Dual Cache Architecture for Low Cost and High Performance

Jung-Hoon Lee, Gi-Ho Park, and Shin-Dug Kim

We present a high performance cache structure with a hardware prefetching mechanism that enhances exploitation of spatial and temporal locality. Temporal locality is exploited by selectively moving small blocks into the direct-mapped cache after monitoring their activity in the spatial buffer. Spatial locality is enhanced by intelligently prefetching a neighboring block when a spatial buffer hit occurs. We show that the prefetch operation is highly accurate: over 90% of all prefetches generated are for blocks that are subsequently accessed. Our results show that the system enables the cache size to be reduced by a factor of four to eight relative to a conventional direct-mapped cache while maintaining similar performance.

Keywords: Memory hierarchy, dual data cache, temporal locality, spatial locality, prefetching.

Manuscript received Jan. 15, 2003; revised June 11, 2003.

This work was supported by postdoctoral fellowship program from Korea Science & Engineering Foundation (KOSEF).

Jung-Hoon Lee (phone: +82 2 2123 2718, email: ljh@yonsei.ac.kr) and Shin-Dug Kim (email: sdkim@yonsei.ac.kr) are with the Department of Computer Science, Yonsei University, Seoul, Korea.

Gi-Ho Park (email: gihopark@samsung.com) is with Samsung Electronics Co., Suwon, Korea.

I. Introduction

Cache memory is a key mechanism for improving overall system performance. A cache exploits the locality inherent in the reference stream of a typical application. Two primary types of locality are available, and the degree to which they can be exploited depends on program execution characteristics. Temporal locality relies on the greater probability that recently accessed data will be accessed again in the near future. Spatial locality refers to the tendency for adjacent or nearby memory locations to be referenced close together in time. However, most cache systems have shown a tendency to emphasize only one or the other of the two types of locality because they place contradictory requirements on the structure of the hardware.

Prefetching mechanisms can be used to reduce cache misses. Prefetching also reduces processor stall time by bringing data into the cache before its use, so that it can be accessed without delay. Hardware-based prefetching [1] requires some modification of the cache but little modification to the processor core. Its main advantage is that prefetches are handled dynamically at run time without compiler intervention. The drawbacks are that extra hardware is needed and that memory references for complex access patterns are difficult to predict. In contrast, software-based approaches [2] rely on compiler technology to perform static program analysis and to selectively insert prefetch instructions. The drawbacks are that non-negligible execution overhead is introduced due to the extra instructions, and that static analysis may miss some prefetch opportunities that are obvious at run-time.

Most hardware prefetching mechanisms generate a prefetch signal when either a cache hit or a miss occurs. Therefore, a large number of prefetch signals are generated, leading to higher power consumption and cache pollution [3]. Prefetching can also cause an increase in the memory cycles per instruction (MCPI) in the worst case. Our intelligent prefetching mechanism avoids these problems by reducing the prefetch generation rate. As a result, power consumption is also reduced and the increase in MCPI is negligible.

The proposed cache is constructed in three parts: a conventional direct-mapped cache with a small block size, a fully associative spatial buffer with a large block size at the same cache level, and a hardware prefetching unit. The improvement in performance is achieved by exploiting the basic characteristics of locality. Specifically, two different block sizes are used, i.e., a small block size to exploit temporal locality and a large block size, which is a multiple of the small block size, to exploit spatial locality. In addition, temporal locality is enhanced by selectively retaining blocks with a high probability of a repeated reference in the time domain, and spatial locality is enhanced by both a large fetch size and an intelligent prefetching mechanism.

The technique we use for enhancing temporal locality is to monitor the behavior of a block over a time interval before choosing to store it into the direct-mapped cache. In this way, instead of placing every missed block directly into the direct-mapped cache, we first load a large block including the missed block into the spatial buffer. The missed block is not moved into the direct-mapped cache until the large block is replaced from the spatial buffer, and then only if it has shown evidence of temporal locality while it was resident in the spatial buffer. Thus the proposed cache exploits information about utilization of the block that is obtained during this time interval. The interval itself is proportional to the number of entries in the spatial buffer. The behavior that we observe during this time enables us to determine which blocks show strong temporal locality.

II. Related Work

The stream cache [4] is a small fully-associative cache containing on the order of two to five lines of data. When a miss occurs, data are returned to the direct-mapped cache and then several consecutive words from the data stream are prefetched to the stream cache. To make better use of the stream cache, Jouppi uses a different replacement algorithm, that is, victim caching. The victim cache [4] reduces the delay associated with cache misses due to line conflicts. It has been shown to be effective in improving overall system performance while requiring only a small additional buffer. Conflict misses may also be reduced by increasing cache associativity. Selective victim caching [5] places incoming blocks selectively in the main cache or a small victim buffer using a prediction scheme. This scheme swaps a block from the victim buffer to the main cache based on its history of use. For instruction caches, the selective victim cache shows an effective performance improvement compared to conventional victim caches. However, it shows no performance improvement for data caches.

The selective cache [6] consists of a spatial cache with a large block size, a temporal cache with a small block size, and a locality prediction table. The data may be placed in just one of the two subcaches or may not be cached anywhere, depending on the predicted type of locality for a given memory access. The prediction is made by means of a history table, which is called a locality prediction table. The locality prediction mechanism is designed for numeric codes with constant stride vectors.

The split temporal/spatial cache (STS) [7] is organized as two parts, i.e., a spatial cache with a prefetch mechanism and a temporal cache. The temporal cache is organized as a two-level hierarchy, with a one-word block size at each level. The spatial cache has the usual block size and includes a hardware prefetching mechanism. At compile time, with some estimation of the access probability, data accesses are classified as exhibiting predominantly either temporal locality or spatial locality, and are tagged for one or the other of these caches.

The HP-7200 assist cache [8] provides a methodology that seeks to avoid both block interference and cache pollution before they happen. This design places the primary direct-mapped cache in parallel with a small fully associative buffer, guaranteeing single-cycle lookup at both units. Blocks requested from the cache controller, due to a cache miss or a prefetch, are first loaded into the assist buffer, and are only promoted into the direct-mapped cache if they exhibit temporal locality. Data with no temporal reuse bypass the direct-mapped cache and are moved directly back to memory with a FIFO replacement algorithm.

The NTS cache proposed by Rivers and Davidson [9] supplements the conventional direct-mapped cache with a parallel fully associative cache. This scheme separates the reference stream into temporal and non-temporal block references. Blocks are treated as non-temporal until they become temporal. Cache blocks that are identified as non-temporal when they are replaced are allocated to fully associative cache on subsequent requests. However, NTS caching does not allow swaps between cache A and cache B. This becomes critical when we have conflicts among temporal blocks or conflicts among non-temporal blocks. This degrades the performance by not utilizing cache space that might be available in the L1 cache. Although they use a mechanism similar to tag blocks as temporal, they improve this scheme by allowing both temporal and non-temporal blocks to reside in cache A and in cache B.

The difference between the proposed cache and these approaches is explained as follows. The proposed cache has different associativities and block sizes. We also include hardware prefetching. In contrast, the selective cache [6] and the STS cache [7] use the same associativity with different block sizes. The stream cache [4], victim cache [4], and assist cache [8] use different associativities with the same block size. Additionally, the hardware mechanisms of these schemes differ from each other in many aspects. First of all, the selective cache system has a locality prediction table that decides whether the requested data has either temporal locality or spatial locality. The STS cache uses a prefetching mechanism to exploit spatial locality but cannot exploit temporal locality effectively. The cited cache systems focus on how to detect the type of locality for a certain memory reference and how to deal with the references based on the predicted locality. The proposed cache effectively exploits both types of locality via a hardware control mechanism that employs a time interval during which a location is accessed from the spatial cache to determine whether it also has temporal locality. The HP-7200 assist cache system employs a time interval and a prefetching mechanism, but the information needed to detect temporal locality is provided statically by the compiler.

