodied by embedded processors, one typically is concerned with high performance, power efficiency, better execution determinism, and minimized area. Unfortunately, these characteristics are often adversarial, and focusing on improving one often results in worsening the others. For example, in order to increase performance, one adds a more complex cache hierarchy to exploit data locality, but introduces larger power consumption, more indeterminism in the data access time, and increased area. However, if an application is highly regular and contains an abundance of both spatial and temporal data locality, then the advantages in performance greatly outweigh the drawbacks. On the other hand, as these applications become more complex and irregular, they are increasingly prone to thrashing. For example, video codecs, which are increasingly being included in wireless devices like cell phones, utilize large data buffers and significantly suffer from cache thrashing [1].

In particular, embedded systems, where power efficiency is paramount, mostly choose primary (L1) caches that are direct-mapped as opposed to being associative. Earlier researchers realized that direct-mapped caches are more predisposed to thrashing, and proposed solutions such as the victim cache [2]. These solutions were useful in alleviating thrashing and cache pollution in simple applications, where the algorithms were short and localized. Yet, with today's more aggressive and highly irregular applications, the potential inadequacy of the victim cache and the possibility for improved solutions comes to bear.

In this paper, we propose a solution that will not only improve performance, power efficiency, and determinism, but also minimize the cost in area. We present an architecture capable of significantly reducing cache misses, which in turn reduces the overall execution time and power consumption by avoiding secondary memory accesses. This is done by adding a nominal amount of hardware. The architecture is capable of dynamically detecting the necessity for cache reconfiguration and is able to leverage preexisting hardware in order to accomplish greater cache utilization. We show the implementation of this architecture and provide experimental data taken over a general sample of complex, real-world applications to show the benefits of such an approach. The simulation results show significant improvement in cache miss ratios and reduction in power consumption of approximately 30% and 15%, respectively.


*DAC 2008*, June 8–13, 2008, Anaheim, California, USA

## 2. RELATED WORK

Various techniques have been proposed in the computer architecture community to attack the problem of cache conflict and cache pollution for data-intensive applications, both at the general-purpose and the embedded systems levels. In the last five years, the industrial embedded processor space has seen enormous expansion. Processors provided by companies such as ARM and Tensilica have become increasingly more powerful and complex, and are used in a wide variety of industrial applications. For example, current cell phone technology often incorporates both ARM9 and ARM11 embedded processors, along with a number of sophisticated specialized DSP processors, such as Qualcomm's QDSP6. These cell phones are expected to handle a wide variety of purposes, from data communication to audio/video processing, and even GPS tracking. Numerous cache optimization techniques have been proposed to address this class of high-performance embedded processors, where simply having a larger and more-associative cache is precluded by tough design constraints.

We already mentioned the victim cache, which retains data evicted from the cache in a small associative "victim" cache, typically containing 8 to 64 entries [2]. Victim caches are typically employed alongside direct-mapped L1 caches, and are used to combat the *brittleness* of having a direct-mapped cache by providing associativity for very localized cache conflict regions. For example, when a small loop is processing two arrays back-to-back that happen to have the same cache mapping, a victim cache would effectively expand the direct-mapped L1 cache to have a few extra associative entries for that mapping. However, as the applications expected to run on embedded systems become more complex and data-intensive, these cache conflicts are distributed and quite numerous, and a single associative location is quickly polluted and rendered ineffective.

The dual data cache scheme [3] proposes an architecture that can distinguish between spatial, temporal, or single-use memory references. This approach results in improved cache utilization, but the cache interference problem still remains.

Application-specific partitioning of caches is proposed in [4], but requires compile-time analysis and is unable to extend to dynamically created memories (such as pointers and heap elements). Furthermore, as embedded processors continue to be expected to run more complex, algorithmically intensive applications, the ability to find and exploit application specificity becomes increasingly difficult.

