TWO FAST AND HIGH-ASSOCIATIVITY CACHE SCHEMES

Chenxi Zhang

Changsha Institute of Technology

Xiaodong Zhang

Yong Yan

College of William and Mary

Parallel and sequential multicolumn cache schemes increase associativity while keeping cycle times low.

In the race to improve cache performance, many researchers have proposed schemes that increase a cache's associativity. The associativity of a cache is the number of places in the cache where a block may reside.[1] In a direct-mapped cache, which has an associativity of 1, there is only one location to search for a match for each reference. In a cache with associativity n—an n-way set-associative cache—there are n locations (see the boxes on definitions and set-associative caches). Increasing associativity reduces the miss rate by decreasing the number of conflict, or interference, references.

The column-associative cache[4] and the predictive sequential associative cache[5] seem to have achieved near-optimal performance for an associativity of two. Increasing associativity beyond two, therefore, is one of the most important ways to further improve cache performance.

We propose two schemes for implementing associativity greater than two: the sequential multicolumn cache, which is an extension of the column-associative cache, and the parallel multicolumn cache. For an associativity of four, they achieve the low miss rate of a four-way set-associative cache. Our simulation results show that both schemes can effectively reduce the average access time. With a 4-Kbyte cache and a miss penalty of 20 cycles, the improvements in average access time over a direct-mapped cache were 9.8% for the SMC and 10.8% for the PMC. The improvement of the SMC in average access time reached 22.4% when the associativity was eight and the miss penalty increased to 100 cycles. The two schemes are effective for a wide range of cache sizes, from 1 to 128 Kbytes.

Advantages and disadvantages of high associativity

Hill and Smith[6] reports that using a two-way set-associative cache instead of a direct-mapped cache reduces misses by about 30%. Increasing associativity, however, also increases hit access time. This generally leads to a longer cycle time because in most microprocessors access to the cache lies in the critical path. There are two reasons for the longer hit access time:

- The multiplexing logic that selects the correct data point from the referenced set causes a delay.
- Only after completing tag checking does the multiplexing logic know which data point to select and dispatch to the CPU. Therefore, it is difficult or impossible to implement a speculative dispatch of data to the CPU (or optimistic execution[2]).

In contrast, a direct-mapped cache requires no such multiplexing logic; the cache system can speculatively dispatch data to the CPU as soon as it reads the data.

Higher associativity is important when the miss penalty is large and when memory and memory interconnect contention delay are significant or sensitive to the cache miss rate. Both situations may occur with shared-memory multiprocessors.[1] A uniprocessor also may have a large miss penalty when it has only a first-level cache and the speed gap between the processor and the memory is large.

Increasing associativity also has the advantage of reducing the probability of thrashing. Repeatedly accessing m different blocks that map to the same set will cause thrashing. A

0272-1732/97/$10.00 © 1997 IEEE

cache with an associativity of n can avoid such thrashing if $n \geq m$ and LRU (least recently used) replacement is guaranteed.

Researchers have proposed various cache designs that aim at increasing associativity without sacrificing the advantages of a direct-mapped cache (see the cache schemes box, next page). These designs succeed in retaining the low cycle times of a direct-mapped cache while achieving miss rates comparable to those of two-way set-associative caches. Agarwal and Pudar also shows that the column-associative cache can overcome secondary thrashing, the main drawback of the hash-rehash cache.[4] None of these designs, however, can keep cycle times low for associativity greater than two.

Multicolumn caches

Both the SMC and PMC cache schemes use the multiple MRU block technique and the LRU replacement algorithm.

Definitions and variables

Throughout this article, we use the following definitions and variables:

A *location* is a place in a cache where a block may reside. This definition does not restrict how to determine locations or how to search them for the desired data when servicing a reference.

A *miss penalty* is the time required to service a miss from lower-level memory.

The *average access time* for a cache is the average number of cycles required to complete a reference.

If C is the size of a given cache in blocks, we define c to be the integer such that $C = 2^c$. Similarly, for associativity n, n' is the integer such that $n = 2^{n'}$.

We use x for the block address of a reference.

Set-associative caches

The recently proposed column-associative cache, the predictive sequential associative cache, and others are generally considered to be improvements of direct-mapped caches, and they do not address the implementation of higher associativity. Here, we give a systematic and generalized discussion of the implementation of associativity.

In a traditional n-way set-associative cache, a data array holds data blocks and a tag array holds their associated tags, which are all organized into n banks. There is a one-to-one correspondence between data banks and tag banks. On a reference, the cache system uses the lower $c - n'$ bits of the block address to select a row in the tag array of n tags and a row (of the same row number) in the data array of n data items. The system reads the n tags simultaneously and compares them against the tag bits of the address in parallel. At the same time, it reads n candidate data items, one of which a multiplexer later selects using the outcome of the tag comparisons.

In an n-way set-associative cache, mapping functions determine the n candidate locations, which can be any blocks in the cache, that make up a set. These functions, together with the search approach determining how to examine these locations, distinguish the various implementations of associativity. During a search, the relationships and orders among these locations are open.

Mapping functions

Because the mapping function lies in the critical path of cache accesses, it must work as quickly as possible. Therefore, most cache designs use bit selection as the mapping function. Bit selection works by conceptually dividing the cache into n equal banks, each of size $2^{c-n'}$. It then uses the lower $c - n'$ bits of the block address as an index to select one candidate location from each bank.

This design uses the same mapping function for each of the banks if the blocks are all relatively addressed, and this mapping requires no extra time. In contrast, some cache designs, such as the skewed-associative cache,[2] use different mapping functions for different banks as a way to increase associativity.

Search approach

The search approach determines how the cache system examines candidate locations for a match when servicing a reference. There are two main kinds of search approach: parallel and sequential.

Parallel search. This approach examines all locations in parallel. While the cache system reads and compares the tags, it also reads all the candidate data; the multiplexer later selects one data point if the reference turns out to be a hit. Each location in the set is of equal importance. Traditional implementations of set-associative caches use this search approach. These implementations organize both tag memory and data memory into multiple banks.