III. New Cache System

1. Motivation

Common design objectives for a cache are to improve utilization of the temporal and spatial locality inherent in applications. However no single cache organization exploits both temporal and spatial locality optimally because of their contradictory characteristics. Increasing block size reduces the number of cache blocks. Thus, conventional cache designs attempt to compromise by selecting an adequate block size. The cache's lack of adaptability to patterns of references with different types of locality poses a further drawback. A fixed block size results in suboptimal overall access time and bandwidth utilization because it fails to match the varying spatial and temporal localities across and within programs [10], [11].

A significant reason for using a direct-mapped cache with a small block size is to reduce power consumption and cycle time. To make up for the poor spatial locality of such a direct-mapped cache, we provide a fully associative spatial buffer with a large block size. The small block size exploits temporal locality and the large block size exploits spatial locality. Small blocks are first loaded as part of larger blocks that are fetched into the spatial buffer. We selectively extend the lifetime of those small blocks that exhibit high temporal locality by storing them in the direct-mapped cache. When a large block is replaced in the spatial buffer, small blocks belonging to it that have been accessed at least once during its residency are moved into the direct-mapped cache. This approach effectively reduces conflict misses and thrashing at the same time.

Making the block size of the direct-mapped cache as small as possible improves temporal locality by increasing the number of available entries for a given cache space. For example, the number of entries for 4B blocks is sixteen times more than for 64B blocks. Therefore, the lifetime of a data item in a direct-mapped cache with an s-byte block size is at most l/s times that for a cache with an l-byte ($l > s$) block size, for a given cache size. In conventional direct-mapped caches, the increase in spatial locality due to a larger block size gives a greater performance improvement than the increase in temporal locality resulting from increasing the number of entries. However, the spatial buffer of the proposed cache provides the necessary spatial locality, and thus we choose the smallest block size possible for the direct-mapped cache.

Fetching a large block when a cache miss occurs increases spatial locality. Our simulations show the probability that neighboring small blocks will be accessed during the lifetime of the initial block is more than 50% for most benchmarks. An intelligent prefetching approach that emphasizes prefetching of data that has exhibited evidence of spatial locality can also improve cache performance.

Instead of loading a missed block directly into the direct-mapped cache, we load it into the fully-associative spatial buffer. Then a time interval proportional to the number of entries in the buffer elapses before it is evicted. The small blocks, (which are individual words in the simple case) within the large block, that have been referenced during that time are moved into the direct-mapped cache when the large block is evicted.

2. Cache System Structure

The proposed cache structure is shown in Fig. 1. The direct-mapped cache is the main cache and its organization is similar to a traditional direct-mapped cache, but it has a smaller block size. The spatial buffer is designed so that each entry is a collection of several banks, each of which is the size of a block in the direct-mapped cache. The tag space of the buffer is a content addressable memory (CAM). The small blocks in each bank include a hit bit (H) to distinguish referenced blocks from unreferenced blocks. Each large block entry in the spatial buffer further has a prefetch bit (P), which directs the operation of the prefetch controller. The prefetch bits are used to dynamically generate prefetch operations.

When the CPU performs a memory reference, the direct-mapped cache and the spatial buffer are searched in parallel.

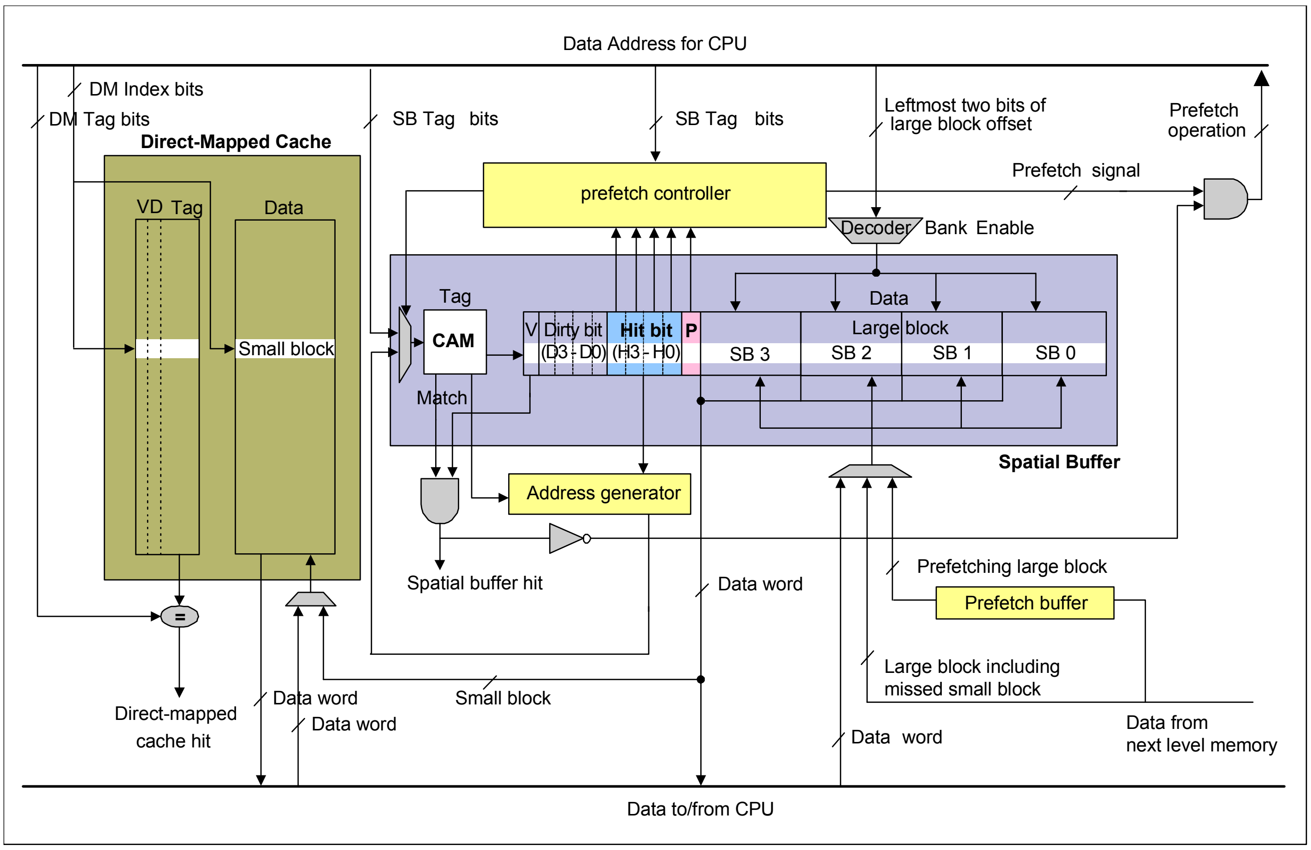

Fig. 1. Proposed cache structure.

