LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

# Dynamic Cache Partitioning via Columnization

Computation Structures Group Memo 430

**Derek Chiou†**

**Larry Rudolph†**

**Srinivas Devadas†**

**Boon S. Ang‡**

**†Massachusetts Institute of Technology**
**Laboratory for Computer Science**
**545 Technology Square, Cambridge, Massachusetts**
**{derek,rudolph,devadas}@lcs.mit.edu**

**‡HP Labs**
**Palo Alto, California**
**boonang@exch.hpl.hp.com**

This paper describes research performed at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for this work is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-92-J-1310.

# Dynamic Cache Partitioning via Columnization

Derek Chiou† Larry Rudolph† Srinivas Devadas†
Boon S. Ang‡
†Massachusetts Institute of Technology

Laboratory for Computer Science
545 Technology Square, Cambridge, Massachusetts
{derek,rudolph,devadas}@lcs.mit.edu
‡HP Labs

Palo Alto, California
boonang@exch.hpl.hp.com

**Abstract**

This paper introduces column caching, a flexible mechanism that allows software to dynamically customize cache behavior through fine-grain control of its placement policy. For a set-associative cache, specific data can be restricted to a subset of the usual target cache set during replacement. Through this simple enhancement, column caching enables the cache to be partitioned. When done properly, this improves cache utilization through both constructive and destructive interference, leading to overall better system performance. Column caching provides a basic mechanism which can emulate many different hard-wired specializations, such as dedicated SRAM and separate temporal and spatial cache, and further offers the advantage of dynamical repartitioning under software control. This paper introduces column caching, describes possible implementations and presents a few example usages including preliminary performance numbers.

# 1 Introduction

This paper proposes *column caching* as a simple enhancement to conventional cache design that enables flexible, dynamic partitioning of a cache under software control. With this general mechanism the same cache hardware can emulate several distinct specialized hardware mechanisms that are commonly used to deal with cache conflicts when memory reference patterns do not conform to the assumptions of LRU cache-line replacement strategies. Existing solutions typically utilize separate, dedicated, associative-memory hardware structures. Unfortunately, fixed division of fast memory into separate pieces cannot achieve efficient utilization of fast memory across all applications. In contrast, column caching pools the fast memory resources, deferring its partitioning to run-time so that the system only allocates what is needed, when it is needed.

As software and microprocessor architecture evolved over the years, memory reference pattern has changed too. The presence of an increasing amount of streaming data and the incorporation of multi-threaded parallel processing into microprocessors results in memory reference pattern that have less temporal locality. For example, processing of a data stream can result in a long series of accesses to data that will not be used again in the near future and simply thrashes the cache. Another example is garbage collection, during which a large number, and possibly all, items in an execution are examined, again thrashing the cache. Multitasking and multithreading makes the problem worse by interleaving logically unrelated, but possibly physically interfering cache accesses. Such reference patterns pollute standard caches with data that will not be accessed again in the near future, potentially replacing data that will soon be accessed again.

One perception of this problem is that these applications and systems have larger cache footprints. A typical response is to adopt the brute-force solution of increasing the cache size. Although this is a logically simply solution, caches currently already occupy nearly three quarters of the chip area[17] and an even larger fraction of the transistors on a processor die[11, 5]. A better approach is to seek ways of improving cache utilization, effectively reducing application cache footprints. This paper shows, in Section 3.3, that appropriate cache partitioning can have this beneficial effect.

Static cache partitioning is an old idea. Instruction and data caches have long been split in Harvard architectures and spatial/temporal caches are becoming popular[26, 29, 16, 2, 21, 13]. The static nature of these partitionings, however, often waste resources, allocating too much to one partition and not enough to another. Column caching, with its dynamic partitioning capability is thus vital for achieving the best partitioning.

The unpredictability of caches have made them unusable for time-critical data in real-time embedded systems. Cache partitioning can overcome this unpredictability by ensuring that a time critical data region is given its own region of cache where conflict will not occur. Section 3.1 shows how the column caching mechanism can be used in this manner to emulate RAM.

The next section describes column caching, how it works, and how it is implemented. In Section 3, several examples of how column caching can be used to emulate several existing special-purpose solutions are described. Experiment results and comparison to related work can also be found in this section. As this is still an early piece of work, many open questions remain; some of these are mentioned in the conclusion (Section 4). Additional information can be found elsewhere [6, 7, 8].