Sequential search. A sequential search examines locations one by one until it finds a match or exhausts all the locations. This approach organizes both tag memory and data memory into a single bank. To implement an associativity of n, however, the tag memory and data memory are divided into n banks.

With a sequential search, the cache access time for a hit may vary for different references, depending on how many locations the cache system examines before it finds a hit. We call a reference an ith hit if the system finds a match in the ith location; its access time is the ith hit time. The ratio of the number of ith hits to the total number of hits is the ith hit rate. We call a reference the ith fetch if it turns out to be a miss after the system examines i locations, in which case the system must fetch a block from lower-level memory. In addition, we say one location is more important than another for a reference if the system is more likely to find a hit in that location. Usually, the more recently a location was accessed, the more important it is, due to the temporal locality of processor reference streams. To achieve the shortest average access times, the ith hit rates

continued on p. 42

Set-associative caches (continued)

must be highest for small values of *i*. That is, the processor should make as many first hits as possible.

If the candidate locations are of similar importance for each reference, or if the cache system examines a less important location first, there will be little gain and even some degradation in performance. Fortunately, not all candidate locations are of equal importance. Only one location, the most recently used location, has distinguished importance. So and Rechtschaffen[3] report that for an associativity of four, most of the hits to the cache happen to the MRU locations for their traces. Our simulation produced a similar result. To ensure that most hits are first hits, the MRU location should be the entry point for the search. Therefore, in designing a sequential search approach, it is important to know how to find the MRU location.

The cache system must determine the search order, including the entry point, before a search begins. There are two kinds of approaches to determine the search order: mapping and swapping.

Mapping approach. This approach determines the search order dynamically on a reference by using steering information in a mapping table. The mapping will lengthen the cache access cycle, however, if it uses the effective address as the index to look up the steering information. This kind of approach is not effective for first-level caches. If the mapping can use as the index some other sources available at earlier pipeline stages, however, the steering information may be available on time. Such an approach will add no additional delay to the access cycle time, but requires extra memory for the mapping table.

Swapping approach. This approach determines the search order statically. In other words, it uses a fixed order and assumes that the most recently used block is in location 0, the second most recently used block is in location 1, and so forth. Performing appropriate swaps of blocks after each reference can guarantee the correct order. This approach is impractical for associativity larger than two, however, if a complete order is to be maintained.

To reduce complexity and cost, designers can simplify the search approach so that it only searches the most important locations in a strict order of importance. It can then search the remaining locations in an arbitrary order if the performance degradation that the search causes is acceptable.

Cache schemes

Designers have proposed various schemes for improving cache performance. Most of the following use sequential search and have an associativity of two.

Hash-rehash cache

A hash-rehash cache[7] uses two mapping functions to determine the candidate locations. It uses bit selection, however, to reduce the overhead. For a reference, the cache system derives the address of the first location directly from the lower *c* bits of *x* and the address of the second location by flipping the most significant bit of the derived address. Most of the following cache schemes determine candidate locations in this way.

The hash-rehash cache uses a sequential search. It tries to keep the MRU blocks in the first location for references by swapping blocks, examining the second location only if the first probe is a miss. If the second probe is a hit, the system swaps the contents in the two locations because the second location now contains the MRU block. If the second probe is a miss, the system moves the data in the first location to the second location, which is the former MRU block, and places the newly fetched block in the first location.

Agarwal et al.[7] found that the hash-rehash cache has a higher miss rate than a two-way set-associative cache with LRU replacement, and has an average access time similar to that of a direct-mapped cache. The higher miss rate results from the non-LRU replacement and the lack of an index to filter out unnecessary probes.

Column-associative cache

The column-associative cache, proposed by Agarwal and Pudar,[4] improves the hash-rehash cache by adding a rehash bit, which it uses to implement approximate LRU replacement and to filter out unnecessary probes. In the column-associative cache, each location has an associated rehash bit that indicates whether it is a rehashed location. The cache system sets the rehash bit for a location, marking it as a rehashed location, when the system probes it second on a reference. The column-associative cache differs in operation from the hash-rehash cache only when the rehash bit of the first location is set, which means both blocks in the set are useless for the current reference. In this case, the system does not probe the second location, and it replaces the block in the first location with the newly fetched block and clears the rehash bit.

Agarwal and Pudar found that the miss rate and average access time of the column-associative cache are comparable to those of the traditional two-way set-associative cache.

Predictive sequential associative cache

The PSA cache, proposed by Calder et al.,[5] also uses bit selection and a sequential search. This cache, however, uses the mapping approach instead of the swapping approach to determine search order. The steering bit table determines the entry point for a search. The PSA cache reads the steering bit using a prediction index produced from predictive sources available at earlier pipeline stages, so it determines the search order when the cache is accessed. This approach

Multiple MRU block technique. As we described in the box on set-associative caches, the cache system usually searches the candidate locations in a set in the order of importance. The MRU caches maintain only one order for each set. This means that for any reference mapping into the set, the search begins from the same entry point—the MRU location for the set (see Figure 1a). Thus, they suffer from a low first hit rate in some cases. For example, consider a long sequence of references with an interval of d between the addresses of each pair of consecutive references. If the sequence can fit into a direct-mapped cache, subsequent references to any of these locations will result in first hits. In an MRU cache with block size b, however, 1/(b/d) of all the subsequent references will be non-first hits. The worst case of $b = d$ will have no first hits; in fact, all hits will be the nth hits. The problem

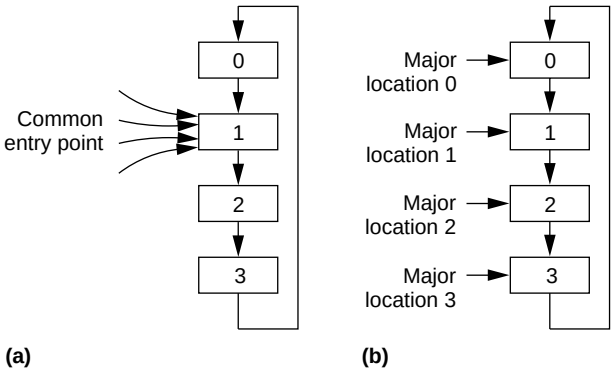


Figure 1. One entry point (a) and multiple entry points (b).