Pseudo-associative caches [5] provide extended associativity, but typically suffer from poor overall cache utilization. Allowing every set to remap to another set regardless of cache utilization pressure causes larger power consumption.

```
#define N = 1000;

int A[N], B[N], C[N];
int D[N], E[N], F[N];

for(int i = 0; i < N; ++i)
{
    A[i] = B[i] + C[i];
    D[i] = E[i] + F[i];
}
```

**Figure 1: Example Thrashing Code**

```
Miss on B[i], loaded into set-S
Miss on C[i], loaded into set-Q
Miss on A[i], loaded into set-R
Miss on E[i], loaded into set-S (with eviction)
    - B[i] is evicted and moved into Victim Cache
Miss on F[i], loaded into set-Q (with eviction)
    - C[i] is evicted and moved into Victim Cache
Miss on D[i], loaded into set-R (with eviction)
    - A[i] is evicted and moved into Victim Cache
       o B[i] is evicted from Victim Cache
Miss on B[i], loaded into set-S (with eviction)
    - E[i] is evicted and moved into Victim Cache
       o C[i] is evicted from Victim Cache
Miss on C[i], loaded into set-Q (with eviction)
    - F[i] is evicted and moved into Victim Cache
       o A[i] is evicted from Victim Cache
Miss on A[i], loaded into set-R (with eviction)
    - D[i] is evicted and moved into Victim Cache
       o E[i] is evicted from Victim Cache
...
```

**Figure 2: Cache Trace of Example Thrashing Code**

Furthermore, pure pseudo-associative caches are always accessed in a fixed order, starting with the initial location, and then searching in the secondary location if the first was unsuccessful. This rigidity in lookup order tends to detract from the performance benefit, since there is an inherent penalty for choosing the incorrect initial lookup location.

In the arena of power efficiency, filter caches [6] are able to reduce power consumption, but suffer from a significant decrease in performance. This is typically unacceptable in modern embedded processors. Similarly, the on-demand selective cache [7], which shuts down parts of the cache according to application demand, suffers in performance in order to achieve power efficiency.

## 3. MOTIVATION

A typical data-processing algorithm consists of data elements (usually part of an array or matrix) being manipulated within some looping construct. These data elements each effectively map to a predetermined row in the data cache. Unfortunately, different data elements may map to the same row due to the inherent design of caches. In this case, the data elements are said to be in "conflict". This is typically not a large concern if the conflicting data elements are accessed in disjoint algorithmic hot-spots, but if they happen to exist within the same hot-spot, each time one is brought into the cache, the other will be evicted, and this *thrashing* will continue for the entire hot-spot.

Additionally, with structures such as a victim cache [2], which use a central repository to hold recently evicted data, unrelated data accesses may cause this repository to become polluted. This becomes increasingly prevalent as the complexity of an algorithm grows.

For example, figure 1 shows an excerpt of code for a particularly unfortunate memory layout, assuming arrays B and E map into set-S, arrays C and F map into set-Q, and arrays A and D map into set-R. For the sake of this example, assume the L1 data cache is direct-mapped and we have a 2-entry victim cache. The trace of this code is shown in figure 2. As can be seen, the above code will display abysmal cache performance, since the number of active elements exceeds the available storage space for that particular section of the cache. Furthermore, the victim cache is also rendered

