Design and Implementation of Software-Managed Caches for Multicores with Local Memory[*]

Sangmin Seo[†] Jaejin Lee[†] Zehra Sura[‡]

[†]School of Computer Science and Engineering, Seoul National University, Seoul, Korea
[‡]IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA
sangmin@aces.snu.ac.kr, jlee@cse.snu.ac.kr, zsura@us.ibm.com
http://aces.snu.ac.kr

Abstract

Heterogeneous multicores, such as Cell BE processors and GPGPUs, typically do not have caches for their accelerator cores because coherence traffic, cache misses, and latencies from different types of memory accesses add overhead and adversely affect instruction scheduling. Instead, the accelerator cores have internal local memory to place their code and data. Programmers of such heterogeneous multicore architectures must explicitly manage data transfers between the local memory of a core and the globally shared main memory. This is a tedious and error-prone programming task. A software-managed cache (SMC), implemented in local memory, can be programmed to automatically handle data transfers at runtime, thus simplifying the task of the programmer. In this paper, we propose a new software-managed cache design, called extended set-index cache (ESC). It has the benefits of both set-associative and fully associative caches. Its tag search speed is comparable to the set-associative cache and its miss rate is comparable to the fully associative cache. We examine various line replacement policies for SMCs, and discuss their trade-offs. In addition, we propose adaptive execution strategies that select the optimal cache line size and replacement policy for each program region at runtime. To evaluate the effectiveness of our approach, we implement the ESC and other SMC designs on the Cell BE architecture, and measure their performance with 8 OpenMP applications. The evaluation results show that the ESC outperforms other SMC designs. The results also show that our adaptive execution strategies work well with the ESC. In fact, our approach is applicable to all cores with access to both local and global memory in a multicore architecture.

1 Introduction

Multicores have been widening their user base, with many processor vendors having released their multicore processors [13]. There are two types of multicore architectures: homogeneous architectures that resemble symmetric-multiprocessors, and heterogeneous architectures that use distinct accelerator cores, such as Cell BE processors [6, 9] and GPGPUs [4, 19, 23]. Unlike homogeneous multicore architectures, heterogeneous multicores typically do not have caches for their accelerator cores [16]. This is because cache coherence traffic, cache misses, and latencies from different types of memory accesses add overhead and adversely affect instruction scheduling. Instead of using caches, accelerator cores typically access data from local memory that is not part of the system memory hierarchy.

Programmers of such heterogeneous multicore architectures must explicitly manage data transfers between the local memory of a core and the globally shared main memory. This is a tedious and error-prone programming task. A software-managed cache (SMC), implemented in local memory, can be programmed to automatically handle data transfers at run time, thus simplifying the task of the programmer. The SMC can also enable performance gains by hiding some of the long latency of accesses to globally shared main memory.

Similar to hardware caches, SMCs provide fast memory access when the requested data item resides in the cache. However, there are major differences between hardware caches and SMCs. First, the organization of the SMC is flexible while the structure of the hardware cache is typically fixed. The SMC organization can be largely different from the conventional hardware cache. The programmer can customize the structure even at run time to make it suitable for the target application. For example, the programmer can resize the SMC as long as its size fits in the local memory, or adjust its line size at run time. The programmer can even choose a proper line replacement policy at run time. Second, the operation of the SMC is not fully transparent to the user. The user has to insert API functions for accesses that need to go through the SMC. The API functions differentiate accesses that use the SMC from other accesses that directly refer to local memory. Finally, the performance of the SMC is dependent not only on the number of misses but also on the way in which the cache operations are implemented. More sophisticated algorithms may degrade the performance of the SMC due to the cost of the instructions and memory operations used in the software implementation. Consequently, identifying a cache algorithm that has a sufficiently low implementation complexity as well as a low cache miss rate is critical in the design and implementation of the SMC to achieve high performance.

[*]This work was partly supported by the IT R&D program of MIC/IITA [2006-S-040-01, Development of Flash Memory-based Embedded Multimedia Software], by the IBM Corporation, and by the Ministry of Education, Science and Technology under the BK21 Project. ICT at Seoul National University provided research facilities for this study.

978-1-4244-2932-5/08/$25.00 ©2008 IEEE

In this paper, we propose a new SMC design, called *extended set-index cache* (ESC) for heterogeneous multicore architectures. The ESC is based on the 4-way set-associative cache. Its tag lookup table has the same structure as that of the 4-way set-associative cache. However, it decouples the tag lookup table entries and the data blocks (i.e., the cache lines). The mapping between the tag entries and the data blocks is dynamically determined. This enables the ESC to have a larger number of entries in the tag lookup table than the number of cache lines. Using a larger lookup table (i.e., a larger number of sets) than the 4-way set-associative cache with the same number of data blocks results in miss rates for the ESC that are comparable to miss rates of a fully associative cache.

Even though this decoupled cache structure is essentially the same as that of the V-Way cache proposed by Qureshi *et al.*[21], there are major differences. First, the V-Way cache is a hardware cache while the ESC is implemented entirely in software. We can configure cache parameters, such as number of sets, cache size, line size, and replacement policy, both on-line and off-line in the ESC. Consequently, the ESC is more flexible than the V-Way cache. Second, the replacement policy that best suits the V-Way cache is not appropriate for the ESC due to the cost of instructions and memory operations used in the software implementation. Finally, the ESC can fully utilize the available local memory space depending on the application needs. The available local memory space for the ESC is typically variable because the local memory is partitioned between code, stack (local data), and globally shared data.

When the number of sets is a power of two, it enables using a cheap bit-wise AND operation to compute the set index in the ESC instead of the more expensive modulo operation. Since the structure of the ESC tag lookup table is the same as that of the 4-way set-associative cache, the speed of the ESC tag lookup is as fast as that of the 4-way set-associative cache. For tag lookups, we exploit SIMD comparison operations typically available in accelerator cores.

We examine various line replacement policies for the SMC, and discuss their trade-offs. In addition, we show that a single cache line size or a single replacement policy does not fit all applications. Based on this observation, we propose an adaptive execution strategy that selects an optimal cache line size and replacement policy for each program region at run time.

To evaluate the effectiveness of our approach, we implement the ESC and other cache designs fully in software on the Cell BE processor, and measure their performance with 8 OpenMP applications running on the 8 accelerator cores in the Cell BE processor. We also show the effectiveness of our adaptive execution strategy with the ESC.

The evaluation results show that the ESC significantly reduces the number of misses and improves performance compared to other SMC designs. The result of adaptive execution shows that it achieves nearly the same performance as an execution that uses specific replacement policies and line sizes for each parallel loop. These specific replacement policies and line sizes are manually pre-determined to be optimal.

Our approach is applicable to all cores with access to both local and global memory in a multicore architecture, even though our implementation targets a specific heterogeneous multicore architecture.

The rest of this paper is organized as follows. In the next section, we discuss related work. Section 3 describes the target heterogeneous multicore architecture and its runtime. The design and implementation of the software-managed caches is presented in Section 4. Our adaptive execution strategy is described in Section 5. Section 6 presents the evaluation results. Finally, Section 7 concludes the paper.

2 Related Work

There have been many proposals in hardware direct-mapped caches and set-associative caches to reduce cache misses, especially conflict misses[8]. These hardware cache proposals can be divided into two categories: new cache organizations and new hash functions.

Jouppi[10] proposes a victim cache that can be added to a direct-mapped cache to reduce conflict misses. The victim cache is a small fully-associative cache containing recently evicted lines from the direct-mapped cache. On a miss in the direct-mapped cache, if the missed line hits in the victim cache, the matching victim cache line is loaded into the direct-mapped cache. Seznec[24] introduces the skewed-associative cache that is an organization of multi-bank caches. Based on a 2-way set-associative cache that has two distinct banks, the cache uses a different hash function for each bank. It has a miss ratio close to that of a 4-way set-associative cache.

Pudar *et al.*[24] introduce the column-associative cache to reduce the miss rate of direct-mapped caches. The column-associative cache allows conflicting addresses to dynamically choose different hashing functions. Parcerisa *et al.*[5] show that XOR-based hashing functions can eliminate many conflict misses. They evaluate a hashing function that uses a simple bit-wise XOR of two portions of the requested address and the polynomial hashing mechanism proposed in [22]. Kharbutli *et al.*[11, 12] propose two hashing functions based on prime numbers to reduce conflict misses. One is a prime modulo function that uses a prime number instead of a power of two when performing the modulo operation. The other is the odd-multiplier displacement hashing function in which a modulo operation with the number of sets is performed after transforming the original set index in some special way.