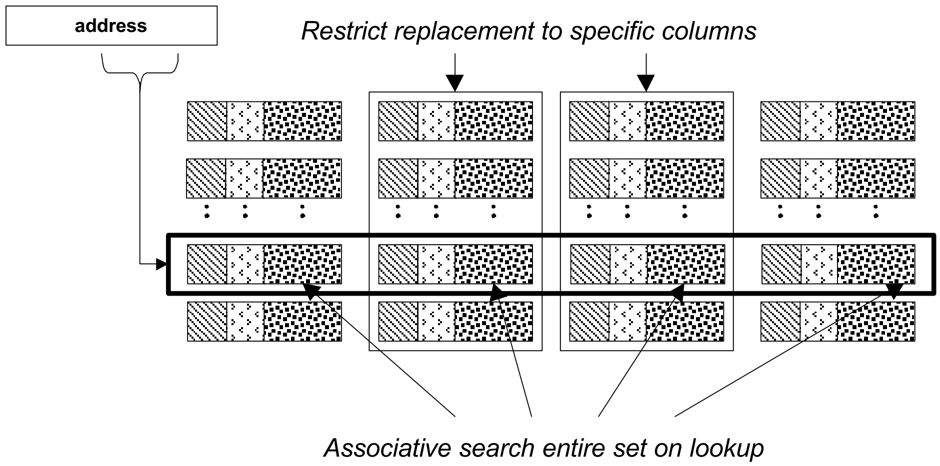


Figure 1: Basic Column Caching: Restricting Replacement in a Set-Associative Cache

# 2 Column Caching

This section describes the basic column caching mechanism that implements dynamic cache partitioning. Our unit of partition granularity is that of a "way" or *column* of a set-associative cache; hence the name "column caching." A column cache behaves just like any set-associative cache, except in its placement/replacement policy – the decision as to where a new item is placed (Figure 1). The innovation is that specific data can be restricted to be only placed into a particular subset of columns. A column cache can be almost instantaneously remapped to a normal cache by removing all restrictions (which we call *mappings*). An implementation of column caching is unlikely to lengthen critical timing because replacement decisions are only made on cache misses. A cache miss generally requires at least a few cycles to service during which the replacement decision can proceed in parallel.

Column caching allows software to specify that certain data, characterized by criteria such as address, memory operation type and instruction address, are restricted to specific columns defined by a bit-vector, one bit per column. Different data can be mapped to exactly the same columns, completely different columns or a mix of both. Column caching can also be used by hardware-implemented policies that dynamically determine in which columns certain data can reside.

## 2.1 Tints

The simplest form of column caching associates all cache-lines in a page with the same set of columns. Rather than directly mapping pages to columns, each page is mapped to a *tint*[1]. Each tint, in turn, is mapped to a set of columns. For example, an entire streaming data structure may be tinted red, while all other pages tinted blue. If the red tint is mapped to just column 1 and the blue tint to columns 2, 3, and 4, then accesses to items in blue pages will not evict cache items that came from red pages and vice-versa. A traditional cache can be simulated with a white tint specifying all columns. The use of tint introduces a level of indirection achieve the following: (i) isolate the user from machine-specific information such as the number of columns or the number of levels of the memory hierarchy and (ii) make re-mapping easier.

Tints are stored in page table entries. To change the tint of a region of memory, a process called *re-tinting*, the tint entry in each page table entry of that region needs be updated and any TLB's caching those page table entries either updated or flushed. Re-tinting, which corresponds to redefining memory region boundaries should occur very infrequently compared to remapping tints to bit-vectors.

The mapping between memory and cache columns *partitions* the cache; modifying the mappings *repartitions* the cache. A strength of column caching is its ability to quickly repartition. A tint to bit-vector table is used to map each tint to a set of

[1] We use the term tint to distinguish it from page "coloring".

columns. A table of size 4KB is required to support 1024 tints and a 32-way associative cache. Changing the tint to column set mapping can be accomplished quickly by updating the table entry.

Remapping only affects future replacements. Since an associative lookup is performed on each cache access, items present in the cache will always be found whether or not they are in the correct column. Repartitioning is thus a lazy operation. Some transient thrashing due to repartitioning is possible as active data is replaced then brought back into the correct set of columns, but can be mitigated through the use of a victim cache [19].

A protection mechanism enables user-level access to mapping structures. A process is allocated a set of columns and tints. It is free to map them in any way and at any time. By spreading the entries of the mapping table over several pages, the virtual memory system limits which tints a process may map. A bit mask, associated with each process limits the values that a process may store into the table. When a process stores a new bit vector into the mapping table, hardware automatically ANDs this value with the bit mask to produce the value stored. This bit mask requires maintenance. As columns are allocated and deallocated to a process, the operating system must update the bit mask and it must be saved and restored on each context switch. If the columns assigned to a process change, the corresponding tint-to-bit-vector table entries must be updated accordingly and the process must be notified that its columns have changed. Such notification can be performed via a callback routine provided by the application for the operating system to reclaim columns. Such routines have been proposed for user-level page management [20].