Cache schemes (continued)

eliminates swapping without lengthening the cache access cycle. Calder et al. examines four prediction sources: effective address, register contents and offset, register number and offset, and instruction and previous references. The effective address is usually impractical, however, especially in a pipelined cache.

In addition to the rehash bit, which is similar to that of the column-associative cache, the PSA cache uses the MRU bit to implement a true LRU replacement. It uses the rehash bit only to filter out unnecessary probes.

The PSA cache has the same miss rate as a two-way set-associative cache. However, simulation results show that for an 8-Kbyte cache, the PSA cache has a slightly longer average access time than the column-associative cache, unless it uses the effective address as the prediction source.[5] This is due to the predictive nature of the PSA cache: sometimes the prediction is incorrect. Calder et al. does not report performances for other cache sizes or extend the PSA cache to higher associativity.

MRU cache

Both Kessler et al.[1] and Chang et al.[8] propose cache schemes that address associativity greater than two. Though both are named the MRU cache, they differ significantly from each other.

Kessler's scheme uses a sequential search and determines the search order with a mapping approach. The cache system maintains an MRU list, indexing it with the effective address, to give information on the search order. Because the cache system must fetch the MRU information before accessing the cache, either the cache access cycle is lengthened or one more cycle is necessary to access the MRU list. Therefore, Kessler et al. states that this scheme may not be suitable for first-level caches.

Chang's MRU cache, proposed for a multichip implementation of the cache in CMOS System/370, is an improvement of the traditional four-way set-associative cache. This cache uses a parallel search. It uses MRU information not to determine the search order but to guide the speculative selection of data. When the cache system fetches the candidate data items from data banks, it looks up the MRU table and uses the MRU information to speculatively select the desired data from the four candidates. Concurrently, it reads and compares the tags. If the system selects the wrong data item, it will detect the error on the next cycle and use the result of the two-cycle backup access. This design also uses MRU information in the address translation using a translation look-aside buffer. Though Chang et al. reports that this scheme reduces access time by 30 to 35 percent, this estimation is for their multichip implementation.[8] The reduced access time is still larger than that of a direct-mapped cache due to the multiplexing of data.

Both MRU cache schemes suffer from a serious drawback: the performance of an MRU cache can often be worse than that of a direct-mapped cache for long sequences of consecutive addresses, a common occurrence.[4] In some extreme cases, it is possible for all the hits to be the nth hits for an MRU cache with an associativity of n. These may result in a longer average access time when n is greater than two.

Skewed-associative cache

The previous schemes all increase associativity by increasing the number of candidate locations in a set. In contrast, the skewed-associative cache, proposed by Seznec,[2] increases associativity in an orthogonal dimension, using carefully selected skewing functions instead of bit selection to determine candidate locations. Different mapping functions operate on different banks. This cache can reduce the possibility of conflicts by mapping onto different sets a group of conflicting references that a traditional cache would map onto one set. Seznec finds that the two-way skewed-associative cache has a miss rate close to that of a four-way set-associative cache.

The major disadvantages of the skewed-associative cache are a longer cycle time and the mapping hardware necessary for skewing. Seznec only reports performance for caches in the range of 4 to 8 Kbytes.

here is that a block in a set may have different importance to different references mapping into the same set, but the single MRU search order for the set does not account for this.

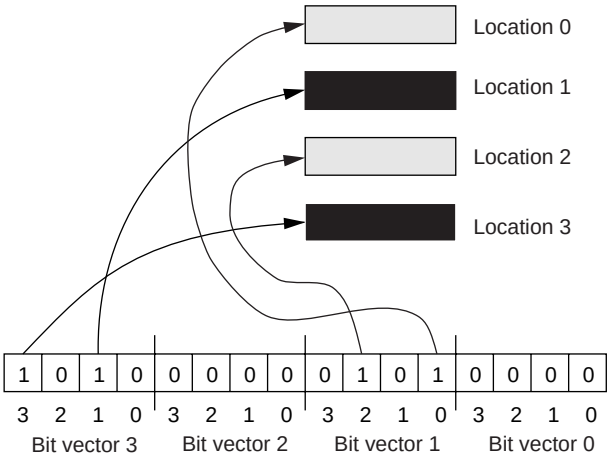


Figure 2. Index for a set in a four-way SMC.