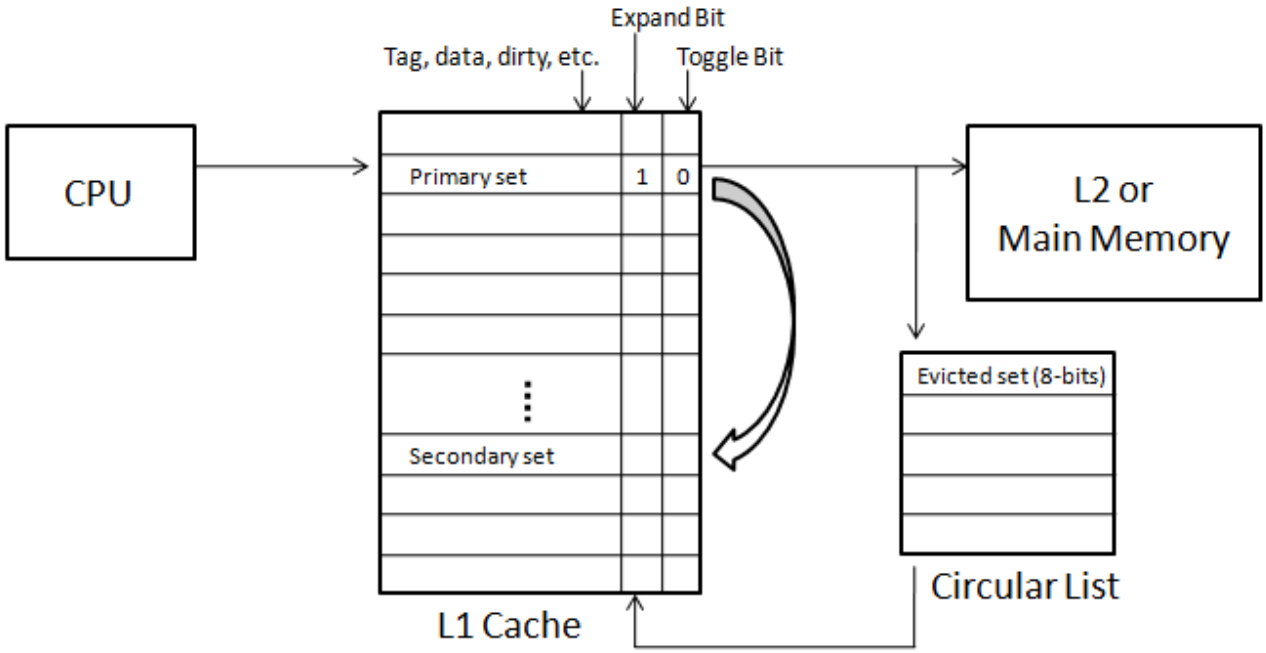


**Figure 3: Dynamically Expandable L1 Cache Implementation**

ineffective due to uncorrelated evicted data polluting and overwhelming the central victim storage.

Obviously, this is an extreme example. Nevertheless, given large, complex, data-intensive applications, the probability of multiple cache lines being active within a hot-spot, as well as the probability of those cache lines mapping to the same cache set, increases dramatically. Given the current trajectory of application complexity expected to run on embedded systems, the possibility of such data conflicts and pollution occurring should be investigated in order to ascertain if additional architectural improvements are warranted.

# 4. IMPLEMENTATION

The proposed architecture is composed of two mechanisms: a circular *recently-evicted-set* list and an expandable cache lookup. Each mechanism is described in detail in the following sections. A high-level view is provided in figure 3.

## 4.1 Circular Recently-Evicted-Set List

The first mechanism is a very small circular list that will keep track of the most recently evicted sets from the cache (due to a cache miss). This is shown in figure 3 on the bottom, right-hand side. We only need to keep track of the set (as opposed to the full address), since that will encompass items that map to the same row within the cache. The point of this list is to maintain a record of valid cache entries that were displaced by another entry with the same set value. When we encounter a cache eviction, we can look and see if that set is already present in the list. If so, we can conclude that the current set is in a probable state of thrashing and should dynamically be expanded. When this happens, we enable the *expand bit* for that set on the cache.

One can control the threshold for probable thrashing detection by adjusting the length of the circular list. For the purposes of this paper, we chose a length of only five entries, but this can be any positive integer. The width of each of these entries is dependent on the number of sets within the L1 cache. For example, an L1 cache with 256 sets requires an encoding of 8-bits, which equates to a total size of 40-bits for the circular list. Additionally, since this is a circular list, we need a list index, the size of which is dependent on the number of entries in the list. The width of this list index is simply the log-base-2 of the number of entries rounded to the next integer value. If we take a five-entry list, it requires 3-bits for the list index. The generalized equation for determining the total size of this list structure is provided below, where $L$ is the number of entries in the list, $S$ is the number of sets in the cache, and B is the resulting number of additional bits required:

$$B = \lceil \lg L \rceil + (L * \lceil \lg S \rceil)$$

For example, a 256-set cache and a five-entry list would result in a total size of only 43-bits, which is rather nominal.