Qureshi *et al.*[21] propose the V-Way cache. They increase the number of tag-store entries relative to the number of data lines. This increases the number of sets and decouples the tag store and data blocks to achieve high associativity. The V-Way cache achieves the performance benefit of a fully-associative cache while maintaining the constant hit latency of a set-associative cache. They also propose a frequency based global replacement scheme called *reuse replacement*. We implement the reuse replacement policy in the ESC and compare its performance to other replacement policies in Section 6. The differences between the V-Way cache and the ESC have been described in the previous section.

Hallnor *et al.*[7] propose a software-managed hardware cache called indirect index cache (IIC). The IIC extends the number of ways in a set on demand. As a result, its observable behavior is the same as a fully associative cache. They propose a software management scheme called *generational replacement* to fully utilize the IIC. While the IIC extends the number of ways in a set, the ESC extends the number of sets and gives a miss rate that is close to the fully associative cache. In addition, the ESC is implemented completely in software. Dynamic extension of ways in a set adversely affects the performance when implemented in software. We implement the IIC in software and compare its performance with that of the ESC in Section 6.

The proposals that target hardware caches can be implemented in software. However, performance may degrade because the underlying machine may not have fast specialized operations that need to be used to make the implementation efficient and feasible. This paper focuses on the design and implementation of SMCs that overcome these difficulties.

There are some proposals for software-managed caches. Miller *et al.*[17] demonstrate software-based instruction caching schemes for processors without any caching hardware. In their system, software automatically manages all or part of scratchpad memory as a cache. While their system focuses on instruction caching, our approach focuses on data caching with a different cache organization.

Eichenberger *et al.*[3] propose a compiler-controlled software cache for the Cell BE architecture. It is a typical 4-way set-associative cache implemented in software, using a FIFO replacement policy. It exploits SIMD instructions to look for a match among the four tags in a set. The ESC exploits the same SIMD parallelism to compare tags, but its organization and performance are quite different from the 4-way set-associative cache.

The software-managed cache proposed by Balart *et al.*[2] also targets the Cell BE. This cache uses hashed list structures for lookup and provides full associativity and the ability to lock cache lines. These features enable a user to determine code regions guaranteed not to have any cache conflicts. The user can re-order cache lookup and miss handling operations related to multiple accesses in such regions, so as to generate code with greater overlap between communication and computation. Additionally, the cache is configured to delay re-using a line until all prior free lines have been used, and to initiate early write-backs for dirty cache lines. This enables further communication-computation overlap. Detecting when to initiate early write-back for a line is made possible by code transformations that label and classify cache accesses in the code. The proposed cache is likely to perform well for specific loops containing few cache accesses with high temporal locality, but it has a large implementation overhead in the general case. In contrast, the cache mechanism we propose is comparatively low overhead, adaptive and likely to perform reasonably well across a wide range of access patterns, and does not rely on code transformations specific to the cache configuration.

There are some proposals that are related to our adaptive execution strategy. Lee *et al.*[15, 26] present an adaptive execution algorithm to automatically map code on a heterogeneous intelligent memory architecture that consists of a high-end host processor and a simpler memory processor. The algorithm partitions the application code, and adaptively schedules the partitions on the most appropriate processor. Veidenbaum *et al.*[28] introduce an algorithm that adaptively chooses a proper cache line size for an application. The algorithm is implemented in hardware and changes the size of each cache line according to the portion of the line accessed between cache misses. If half of the line is not accessed, its size decreases by half. Our adaptive execution scheme is not for code mapping. It uses runtime information to dynamically select an optimal cache line size and replacement policy for each region in an application. The selected line size is applied to the entire cache for the region.

3 The Target Architecture and Runtime

In this section, we briefly describe our target processor architecture and its runtime.

We target any multicore architecture that is similar to the architecture shown in Figure 1. This architecture consists of a general-purpose processor element (PE) that can perform system management tasks and execute privileged kernel routines, and multiple accelerator processor elements dedicated to compute intensive workloads. The processor elements are connected by a high speed interconnect bus. Typical examples of this architecture are the Cell BE architecture[6, 9], and the GPGPU multiprocessor architecture[19, 23] in which a general-purpose processor is connected to multiple graphics cards through a PCI express bus.

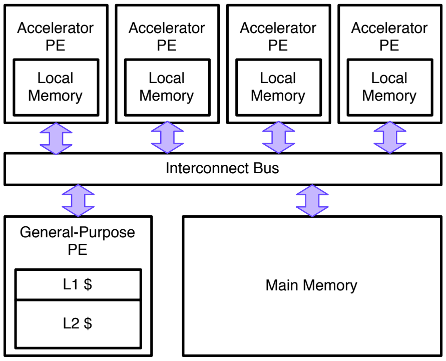


Figure 1. The target architecture.

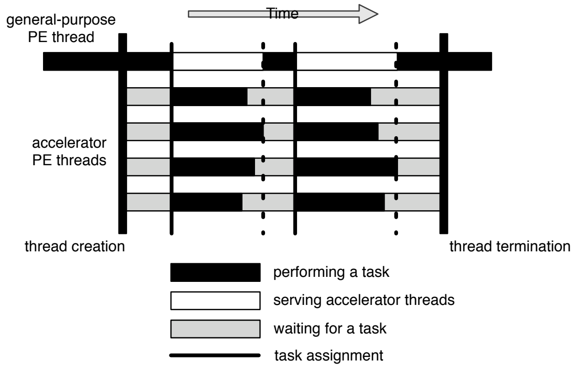


Figure 2. The thread model.

The general-purpose PE is typically backed by a deep on-chip cache hierarchy. The hardware guarantees coherence between these caches and the external main memory. On the other hand, the accelerator PEs typically do not have any caches because coherence traffic, cache misses, and latencies from different types of memory accesses adversely affect instruction scheduling, resulting in poor performance. Instead, each accelerator PE has internal local memory that is not coherent with the external main memory. Instead of using load and store instructions, data is transferred between the local memory and the main memory using DMA operations. Henceforth, we use a *global address* to refer to an address in the main memory and a *local address* to refer to an address in the local memory of an accelerator core. The accelerator PEs are capable of extremely high performance computation, and support SIMD operations.

Programmers who write code for such an architecture need to explicitly manage data transfers between the local memory of an accelerator PE and the main memory. Moreover, the programmer has to ensure memory consistency and coherence because they are not supported by the hardware. A runtime, such as a software shared virtual memory[14], can provide a coherent and consistent view of the main memory to the accelerator PEs. As a result, the programmer sees a global shared address space between the general-purpose PE and the accelerator PEs. The runtime provides a page-level (i.e., cache line level) coherence and consistency mechanism, and the SMC is a critical component of this runtime.

Whenever an accelerator PE needs a page from the main memory, it fetches the page from the main memory and saves it in the

SMC implemented in its local memory. When the PE accesses the page again, it accesses the page saved in the SMC. Fetching pages from and flushing pages to the main memory follow the coherence and consistency protocol defined by the runtime. The coherence and consistency protocol is implemented using the communication mechanisms between the general-purpose PE and the accelerator PEs provided by the underlying architecture. The miss penalty of the SMC mainly consists of the protocol overhead and the latency of a DMA operation.

The runtime API provides functions for reading and writing shared variables, thread creation and destruction, page size variation, and synchronization. The programmer only needs to distinguish shared memory reads and writes and replace them with the read and write API functions. More details about the runtime and its API functions can be found in [14]. Figure 2 shows the thread model used in our runtime. All coherence actions are performed by the general-purpose PE. The model is similar to the OpenMP thread model[20].

4 Software-Managed Caches

The design of the Extended Set-Index Cache (ESC) is based on the conventional 4-way set-associative cache[8]. Before we describe the design of our ESC, we introduce feasible software implementations of the 4-way set-associative cache and the fully associative cache, and address issues specific to the software implementation.

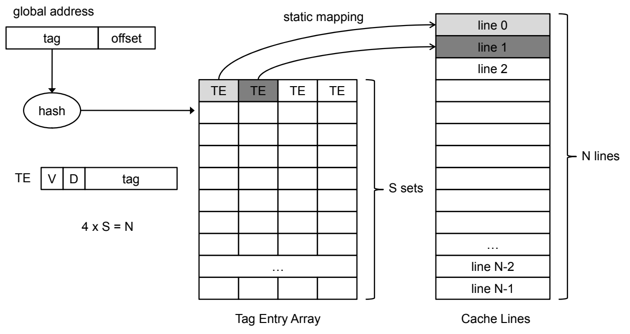


Figure 3. The 4-way set-associative cache.