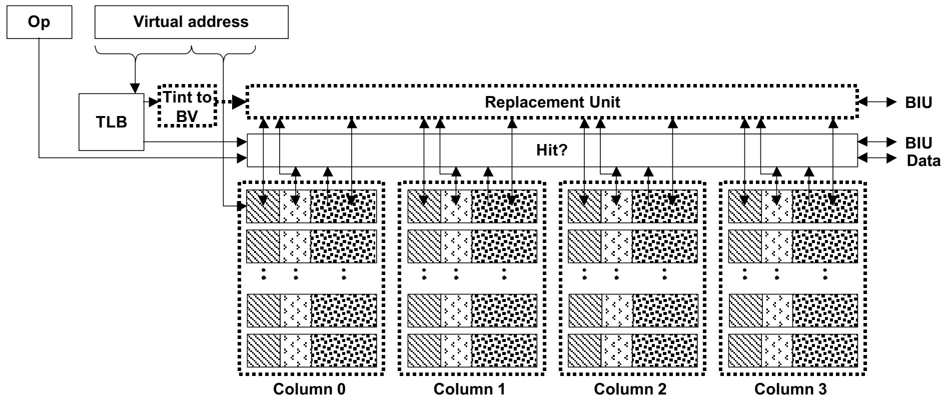


Figure 2: Basic Column Caching. Three modifications to a set-associative cache are necessary: (i) augmented TLB to hold tints, (ii) a tint-to-bit-vector unit between the TLB and the replacement unit, and (iii) modified replacement unit that uses mapping information.

## 2.2 Implementation

In a standard cache, lower-order bits are used to select a *set* of cache-lines which are then associatively searched for the desired data. There are two control units normally associated with the cache. The *hit unit* determines whether or not there is a hit on each access using the address tags and cache state tags stored in the cache itself along with the requested physical address and the opcode of the request. Though physical caches are assumed throughout, column caching works just as easily with virtual caches. The *replacement unit* determines which cache-line should be replaced if replacement is necessary.

Column caching is implemented by three small modifications to a set-associative cache (Figure 2). The TLB must store the tints, a tint-to-bit-vector unit must be added between the TLB and the replacement unit and the replacement unit must respect the mapping restrictions.

The replacement algorithm is modified to limit replacement to the columns specified by the bit vector. Column caching is compatible with any replacement policy but may require changes to certain implementations of those replacement policies to emphasize repartitioning. For example, a standard LRU algorithm may prevent the repartitioning from occurring when frequently accessed memory locations are still cached in old columns. Assume a red tint that is initially mapped to two columns. A remapping occurs that reduces the red tint to a single column. If data from red tinted pages that reside in the abandoned column are accessed sufficiently often, a standard LRU scheme would likely not replace these items. The repartitioning basically has no effect. One way to solve this problem is to not update the LRU state of a cache-line that is caching data not currently mapped to the column it resides in.

Basic column caching depends on highly-associative caches to provide sufficiently small mapping granularities. High-associative caches, however, are becoming more common, especially in embedded processors [16, 1]. Such associativity is generally implemented with Content Addressable Memories (CAMs) that require between two and four times more area but consume less power than traditional associative structures.

It may be desirable to map regions of memory smaller than a page. To achieve finer granularity mapping, additional tints can be provided within each page. Such additional tints are trivial to implement if the smaller granularity regions are constant sized and aligned. Additional tints can also be provided for different memory operation types or memory operation modifiers (additional bits such as are found in the IA-64 that further specify memory operations).

## 2.3 Effect of Multi-Level Caches

Column caching can be applied independently to each level of the cache hierarchy since they differ in size and in associativity, making it both undesirable and impractical to reuse the same partitioning across all levels. Tint information from the TLB can be easily used by all levels that are on the same chip, although each level will have its own tint-to-bit-vector. Tint information may be duplicated in each memory hierarchy level off-chip to eliminate the need to pass tint information between chips.

Inclusive caches can implement a form of column caching by just implementing column caching in lower-levels of the memory hierarchy. For example, restricting the amount of space that a region of memory can consume in L2 automatically limits the amount consumed in L1.

## 2.4 Low Associativity

Though it is desirable to have some associativity in a cache to avoid conflict misses, associativity is fairly expensive to implement, requiring area proportional to the amount of associativity. Modern set-associative caches are generally low in associativity, perhaps four to eight way, making each column fairly large. For example, the HP PA-8500 with a four way set-associative 1MB cache will have columns of 256KB each [17].

Column caching can be implemented in its basic form in a low-associative cache but then suffers from the inability to have any two pages conflict within the cache since two physical pages do not necessarily map to the same sets in the cache. Even if they do, the amount of space that those pages take up in the cache can only vary by the amount of associativity in the cache. For example, an 8 page contiguous region of memory mapped onto a 4-way set-associative column cache where each column is four pages long can only be set to consume 0, 4 or 8 pages.