We assumed that the tag match with the direct-mapped cache and the spatial buffer should be as fast as the time used in the direct-mapped cache only. A hit in the direct-mapped cache is processed in the same way as a hit in a conventional L1 cache. When a miss occurs in both the direct-mapped cache and the spatial buffer, a large block is fetched into the spatial buffer. If a reference misses in the direct-mapped cache but hits in the spatial buffer, its corresponding small block is simply fetched from the spatial buffer and the hit bit for that small block is set at the same time. The prefetch controller generates a prefetch signal when a large block is accessed and found to have a multiple hit bits set. Two operations are performed consecutively by the prefetch controller. Given a hit in the i-th large block, the first operation searches the tags of the spatial buffer to detect whether the $(i+1)$th large block is already present. A one-cycle penalty occurs in this case, but the overhead is negligible because prefetching is initiated in response to only about 1.5% to 2.5% of the total number of addresses generated by the CPU. Thus, the average MCPI is increased by about 0.06%. If the $(i+1)$th large block is not already in the spatial buffer, the second operation is performed: it is fetched into a prefetch buffer and the P bit in the i-th large block is set to prevent it from generating further prefetches. The number of prefetch buffer entries is assumed to be one.

If misses occur in both the direct-mapped cache and the spatial buffer, the cache controller initiates miss handling. When this occurs, a block that was already in the prefetch buffer is transferred to the spatial buffer. Therefore, the transfer time is hidden because 19 clock cycles are required for handling a miss. But the missed block may already be present in the prefetch buffer. Therefore, when the block in the prefetch buffer is transferred to the spatial buffer, the tag value should be simultaneously compared with the generated address. This comparison can be implemented either by using the comparator in the direct-mapped cache or an additional comparator. If the comparator shows a match, the requested data item is transmitted to the CPU and transferred to the spatial buffer at the same time. In addition, the ongoing miss signal should be canceled by the cache controller.

3. Operational Model of the Proposed Cache

Here we describe in detail the algorithms for managing the proposed cache. Prefetching is performed dynamically depending upon both the H and the P bits for a particular large block in the spatial buffer. On every memory access, both the direct-mapped cache and the spatial buffer are accessed at the same time. The different cases for the operational model are explained as in Fig. 1.

A. Case of Cache Hits

On every memory access, a hit may occur at the direct-mapped cache or the spatial buffer. First, consider the case of a hit in the direct-mapped cache. If a read access to the direct-mapped cache is a hit, the requested data item is transmitted to the CPU without delay. If a write access to the direct-mapped cache is a hit, the write operation is performed and the dirty bit for the block is set.

Second, consider the case of a hit in the spatial buffer. The data in the small block are sent to the CPU, and the hit bit is set to mark it as referenced. As an aside, we also reduce power consumption by using the most significant two bits of the large block offset to activate just one of the banks in the spatial buffer. For example, if the size of a small block is 8B and a large block is 32B, there are four banks in the spatial buffer. When a large block is accessed, and its P bit is in the reset state, a prefetch operation is initiated if a hit occurs in any bank of the spatial buffer and one or more hit bits are already set. At the same time, the tags of the spatial buffer are searched for the prefetch address to check whether it is already present, in which case the prefetch is squashed and the P bit is set. If the address is not in the spatial buffer, the prefetch controller generates the prefetch signal and the target address to fetch the large block into the prefetch buffer from the next level of memory. At the same time, the P bit of the original large block is set to prevent repetition of the prefetch. If the P bit of a large block is set, the consecutive large block must be present in either the spatial buffer or the prefetch buffer, and there is no need to search the tags within the spatial buffer.

If a block is in the prefetch buffer when a subsequent prefetch signal is generated as a result of a cache miss, then the miss stalls while the contents of the prefetch buffer are moved into the spatial buffer. According to our simulation results, this case almost never occurs, even for a prefetch buffer with just one entry. This is because the overall rate of prefetch operations is only about 0.3%, and the miss ratio is about 1.7%. Therefore, the probability for a prefetch to be initiated in this manner is about 6 times smaller than the miss ratio. Because the utilization of prefetched blocks is over 90%, we concluded that it is not worth adding hardware specifically to handle this rare case, relying instead on the existing cache stall mechanism.

When either cache misses, then while the cache controller is handling the miss, a large block is loaded into the spatial buffer from the prefetch buffer. All of the hit bits of the prefetched block are set to zero. Finally, if either cache misses while a prefetch operation is being performed, miss handling is deferred until the ongoing prefetch operation completes.

B. Case of Cache Misses

If a miss occurs in both caches, a large block including the missed small block is brought into the spatial buffer from the next level of memory. We use as an example an 8kB direct-mapped cache with a small block size of 8B and a 1kB spatial buffer with a large block size of 32B, so four sequential small blocks are contained within a 32B block. We consider two cases, depending on whether the spatial buffer is full.

- Case 1: The spatial buffer is not full.

If at least one entry in the spatial buffer is in the invalid state, a large block is fetched and stored in the spatial buffer. When a particular small block is accessed by the CPU, the corresponding hit bit is set to one. Thus, the hit bit of the small block identifies it as a referenced block.

- Case 2: The spatial buffer is full.

If the spatial buffer is full, the oldest large block is replaced. Each small block whose hit bit is set in the about-to-be-evicted large block is loaded into the direct-mapped cache, because those blocks have shown temporal locality. This loading time is also hidden by the miss-handling time. If the spatial buffer is full, the oldest entry is replaced according to a FIFO policy. At that point, the blocks in the entry whose hit bits are set are moved into the direct-mapped cache. Because these actions are accomplished while the cache controller is handling a miss, this operation does not introduce any additional delay. The move operations between the two caches are illustrated as follows. For our example configuration, when a 32-bit memory address is generated, such as FFFFFF80, in the direct-mapped cache, the tag field is 19 bits (A: 7FFFF), the index field is 10 bits (B: 3F0), and the offset field is 3 bits. In the spatial buffer, the tag field is 27 bits (C: 7FFFFFC) and the offset field is 5 bits. Therefore, the high order two bits of the large block offset are 00. These two bits are used to search one of the four banks selectively. If a miss occurs in both caches, data corresponding to the tag value of the large block (C) are fetched and only the hit bit of the first (1FFFFFF0) of the four small blocks (1FFFFFF3, 1FFFFFF2, 1FFFFFF1, and 1FFFFFF0) is set. Now consider what happens when this large block in the spatial buffer is replaced. If the hit bit of the first small block is only one set, the bits 00 corresponding to the first small block are added to the tag value of the spatial buffer (C) by the address generator. The two-bit offsets corresponding to the four small blocks are 00, 01, 10, and 11, respectively. Therefore, a new memory address (A+B) without an offset is formed, and the corresponding tag and index values for the direct-mapped cache are represented as A and B, respectively, through the process of redecoding.

Cache write-back does not occur from the spatial buffer because any modified or referenced small block is always moved to the direct-mapped cache before its corresponding large block is replaced. Write-back occurs only from the direct-mapped cache. In a conventional direct-mapped cache or victim cache, which would typically have the same block size (e.g., 32B) as the spatial buffer, write-back must be performed for the full 32B block, even though only one word requires write-back. In contrast, the proposed cache executes the write-back operation only for the marked 8B small blocks. Therefore, write traffic into memory is potentially reduced to a significant degree.

It should be noted that the potential exists in any split cache for blocks of incoherent copies to appear in the different subcaches. Thus, to avoid this problem, we chose and simulated a simple mechanism, which we describe as follows. When a global miss occurs, the cache controller searches the tags of the temporal cache to detect whether any of the four small blocks belonging to the particular large block being fetched are present in the temporal cache. If a match is detected, then all of the corresponding small blocks in the temporal cache are invalidated. Each of these small blocks that is also dirty is then used to update its corresponding entry in the spatial buffer once the large block has been loaded. This search operation can be accomplished while the cache controller is handling a miss. Further, the power consumption overhead is negligible, because the miss ratio is only about 1.7% of the total number of addresses generated by the CPU.