Furthermore, timing is not affected, since accessing and updating this list only occurs during a cache miss. Thus, the access and update is effectively masked by the associated timing penalty for the miss.

## 4.2 Expandable Cache Lookup

The second mechanism allows for the expansion of a particular cache set into a second cache set. Once a probable thrashing set is detected by the first mechanism, the *expand bit* is turned on. On a cache miss, if this bit is enabled, a secondary set within the same cache is accessed on the

next cycle, similar to a pseudo-associative cache. This secondary set is determined by a fixed mapping function. For simplicity, we chose to have the mapping function pick the complementary set based on the cache size, which is achieved simply by flipping the high-order bit for the set. For a cache with 256 sets, set-0's complement would be set-128. Similarly, set-250's complement would be set-122.

If the data is found within the secondary set, you effectively have a cache hit, but with a one cycle penalty. If after looking into the secondary set, the data is still not found, a full cache miss occurs and the next memory device in the hierarchy (L2) is accessed with the associated cycle penalties.

Additionally, there is a *toggle bit* on each of the sets within the cache. This bit is enabled whenever a cache hit occurs on the secondary cache set; it is disabled when a cache hit occurs on the primary cache set. The *toggle bit* is also used on subsequent cache lookups to determine whether to initially do a lookup on the primary cache set or the secondary cache set. This toggle functionality effectively encodes an MRU scheme into the expandable cache. If the last hit occurred in the secondary set, the next access has a higher probability to also occur in the secondary set, and vice versa.

In addition, we implemented our caches to use an LRU replacement policy. When a set has its *expand bit* on, the LRU set (primary or secondary) is used when a replacement is necessary. Furthermore, for associative caches, after determining which set to use, the normal LRU rules for which way to use within that set apply. The set-wise LRU is supported by the *toggle bit*.

To implement this expandable cache, one will need two extra bits for every cache set (one for the *expand bit* and one for the *toggle bit*). Since we have 256 sets in our implementation, this results in an additional 512-bits added to our L1 cache. Thus, even though we introduce a minimal amount of extra storage, by adding a dynamic mechanism that allows a set to re-lookup into a predefined secondary set, we effectively double the size of the given set without needing to double the hardware. This works in principle due to the typical locality of caches. Since our complement set is chosen to be the bidirectionally farthest distance from the current set, the probability of both the primary set and the complement set being active at the same temporal moment in an application is quite small. Thus, we do not need to worry that by using the complement set we are polluting a currently active set in our cache. Later in the program's progression, if the complement set is then used, it will function normally and evict the older data from the prior expansion out of its space.

It is important to note that once a set has its *expand bit* enabled, it remains enabled until that cache is flushed or reset. Furthermore, this expansion need not be symmetrical. Even though *set-A* may have its *expand bit* turned on and be utilizing *set-A* and its complement *set-B*, *set-B* can still be acting as a normal non-expanded set. Only if primary accesses to *set-B* also trigger the expansion threshold will *set-B*'s *expand bit* also be enabled to allow expansion into *set-A*.

## 4.3 A Demonstrative Example

Figure 4 provides the trace of this proposed cache architecture using the code excerpt from figure 1. In this example, a direct-mapped L1 cache is assumed, and *set-S'* (*S-prime*) refers to the complement of *set-S*.

```
Miss on B[i], loaded into set-S
Miss on C[i], loaded into set-Q
Miss on A[i], loaded into set-R
Miss on E[i], loaded into set-S (with eviction)
    - set-S added to circular list due to eviction
Miss on F[i], loaded into set-Q (with eviction)
    - set-Q added to circular list due to eviction
Miss on D[i], loaded into set-R (with eviction)
    - set-R added to circular list due to eviction
Miss on B[i], but set-S also detected in circular list
    - set-S expand bit enabled, loaded into set-S'
Miss on C[i], but set-Q also detected in circular list
    - set-Q expand bit enabled, loaded into set-Q'
Miss on A[i], but set-R also detected in circular list
    - set-R expand bit enabled, loaded into set-R'
Hit on E[i], loaded from set-S
Hit on F[i], loaded from set-Q
Hit on D[i], loaded from set-R
Hit on B[i], loaded from set-S'
Hit on C[i], loaded from set-Q'
Hit on A[i], loaded from set-R'
Hit on E[i], loaded from set-S
Hit on F[i], loaded from set-Q
Hit on D[i], loaded from set-R
...
```