Using a separate cache address to specify the sets in each column, i.e. *retargetting* [7], can dramatically reduce this problem by allowing each page to reside in a different cache page within each column. A simple version makes the cache address a cache page number for each column. Retargeting enables any set of pages to share the same region of the cache. A level of indirection similar to tints either in the page table entries or in the TLB that is then converted to the respective page numbers can simplify remapping. Separate cache addresses are needed at each level of the memory hierarchy that supports retargetting. An identical mechanism to reduce conflicts in direct-mapped caches has been independently proposed [28].

Using retargetting, our 8 continuous pages example can be mapped to 0, 1, 2, 3, 4, 5, 6, and 7 cache pages by mapping the 8 pages across all four cache pages in column 0, 2 cache pages in column 1 and 1 cache page in column 2 (Figure 3). To map to 1 page, set the tint-to-bit-vector to replace to column 2; to map to 3 pages set the tint-to-bit-vector to replace columns 1 and 2; to map 7 pages set the bit vector to map columns 0, 1, and 2. Of course, the extra column is still available for mapping.

Such a mechanism requires that the cache addresses be generated before the cache is accessed, since the cache addresses determine where to look in the cache for the data. This fact, however, makes it expensive to remap a page into a different cache page, since the original cache page must be flushed to avoid coherence problems.

# 3 Using Column Caching

With column caching, evictions occur only in the partition where the newly inserted data can reside. By careful mapping, memory regions can be isolated in the cache to eliminate replacement errors. This simple ability enables the emulation of more complex mechanisms.

In this section we examine three uses of column caching: scratchpad memory emu-

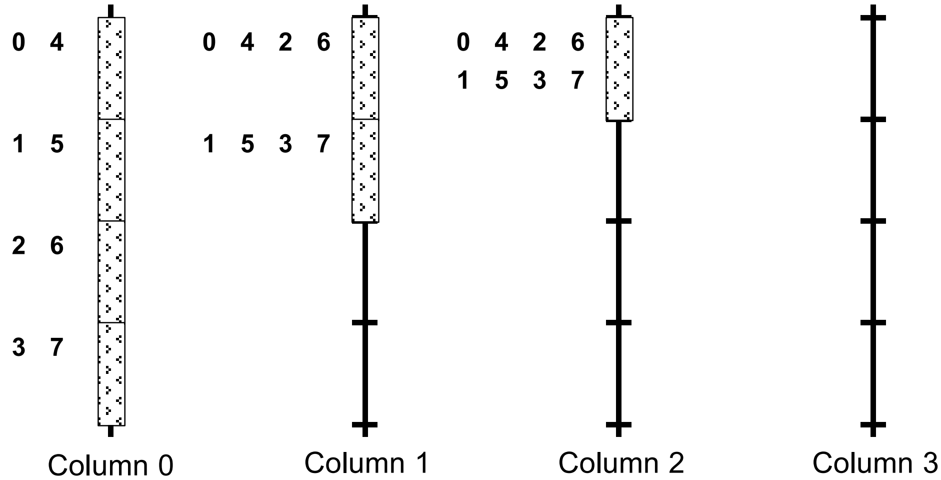


Figure 3: In column 0, pages 0 and 4 are mapped to cache page 0, pages 1 and 5 are mapped to cache page 1, pages 2 and 6 are mapped to cache page 2 and so on. Such a mapping, made possible by retargeting, allows a three column cache to vary the amount of cache space allocated to this eight page region of memory to any integer number of pages up to 7.

lation, optimization of multitasking/multithreading and optimization of stream-based computing.

## 3.1 Scratchpad Memory

Scratchpad memory is explicitly addressed and managed memory found in many embedded processors and virtually all DSPs. Since the performance of scratchpad memory is extremely easy to predict, it is used extensively to hold the data of timing-critical applications. For programming ease, a cache is used to hold data for which predictable access time is less important.

Though both types of memory are generally useful, different applications require different amounts of each. Standard caches and scratchpad memories, however, are static structures hard-wired into the processor with no way to convert one into the other, often leading to sub-optimal allocation of resources. Even if the static partitioning is customized for a specific application's needs, the next generation of software will likely have a new set of needs.

Current solutions exist that allow cache to be used as scratchpad memory by pinning cache-lines or cache columns. Cache-line pinning is available in processors such as the Cyrix MII[12], while column-pinning for instructions is provided in processors such as the Motorola 8240[22]. The 8240 column-pinning allows software to specify that a specific column in the instruction cache not be replaced. Pinning eliminates a particular cache-line as a candidate for replacement. These mechanisms, however, do not provide a way to determine whether the right data is in the cache when the pinning occurs. In addition, they can increase execution time by requiring one instruction to pin and another to unpin each cache-line.