4.1 The 4-Way Set-Associative Cache

Figure 3 shows the structure of the 4-way set-associative cache. This type of SMC has been used in the Cell BE architecture and provided in its SDK[6, 9]. It has S sets and N data blocks (cache lines), and consists of a hash function h, an array of $4 \times S$ tag entries and an array of N cache lines. Each tag entry (TE) contains a tag, a valid bit, and a dirty bit. There is a one-to-one correspondence between the tag entries and the cache lines, and the mapping is static. In other words, it has the property that the number of TEs $(4 \times S)$ is equal to the number of cache lines (N). The tag in a TE is the global address of the cache line that corresponds to the TE.

When the cache receives a memory request with a global address x, the hash function returns the set index corresponding to the requested global address. The valid tags in the selected set are compared to the tag in the global address. These tag comparisons are performed in parallel with the SIMD comparison instruction supported by the accelerator core. If there is a valid matching TE, the request is a hit and the local address is determined by adding the block offset (in the global address) to the local address of the data block that corresponds to the matching TE.

Otherwise, it is a miss, and the miss handler is invoked. It selects a victim data block following the predefined replacement policy. It evicts the victim from the cache and flushes the victim to the main memory. Then, it fetches the data block that corresponds to the requested global address from the main memory and places it in the evicted location.

The hash function is typically defined by,

$$h(x) = x \textbf{ modulo } S$$

The hash function can use a bit-wise AND operation to improve performance if S is a power of 2. Otherwise, it uses the expensive modulo operation to compute the set index.

Note that to fully utilize the available local memory space for the cache, often the number of sets is not a power of 2. The modulo operation causes a significant computational overhead because it has to be performed for each data request.

4.2 The Fully Associative Cache

The structure of the software-managed fully associative cache is similar to that of the 4-way set-associative cache, but it has only one set. In this case, performance is degraded by the sequential search to find the valid matching tag for the request. Since there is typically no machine level operation that supports more than four tag comparisons in parallel, almost all tags need to be compared with the requested tag in the worst case.

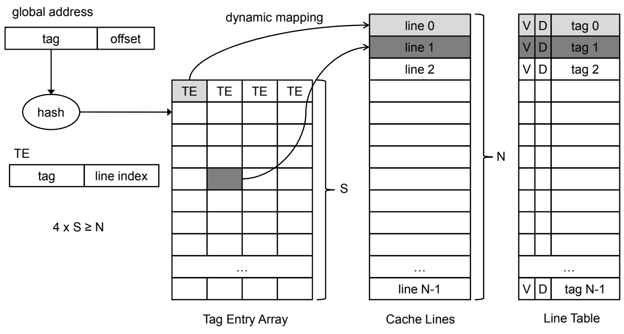


Figure 4. The Extended Set-Index Cache.

4.3 The Extended Set-Index Cache

Figure 4 shows the structure of the ESC. Its structure is similar to the 4-way set-associative cache. However, it decouples the TEs and the data blocks. The mapping between the TEs and the cache lines is determined at run time on demand. Moreover, the number of TEs can be greater than the number of cache lines (i.e., $4 \times S \geq N$). Each TE contains a tag and a cache line index (i.e., the local address of the data block). The line index fields record the mapping between the TEs and cache lines. Each cache line has an entry in the cache line table. The line table entry (LTE) contains a valid bit, a dirty bit, and a tag that is identical to the tag saved in the corresponding TE. The tag is the global address of the data block saved in the cache line.

When the cache receives a memory request with a global address x, the process of determining whether the request is a hit or miss is the same as that for the 4-way set-associative cache.

On a hit, the local address is computed by adding the block offset to the local address of the cache line saved in the matching TE. Otherwise, the miss handler is invoked. It selects a victim among the cache lines following the predefined replacement policy. If the victim's dirty bit is set, it flushes the data block of the victim to main memory using the tag (the global address) saved in the LTE. Then, the victim's TE is located by performing a hash operation and SIMD lookup using the same tag from the LTE. The victim's line index saved in the TE is invalidated. Next, the miss handler fetches the requested data block from main memory. Finally, it sets the corresponding TE and LTE fields appropriately.

Instead of saving the tag for each data block in its LTE, a pointer to the cache line's TE can be saved in the LTE. In this case, when a victim is selected, its tag (its global address) is accessed by following the pointer in the victim's LTE, and the victim data block is flushed to main memory. Then the line index saved in the TE can be invalidated. It is implementation dependent whether to save the tag in the LTE or to save the pointer to the TE. However, note that when the tag is saved in the LTE, the handler can use it to overlap the DMA operation for flushing the victim and the computation required to find the victim's TE.

The replacement policy applies to the cache lines as if the ESC were a fully associative cache. An exception is the case when the requested set has no empty TE. In this case, a victim is selected in the same way as in the 4-way set-associative cache.

The benefit of having the number of TEs greater than the number of cache lines is that collisions between addresses can be significantly reduced compared to the 4-way set-associative cache with the same number of cache lines. This implies that the ESC can reduce the effect of conflict misses which often adversely impact the performance of set-associative caches. With a sufficiently large number of TEs, we find the number of misses in the ESC to be close to the number of misses in the fully associative cache with the same number of cache lines.

Another benefit is that the ESC can use any number of cache lines less than or equal to the number of TEs. This improves the utilization of the accelerator core's local memory space.

Furthermore, since the ESC is not constrained to use a number of TEs equal to the number of cache lines, we can choose the number of sets to be equal to a power of 2. This enables us to use a low cost bit-wise AND operation in the set index computation. Making the number of sets equal to a power of 2 is easier than making the number of cache lines equal to a power of 2 because a TE consumes less space than a cache line.

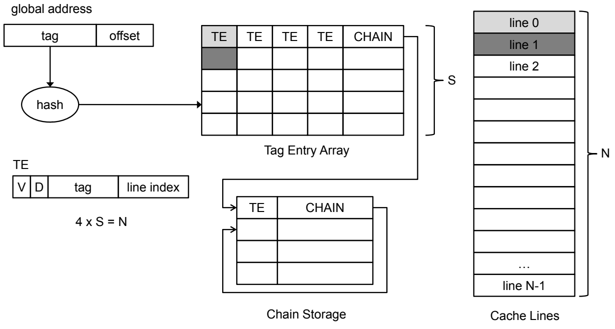


Figure 5. The Indirect index cache.

4.4 The Indirect Index Cache

Hallnor *et al.*[7] propose the indirect index cache (IIC). It is a hardware cache, and Figure 5 shows its structure. Similar to the ESC, it is based on the 4-way set-associative cache, and its mapping between TEs and cache lines is dynamically determined. The difference is that it has a chain pointer in each set to extend the number of ways in the set when there is no empty slot available in the set.

Table 1. Tag comparison overhead and number of misses.

	FAC	4WC	IIC	ESC
Tag comparison overhead	High	Low	Medium	Low
Expected # of misses	Low	Medium	Low	Low

Table 2. Computational overhead and number of misses.

	FIFO	Clock	Reuse	LRU
Searching overhead for a victim (when the # of lines is large)	Low	Medium-low	Medium-high	High
Searching overhead for a victim (when the # of lines is small)	Low	Medium-low	Medium-high	Low
Bookkeeping overhead	Low	Medium-low	High	Medium-high
Expected # of misses	High	Medium-high	Medium-low	Low

When there is a memory request and there is no empty TE available in the requested set, the IIC allocates a TE from the chain storage for the request, and links it to the TE chains in the set. The size of the chain storage is $N-4$, where N is the number of cache lines. Note that 4 TEs are already allocated statically and we do not need storage for more than N chained TEs. When the IIC performs the tag comparison for a request, if there is no matching tag in the set of statically allocated TEs and the requested set has TE chains, it needs to search the TE chain for a matching tag by following the chain pointers.

The IIC is essentially the same as a fully associative cache. The pointer chasing incurs significant overhead when the IIC is implemented in software. Increasing the number of ways in a set implies more tag comparison overhead due to the sequential nature of the tag comparison when implemented in software.

Table 1 summarizes the relative computational overhead of tag comparison and the degree of the number of misses expected by the fully associative cache (FAC), 4-way set-associative cache (4WC), IIC, and ESC when they have the same number of cache lines and are implemented in software.

4.5 Using a Fetch Buffer

In our SMC implementation, the requested data block cannot be fetched into the cache until the selected victim is flushed to main memory to make room for the fetched data block. This means that victim selection, flushing the victim to main memory, and fetching the requested data block occur sequentially in our implementation. However, flushing the victim to main memory and fetching the requested data block can be overlapped if we use a fetch buffer. Moreover, victim selection and fetching the requested data block can also be overlapped.