A small block may thus temporarily exist in the temporal cache in the invalid state, while its valid copy is being referenced in the spatial buffer. When its corresponding large block is replaced, the small block is copied into the temporal cache once again. Therefore, there is almost no performance decrease. Of course, if three or four small blocks are present in both the temporal cache and the spatial cache, then the effective utilization of total cache space decreases a bit more, but is still negligible. This mechanism also applies in the case of transferring a prefetched block into the spatial buffer.

IV. Performance Evaluation

Benchmarks used in the trace-driven simulation include six of the SPECint95 benchmarks and two from SPECfp95 (applu and tomcatv), representing general-purpose applications, and ten of the Media benchmarks, representing embedded multimedia and communications applications. The Media benchmarks are representative of image compression, voice, video transmission, 3D text mapping, cryptography, and so forth. Only data references are collected and used for the simulation. The DineroIV cache simulator was modified to simulate the proposed data cache system. We have chosen two common approaches, the direct-mapped cache and the victim cache, for comparison in terms of performance.

1. Time of Prefetch Signal Generation and Overhead

We used simulation to determine the threshold for the number of hit bits that should be set before we initiate a prefetch signal. Figures 2 and 3 show the miss ratio and the average memory access time for variations of initiating a prefetch signal based on the number of set hit bits. Generally, the more meaningful measure to evaluate the performance of any given memory-hierarchy is the average memory access time. The basic parameters for the simulation are presented as follows: the hit times of direct mapped cache and fully

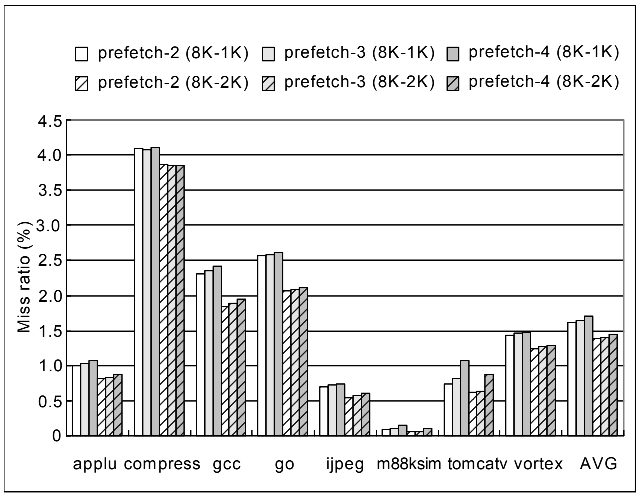

Fig. 2. Miss ratio for various prefetch configurations.

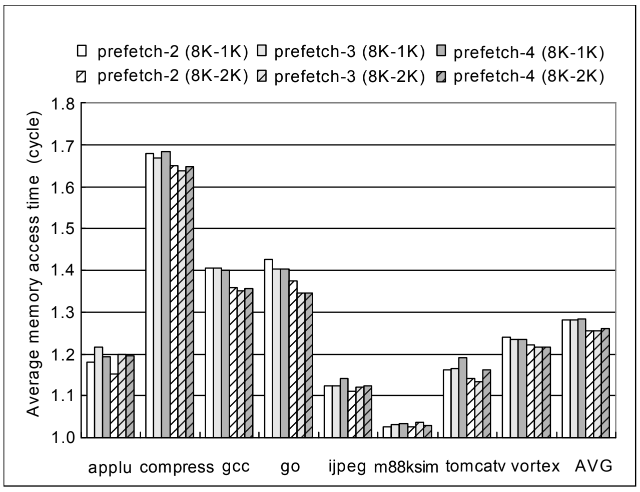

Fig. 3. Average memory access time for various prefetch configurations.

associative buffer are both assumed to be one cycle. We assume 15 cycles are needed for a miss. Therefore, each 8B block is transferred from the off-chip memory after a 15 cycle penalty. These parameters are based on the values for common 32-bit embedded processors (e.g., Hitachi SH4 or ARM920T).

The notation "8K–1K" denotes our example configuration (8kB direct-mapped cache with a 1kB spatial buffer). Also the notation "prefetch-2" denotes that the prefetch controller generates a prefetch signal when two of the four hit bits are set, "prefetch-3" denotes a threshold of three set hit bits, and so on. Our simulations show that prefetching when the number of hit bits is two achieves a more significant miss gain than the other cases, in spite of the potential for greater overhead due to increased prefetch frequency. With respect to power consumption, memory traffic, and the accuracy of the prefetching operation, the "prefetch-4" mechanism provides the most significant effect.

When a prefetch operation is performed, the overhead can be determined as follows. If the P bit is reset, the prefetch controller searches the tag part of the spatial buffer, resulting in an additional one-cycle penalty beyond the single cycle required for normal access. Search overhead is shown in Fig. 4. We count both the access cycle and the search cycle, for a total of two cycles, in determining the overhead. However, this overhead turns out to be negligible because it applies to only 1.5% to 2.5% of all addresses generated. Also, using the P bit can reduce the two-cycle overhead to a single cycle by eliminating searching when a block is already present. Overall, use of the P bit reduces the search overhead by around 65% to 80%.

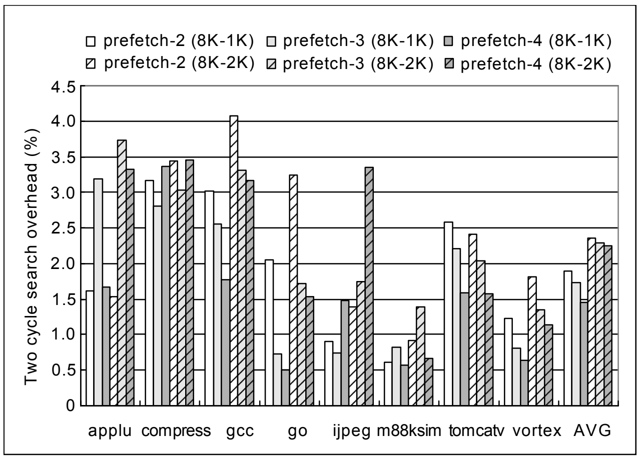

Fig. 4. Tag search overhead for various prefetch configurations.

In general, the search overhead of the "prefetch-2" case tends to be greater than the other configurations because of the higher rate of prefetch generation, but not in every case. For example, the detailed breakdown of overhead for the "applu" benchmark is shown in Table 1. The figures in the row for case A show the rate at which prefetch operations actually occurred after the tags of the spatial buffer were searched, with a two-cycle overhead. Case B shows the rate when the block to prefetch already exists in the spatial buffer, but the P bit is not set, so an extra-cycle search is performed before a prefetch signal is generated. In case C, the P bit of the block has already been set, so there is no need to search the tags. In this case there is only one cycle of overhead, as with normal accesses. Thus, the actual overhead in Table 1 is reduced by the amount in case C.

Table 1. Various cases of the two cycle search overhead in Applu.

Cases	prefetch-2	prefetch-3	prefetch-4	Two cycle overhead
A (P bit: 0→1)	0.436%	0.324%	0.196%	Yes
B (P bit: 0→1)	1.176%	2.866%	1.469%	Yes
C (P bit: 1→1)	4.041%	2.623%	4.539%	No
Actual two cycle search overhead (A+B)	1.612%	3.190%	1.665%	

Finally, in the case of prefetching a block that does not exist in the spatial buffer, the target block is simply fetched into the prefetch buffer. The rates at which prefetch operations actually occurred and prefetched blocks are actually referenced are shown in Figures 5 and 6. With only a small number of prefetch operations, the proposed cache system achieves a significant performance gain with low overhead. For the "prefetch-4" mechanism, the utilization of prefetched blocks is over 90%. This data clearly shows that spatial locality is enhanced by prefetching a neighboring block intelligently when a spatial buffer hit occurs.