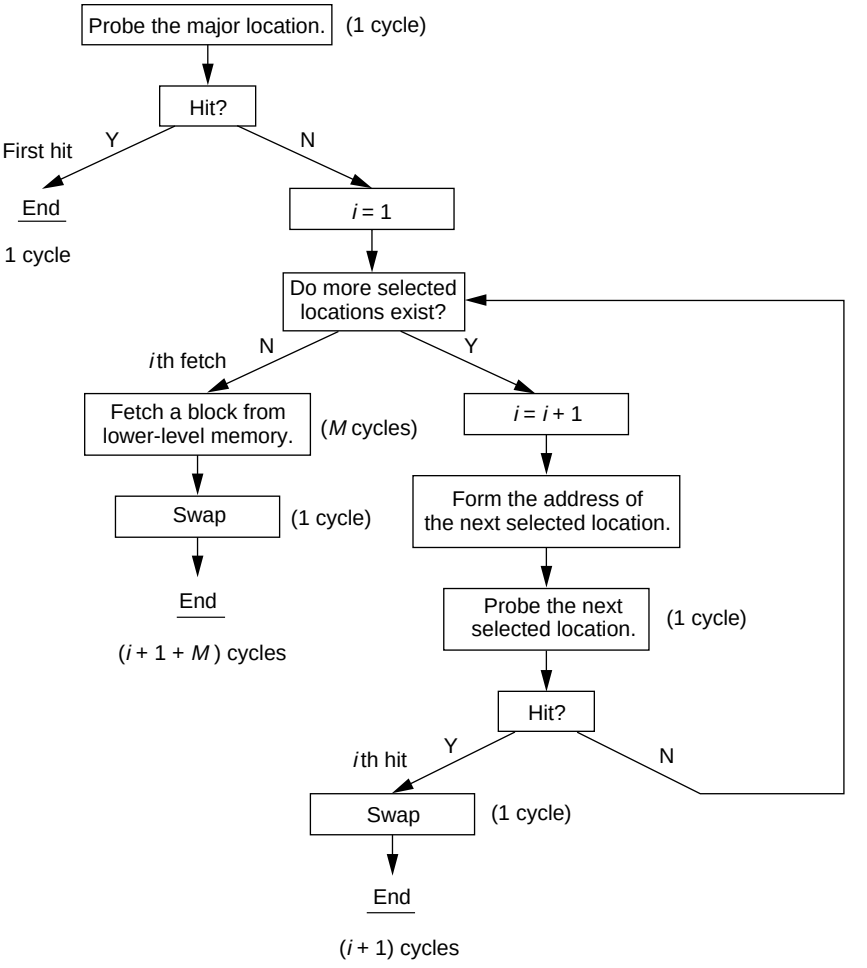


Figure 3. Search algorithm of the SMC. *M* represents the miss penalty.

We propose a multiple MRU block technique to solve this problem. In our approach, as in a direct-mapped cache, the major location is the location onto which a reference is mapped. The other locations in the same set are nonmajor locations for the reference. We classify references to one set into different groups in terms of major locations, with each group of references corresponding to one major location. Maintaining an independent search order for each group of references, instead of one search order for all, would optimize performance. The MRU block is the most important one in a search order, and the possibility that a reference will find a hit in a location other than the MRU block is very small. Therefore, we only keep information about multiple MRU blocks.

In our scheme, when the cache services a reference, the major location for the reference serves as the entry point for the search. We have multiple entry points, each corresponding to a group of references (see Figure 1b). We use the swapping approach to guarantee that the MRU block always remains in the major location after a reference, though other data blocks may later replace it. The system performs a single swap operation only when a nonfirst hit or a miss occurs. Therefore, in most cases, no swap is necessary.

During swapping, the cache cannot respond to accesses from the CPU. Hardware support will reduce the time required to perform swaps. We use the same hardware as the column-associative cache does; this consists primarily of an additional buffer, and the cost is very small. A swap requires, on average, only one cycle. We have accounted for this one cycle in our design and simulation.

With our multiple MRU block approach, all hits for the above cases of reference sequences will be first hits, including the worst case. Designers can also use this approach to extend the original MRU caches or other schemes for performance improvement.

Sequential multicolumn cache. The SMC is an extension of the column-associative cache. It uses a sequential search approach starting from the major location. Though the nonmajor locations are much less important than the major ones, we find that the way the cache system searches them significantly affects cache performance. Our simulation experiments indicate that a naive sequential search of the nonmajor locations will degrade performance for higher associativity. Therefore, we use a search index to improve search efficiency.

A location in the set can only contain a match for a reference if the block in the location is loaded from

the lower-level memory on a miss for a previous reference that has the same major location as the current reference. We refer to these locations as selected locations belonging to the corresponding major location. We use an index table to maintain information about selected locations. The cache system uses this index table to find the addresses of locations it will examine and to skip over those locations that cannot contain a match.

The address of a selected location has two parts: the higher n' bits are the bank number, and the lower $c - n'$ bits are the relative block address within the bank. The cache system derives the relative block address directly from the lower $c - n'$ bits of x, and it generates the bank number using information in the index table during the search.

In the index table, we allocate a bit vector of length n for each location, with each bit corresponding to a location in the set. The value of a bit in the vector indicates whether the corresponding location is a selected location (except the major location). To support fast modification of the index table, we concatenate all the bit vectors for locations in a set into an item. Figure 2 illustrates the index for a set in a four-way SMC. In this figure, bit vector i (i = 0, 1, 2, 3) is the index for location i, bit j (j = 0, 1, 2, 3) in a bit vector corresponds to the jth location in the set, and a 1 in the bit indicates that the corresponding location is a selected location. The index in Figure 2 gives the following information:

- For references whose major location is 1, there are two selected locations: 0 and 2;
- For references whose major location is 3, 1 is the only selected location (there is a 1 in bit 3 of bit vector 3, but location 3 is the major location);
- For references whose major location is 0 or 2, there is no selected location.

Figure 3 shows the search algorithm of the SMC. On a reference, the search examines the major location. At the same time, the cache system reads index information about the corresponding set in the index table and uses it to form the address of the next selected location. If the first probe is a miss, the system examines the next selected location while it generates the address of the further next selected location, again using the index information. This process continues until it produces a match or until it has examined all the selected locations, in which case the system fetches a block from lower-level memory. Each probe requires one cycle. For a nonfirst hit or a miss, a swap operation is necessary to move the new MRU block to the major location for the current reference.