Therefore, we modify our implementation to maximize computation and communication overlap by using a fetch buffer, as follows. When a miss occurs, the requested data block is fetched into the fetch buffer. In parallel, a victim cache line is selected and flushed to main memory. This works with the ESC and IIC perfectly because the mapping between TEs and cache lines is dynamically determined, but this does not work with the fully associative and 4-way set-associative caches. To make it work for them, we add the line index field to each TE and dynamically set the line index field. The local address of the current fetch buffer becomes the line index of the requested TE, and the space made available by the victim cache line becomes the fetch buffer for the next miss.

4.6 Replacement Policies for SMCs

One important factor that affects cache performance is the replacement policy that determines which line will be evicted when the cache is full. In this section, we describe some representative replacement policies that are applicable to SMCs: first-in-first-out (FIFO), least recently used (LRU), and the clock algorithm.

FIFO selects a victim according to the order in which the cache lines are fetched into the cache. FIFO can be applicable across all cache lines (for fully associative caches), or across cache lines in a set (for set-associative caches).

In general, LRU has a lower miss rate than FIFO. Similar to FIFO, LRU can be applicable across all cache lines (for fully associative caches), or across cache lines in a set (for set-associative caches). To implement LRU, a timestamp needs to be maintained for each cache line. Whenever a cache line is accessed, its timestamp is updated with the current timestamp. On a cache miss, the cache line with the earliest timestamp will be selected as the victim. There are two types of computational overhead in the software implementation of LRU. One is the overhead of maintaining the timestamp whenever a cache line is accessed. The other is the overhead of comparing timestamps to find the earliest one.

Instead of using timestamps, one can maintain a stack whose bottom contains a pointer to the least recently used cache line. Whenever a cache line is accessed, it is moved to the top of the stack. In either case, the software implementation of LRU is costly for large number of cache lines due to overheads for searching and bookkeeping.

The clock algorithm is an approximation of the LRU replacement policy with a low maintenance overhead[25]. It maintains a reference bit for each cache line and sets the reference bit when the line is accessed. It also maintains a pointer to the cache lines. On a cache miss, it looks for an unreferenced line by moving the line pointer in a round-robin manner. When the pointer moves to the next cache line, it checks the line's reference bit and resets the bit if the line has been referenced.

Qureshi *et al.*[21] propose a frequency based global replacement scheme, called *reuse replacement*, for their V-Way cache. Every cache line is associated with a two-bit reuse counter. A cache line pointer is also maintained. When a cache line is newly filled, the reuse counter associated with the line is initialized to zero. For each subsequent access to the line, the reuse counter is incremented using saturating arithmetic. On a miss, starting with the line pointed to by the cache line pointer and proceeding in a round-robin manner, the reuse replacement algorithm linearly searches for a counter whose value is equal to zero. The algorithm tests and decrements each non-zero reuse counter until a victim is found. The reuse replacement algorithm is a closer approximation of the LRU replacement policy than the clock algorithm, but it incurs more overhead due to counter maintenance.

Since the ESC has a relatively large cache line size (i.e., a few kilobytes) that results in a relatively small number of cache lines (i.e., several tens), the original reuse replacement algorithm may introduce more misses by selecting the most recently used cache line whose counter value is zero. To avoid this, we slightly modify the original reuse replacement policy. When a cache line is newly filled, its reuse counter is initialized to a non-zero value (less than or equal to three), to prevent the most recently filled cache line from being kicked out from the cache on the next miss.

Hallnor *et al.* [7] also propose a replacement policy called *generational replacement* that is implemented in software for the IIC. Generational replacement uses a reference bit and is a variation of the clock algorithm. It maintains priority pools of cache lines and chooses an unreferenced line as a victim from the lowest priority pool. A pool is implemented using a linked list. On a cache miss, after checking the head of each pool, if the head was referenced, it is moved to the tail of the one-level higher pool. If not, it is moved to the tail of the one-level lower pool. Generational replacement is too costly to implement in software due to the bookkeeping overhead for priority pools. In addition, it does not fit the ESC design when the requested set is full, because it requires pointer-chasing linked lists to search for the victim cache line with the lowest priority in the set. This results in a significant overhead in the ESC implementation.

Table 2 summarizes the relative computational overhead and the degree of the number of misses expected by the replacement policies we use.

4.7 Thrashing Avoidance

The primary targets for our runtime on heterogeneous multicore architectures are applications with loop-level parallelism. When the working set of a parallel loop is larger than the cache, it causes thrashing and generates many cache misses. To solve this problem, we apply loop distribution[1] or loop fission to certain parallel loops in the applications If the number of different global variables (e.g., global arrays) in a parallel loop is larger than the number of cache lines in the SMC, the probability of thrashing is very high. We apply loop distribution to this type of parallel loop. Loop distribution splits a loop into multiple loops having the same index range, but each split loop executes only a part of the original loop body. This technique decreases the size of the working set, and as a result, thrashing can be avoided. We apply loop distribution to the parallel loops in swim when we evaluate the SMCs in Section 6.

5 Adaptive Execution Strategies

Data access patterns differ from application to application. Even different regions in an application have different degrees of locality. Therefore, a single cache line size or a single replacement policy does not fit all applications or all regions in an application. One of the advantages of using an SMC is the flexibility to adjust its parameters. For example, one can adjust the cache line size or replacement policy on demand at run time depending on the access patterns of the application.

However, it is difficult for the programmer to identify an optimal cache line size or replacement policy by just looking at the application code. Instead of relying on the programmer, we propose an adaptive execution algorithm to identify an optimal cache line size and replacement policy at run time for each region in an application. The regions considered are parallel loops whose iterations are distributed across multiple accelerator cores in our heterogeneous multicore architecture. It is often the case that these parallel loops are invoked many times during program execution, thus enabling our runtime to learn and adapt to characteristics specific to the loop.

To measure the performance of the parallel loop with different cache line sizes or replacement policies, we use execution time per iteration (TPI) as our metric. This metric is robust against workload variation (i.e., access pattern variation) in a parallel loop across invocations of the loop. However, the metric is not sensitive to workload variation across iterations of the loop. In general, we

```
/*********************************************************/
/***  The levels of the line size: 0, 1, 2, 3, and 4  ***/
/***  Replacement policies: CLOCK, LRU, and FIFO      ***/
/*********************************************************/

/***   INITIALIZATION   ***/
MAX_LINE_LEVEL      : 4
MAX_NUM_REPLACEMENT : 3
initial_line_size   : 2
candidate_line_size : 2
opt_line_size       : 2
/*** -1: decrease, +1: increase  ***/
direction           : -1
/*** FALSE: not done, TRUE: done ***/
optimal             : FALSE
replacement         : CLOCK
opt_replacement     : CLOCK
replacement_trial   : 1
T_iter              : infinite

/***   ADAPTIVE EXECUTION   ***/
if( optimal ) {
   line_size = opt_line_size;
   if( replacement_trial >= MAX_NUM_REPLACEMENT )
      replacement = opt_replacement;
   else {
      replacement = next replacement policy;
      timer on;
   }
} else {
   line_size = candidate_line_size;
   timer on;
}

set the cache's line size with line_size and
   replacement policy with replacement;
execute the loop in parallel;

if( !optimal ) {  /* STAGE 1: select the line size */
   timer off;
   new_T_iter = the execution time per iteration;
   if( new_T_iter < T_iter ) {
      T_iter = new_T_iter;
      opt_line_size = line_size;
      candidate_line_size = line_size + direction;
      if( candidate_line_size < 0 or
          candidate_line_size > MAX_LINE_LEVEL )
         optimal = TRUE;
   } else {
      if( direction < 0 ) {
         if( | line_size - opt_line_size | > 1 or
             initial_line_size == MAX_LINE_LEVEL )
            optimal = TRUE;
         else {
            direction = +1;
            candidate_line_size = initial_line_size + 1;
         }
      } else optimal = TRUE;  /* we tried all the levels */
   }
} else {  /* STAGE 2: select the replacement policy */
   if( replacement_trial < MAX_NUM_REPLACEMENT ) {
      timer off;
      new_T_iter = the execution time per iteration;
      if( new_T_iter < T_iter ) {
         T_iter = new_T_iter;
         opt_replacement = replacement;
      }
      replacement_trial++;
   }
}
```

Figure 6. The adaptive execution algorithm to determine an optimal cache line size and replacement policy for each parallel loop.

rarely find workload variation across iterations of a parallel loop in our target application domain.

Figure 6 is our adaptive execution algorithm to find the optimal cache line size and replacement policy at runtime. The algorithm is applied to each parallel loop independently. The algorithm consists of two separate stages. The first stage looks for the optimal cache line size and the second looks for the optimal replacement policy.