Column caching can emulate scratchpad memory by dedicating a region of cache to an equal-sized region of data that maps to the cache with no conflicts. Thus, once the data is brought into the cache it will remain there. After it is no longer needed, a remapping will eventually cause the data to be evicted. To provide performance guarantees, software must first explicitly load the relevant data into cache, in a manner similar to a dedicated SRAM an ensure exclusive allocation of the columns. Column caching emulation of scratchpad memory is superior to scratchpad memory, however, since the data will be automatically (but lazily) recopied back to memory upon remapping.

Panda [24] has demonstrated the variation in performance of embedded software for different cache to scratchpad memory ratios. We replicate the graphs for three routines namely `dequant`, `plus` and `idct` in Figure 4. For the first two routines, optimal performance is obtained when the scratchpad memory is 1KB and the cache is 0KB. For the `idct` routine, however, optimal performance is obtained when both the scratchpad and the cache are are 512B. Fixing the ratio of scratchpad and cache

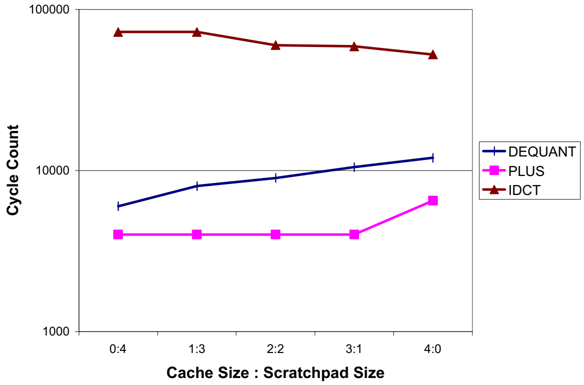


Figure 4: Performance of `Dequant, Plus, and Idct` routine with varying the ratio of cache to scratchpad memory.

causes one or more routines to have sub-optimal performance.

Using a column cache, one can dynamically vary the amount of available scratchpad memory and cache, prior to executing the routine. Thus, optimal performance can be achieved for all three routines, improving overall application performance.

## 3.2 Multitasking/Multithreading

Standard cache management allows each currently running process to consume the entire cache. In a heavily used timesharing or multitasking system, it is unlikely that a process will find any of its data in the cache when it returns from a context switch. The common solution is to schedule in large time quantums to amortize the cost of cache starts. As caches get bigger and the number of processes increases, time quantums must get even bigger still.

A few select processes can be exclusively assigned dedicated columns to avoid cold cache starts. Short time quantums are then practical for such jobs, while the other jobs being scheduled at the old time quantums will behave as usual. Thus, both overall system throughput and the response time of the select jobs improves.

Sun Microsystems Corporation holds a patent on a mechanism [23] very similar to column caching that allows partitioning of a cache between processes at cache column granularity. As part of a process state, a bit mask is specified that indicates which columns can be replaced by that process. The Sun technique is limited to partitioning between processes and thus cannot emulate scratchpad memories or help within a single application.

To demonstrate how column caching can improve multitasking, consider two concurrently executing applications, GZIP that requires a significant amount of cache and MGRID that does not. Assume further that MGRID is the critical application. The two applications are time sliced and scheduled in a round-robin fashion. The quantum execution time slice is varied. Figure 5 presents the results for a 32KB, 8-way set-associative cache. Two memory traces were generated by the SimpleScalar simulator[4] running -O3 optimized codes, then processed by our own cache simulator[7] at varying time quantums. Two sets of experiments were performed. The first assumed a standard LRU cache. For very small time quantums, the cache misses of each application evict data important to the other application. Larger time quantums are required to make effective use of the cache (as seen by the dip in the miss rate starting at about 4096). Such time quantums and shorter will become more common with the advent of multithreaded architectures such as the Compaq Alpha EV8.

With column caching, the MGRID application is exclusively allocated two columns and six columns are dedicated to the GZIP application. By separating the applications, the performance of the critical application is improved at the expense of the less critical application.

Partitioning the cache in such a fashion essentially breaks the cache into multiple