The index table and block number generation require hardware support. For an n-way SMC with S sets, the size of the index table memory is S words $\times$ n $\times$ n bits, with each word corresponding to a set. Figure 4 shows the organization of the index mechanism in a four-way SMC. Upon a reference, the mechanism fetches the corresponding word in the index memory, latches it into IR2, and latches the corresponding bit vector into IR1. It reserves the word in IR2 for use in updating the index memory if necessary, whereas the block number logic uses the bit vector in IR1 immediately to generate the block number of the next selected location. These operations

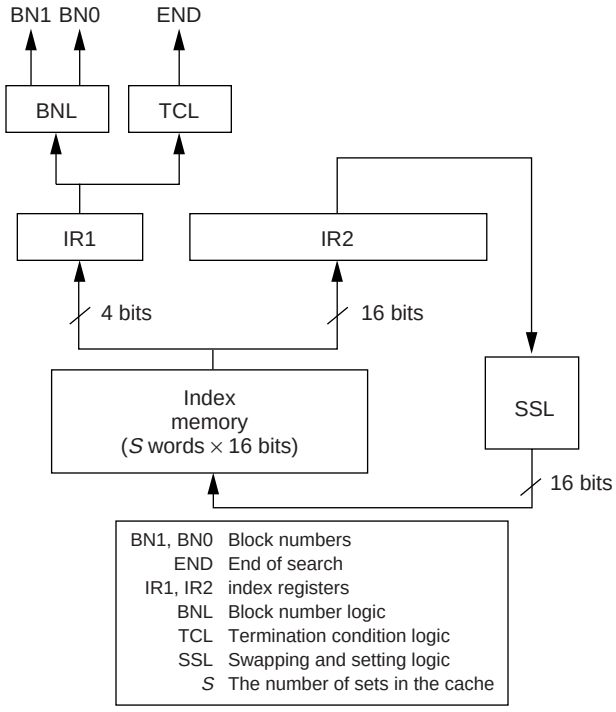


Figure 4. Block diagram of the index mechanism in a four-way SMC.

proceed in parallel with the probe of the major location, and finish in the same cycle. At the end of this cycle, the index mechanism clears the corresponding bit in IR1. It uses the content in IR1 again in the next cycle to generate the block number of a further selected location, if there is one.

The BNL is actually a priority encoding logic. It can be implemented with 10 AND gates and two OR gates, and with a delay of two levels of gates. Because the size of the index table is only 1/32 of the cache size if the block size is 16 bytes, the access time of the index table is much smaller than that of the cache memory. For example, Wilton and Jouppi[9] reports that reducing the size of a direct-mapped cache by half, in the range of 4 to 128 Kbytes, reduces the access time of the cache memory by 10 to 15%. Therefore, the index memory can be implemented in such a way that the access of the index memory and the generation of the block number do not become the critical path.

The END signal that the termination condition logic generates controls the termination of the search process. The END is valid when there are no unprobed selected locations. The TCL can be implemented with two AND gates and two OR gates by using some signals generated in the BNL. With these supports, the search engine is quite simple and adds little complexity to that of a direct-mapped cache.

The cache system updates the index memory when a swap or a miss occurs. The swapping and setting logic component, which is actually a 16-bit, 6-input multiplexer, performs this function. When a swap occurs, the SSL must swap the contents of the corresponding two bits in each bit vector for the

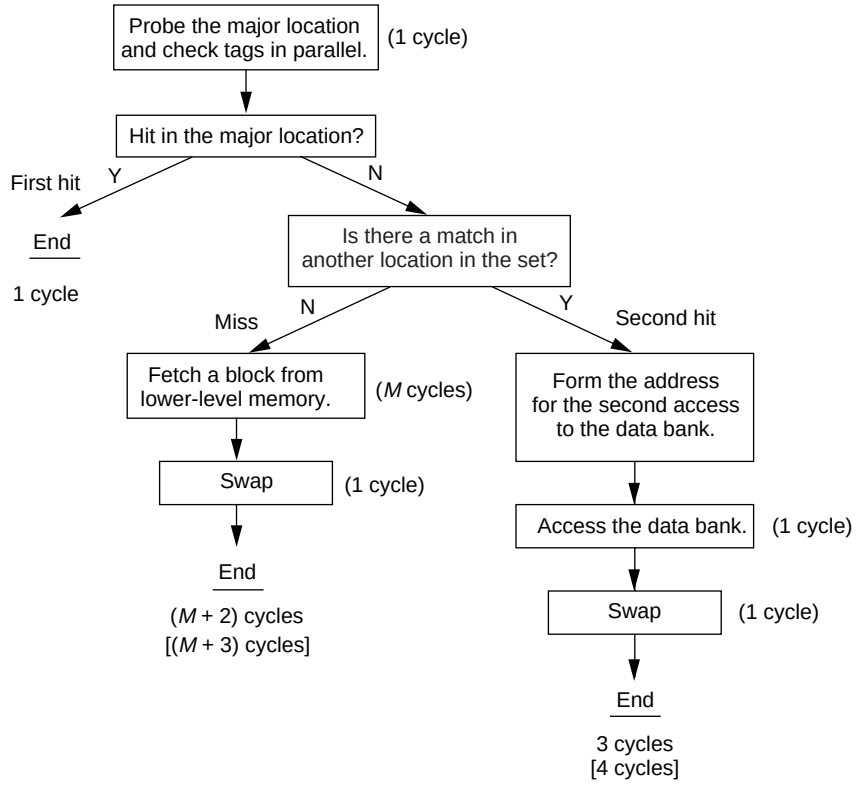


Figure 5. Search algorithm of the PMC. *M* represents the miss penalty. Square brackets indicate the time required for the PMC 2.

set, while the others remain unchanged. The index mechanism does this by writing the content in IR2, which contains the bit vectors for the set under consideration, back to the index memory, and controlling the multiplexing in the SSL.

When a miss occurs, the system fetches a new block into a location in the set. This sets the corresponding bit in the bit vector for the major location of current reference, and clears the corresponding bits in remaining bit vectors for the set. Again, the index mechanism does this by writing IR2 back to the index memory, with one bit of the multiplexer in SSL selecting a 1, three bits of the multiplexer selecting 0s, and the remaining selecting the content in IR2. The mechanism can perform these operations during the swap operation or the fetching of the new block, and hence do not increase the access time of the SMC.

A four-way SMC does not require much hardware beyond that included in the column-associative cache: an index memory whose size is only 1/32 of the cache size if the block size is 16 (and 1/64 if the block size is 32), two registers (one of 16 bits, the other of 4 bits), a 16-bit multiplexer, and a dozen AND/OR gates. Compared with a direct-mapped cache, an additional swapping buffer is necessary. The cost of this hardware is small and acceptable.