The first stage has five levels (level 0 to level 4) of different cache line sizes, with each higher level corresponding to a larger cache line size, and the replacement policy is fixed to be the clock algorithm. It starts executing the parallel loop with the initial level (i.e., level 2) when the loop is first invoked. In this first invocation, it measures the TPI, TPI_1. When the loop is invoked the next time, it lowers the line size by one level. It measures the TPI, TPI_2, for this second invocation. Then the algorithm compares TPI_1 with TPI_2. If TPI_1 is greater than TPI_2, this tells us that a smaller cache line is better for the loop. In the next invocation of the loop, the algorithm lowers the cache line size by one level again and compares the current TPI with the previous TPI. If the previous TPI is greater than the current TPI, it determines that the current cache line size is the optimal for the loop, else the cache line size corresponding to level 1 is optimal. If TPI_1 is smaller than TPI_2, the algorithm searches for the optimal cache line size in the opposite direction.

After selecting the optimal line size, the second stage of the algorithm rolls in. For the next two invocations, the loop is executed with the optimal line size and LRU, and then with the optimal line size and FIFO. The algorithm compares the three TPIs of different replacement policies and selects the replacement policy that has the lowest TPI as the optimal one. The remaining invocations of the loop are executed with the selected optimal cache line size and replacement policy.

Table 3. System configuration.

System	Cell Blade Server
Configuration	Dual 3.2GHz Cell BE, 8 SPEs each (only one processor is used) 512KB L2 cache, 2GB main memory RedHat Linux ES 5.1
Compiler	IBM XL Cell ppuxlc, -O5 IBM XL Cell spuxlc, -O5 IBM XL Cell ppu32-embedspu
Library	Cell SDK 3.0

Table 4. Applications used.

Application	Source	Input
CG	NPB	class A
equake	SPEC	reference
FT	NPB	class A
IS	NPB	class A
jacobi	OpenMP	2000 × 2000
md	OpenMP	8192 particles
MG	NPB	class A
swim	SPEC	reference

6 Evaluation

In this section, we evaluate the effectiveness of our SMCs and adaptive execution strategy.

6.1 Evaluation Environments

We implement the SMCs and the adaptive execution strategy on the Cell BE processor. The Cell BE processor consists of one Power Processor Element (PPE) and eight Synergistic Processor

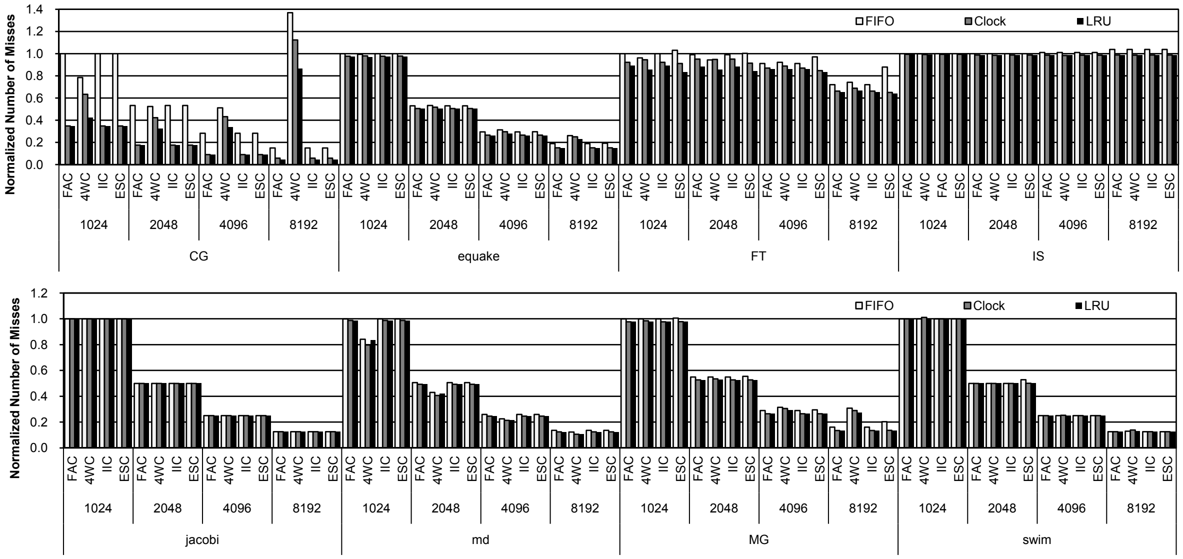


Figure 7. Normalized number of misses.

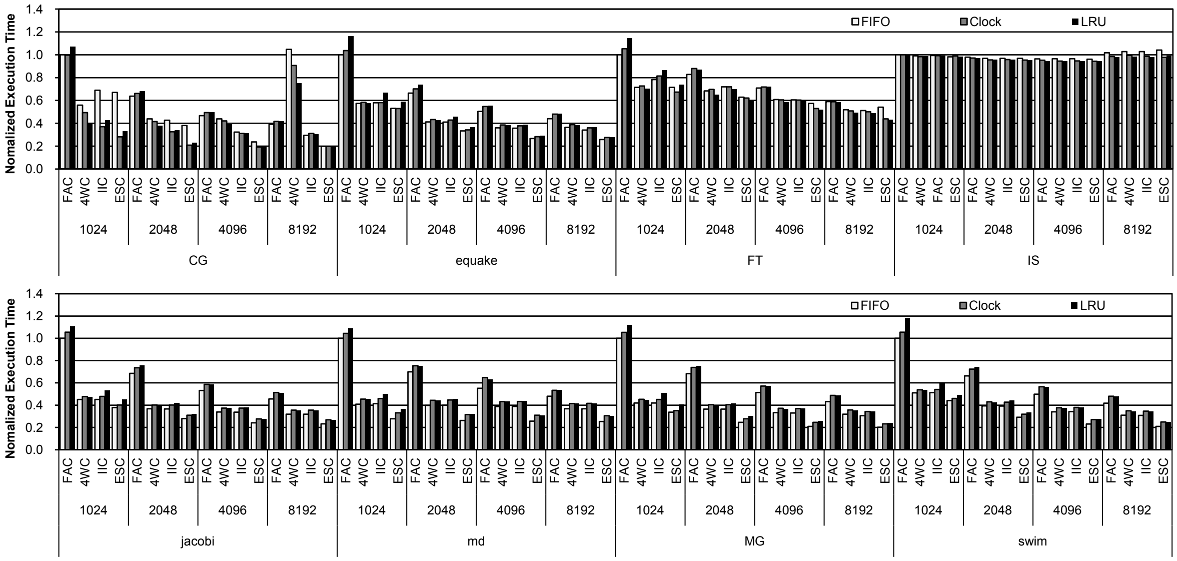


Figure 8. Normalized execution time.

Table 5. Miss rates (%).