**Figure 4: Cache Trace of Example Trashing Code on Proposed Architecture**

As can be seen, once the thrashing behavior was detected, the cache was dynamically able to expand *set-S, set-Q, and set-R* to also include *set-S', set-Q' and set-R'*, respectively. Thus, by the third loop iteration, all data was able to fit within the cache without parasitic eviction.

This example illustrates how our proposed architecture is dynamically able to detect thrashing behavior and adjust accordingly. Obviously, this particular example is an extreme case, but as embedded applications continue to grow in complexity and data-intensity, scenarios similar to the one shown above will become increasingly more common. As we will show in the following section, current large-scale applications do indeed suffer from this type of interference and pollution, and our proposed design is able to resolve much of this contention.

# 5. EXPERIMENTAL RESULTS

In order to assess the benefit from this proposed architectural design, we used the SimpleScalar toolset [8]. We chose two representative system configurations. The first configuration used a 256-set, direct-mapped L1 data cache with a 32-byte line size. The second configuration used a 256-set, 4-way set-associative L1 data cache with a 32-byte line size and an LRU replacement policy. Both configurations utilized a typical in-order simulation engine, with dedicated L1 caches for data and instruction memory, backed by a unified L2 cache.

Seven representative programs from the SPEC CPU2000 benchmark suite [9] are used: *bzip2* - compression program; *crafty* - high-performance chess playing application; *gcc* - C compiler; *gzip* - LZ77 compression program; *parser* - word processing application; *twolf* - CAD place and route simulation; and *vpr* - FPGA place and routing.

Table 1 utilized the direct-mapped L1 data cache configuration and provides the total number of L1 accesses, baseline miss rates, miss rates using an 8-entry victim cache, the vic-

| | Baseline | | Victim Cache | | Our Design | |
|---|---|---|---|---|---|---|
| Benchmark | Accesses | Miss Rate | Miss Rate | Improvement | Miss Rate | Improvement |
| bzip2 | 7.69e10 | 0.0482 | 0.0358 | 25.73% | 0.0339 | 29.67% |
| crafty | 9.67e9 | 0.1314 | 0.1141 | 13.17% | 0.0881 | 32.95% |
| gcc | 8.90e8 | 0.0599 | 0.0483 | 19.37% | 0.0363 | 39.40% |
| gzip | 3.46e10 | 0.0483 | 0.0430 | 10.97% | 0.0344 | 28.78% |
| parser | 4.83e9 | 0.0843 | 0.0687 | 18.51% | 0.0518 | 38.55% |
| twolf | 4.18e9 | 0.1426 | 0.1278 | 10.38% | 0.1211 | 15.08% |
| vpr | 4.72e9 | 0.1052 | 0.0852 | 19.01% | 0.0728 | 30.80% |

**Table 1: Direct-Mapped L1 Miss Rates**

| | Baseline | | Victim Cache | | Our Design | |
|---|---|---|---|---|---|---|
| Benchmark | Accesses | Miss Rate | Miss Rate | Improvement | Miss Rate | Improvement |
| bzip2 | 7.69e10 | 0.0270 | 0.0266 | 1.48% | 0.0191 | 29.26% |
| crafty | 9.67e9 | 0.0102 | 0.0060 | 41.18% | 0.0060 | 41.18% |
| gcc | 8.90e8 | 0.0157 | 0.0144 | 8.28% | 0.0099 | 36.94% |
| gzip | 3.46e10 | 0.0312 | 0.0308 | 1.28% | 0.0224 | 28.21% |
| parser | 4.83e9 | 0.0396 | 0.0387 | 2.27% | 0.0257 | 35.10% |
| twolf | 4.18e9 | 0.0876 | 0.0865 | 1.26% | 0.0776 | 11.42% |
| vpr | 4.72e9 | 0.0450 | 0.0444 | 1.33% | 0.0427 | 5.11% |

**Table 2: 4-way Set-Associative L1 Miss Rates**

tim cache's improvement over the baseline, our implementation's miss rates using a 5-entry recently-evicted-set list, and our percentage improvement over the baseline. In addition, figure 5 graphically compares the percent improvement in miss rate between the victim cache and our implementation. The arithmetic mean of the miss rate improvement across the benchmarks for our proposed implementation is 30.75%.