Parallel multicolumn cache. A PMC separates the implementation of the tag array from that of the data array. Though it keeps the data memory in one bank, it organizes the tag memory into *n* banks. The cache system accesses the data bank sequentially, one word at a time, but performs tag checking in parallel. This decoupling makes it possible to dispatch data speculatively to the CPU. No multiplexing operation on the data is necessary.

The PMC uses a parallel search, as shown in Figure 5. On a reference, it reads the data in the major location from the data bank, the same way a direct-mapped cache does, and dispatches it to the CPU. At the same time, it reads and checks *n* tags in parallel, using a method similar to that of a conventional *n*-way set-associative cache. A first hit occurs if the tag for the major location matches the tag of the reference. If any of the other tags match the tag of the reference, a second hit occurs, requiring one more cycle to access the data bank again for the desired data. The cache system generates the address for this access using the outcome of the tag checking. If the reference is a miss, it will fetch a new block from lower-level memory. As in the SMC, a swap operation is necessary in the case of a miss or a second hit.

The PMC generates the address for the second access to the data bank after tag checking with a delay of an OR gate. This adds little to the cycle time in a VLSI processor that does not implement optimistic execution. If the processor uses optimistic execution, however, the cache system checks tags in the next cycle. The hardware for generating the address for the second access adds a delay to the cycle time. To obtain the same cycle time as that in a direct-mapped cache, we allocate a complete cycle for the generation of the address for the second access. In this case, four cycles, not three, are necessary for a second hit. Figure 5 shows the time required for this case in square brackets. In our simulation, we consider both cases, which we call the PMC 1 and the PMC 2.

The PMC essentially combines the direct-mapped cache and the traditional *n*-way set-associative cache with the application of the multiple MRU block technique. It inherits both the short cycle time of the direct-mapped cache and the parallel search of the traditional *n*-way set-associative cache. In complexity and hardware cost, the PMC falls between the direct-mapped cache and the traditional *n*-way set-associative cache. It eliminates the multiplexing logic in a traditional *n*-way set-associative cache and organizes the data array into one bank, but adds a buffer for swapping.

Performance evaluation

We chose to compare the performance of our cache schemes with that of the column-associative cache because that cache performed best in two other comparison tests.

Agarwal and Pudar[4] compared the performance of the hash-rehash cache, the column-associative cache, and Jouppi's victim cache[10] by simulation using ATUM (Address Traces Using Microcode)[11] traces. The results show that the column-associative cache has the lowest miss rate of the three. Calder et al. used the SPEC92 benchmarks to compare the performance of the hash-rehash cache, the column-associative cache, Kessler's two-way MRU cache, and the PSC.[5] The column-associative cache proved to have the lowest average access time. An exception was the PSC when it uses the often impractical effective address as the predictive source.

For our comparisons, we used the ATUM traces, which include traces for the following 10 programs: dec0, fora, forf.3, fsxzz, macr, memxx, mul8.3, pasc, spic, and ue02. These traces comprise realistic workloads and include both operating system and multiprogramming activity. Agarwal and Pudar used the ATUM traces to evaluate the column-associative cache. Thus, using the same traces would be fair for our comparisons. We extended the Dinero III simulator extensively to support our schemes and the column-associative cache.

We tested associativities of 4, 8, and 16, and cache sizes from 1 to 128 Kbytes. We chose a block size of 16 for our simulation, the same as that in Agarwal and Pudar's simulation.[4] We also studied performance of the SMC and PMC for miss penalties ranging from 20 to 100 cycles with the cache size fixed at 4 Kbytes.

Performance metrics. The miss rate and the average access time are two important measurements of cache performance; we used both in our evaluation. The average access time, though dependent on the cache's organization and on a particular address stream, reflects the actual performance of the cache design more directly and more accurately than does the miss rate. Therefore, we paid more attention to this measurement in our evaluation.

Suppose M is the miss penalty (in cycles), R is the total number of references in a trace, H_i is the number of ith hits, and F_i is the number of ith fetches in the simulation. Also suppose that T_{COL}, T_{SMCn}, T_{PMCn}, and T'_{PMCn} are the average access times for the column-associative cache, the SMC (with associativity n), the PMC 1, and the PMC 2. Based on Agarwal and Pudar's work[4] and the search algorithms described earlier, we have

$$T_{\mathrm{COL}} = \frac{1}{R}\left[H_1 + 3H_2 + (M+1)F_1 + (M+3)F_2\right],$$

$$T_{\mathrm{SMC}n} = \frac{1}{R}\left[H_1 + \sum_{i=2}^{n}(i+1)H_i + \sum_{i=1}^{n}(i+1+M)F_i\right],$$

$$T_{\mathrm{PMC}n} = \frac{1}{R}[H_1 + 3H_2(M+2)F],$$

$$T'_{\mathrm{PMC}n} = \frac{1}{R}[H_1 + 4H_2(M+3)F]$$

The PMC has only one kind of fetch, whose number we denote by F. In our simulations, we accumulated the values of H_i, F_i, and R, and calculated T_{COL}, T_{MSCn}, T_{PMCn}, and T'_{PMCn} using these formulas.

To compare the performance of our multicolumn caches with that of the direct-mapped cache and the column-associative cache, we define improvement in average access time over the direct-mapped cache (denoted by *IMP*) as follows:

$$IMP = \frac{T_{\mathrm{DIR}} - T}{T_{\mathrm{DIR}}} \times 100\%.$$

Here, T_{DIR} is the average access time of a direct-mapped cache, and T is T_{COL}, T_{SMCn}, T_{PMCn}, or T'_{PMCn}, depending on the cache under consideration. We used the same cache size to calculate T_{DIR} and T for *IMP* calculations.