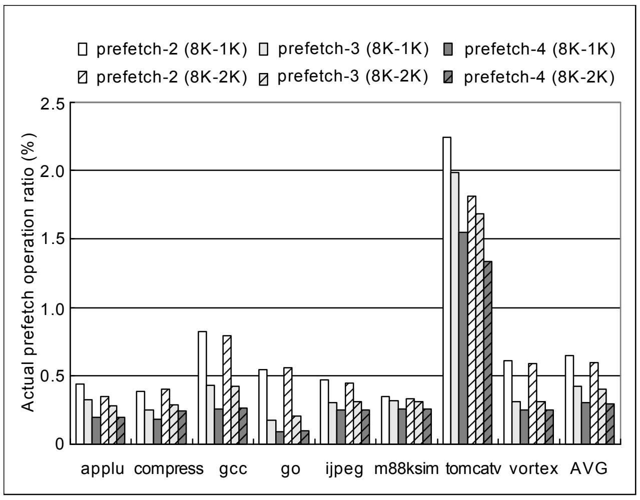

Fig. 5. Ratio of the actual number of prefetch signals generated by prefetch operation.

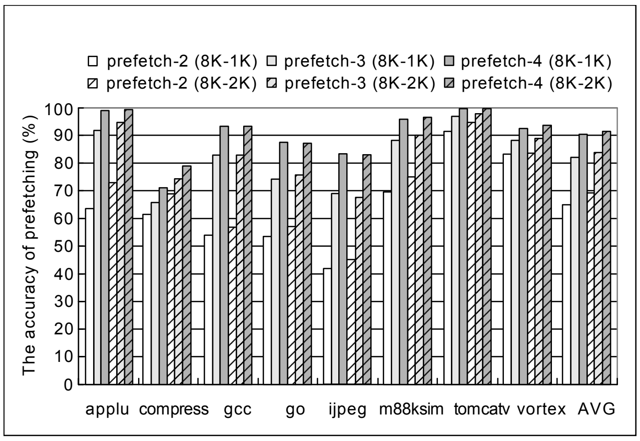

Fig. 6. Actual hit ratio of prefetched blocks.

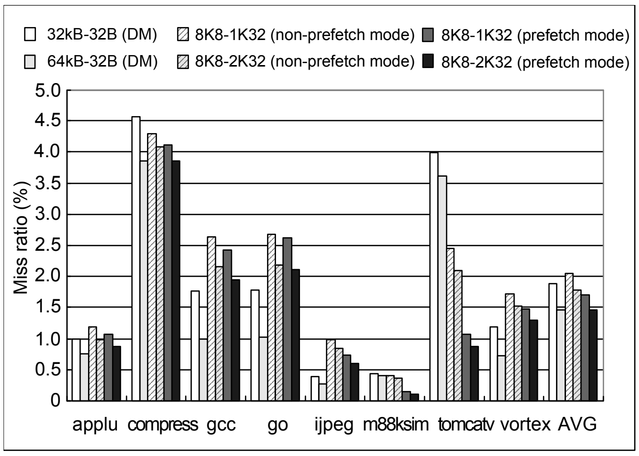

Fig. 7. Miss ratios of the direct-mapped cache and proposed cache.

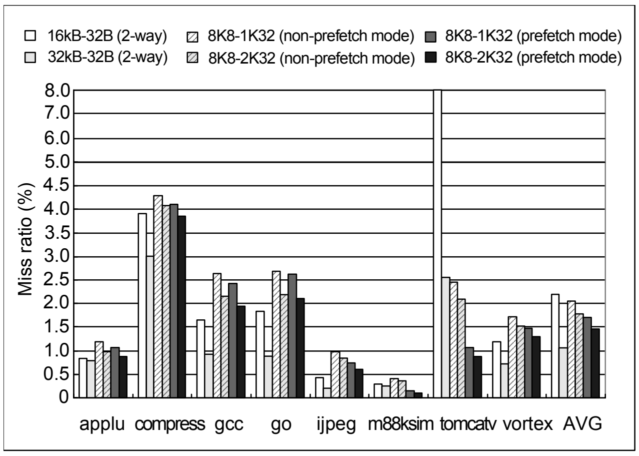

Fig. 8. Miss ratios of the 2-way set associative cache and proposed cache.

2. Comparison of a Conventional Cache with the Proposed Cache Configuration

Two common performance metrics, the miss ratio, and the average memory access time, are used to evaluate and compare the proposed cache system operating in a “prefetch-4” configuration with other approaches.

A. Miss Ratio and Average Memory Access Time

Several experiments were performed to determine the optimum block sizes for the proposed direct-mapped cache and spatial buffer. The combination of an 8B small block and a 32B large block shows the best performance for most cases. To clarify the impact of the prefetch operation, we evaluated the cache both with and without prefetching.

The cache miss ratios for the conventional direct-mapped cache and the proposed cache are shown in Fig. 7. For the direct-mapped cache, denoted as DM, the notation “32kB–32B” denotes a 32kB direct-mapped cache with a block size of 32B. The proposed cache notation “8K8–1K32” denotes an 8kB direct-mapped cache with a block size of 8B, and a 1kB spatial buffer with a block size of 32B. Notice that the average miss ratio of the proposed cache for a given size (e.g., 8kB) is equal to a conventional direct-mapped cache with a cache size of four or eight times as much space (e.g., 32kB, or 64kB) in a non-prefetching mode and prefetching mode, respectively.

The miss ratios for a conventional 2-way set-associative cache and the proposed cache are compared in Fig. 8. The 2-way set-associative cache greatly reduces the miss ratio, but because of its slower access time and higher power consumption, embedded processors typically do not employ this organization. The results of simulation show that the proposed cache can achieve better performance than a 2-way set associative cache with double the space. A 4-way set associative cache shows results similar to a 2-way set associative cache, given the same cache sizes (16kB–32kB).

The average memory access times for the conventional direct-mapped cache and the proposed cache are compared in Fig. 9. The average access time was obtained using the following simulation mechanism. First, a block to prefetch is loaded into the prefetch buffer 19 cycles after the initiation of a prefetch signal, using a cycle counter. If no cache miss or reference to the block being prefetched occurs during this prefetch operation, then the prefetch controller needs to check the tag part of the spatial buffer to see whether the block is present, resulting in a single-cycle penalty being added to the normal access time. However, if a cache miss occurs while a prefetch operation is being performed, its miss handling is deferred until the ongoing prefetch operation completes. In this case, we assume that the average access time is increased by at most 18 cycles, as would occur when a miss arrives on the cycle following the start of the prefetch operation. The precise

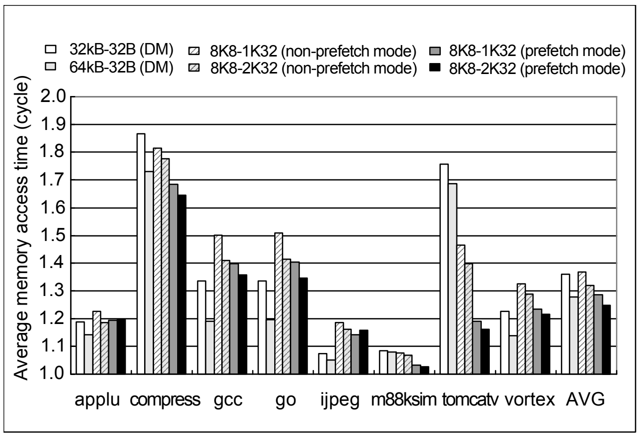

Fig. 9. Average memory access time of the direct cache and proposed cache.