		FAC			4WC			IIC			ESC					FAC			4WC			IIC			ESC		
		FIFO	Clock	LRU	FIFO	Clock	LRU	FIFO	Clock	LRU	FIFO	Clock	LRU			FIFO	Clock	LRU	FIFO	Clock	LRU	FIFO	Clock	LRU	FIFO	Clock	LRU
CG	1024	1.21	0.42	0.42	0.95	0.77	0.51	1.21	0.42	0.42	1.21	0.42	0.42	jacobi	1024	0.44	0.44	0.44	0.44	0.44	0.44	0.44	0.44	0.44	0.44	0.44	0.44
	2048	0.64	0.21	0.21	0.63	0.51	0.39	0.64	0.21	0.21	0.64	0.21	0.21		2048	0.22	0.22	0.22	0.22	0.22	0.22	0.22	0.22	0.22	0.22	0.22	0.22
	4096	0.34	0.11	0.11	0.62	0.52	0.41	0.34	0.11	0.11	0.34	0.11	0.11		4096	0.11	0.11	0.11	0.11	0.11	0.11	0.11	0.11	0.11	0.11	0.11	0.11
	8192	0.18	0.07	0.05	1.65	1.36	1.04	0.18	0.07	0.05	0.18	0.07	0.05		8192	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05
equake	1024	0.74	0.72	0.72	0.73	0.72	0.72	0.74	0.72	0.72	0.74	0.72	0.72	md	1024	0.20	0.20	0.20	0.17	0.16	0.16	0.20	0.20	0.20	0.20	0.20	0.20
	2048	0.39	0.37	0.37	0.39	0.38	0.37	0.39	0.37	0.37	0.39	0.37	0.37		2048	0.10	0.10	0.10	0.09	0.08	0.08	0.10	0.10	0.10	0.10	0.10	0.10
	4096	0.22	0.20	0.19	0.23	0.22	0.21	0.22	0.20	0.19	0.22	0.20	0.19		4096	0.05	0.05	0.05	0.04	0.04	0.04	0.05	0.05	0.05	0.05	0.05	0.05
	8192	0.14	0.11	0.11	0.19	0.18	0.17	0.14	0.11	0.11	0.14	0.11	0.11		8192	0.03	0.02	0.02	0.02	0.02	0.02	0.03	0.02	0.02	0.03	0.02	0.02
FT	1024	1.04	0.96	0.93	1.00	0.98	0.89	1.04	0.96	0.93	1.07	0.95	0.87	MG	1024	0.40	0.39	0.39	0.40	0.39	0.39	0.40	0.39	0.39	0.40	0.39	0.39
	2048	1.03	0.99	0.92	0.98	0.99	0.89	1.03	0.99	0.92	1.04	0.95	0.88		2048	0.22	0.21	0.21	0.22	0.21	0.21	0.22	0.21	0.21	0.22	0.21	0.21
	4096	0.95	0.91	0.89	0.96	0.92	0.90	0.95	0.91	0.89	1.01	0.88	0.87		4096	0.11	0.11	0.10	0.13	0.12	0.12	0.11	0.11	0.10	0.12	0.11	0.10
	8192	0.75	0.69	0.68	0.77	0.72	0.69	0.75	0.69	0.68	0.92	0.68	0.67		8192	0.06	0.05	0.05	0.12	0.12	0.11	0.06	0.05	0.05	0.08	0.05	0.05
IS	1024	12.59	12.52	12.51	12.58	12.49	12.51	12.59	12.52	12.51	12.59	12.52	12.51	swim	1024	0.44	0.44	0.44	0.44	0.45	0.44	0.44	0.44	0.44	0.44	0.44	0.44
	2048	12.59	12.44	12.43	12.59	12.41	12.42	12.59	12.44	12.43	12.59	12.44	12.43		2048	0.22	0.22	0.22	0.22	0.22	0.22	0.22	0.22	0.22	0.23	0.22	0.22
	4096	12.73	12.41	12.40	12.73	12.39	12.40	12.73	12.41	12.40	12.73	12.41	12.40		4096	0.11	0.11	0.11	0.11	0.11	0.11	0.11	0.11	0.11	0.11	0.11	0.11
	8192	13.07	12.44	12.43	13.08	12.42	12.43	13.07	12.44	12.43	13.07	12.44	12.43		8192	0.06	0.06	0.06	0.06	0.06	0.06	0.06	0.06	0.06	0.06	0.06	0.06

Elements (SPEs). PPE has two levels of cache hierarchy that are coherent with the main memory, but each SPE has 256KB of local memory without any cache. An SPE can transfer data from/to the external main memory using an asynchronous DMA operation, and it supports 128-bit SIMD instructions.

In our evaluation, we use the runtime system that provides programmers with an illusion of a global shared address space on the Cell BE processor. This runtime was developed by Lee *et al.*[14], and it is briefly described in Section 3. The SMC is a critical component of this runtime system. Table 3 shows the configuration of our evaluation system.

Eight OpenMP applications written in C are ported to our runtime environment and used in our evaluation. These applications include benchmarks from the SPEC OpenMP suite[27], OpenMP website[20], and the NAS Parallel Benchmarks suite[18]. They are summarized in Table 4. The iterations in each parallel loop are distributed among 8 SPEs and they are executed in parallel. Each chunk of iterations on an SPE uses the SMC to access main memory. The sequential regions are executed on the PPE. To exclude the effect of coherence misses, the SMC is flushed to main memory at the end of each parallel loop.

6.2 Number of Misses and Execution Time

Figure 7 shows the normalized number of misses of the fully associative cache (FAC), 4-way set associative cache (4WC), IIC, and ESC with FIFO, Clock, and LRU replacement policies. All the caches have the fetch buffer. To fully utilize the 256KB local memory, each cache has 160KB of space for cache lines. The remaining space is used by code and stack variables for the SPE. Thus, the number of sets is not a power of two in the fully associative cache, 4-way set-associative cache, and IIC. For the ESC, the number of sets is set to $4 \times K$, where K is the power of 2 that is closest to the number of sets in the 4-way set-associative cache. We vary the cache line size from 1KB to 8KB. We do not use the 16KB line because it does not fit the 4-way set-associative cache with 160KB line space, even though 16KB is the maximum amount of data that can be transferred with a single DMA operation in the Cell BE processor. All the numbers are normalized to the case of the fully associative cache with FIFO replacement policy and with 1KB line size. Table 5 shows the miss rate and Figure 8 shows the normalized execution time for the same setting as in Figure 7.

In Figure 7 and Table 5, we see that the number of misses and miss rate in the IIC and ESC are almost the same as those in the FAC, independent of the cache line size and the replacement policy. The number of misses in 4WC is sometimes smaller than that of the FAC (e.g., 1KB of md), and sometimes much larger than that of the FAC (e.g., 8KB of CG). The number of misses in 4WC depends on the line size because 4WC is very sensitive to conflict misses.

We also see that LRU is the best replacement policy in terms of the number of misses, independent of the cache type and the line size. The next best is the clock algorithm. The clock algorithm is often as good as LRU. The worst is FIFO.

For jacobi, there is no variation in the number of misses across the caches or the replacement policies because its accesses are mostly sequential accesses and there is no temporal locality in them. IS is an irregular application and its access pattern is subscript of subscript. Thus, the number of misses it suffers does not largely depend on the cache type or the line size. FT is also an irregular application and the trend of its number of misses is similar to that of IS.

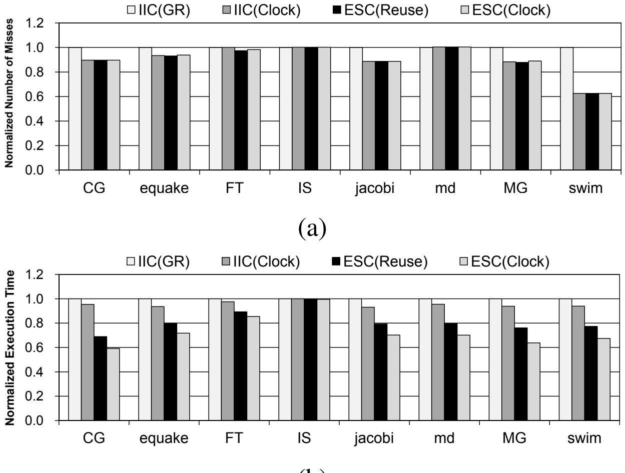
When we consider the execution time in Figure 8, the ESC is the best and the FAC is the worst, independent of the replacement policy and the line size. The FAC and the IIC are worse than the ESC even though their number of misses is almost the same as that of the ESC. This is due to computational overhead. There is a lot of computational overhead due to searching for the matching tag with a linear search algorithm in the FAC. For the IIC, the overhead of chasing pointers for TEs offsets the gains due to a reduced number of misses.

The execution time of FIFO is the best when its number of misses is close to those of the clock algorithm and LRU (i.e., for equake, jacobi, md, MG, and swim). This is because FIFO has the lowest computational overhead to select a victim cache line and no bookkeeping overhead.

When the number of misses is close to that of LRU and the line size is relatively small (1KB, 2KB, and 4KB), the clock algorithm is often better than or as good as LRU. With a relatively small cache line size, the computation overhead of LRU is relatively high because the number of cache lines to be searched for the earliest timestamp becomes large.

We see that both the number of misses and the computational overhead from cache management determine the performance of SMCs. In addition, a single line size or a single replacement policy does not fit all applications.

The ESC gives the number of misses and miss rate comparable to those of the fully associative cache and has a low computational overhead to manage it. Both of them synergistically affect the performance of the ESC.



(a)

(b)

Figure 9. Comparison between the IIC with the generational replacement policy, the IIC with the clock algorithm, the ESC with the reuse replacement policy, and the ESC with the clock algorithm. (a) Normalized number of misses. (b) Normalized execution time.

6.3 Generational and Reuse Replacement Policies

Figure 9 shows the result of comparison between the IIC with the generational replacement policy (IIC(GR)) (with four priority pools of cache lines), the IIC with the clock algorithm (IIC(Clock)), the ESC with the reuse algorithm (ESC(Reuse)), and the ESC with the clock algorithm (ESC(Clock)). For this experiment, the best line size is selected for each application based on the results in Figure 8. The generational replacement policy is proposed by Hallnor *et al.*[7] and the reuse replacement policy is proposed by Qureshi *et al.*[21] for hardware caches. They are described in Section 4.6. Figure 9 (a) shows the number of misses and (b) shows the execution time. All numbers are normalized to IIC(GR).