Similarly, table 2 shows the same fields, but for the 4-way set-associative L1 data cache configuration. This configuration employed a 64-entry victim cache and an 8-entry recently-evicted-set list. In addition, figure 6 graphically compares the percent improvement in miss rate between the victim cache and our implementation. The arithmetic mean of the miss rate improvement across the benchmarks for our proposed implementation is 26.74%.

As one can see, our expandable cache architecture consistently provides significant reductions in the miss rates, both for direct-mapped and set-associative caches.

With regard to power efficiency, we used CACTI [10] to determine an L1/L2 power ratio of 20, meaning it costs 20 times more to access data in L2 than it does in L1. This ratio was determined based on a typical L2 cache configuration[1]. The larger this ratio, the more power improvement will be seen by reducing L1 cache misses[2]. Moreover, if an

[1]L2 was assumed to be 256 KB, 4-way set-associative, and having a 64-byte line size. We also assumed the feature size of the hardware to be 65nm.

[2]Furthermore, we believe that this number will in fact be larger on multi-core embedded systems, since L2 is typically outside the core and is shared, necessitating large power usage to drive the I/O pins.

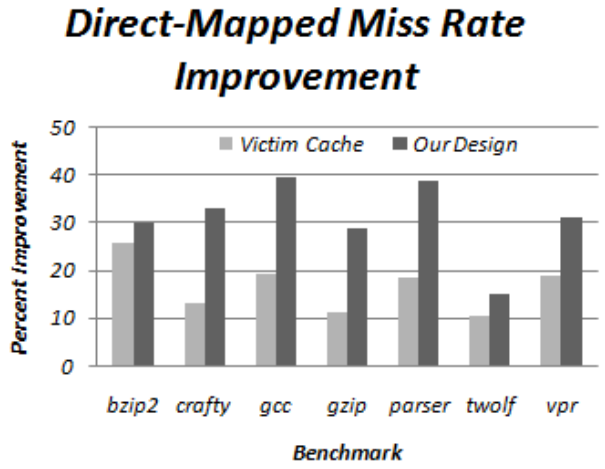


**Figure 5: Comparative Miss Rate Improvement**

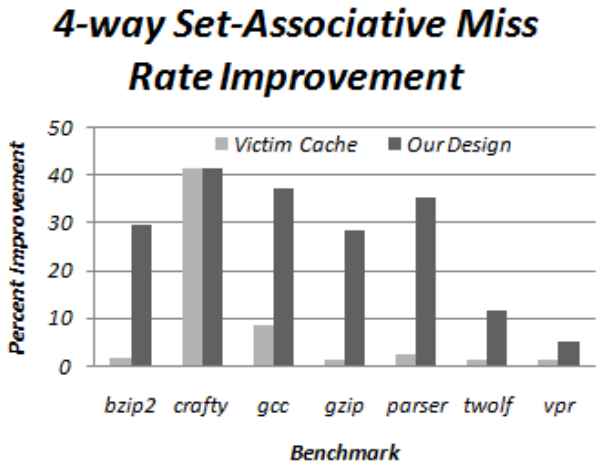


**Figure 6: Comparative Miss Rate Improvement**

| Benchmark | Victim Cache Improvement | Our Design Improvement |
|---|---|---|
| bzip2 | 0.12% | 11.43% |
| crafty | 0.09% | 20.84% |
| gcc | 0.10% | 18.42% |
| gzip | 0.05% | 12.26% |
| parser | 0.11% | 21.20% |
| twolf | 0.07% | 8.50% |
| vpr | 0.12% | 17.47% |