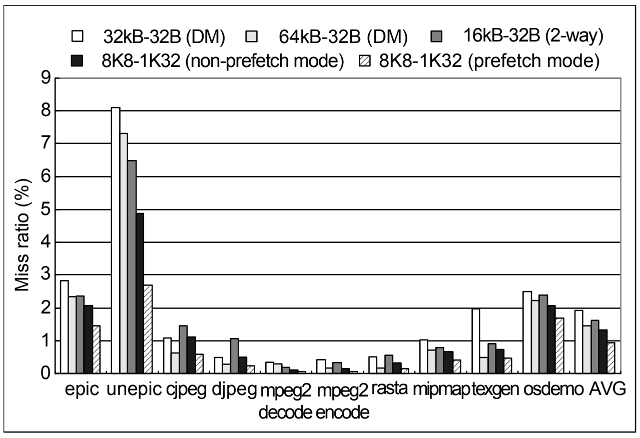

Fig. 10. Miss ratios of the various caches and proposed cache.

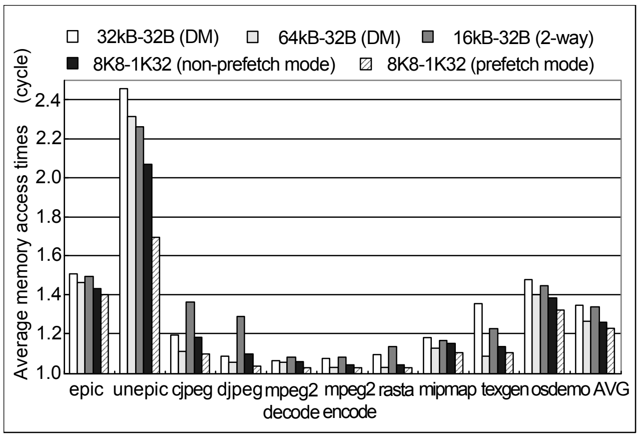

Fig. 11. Average memory access time of the various caches and proposed cache.

miss penalty is 19 cycles minus the number of cycles between the prefetch initiation time and the time when the miss signal was generated. When a reference is made to a block that is being prefetched, the hit time is 19 cycles minus the number of cycles since the prefetch initiation. The exact number of cycles is measured by a counter in the simulation. Our analysis shows that applications with a high degree of locality, such as tomcatv, show an especially strong performance improvement with the proposed cache.

Figures 10 and 11 show the resulting miss ratio and average memory access time for the media benchmarks. Multimedia applications show better performance when larger cache block sizes, e.g., 64B or 128B, are chosen. Therefore, the prefetching mechanism is more prominent, but the non-prefetching mode can also achieve high performance.

3. Comparison of a Victim Cache with the Proposed Cache

We compared several previously proposed cache designs (e.g., NTS cache, victim cache, selective victim cache, assist cache, and so forth) with the proposed cache. Our analysis of the performance improvement achieved by each of these designs showed that one of the most effective is the victim cache [12], [13]. Our results from comparing a victim cache configuration with the proposed cache are presented here. The victim cache can significantly reduce conflict misses and can provide a low overall miss ratio with just a simple hardware mechanism. However, a victim cache does incur a large number of content swaps between the main cache and the victim buffer, and operates with a large block size. Figures 12 and 13 show the resulting miss ratio and average memory access time for the two approaches when the same cache and buffer sizes are used. A victim cache with a 32B block size shows the best performance, but increasing the block size often increases write traffic into memory.

As shown in Figs. 12 and 13, the proposed cache (a 8kB direct cache with an 8B block size and a 1kB spatial buffer with a 32B block size) has better performance than the

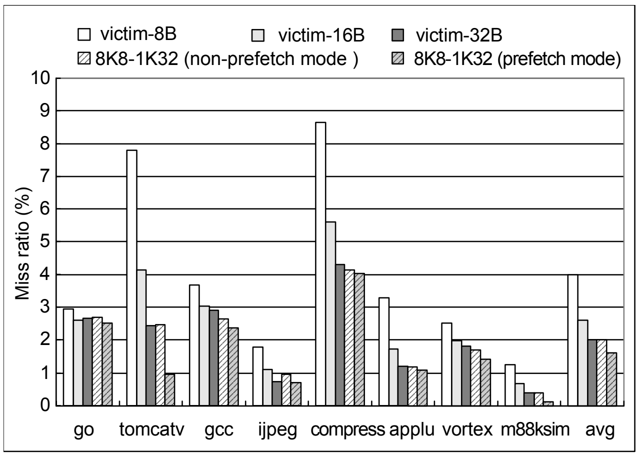

Fig. 12. Miss ratios of victim cache and proposed cache.

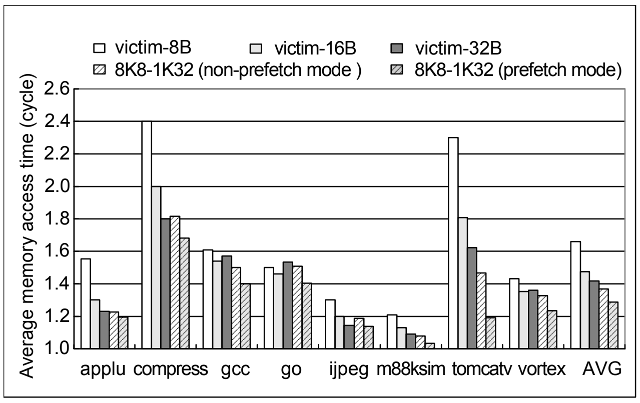

Fig. 13. Average memory access times for victim cache and proposed cache.

8kB victim cache with a 1kB victim buffer. The victim cache employs the same 32B cache block size in the main cache and the victim buffer, while 8B blocks are used in the main proposed cache. The use of a smaller block size in the proposed cache results in a significant reduction in power consumption because write traffic into memory is reduced by 25%.

4. Relation between Cost and Performance

In general, the logic to manage the tags for the fully associative cache is designed as a CAM structure for simultaneous comparison of each entry. Because each CAM cell is a combination of storage and comparison, the size of a CAM cell is double that of a RAM cell [14]. For fair performance/cost analysis, the performance for various direct-mapped cache and buffer sizes is evaluated. The metric is *rbe* (register bit equivalents), and the total area can be calculated as follows:

$$Area = PLA + RAM + CAM. \tag{1}$$

Here, the control logic *PLA* (programmable logic array) is assumed to be 130 *rbe*, a *RAM* cell as 0.6 *rbe*, and a *CAM* cell as 1.2 *rbe*. (2) represents the *RAM* area [14]:

$$\begin{aligned} RAM &= 0.6(\#entries + \#L_{sense\_amp}) \\ &\times((\#data\_bits + \#status\_bits) + W_{driver}), \end{aligned} \tag{2}$$

where L_{sense_amp} is the bit length of a bit-line sense amplifier, W_{driver} the data width of a driver, *#entries* the number of rows of the tag array or data array, *#data_bits* the tag bits or data bits of one set, and *#status_bits* the state bits of one set. Finally, (3) calculates the area of the *CAM*:

$$\begin{aligned} CAM &= 0.6\,(\sqrt{2}\times\#entries + \#L_{sense\_amp}) \\ &\times(\sqrt{2}\times\#tag\_bits + W_{driver}), \end{aligned} \tag{3}$$

where *#tag_bits* is the number of bits for one set in the tag array.