Simulation results. Figures 6a, 6b, and 7 (next page) show that both the SMC and PMC can reduce the miss rate and the average access time. Both have faster average access times than the direct-mapped and column-associative caches. For an associativity of four and a cache size of 4 Kbytes, the *IMP* of the SMC was 9.8%, the PMC 1, 10.9%, and the PMC 2, 7.0%. In contrast, the *IMP* of the column-associative cache was only 4.3%. The SMC has the highest *IMP*s: 12.7% for $n = 4$ and 15.8% for $n = 16$ when the cache size is 16 Kbytes. When n increases from 4 to 8, the gain in performance is moderate, and may still be worth exploiting in some designs. Nevertheless, there is only a small gain in performance when n increases to 16 or larger.

For $n = 4$, the performance of the PMC 1 is comparable to that of the SMC. The performance improvement of the PMC 2, however, is much lower than that of the PMC 1 due to the one extra cycle from second hits. This is also true for the cases of $n = 8$ and $n = 16$, though we did not include the performance of the PMC 1n these cases due to space limitations. Therefore, when the miss penalty is 20 cycles, the SMC scheme is more appropriate for a pipelined cache than is the PMC scheme.

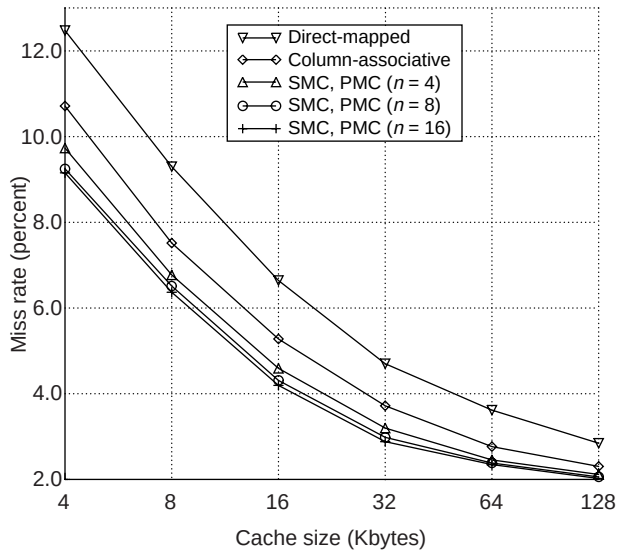
Figure 7 shows that the SMC and PMC can effectively reduce the average access time for both small and large caches. The best improvement occurs with cache sizes in the range of 4 to 32 Kbytes, currently common sizes for on-chip caches.

Figure 8 shows the *IMP* as a function of the miss penalty, from which we conclude the following:

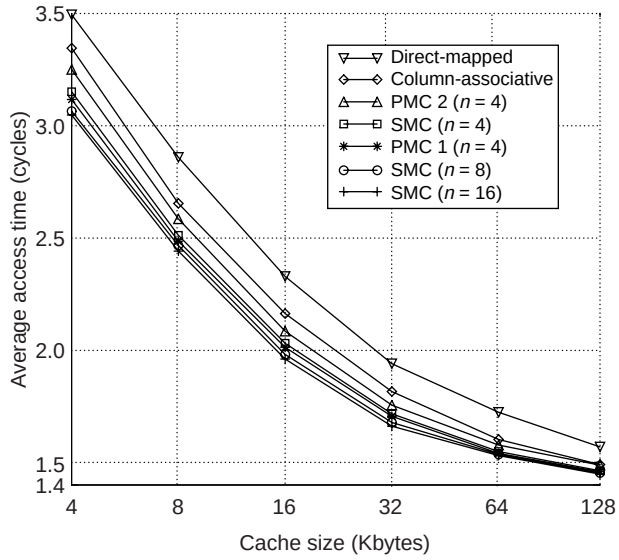
- The curves for the SMC and PMC rise more sharply than that of the column-associative cache as the miss penalty increases, which means that the multicolumn approach brings more benefits for larger miss penalties.
- The improvement of the PMC 1 approaches that of the SMC as the miss penalty increases. Therefore, the PMC scheme works well for a pipelined cache when the miss penalty is large.
- There is still a relatively large gain in performance when n increases from 4 to 8 and the miss penalty is large. The gain is much smaller, however, when n increases to 16.

TWO MAJOR PERFORMANCE BOTTLENECKS

for implementing set-associative caches are the multiplexing operation to select the data item using a parallel search, and the delay for tag comparisons using a sequential search. A



(a)



(b)

Figure 6. Miss rates (a) and average access times (b) for different cache sizes and schemes, averaged over 10 ATUM traces. The miss penalty is 20 cycles, the block size, 16 bytes.

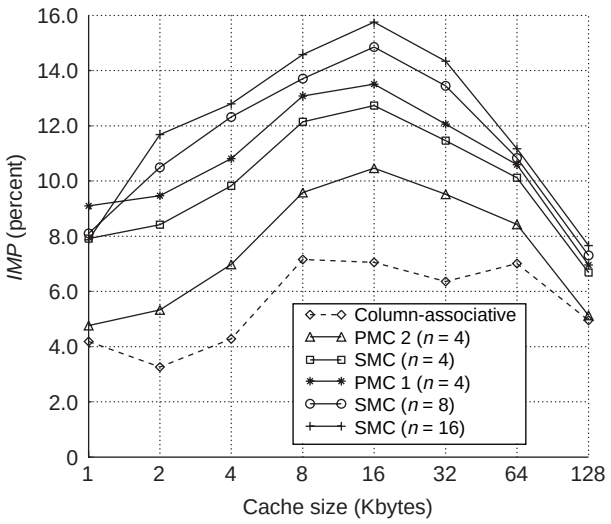


Figure 7. *IMP*s for different cache sizes and schemes, averaged over 10 ATUM traces. The miss penalty is 20 cycles, the block size, 16 bytes.