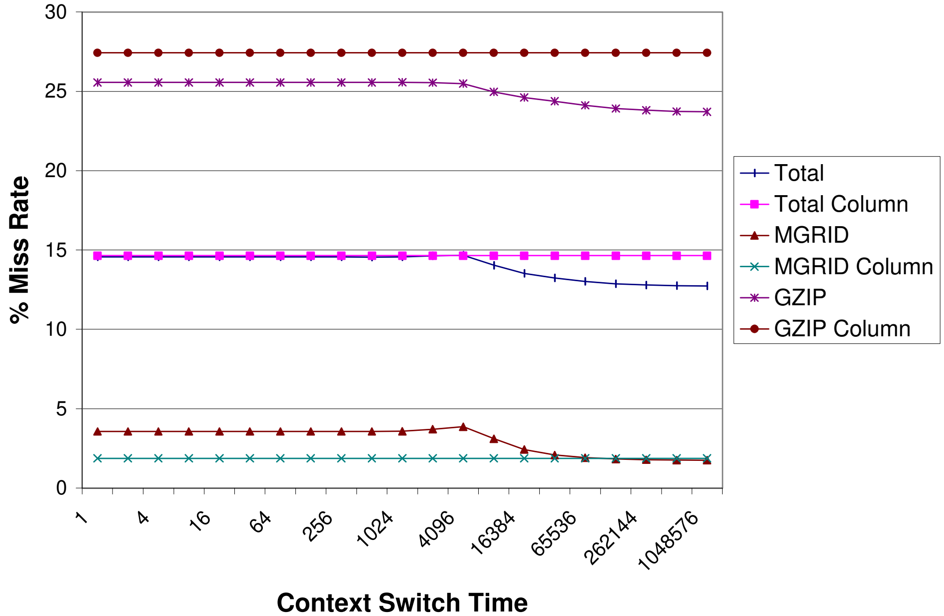


Figure 5: Comparing LRU and Column Caching with respect to multitasking cache performance.

smaller caches. Thrashing between jobs is eliminated, but the application loses the benefit of having the entire cache. Of course, if the allocated cache space is large enough, there is no degradation of performance, just the elimination of thrashing. Partitions can be overlapped to allow processes to have some dedicated cache and some shared cache. Then, the most frequently accessed data tend to end up in a dedicated region that avoids thrashing, while less frequently accessed data tends to end up in a shared region that is more quickly replaced.

## 3.3 Stream Processing

Video, audio and communications applications perform a significant number of accesses to stream data that pollute standard caches. Conventional replacement algorithms are tuned for temporal locality, assuming that all recently accessed data will be accessed again in the near future. This assumption allows stream data to occupy more of the cache than it should and replace temporal data that should not be replaced.

Many solutions have been proposed to address this problem. Perhaps the most common is a split cache, one part for spatial locality and the other part for temporal locality [26, 29, 16, 2, 21, 13]. These designs statically partition the available real-estate between the two caches. Some rely on hardware-based algorithms that separate the reference streams into one or the other cache while others keep information indicating which cache to use in the page table, allowing software to specify the mapping of memory to a specific cache.

Another solution has data bypass certain levels of the cache, allowing pollution to be restricted to a subset of cache levels. The MIPS R8000[14, 27] caches floating point data in the L2 cache but not the L1 since floating point data tends to have very little temporal data that the L1 could exploit. The QED RM7000[25] on the other hand provides caching modes that bypass L2 and L3 caches on writes, updating only memory and sometimes L1. Some processors, such as the Intel IA-64[15] family, provide instructions that can specify in which level of the cache accessed data should be cached.

Another way to determine when to bypass a level of the cache relies on the observation that for some applications only a few memory instructions cause most of the cache misses in specific levels of the cache[30, 18]. Simple hardware (or compilers) can track these "missing" memory instructions to determine whether to cache accessed data on a level-by-level, instruction-by-instruction basis. Bypass techniques attack specific cache pollution causes, but requires additional support, in the form of read buffers, to exploit spatial locality. In addition, they generally do not deal well with data that should be cached, but that should take less space that a standard replacement algorithm allocates it.

Page coloring[3, 28] uses selective mappings of virtual-to-physical addresses in order to reduce conflicts within a direct-mapped cache. It can make a direct-mapped cache perform like a low-way set-associative cache. Page coloring, however, cannot map a contiguous region of address space to a smaller region of cache space and thus cannot reduce certain stream footprints. In order to give a page of memory a dedicated cache page, all other pages that map to that cache page cannot be used. In addition, page coloring requires a memory copy to remap pages from one region of the cache to another.

By appropriate mapping, column caching can emulate a separate spatial/temporal cache, but dynamically alter the amount of space allocated to each. Of course, a dedicated spatial cache often has larger cache-lines which cannot be directly emulated by a column cache[2], but the pollution containment ability of a spatial/temporal cache is supported. Column caching can also provide cache bypass by mapping data to no columns (effectively uncached) in a certain level, though such an ability is likely not to be used often since all spatial data can be mapped to a single column that will exploit spatial locality without causing pollution.

To demonstrate the effectiveness of column caching, consider a synthetic stream application that loosely models a network router lookup code which reads data in, performs some table lookup on the incoming data and outputs the result of the lookup. Eight-word cache-lines, and a 32K lookup table are assumed.

The corresponding miss rates are shown in Figure 6. A column cache that maps the lookup table across the entire cache and the stream data to a single column limits

[2] The addition of curious caching[7] and a slightly modified memory controller, however, enables emulation of larger block sizes.