Table 2. Performance and cost of the proposed cache and various caches. (IR: improvement ratio, AMAT: average memory access time)

	Area (*IR*)	Miss ratio (*IR*)	AMAT (*IR*)
32kB–32B (DM)	177,496 *rbe* (1.00)	1.89% (1.00)	1.34 *cycle* (1.00)
8kB–1kB (victim)	67,431 *rbe* (0.38)	2.00% (1.06)	1.42 *cycle* (1.06)
8kB–1kB (proposed cache)	67,431 *rbe* (0.38)	1.61% (0.85)	1.29 *cycle* (0.96)
64kB–32B (DM)	352,596 *rbe* (1.99)	1.46% (0.77)	1.26 *cycle* (0.94)
8kB–2kB (victim)	73,680 *rbe* (0.42)	1.69% (0.89)	1.38 *cycle* (1.03)
8kB–2kB (proposed cache)	73,680 *rbe* (0.42)	1.37% (0.73)	1.25 *cycle* (0.93)

Table 2 shows the performance/cost for three different cache configurations.

A 32kB direct-mapped cache, an 8kB–1kB victim cache and an 8kB–1kB proposed cache are compared. The improvement for each configuration is normalized to the value of the direct-mapped cache. A 64kB–32B direct-mapped cache, an 8kB–2kB victim cache and an 8kB–2kB proposed cache are compared to a 32kB-32B direct-mapped cache. The proposed cache shows about a 60% area reduction compared with the 32kB–32B conventional direct-mapped cache, even though it provides higher performance. It also offers an 80% area reduction compared with the 64kB–32B configuration, while providing much higher performance. In addition, the improvement ratio for the average memory access time shows that the 8kB–2kB proposed cache is the best configuration.

5. Comparison of Power Consumption

For power consumption analysis, we evaluated various cache sizes using the CACTI-3.0 simulator [15], which can calculate access times, cycle times, area, and power consumption for many types of hardware caches. Our results are based on 0.18 μm technology with a 1.7 V supply voltage.

Table 3 shows the power consumption for various cache configurations. Each entry shows the power dissipation for a cache access and a cache update on a miss case. In the cases of the victim cache and the proposed cache system, the direct-mapped cache and the fully associative cache are searched in parallel at the same level. According to the results of CACTI 3.0, access times for the dual direct-mapped cache (e.g., 8kB–8B) and the fully associative cache (e.g., 1kB–32B) of the proposed cache are 0.953 ns and 1.934 ns, respectively. However, the access time for the tag part of the fully associative cache is 1.372 ns. If a hit occurs at the direct-mapped cache, the data part of the fully associative cache does not need to be driven. That is, the requested data item is transmitted to the CPU without checking for a hit/miss in the fully associative cache. This mechanism offers the fast access time of a direct-mapped cache and low power consumption by using a simple additional unit and an asynchronous SRAM.

Table 3. Power consumption per access for various cache configurations.

Cache configuration	P_{access} (nJ)	P_{cache_write} (nJ)
16kB–32B (DM)	0.4734	0.2220
32kB–32B (DM)	0.6205	0.3726
64kB–32B (DM)	0.9358	0.6909
16kB–32B (2-way)	0.6335	0.2237
16kB–32B (4-way)	0.9349	0.2260
32kB–32B (2-way)	0.7586	0.3402
Victim cache (8kB 32B–1kB 32B)	DM miss: 0.649 DM hit: 0.440	DM write: 0.130 FA write: 0.077
Proposed cache (8 kB 8 B–1 kB 32 B)	DM miss: 0.577 DM hit: 0.409	DM write: 0.106 FA write: 0.077

P_{access} of the victim and the proposed cache can be divided into two parts, i.e., a hit case and a miss case at the direct-mapped cache. If a hit occurs at the direct-mapped cache, accessing power is consumed to access the tag and the data part of the direct-mapped cache and to access the tag part of the fully associative cache (i.e., the "DM hit" case in Table 5). And if a miss case occurs at the direct-mapped cache, power consumption to access the data part of the fully associative cache should be added to that of a hit case (i.e., the "DM miss" case in Table 3). Finally "DM write" at the victim cache denotes two cases, namely, power consumption to update the direct-mapped cache when a global miss occurs or the case for content swapping when a victim buffer hit occurs. "FA write" for the victim cache denotes power consumption for updating the associative cache when the replaced item from the direct-mapped cache is moved into the victim buffer. "FA write" for the proposed cache denotes power consumption for updating the associative cache when a miss occurs. "DM write" for the proposed cache denotes power consumption in the direct-mapped cache when a large block is replaced, that is, when its small blocks that are marked as having been accessed are moved into their corresponding block entries in the direct-mapped cache.

From these values, the average power consumption of the cache system is given by

$$Avg.power = N_{hit} \cdot P_{access} + N_{miss} \cdot P_{miss}, \qquad (4)$$

where N_{hit} and N_{miss} are the ratios of hits and misses in the cache, respectively. P_{access} is the power used to access a cache block and P_{miss} is the power required to process a miss. P_{miss} can be calculated as

$$P_{miss} = P_{access} + P_{cache\_write} + P_{pad}, \qquad (5)$$

where P_{cache_write} is the power for a cache write operation in a cache miss, and P_{pad} is the power dissipated at the on-chip pad slot. P_{pad} can be calculated as in [16], [17],

$$P_{pad} = 0.5\ V_{dd}^{2} \cdot (0.5\ (W_{data} + W_{addr}) \cdot 20\text{pF}, \qquad (6)$$

where W_{data} and W_{addr} are the number of bits for both the data sent/returned and the address sent to the lower level memory on a miss request. The capacitive load for off-chip destinations is assumed to be 20 pF [16]. A data cache with a 32B block size is assumed, where the values of W_{data} and W_{addr} are also 32 bits.

Figures 14 and 15 present the average power consumption of different cache structures compared to the proposed cache for the benchmarks used earlier. Simulation results show that power consumption in the proposed cache is around 10% to 60% lower than these various cache systems. Therefore, the proposed cache shows the lowest power consumption of all the approaches. Overall, the proposed cache shows the best result in terms of both performance and power among all of the approaches.

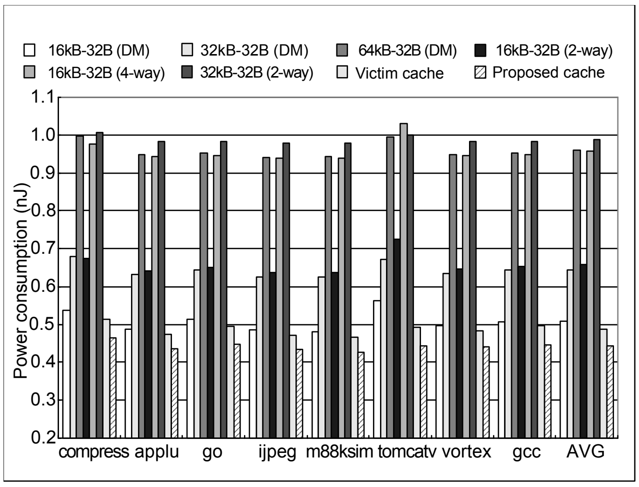

Fig. 14. Spec95 benchmarks: power consumptions of the conventional caches and the proposed cache.