**Table 3: Direct-Mapped Power Usage Improvements**

| Benchmark | Victim Cache Improvement | Our Design Improvement |
|---|---|---|
| bzip2 | 0.00% | 6.92% |
| crafty | 0.07% | -2.89% |
| gcc | 0.02% | 4.40% |
| gzip | 0.00% | 8.61% |
| parser | 0.01% | 11.40% |
| twolf | 0.00% | 3.70% |
| vpr | 0.00% | -2.89% |

**Table 4: 4-way Set Associative Power Usage Improvements**

embedded system does not have an L2 cache, but rather goes directly from L1 to main memory, this ratio would again be much higher.

Using this information, we are able to determine approximate power consumption for the full operation of each of the aforementioned benchmarks. Tables 3 and 4 show the percentage of power improvement achieved by the victim cache and our implementation using the direct-mapped configuration and the 4-way set-associative configuration, respectively. As one can see, our expandable cache architecture consistently provides significant reductions in power for the direct-mapped case. The average power reduction across all benchmarks was 15.73%.

With regard to the set-associative case, our numbers continue to show improvement, albeit to a lesser degree, even though in a couple of instances the proposed technology resulted in higher power costs. The average across the benchmarks for the set-associative setup was still an improvement of 4.19%. Furthermore, set-associative L1 caches typically are not used in power-constrained systems (since they inherently draw too much power to begin with), which ameliorates the necessity for marked power improvements for set-associative designs.

# 6. CONCLUSIONS

We have presented a novel architecture for reducing cache misses and increasing power efficiency for high-performance embedded processors. Preventing cache interference and cache pollution by allowing dynamic expansion of select sets within the L1 data cache have been the main objectives of the proposed architecture. The achievement of these goals has been confirmed by extensive experimental results. A significant increase in cache hit rate and a notable decrease in power usage have been demonstrated by a representative set of simulation results. The proposed technique has significant implications for embedded processors, especially high-performance, power-sensitive devices such as cell phones and MP3 players, as it significantly reduces power consumption while improving execution time and determinism.

As embedded processors continue to spread and become ubiquitous, it is essential to maintain high performance, low power, and small size. The proposed architecture fulfills these requirements and enables embedded processors to continue to mature and be able to handle exceedingly complex and aggressive applications.

# 7. ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their constructive comments in improving the quality of this paper.

# 8. REFERENCES

[1] Li Lee, Srikanth Kannan, and Jose Fridman. MPEG4 video codec on a wireless handset baseband system. In *Proc. Workshop Media and Signal Processors for Embedded Systems and SoCs*, 2004.

[2] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. *SIGARCH Computer Architecture News*, pages 364–373, 1990.

[3] Antonio González, Carlos Aliagas, and Mateo Valero. A data cache with multiple caching strategies tuned to different types of locality. In *ICS '95: Proceedings of the 9th International Conference on Supercomputing*, pages 338–347, 1995.

[4] Peter Petrov and Alex Orailoglu. Performance and power effectiveness in embedded processors – customizable partitioned caches. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1309–1318, 2001.

[5] Anant Agarwal and Stephen D. Pudar. Column-associative caches: a technique for reducing the miss rate of direct-mapped caches. *SIGARCH Computer Architecture News*, pages 179–190, 1993.

[6] Johnson Kin, Munish Gupta, and William H. Mangione-Smith. The filter cache: an energy efficient memory structure. In *MICRO 30: Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 184–193, 1997.

[7] David H. Albonesi. Selective cache ways: on-demand cache resource allocation. In *MICRO 32: Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, pages 248–259, 1999.

[8] Todd Austin, Eric Larson, and Dan Ernst. Simplescalar: An infrastructure for computer system modeling. *Computer*, pages 59–67, 2002.

[9] SPEC CPU2000 Benchmarks. http://www.spec.org/cpu/.

[10] Steven J. E. Wilton and Norman P. Jouppi. CACTI: An enhanced cache access and cycle time model. *IEEE Journal on Solid-State Circuits*, 31(5):677–688, 1996.