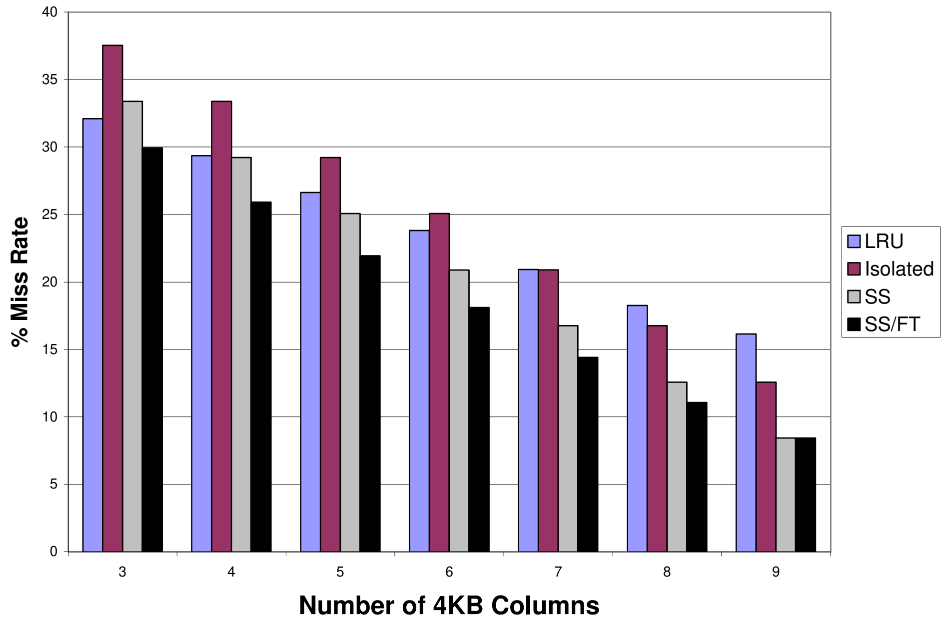


Figure 6: Miss Rates of a Synthetic Streaming Application where data is read in, looked up in a table and written out. The various mapping strategies are (i) standard LRU, no partitioning (ii) Isolated where each stream is allocated one column and the lookup table the rest of the cache (iii) Single Stream column (SS) where the two streams share the same column and the lookup table takes the rest of the cache and (iv) SS/Full Temporal, the same as (SS) but the lookup table uses the entire cache.

the stream data's footprint to a single column while reserving the lookup table the rest of the cache. This substantially improves the hit rate, enough to allow a 32K column cache to perform almost as well as a 64K LRU cache. For a 36KB sized cache, column caching lowers the miss rate from 16 percent to 8 percent.

More telling is the breakdown of the hit rates. The stream data will miss once per cache-line, creating a miss rate of 12.5% with eight-word cache-lines. Without some form of prefetching, this miss rate is unavoidable. The miss rate for the lookup table, however, can be significantly improved if the stream data is kept from interfering with the lookup data. Figure 7 gives miss rates for a standard LRU cache versus a column cache mapped in the same three ways.

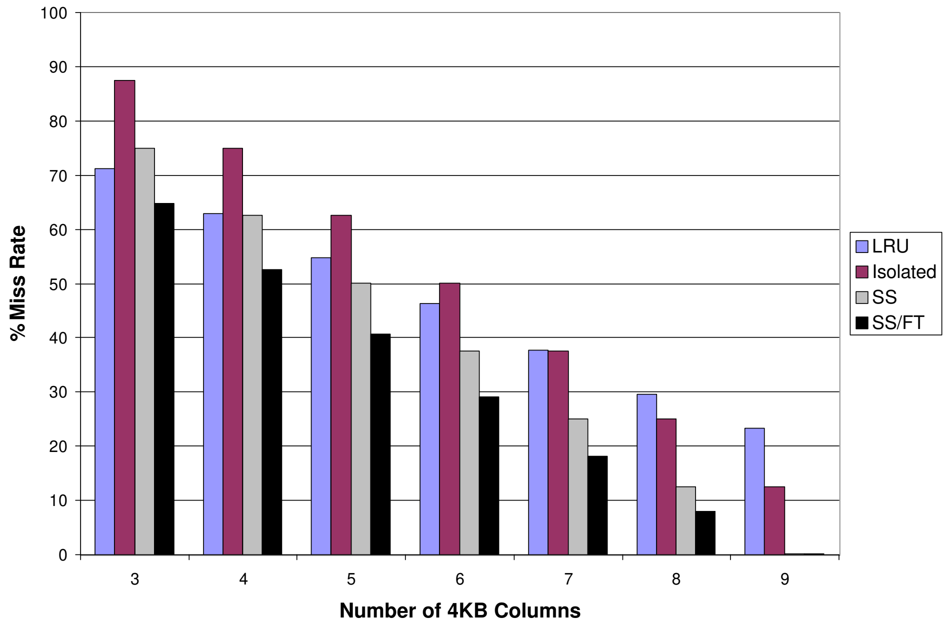


Figure 7: Miss Rates of the temporal regions of a Synthetic Streaming Application. The miss rates of the streaming data cannot be improved without prefetching of some sort.