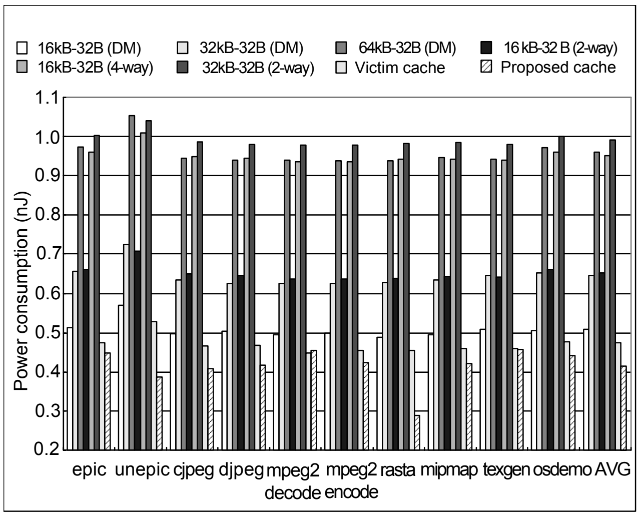

Fig. 15. Media benchmarks: power consumptions of the conventional caches and the proposed cache.

V. Conclusion

The goal of this research was to design a simple but high performance and low power cache system with low cost. To attain this goal, we designed a new caching mechanism for exploiting two types of locality effectively and adaptively; a direct-mapped cache with a small block size for exploiting temporal locality and a fully associative spatial buffer with a large block size for exploiting spatial locality. We used an intelligent hardware-based prefetching mechanism to maximize the effect of spatial locality. We have shown that the proposed cache overcomes the structural drawbacks of direct-mapped caches, such as conflict misses and thrashing. We evaluated the proposed cache system in two configurations, the non-prefetching mode and prefetching mode, to analyze the contribution of intelligent prefetching. Both modes provide high performance, but the non-prefetching mode offers lower power consumption while the prefetching mode offers higher performance. According to our simulation results, the time interval mechanism of the proposed cache decreases conflict misses by about 26%, and the spatial locality miss ratio decreases by about 48%. The average rate of prefetch signal generation is only about 0.3% to 0.7% of the total number of address references generated. For the prefetching mode, the miss ratio is about 21% less and the average memory access time is about 10% less than non-prefetching mode. The average miss ratio and average memory access time of the proposed cache for a given cache space (e.g., 8kB) is equivalent to a conventional direct-mapped cache with four times as much space (e.g., 32kB). We have also shown that at least 60% to 80% area reduction can be obtained as compared with a direct-mapped cache that is large enough to provide similar performance. In addition, the proposed cache can reduce the miss ratio by around 20% and the average memory access time by around 10% (in either the prefetching or non-prefetching mode), versus the victim cache configuration. We have also shown that power consumption in the proposed cache is around 10% to 60% lower than these various cache systems.

References

[1] J.L. Baer and T.F. Chen, "An Effective On-Chip Preloading Scheme to Reduce Data Access Penalty," *Proc. Int'l Conf. on Supercomputing*'91, 1991, pp. 176-186.

[2] T. Mowry, M.S. Lam, and A. Gupta, "Design and Evaluation of a Compiler Algorithm for Prefetching," *Proc. 5th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, 1992, pp. 62-73.

[3] W.Y. Chen, R.A. Bringmann, S.A. Mahlke, R.E. Hank, and J.E. Sicolo, "An Efficient Architecture for Loop Based Data Preloading," *Proc. 25th Int'l Symposium on Microarchitecture*, 1992, pp. 92-101.

[4] Norman P. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully Associative Cache and Prefetch Buffers," *Proc. 17th ISCA*, May 1990, pp. 364-373.

[5] D. Stiliadis and A. Varma, "Selective Victim Caching: A Method to Improve the Performance of Direct Mapped Cache," *IEEE Trans. Comput.*, vol. 46, no. 5, May 1997, pp. 603-610.

[6] A. Gonzalez, C. Aliagas, and M. Valero, "Data Cache with Multiple Caching Strategies Tuned to Different Types of Locality," *Proc. Int'l Conf. on Supercomputing*'95, July 1995, pp. 338-347.

[7] V. Milutinovic, M. Tomasevic, B. Markovic, and M. Tremblay, "The Split Temporal/Spatial Cache: Initial Performance Analysis," *SCIzzL-5*, Mar. 1996.

[8] G. Kurpanchek et al., "PA-7200: A PA-RISC Processor with Integrated High Performance MP Bus Interface," *COMPCON Digest of Papers*, Feb. 1994, pp. 375-382.

[9] Jude A. Rivers and Edward S. Davidson, "Reducing Conflicts in Direct-Mapped Caches with a Temporality-Based Design," *Proc. the 1996 Int'l Conf. on Parallel Processing*, vol. I, 1996, pp. 151-162.

[10] S. Przybylski, "The Performance Impact of Block Sizes and Fetch Strategies," *Proc. 17th ISCA*, May 1990, pp. 160-169.

[11] F. Jesus Sanchez, Antonio Gonzalez, and Mateo Valeo, "Static Locality Analysis for Cache Management," *Proc. PACT*'97, Nov. 1997, pp. 261-271.

[12] G. Albera and R. Iris Bahar, "Power/Performance Advantages of Victim Buffer in High-Performance Processors," *Proc. IEEE Alessandro Volta Memorial Workshop*, Mar. 1999, pp. 43-51.

[13] V. Srinivasan, *Improving Performance of an L1 Cache with an Associated Buffer*, CSE-TR-361-98, University of Michigan, Feb. 1998.

[14] J.M. Mulder, N.T. Quach, and M.J. Flynn, "An Area Model for On-Chip Memories and its Applications," *IEEE J. Solid State*

Circuits, vol. 26, no. 2, Feb. 1991, pp. 98-106.

[15] G. Reinman et al., *CACTI 3.0: An Integrated Cache Timing and Power, and Area Model*, Compaq WRL Report, August 2001.

[16] M.B. Kamble et al., "Energy-Efficiency of VLSI Cache: A Comparative Study," *Proc. IEEE 10th Int'l Conf. on VLSI Design*, Jan. 1997, pp. 261-267.

[17] M.B. Kamble et al., "Analytical Energy Dissipation Models for Low Power Caches," *Proc. ISLPED*'97, Aug. 1997.

Jung-Hoon Lee received his BS degree in control instrumentation engineering from Sungkyunkwan University in Seoul, Korea, in 1999 and the MS degree in computer science from Yonsei University in Seoul, Korea, in 2001. He is currently a PhD student in computer science at Yonsei University. His research interests include advanced architecture, intelligent memory systems, low power architecture, and SoC systems.

Gi-Ho Park received the BS, MS, and PhD degrees in 1993, 1995, and 2000 in computer science at Yonsei University in Seoul, Korea. He is currently a Senior Engineer in Processor Architecture Lab., SOC R&D Center, System LSI Division, Device Solution Network at Samsung Electronics Co. His research interests include advanced computer architectures, memory system design, and realtime architecture.

Shin-Dug Kim received the BS degree in electronic engineering from Yonsei University in Seoul, Korea, in 1982 and the MS degree in electrical engineering from University of Oklahoma in 1987. In 1991, he received the PhD degree from the School of Computer and Electrical Engineering at Purdue University in West Lafayette, Indiana. He is currently a Professor in Computer Science at Yonsei University in Seoul, Korea. His research interests include advanced computer architectures, parallel processing systems, memory system design, and agent based Internet computing.