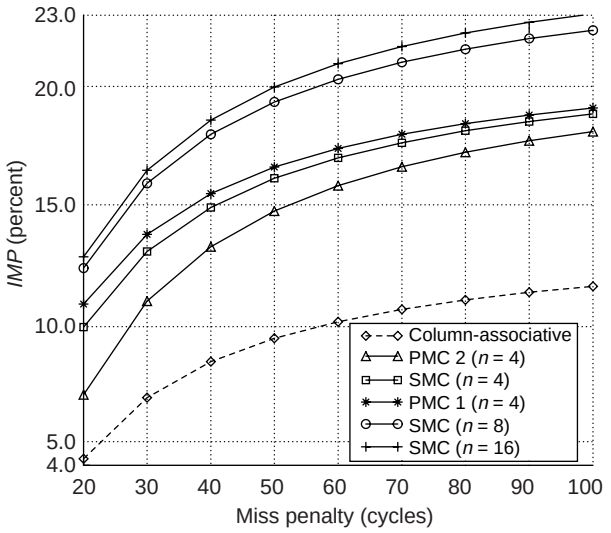


Figure 8. *IMP*s for different miss penalties, averaged over 10 ATUM traces. The cache size is 4 Kbytes, the block size, 16 bytes.

direct-mapped cache does not need the multiplexing, and a hit is always the first hit. However, the miss rate is significantly higher than that of an *n*-way set-associative cache. Our multicolumn cache system gains the advantage of direct-mapped caches by first checking the major location while allowing high-degree *n*-way associativity by minimizing the number of sequential searches. The additional hardware cost is small and acceptable.

We are currently conducting simulations to evaluate a group of representative cache schemes to provide insight into how close to optimal each design is.

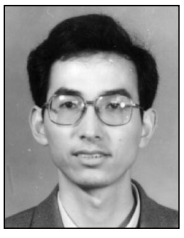
Acknowledgments

We thank Norman P. Jouppi for his technical insights and discussions on issues related to this project.

This work was supported in part by the NSF (grant CCR-9400719), a grant from the NSF International Program Division, the National Science Foundation of China (grants 69525205 and 69473043), and the Air Force Office of Scientific Research (grant AFOSR-95-1-0215).

References

1. R.R. Kessler et al., "Inexpensive Implementations of Set-Associativity," *Proc. 16th Ann. Int'l Symp. Computer Architecture,* IEEE Computer Soc. Press, Los Alamitos, Calif., 1989, pp. 131–139.
2. A. Seznec, "A Case for Two-Way Skewed-Associative Caches," *Proc. 20th Ann. Int'l Symp. Computer Architecture,* IEEE CS Press, 1993, pp. 169–178.
3. K. So and R.N. Rechtschaffen, "Cache Operations by MRU Change," *IEEE Trans. Computers,* Vol. 37, No. 6, June 1988, pp. 700–709.
4. A. Agarwal and S.D. Pudar, "Column-Associative Caches: a Technique for Reducing the Miss Rate of Direct-Mapped Caches," *Proc. 20th Ann. Int'l Symp. Computer Architecture,* IEEE CS Press, 1993, pp. 179–190.
5. B. Calder, D. Grunwald, and J. Emer, "Predictive Sequential Associative Cache," *Proc. 2nd Int'l Symp. High Performance Computer Architecture,* IEEE CS Press, 1996, pp. 244–253.
6. M.D. Hill and A.J. Smith, "Evaluating Associativity in CPU Caches," *IEEE Trans. Computers,* Vol. 38, No. 12, Dec. 1989, pp. 1612–1630.
7. A. Agarwal, J. Hennessy, and M. Horowitz, "Cache Performance of Operating Systems and Multiprogramming," *ACM Trans. Computer Systems,* Vol. 6, No. 4, Nov. 1988, pp. 393–431.
8. J.H. Chang, H. Chao, and K. So, "Cache Design of a Sub-Micron CMOS System/370," *Proc. 14th Ann. Int'l Symp. Computer Architecture,* IEEE CS Press, 1987, pp. 208–213.
9. S.J.E. Wilton and N.P. Jouppi, *An Enhanced Access and Cycle Time Model for On-Chip Caches,* Tech. Report 93/5, Digital Equipment Corporation Western Research Lab, Palo Alto, Calif., 1993.
10. N.P. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," *Proc. 17th Ann. Int'l Symp. Computer Architecture,* IEEE CS Press, 1990, pp. 364–373.
11. R.L. Sites and A. Agarwal, "Multiprocessor Cache Analysis Using ATUM," *Proc. 15th Ann. Int'l Symp. Computer Architecture,* IEEE CS Press, 1988, pp. 186–195.



Chenxi Zhang is a professor of computer science at the Changsha Institute of Technology, China. From 1995 to 1996, he was a visiting scholar in the High Performance Computing and Software Laboratory at the University of Texas, San Antonio. He has received numerous awards for his research activities in China. Zhang's research interests are in the areas of computer architectures, simulations, and logic programming. He received his PhD in computer science from the Changsha Institute of Technology.

Xiaodong Zhang is a professor of computer science at the College of William and Mary. He has also served on the faculty at the University of Texas at San Antonio, where he established and directed the High Performance Computing and Software Lab. His research interests are parallel and distributed computation, computer system performance evaluation, and scientific computing. Zhang received his PhD in computer science from the University of Colorado at Boulder. He is a senior member of the IEEE, and he chairs the Computer Society's Technical Committee on Supercomputing Applications.

Yong Yan is a PhD candidate in computer science at the College of William and Mary. He was a research associate in the High Performance Computing and Software Laboratory at the University of Texas, San Antonio, from 1993 to 1997. Since 1987, he has extensively published in the areas of parallel and distributed computing, performance evaluation, operating systems, and algorithm analysis. Yan received his BS and MS degrees in computer science from Huazhong University of Science and Technology, Wuhan, China. He is a member of the ACM and the IEEE.

Direct questions concerning this article to Xiaodong Zhang, Dept. of Computer Science, College of William and Mary, Williamsburg, VA 23187; zhang@cs.wm.edu; http://www.cs.wm.edu/hpcs/.