In Figure 9 (a), the number of misses in IIC(Clock) is smaller than that of IIC(GR). IIC(Clock) and ESC(Clock) have the same number of misses for all applications because the number of misses in both the IIC and ESC is close to that in the fully associative cache with the same replacement policy. As mentioned before, FT and IS are irregular applications and the number of misses they suffer does not largely depend on the cache type.

For all cases but IS, IIC(Clock) is faster than IIC(GR) due to the reduced number of misses. Even though the overhead of victim selection in IIC(Clock) is much less than that in IIC(GR), the reduction in the execution time is not that significant. This is because the overhead of tag comparison (i.e., pointer chasing) is relatively high in the IIC. For all cases but IS, ESC(Clock) is much faster than IIC(Clock) due to the lower overhead of tag comparison in the ESC.

As expected, the number of misses in ESC(Reuse) is slightly smaller than or equal to that in ESC(Clock). However, ESC(Reuse) is not faster than ESC(Clock) because of the bookkeeping and victim selection overhead.

Table 6. The effect of the number of sets on miss rates.

	FAC	ESC					
		8	10	12	14	16	32
CG	0.45	2.19	0.60	0.50	0.51	0.45	0.45
equake	0.20	0.22	0.21	0.21	0.20	0.20	0.20

6.4 The Number of Sets in the ESC

Table 6 shows the effect of the number of sets (say S) on the miss rate of CG and equake for the ESC. The ESC has 128KB of fixed space for cache lines and the line size is 4KB in this experiment (i.e., the number of lines $= 128\text{KB}/4\text{KB} = 32$, and it is a power of two). Its replacement policy is LRU. We vary S from 8 to 32. When $S = 8$, the number of TEs in the ESC is the same as that of the 4-way set-associative cache. We compare the miss rate of the ESC to that of the fully associative cache (FAC) with the same LRU replacement policy.

Among the eight applications, only CG and equake are sensitive to the number of sets (i.e., they suffer from conflict misses). As the number of sets increases, the miss rate of the ESC approaches that of the the fully associative cache for CG and equake. When the number of sets is double that of the 4-way set-associative cache, the miss rate is the same as that of the fully associative cache. The miss rate of the other applications is the same as that of the fully associative cache when $S = 8$ and does not vary as the number of sets increases.

6.5 Fetch Buffer

Figure 10 shows the normalized execution time of each application for the ESC with the fetch buffer. The experimental setting is the same as that in Figure 7. The execution time is normalized to the case without the fetch buffer. As described in Section 4.5, the fetch buffer maximizes computation and communication overlap. We see that the fetch buffer reduces execution time significantly (up to 26%). However, as the line size increases, its effect on execution time decreases. Since the number of misses decreases as the line size increases (Figure 7), there are less opportunities for computation and communication overlap.

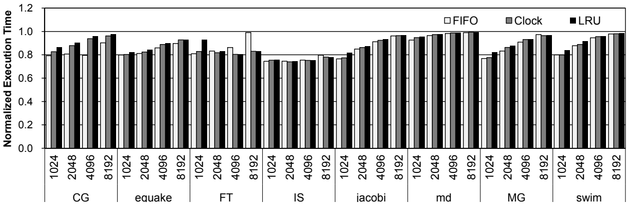


Figure 10. Normalized execution time of each application for the ESC with the fetch buffer.

Table 7. Hit latency in cycles.

	FIFO	Clock	LRU
1024	103	113	113
2048	101	111	111
4096	101	111	111
8192	101	111	111

6.6 Hit Latency and Miss Penalty

Table 7 shows the hit latency of the ESC in core clock cycles (3.2GHz) for different line sizes and different replacement policies. Since the computational overhead is incurred mainly on a miss, the hit latency across different replacement policies and different line sizes is almost the same. The hit latency of FIFO is the smallest because FIFO does not need to maintain the reference bit or timestamp for each access. Since the fetch buffer, line size, and DMA latency significantly affect the miss penalty, we cannot exactly measure the miss penalty for each case. Approximately, it is in the order of a few hundred micro-seconds. As the line size increases, the miss penalty also increases due to the latency of DMA transfer, even though the computational overhead decreases due to the smaller number of cache lines. The computational overhead of LRU replacement is the highest.

6.7 Storage Overhead

Table 8 shows the extra storage overhead for each cache when the size of the storage for N cache lines is excluded. To compute the extra storage overhead, we assume the following:

- To avoid additional masking operations, the tag, line index, and chain pointer are all 4-byte integers. The valid and dirty bits are one byte each.
- All the caches use the fetch buffer. Thus, each TE needs a line index field. Except the 4-way set-associative cache, each LTE needs a tag field to find the global address of the victim line.
- $S = N/4$ is the number of sets in the 4-way set associative cache and IIC. S' is the number of sets in the ESC. It is four times the closest power of two to S.

Table 9 shows the extra overhead computed for different cache line sizes with 160KB storage for the cache lines. Even though the ESC has the largest storage overhead, this overhead is less than 4% of the storage consumed by the cache lines.

Table 8. Extra storage overhead (in bytes).

	Fully associative	4-way set-associative	Indirect index	Extended set-index
TE	tag + line index = 8	tag + line index = 8	tag + line index = 8	tag + line index = 8
TE Array	$TE \times N = 8N$	$(4 \times TE) \times S = 8N$	$(4 \times TE +$ chain pointer) $\times S = 9N$	$(4 \times TE) \times S' = 32S'$
Chain Storage	-	-	$(TE +$ chain pointer) $\times (N - 4) = 12(N - 4)$	-
LTE	$V + D + tag = 6$	$V + D = 2$	$V + D + tag = 6$	$V + D + tag = 6$
Cache Line Table	$LTE \times N = 6N$	$LTE \times N = 2N$	$LTE \times N = 6N$	$LTE \times N = 6N$
Total	$14N$	$10N$	$27N - 48$	$32S' + 6N$

Table 9. Extra storage overhead with a 160KB cache line space (in bytes).

Line size	Fully associative	4-way set-associative	Indirect index	Extended set-index
8 KB (N = 20)	280	200 (S = 5)	492	632 (S' = 16)
4 KB (N = 40)	560	400 (S = 10)	1032	1264 (S' = 32)
2 KB (N = 80)	1120	800 (S = 20)	2112	2528 (S' = 64)
1 KB (N = 160)	2240	1600 (S = 40)	4272	5056 (S' = 128)

Table 10. Characteristics of each parallel loop.

Region	CG	equake	FT	IS	jacobi	md	MG	swim
#1	1KB FIFO (1, 0.0%)	4KB FIFO (3885, 3.7%)	16KB FIFO (2, 0.2%)	1KB LRU (1, 2.9%)	16KB FIFO (100, 16.1%)	16KB FIFO (11, 98.9%)	16KB FIFO (39, 1.2%)	16KB FIFO (100, 24.0%)
#2	1KB FIFO (1, 0.0%)	8KB FIFO (3885, 47.2%)	16KB LRU (6, 25.1%)	4KB LRU (10, 96.9%)	16KB FIFO (100, 81.5%)	8KB FIFO (10, 0.0%)	16KB LRU (119, 2.9%)	16KB FIFO (100, 42.7%)
#3	2KB FIFO (1, 0.0%)	8KB FIFO (3885, 31.0%)	16KB LRU (2, 17.2%)	16KB FIFO (1, 0.1%)			8KB FIFO (40, 19.0%)	16KB FIFO (1, 0.2%)
#4	4KB Clock (1, 6.0%)	8KB FIFO (3885, 14.5%)	16KB LRU (6, 51.4%)				4KB FIFO (42, 35.2%)	16KB FIFO (98, 29.9%)
#5	4KB Clock (15, 88.1%)		16KB LRU (6, 0.0%)				8KB FIFO (35, 3.6%)	16KB FIFO (10, 0.5%)
#6							8KB FIFO (35, 4.3%)	16KB FIFO (1, 0.3%)
#7							16KB FIFO (4, 1.3%)	16KB FIFO (1, 0.2%)
#8							16KB FIFO (2, 0.5%)	16KB FIFO (1, 0.3%)

6.8 Cache Line Size and Replacement Policy

Table 10 shows the optimal cache line size, optimal replacement policy, the number of invocations, and % execution time for each parallel loop in each application with the ESC. We measure the execution times of each loop with different line sizes and replacement policies, and compare them to determine the optimal size and policy. We see that they are different from parallel loop to parallel loop and from application to application. Consequently, a single line size or replacement policy does not fit all loops or all applications. It is very hard to determine the optimal line size or replacement policy by just looking at the code. Since most of the loops are invoked multiple times, our adaptive execution is useful in determining the optimal line size and replacement policy.

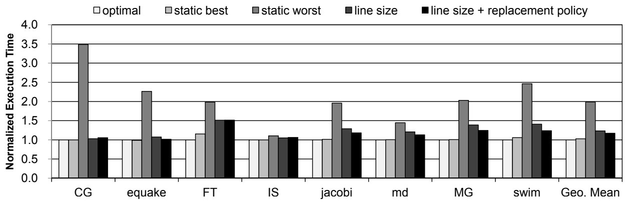


Figure 11. Normalized execution time of each application with our adaptive execution strategy.

6.9 Adaptive Execution

Figure 11 shows the normalized execution time of each application with our adaptive execution strategy. The bar labeled optimal for each application is the execution time measured when each loop in the application is executed with the optimal line size and replacement policy as shown in Table 10. The bar labeled static best for each application is the best execution time obtained in Figure 8 for different line sizes and replacement policies. Similarly, the bar labeled static worst is the worst execution time obtained in Figure 8 for different line sizes and replacement policies. The bar labeled line size is the execution time measured when the adaptive execution strategy is applied to each application only to select the optimal line size, and line size + replacement policy is for the case in which the adaptive execution strategy is applied to each application to select both the optimal line size and the optimal replacement policy. The execution time is normalized to optimal.

The result shows that line size and line size + replacement policy are on average 20% and 14% slower than optimal, respectively. The strategy line size + replacement policy is better than line size for all but CG and FT. For CG and FT, penalties paid by the suboptimal execution of some loops in line size + replacement policy are not amortized by the gains obtained from the optimal replacement policy that is selected by the adaptive execution. Without our adaptive strategy, the execution could use a suboptimal cache line size or replacement policy, and in some cases this may result in significant performance degradation (i.e., sometimes even 3 or 4 times slower than optimal). This shows that our adaptive execution strategy is quite effective.

SMCs are viable in general for currently existing heterogeneous multicores (e.g., the Cell BE architecture). In Lee *et al.*[14], when using the ESC for a mix of 12 regular and irregular OpenMP applications, the speedup is on average 2.57 with 1 PowerPC core and 8 SPEs (a single Cell processor) over a single PowerPC core. In some cases, the SMC may handle only a fraction of the application data, but it does enable programmability/performance for non-streaming or irregular data references.

7 Conclusions

We present a new software-managed cache design, called extended set-index cache, and adaptive execution strategies to dynamically select the optimal cache line size and replacement policy for each code region in an application. Our approach is applicable to all cores with access to both local and global memory in a multicore architecture, even though our implementation targets a specific heterogeneous multicore architecture. Software-managed caches are an aid to improving programmability/performance for such heterogeneous multicore architectures.

Unlike hardware caches, identifying a cache management algorithm that has a sufficiently low implementation complexity as well as a low cache miss rate is critical in the design and implementation of software-managed caches to achieve high performance. In addition, a single cache line size or a single replacement policy for the software-managed cache does not yield the best performance for all applications or all regions in an application.

The evaluation results with 8 OpenMP applications running on a Cell BE processor show that the extended set-index cache significantly reduces the number of misses (to be close to that of the fully associative cache), and it improves performance compared to other cache designs implemented in software. The results also show that our adaptive execution strategies work well with the software-managed cache.

References

[1] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures.* Morgan Kaufmann, 2002.

[2] J. Balart, M. Gonzalez, X. Martorell, E. Ayguade, Z. Sura, T. Chen, T. Zhang, K. O'brien, and K. O'Brien. A Novel Asynchronous Software Cache Implementation for the Cell-BE Processor. In *LCPC'07: Proceedings of the 20th International Workshop on Languages and Compilers for Parallel Computing*, October 2007.

[3] A. E. Eichenberger, J. K. O'Brien, K. M. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, M. K. Gschwind, R. Archambault, Y. Gao, and R. Koo. Using advanced compiler technology to exploit the performance of the Cell Broadband Engine$^{\text{TM}}$ architecture. *IBM Systems Journal*, 45(1):59–84, January 2006.

[4] I. Gelado, J. H. Kelm, S. Ryoo, S. S. Lumetta, N. Navarro, and W. mei Hwu. CUBA: An Architecture for Efficient CPU/Co-processor Data Communication. In *ICS'08: Proceedings of the 22nd International Conference on Supercomputing*, pages 299–308, June 2008.

[5] A. González, M. Valero, N. Topham, and J. M. Parcerisa. Eliminating Cache Conflict Misses Through XOR-Based Placement Functions. In *ICS'97: Proceedings of the 11th International Conference on Supercomputing*, pages 76–83, July 1997.

[6] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. Synergistic Processing in Cell's Multicore Architecture. *IEEE Micro*, 26(2):10–24, March/April 2006.

[7] E. G. Hallnor and S. K. Reinhardt. A Fully Associative Software-Managed Cache Design. In *ISCA'00: Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 107–116, June 2000.

[8] J. L. Hennessy and D. Patterson. *Computer Architectures: A Quantitative Approach.* Morgan Kaufmann, September 2006.

[9] IBM, Sony, and Toshiba. *Cell Broadband Engine Architecture*. IBM, October 2007. `http://www.ibm.com/developerworks/power/cell/`.

[10] N. P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *ISCA'90: Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 364–373, May 1990.

[11] M. Kharbutli, K. Irwin, Y. Solihin, and J. Lee. Using Prime Numbers for Cache Indexing to Eliminate Conflict Misses. In *HPCA-10:Proceedings of the 10th International Symposium on High Performance Computer Architecture*, pages 288–299, February 2004.

[12] M. Kharbutli, Y. Solihin, and J. Lee. Eliminating Conflict Misses Using Prime Number-Based Cache Indexing. *IEEE Transactions on Computers*, 54(5):573–586, May 2005.

[13] R. Kumar, D. M. Tullsen, N. P. Jouppi, and P. Ranganathan. Heterogeneous Chip Multiprocessors. *IEEE Computer*, 38(11):32–38, November 2005.

[14] J. Lee, S. Seo, C. Kim, J. Kim, P. Chun, Z. Sura, J. Kim, and S. Han. COMIC: A Coherent Shared Memory Interface for Cell BE. In *PACT'08: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 303–314, October 2008.

[15] J. Lee, Y. Solihin, and J. Torrellas. Automatically Mapping Code in an Intelligent Memory Architecture. In *HPCA-7:Proceedings of the 7th International Symposium on High Performance Computer Architecture*, pages 121–132, January 2001.

[16] J. Leverich, H. Arakida, A. Solomatnikov, A. Firoozshahian, M. Horowitz, and C. Kozyrakis. Comparing Memory Systems for Chip Multiprocessors. In *ISCA'07: Proceedings of the 34th Annual International Symposium on Computer Architecture*, pages 358–368, June 2007.

[17] J. E. Miller and A. Agarwal. Software-Based Instruction Caching for Embedded Processors. In *ASPLOS-XII: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 293–302, October 2006.

[18] NASA Advanced Supercomputing Division. NAS Parallel Benchmarks. `http://www.nas.nasa.gov/Resources/Software/npb.html`.

[19] NVIDIA. *NVIDIA CUDA Compute Unified Device Architecture*. NVIDIA, June 2008. `http://developer.download.nvidia.com`.

[20] OpenMP. OpenMP. `http://www.openmp.org`.

[21] M. K. Qureshi, D. Thompson, and Y. N. Patt. The V-Way Cache: Demand Based Associativity via Global Replacement. In *ISCA'05: Proceedings of the 32th Annual International Symposium on Computer Architecture*, pages 544–555, June 2005.

[22] B. R. Rau. Pseudo-Randomly Interleaved Memory. In *ISCA'91: Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 74–83, May 1991.

[23] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S.-Z. Ueng, J. A. Stratton, and W. mei W. Hwu. Program Optimization Space Pruning for a Multithreaded GPU. In *CGO'08: Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 195–204, April 2008.

[24] A. Seznec. A Case for Two-Way Skewed-Associative Caches. In *ISCA'93: Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 169–178, May 1993.

[25] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts.* Wiley, 2004.

[26] Y. Solihin, J. Lee, and J. Torrellas. Automatic Code Mapping on an Intelligent Memory Architecture. *IEEE Transactions on Computers*, 50(11):1248–1266, November 2001.

[27] Standard Performance Evaluation Corporation. SPEC OMP. `http://www.spec.org/omp/`.

[28] A. V. Veidenbaum, W. Tang, R. Gupta, A. Nicolau, and X. Ji. Adapting Cache Line Size to Application Behavior. In *ICS'99: Proceedings of the 13th International Conference on Supercomputing*, pages 145–154, June 1999.