## 4 Conclusions and Ongoing Work

Column caching is a mechanism that enables rapid partitioning of cache resources. Column caching is straightforward to implement, the base design requiring only simple changes to a standard set-associative cache. It provides software with substantial control over cache resources enabling new functionality as well as cache allocation tuning for higher performance and a reduction in resource consumption. Our preliminary investigations show that this control can be exploited to improve multi-tasking performance over a wide range of time quanta, to improve the performance of applications with streaming data, and to improve performance and predictability of real-time embedded software.

Column caching was designed to work with curious caching[6, 7], a mechanism that enables caches to incorporate snooped data. Software specifies the memory regions about which it is curious. Any memory accesses to these addresses that are observed by the snooping hardware are brought into the cache. When the curiosity mechanism is combined with the column caching mechanism, message passing buffers can be implemented *within a cache* without causing adverse cache pollution.

We have been investigating implementing memo tables[9] within caches, where the memo table key is used to generate a restricted address while the data is stored at that address as well as for use in data compaction[10]. Column caching is used to control the amount of cache for such exotic uses. Additional support, such as block-address-translation (BAT) structures found in PowerPC processors that enable the block translation of a large address space, along with techniques such as a hashed memory to reduce the memory usage of such tables, can make column caching even more useful.

We have also been looking into using columns for thread-speculation. By providing support to prevent writebacks from occurring and mapping all accessed data after a certain point to a specific set of "clean" columns (with no dirty data) where writebacks are disabled, the cache can be used to store speculative writes. If speculation is successful, the writebacks can be reenabled; otherwise, the columns can be cleared. Limiting writebacks to specific writeback-disabled columns enables cache to be used as a speculative buffer.

In NUMA architectures, non-local variables have a higher reload cost and should remain in the cache even if a more frequently accessed local variable must be evicted. In addition, data that can resolve speculation, such as predicate data, might also be kept cached over other data to reduce speculation and thereby increase potential instruction-level parallelism.

We have also been looking at how column caching can improve bandwidth to the cache by ensuring that data that will be accessed at the same time will reside in different columns of the cache. Cache structures can be designed to allow concurrent access to different columns. Mapping data to different columns enables concurrent access.

# References


[1] ARM. *The ARM 10 Thumb Family*, November 1999.

[2] K. Asanovic. *Vector Microprocessors*. PhD thesis, University of California at Berkeley, May 1998.

[3] B. K. Bershad, B. J. Chen, D. Lee, and T. H. Romer. Avoiding Conflict Misses Dynamically in Large Direct-Mapped Caches. In *ASPLOS VI*, 1994.

[4] D. Burger and T. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report 1342, Computer Sciences Department, University of Wisconsin, Madision, WI, June 1997.

[5] D. Burger, A. Kägi, and J. R. Goodman. The Declining Effectiveness of Dynamic Caching for General Purpose Microprocessors. Technical Report 1261, Computer Sciences Department, University of Wisconsin, Madision, WI, Jan. 1995.

[6] D. Chiou. RISCy Memory: Column and Curious Caching. In *Proceedings of the 1998 MIT Student Workshop on High-Performance Computing in Science and Engineering*, Jan. 1998.

[7] D. Chiou. *Extending the Reach of Microprocessors: Column and Curious Caching*. PhD thesis, Department of EECS, MIT, Cambridge, MA, Aug. 1999.

[8] D. Chiou, P. Jain, S. Devadas, and L. Rudolph. Application-Specific Memory Management for Embedded Systems Using Software-Controlled Caches. Technical Report 427, MIT Laboratory for Computer Science Computation Structures Group, Oct. 1999.

[9] D. Citron, D. Feitelson, and L. Rudolph. Accelerating multi-media processing by implementing memoing in multiplication and divison units. In *Architectural Support for Programminig Languages and Operating Systems (ASPLOS)*, Oct. 1998.

[10] D. Citron and L. Rudolph. Creating A Wider Bus Using Caching Techniques. In *First International Symposium on High Performance Computer Architecture*, Jan. 1995.

[11] D. E. Culler and A. Singh, Jaswinder Pal with Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, 1998.

[12] Cyrix. *Cyrix MII Data Book*, Feb. 1999.

[13] G. Faanes. A CMOS Vector Processor with a Custom Streaming Cache. In *Hot Chips 10*, August 1998.

[14] P.-Y.-T. Hsu. Design of the R8000 Microprocessor. *IEEE Micro*, 1993.

[15] Intel. *IA-64 Application Developer's Architecture Guide*, May 1999.

[16] Intel. *Intel StrongARM SA-1100 Microprocessor*, April 1999.

[17] D. Johnson. Techniques for Mitigating Memory Latency Effects in the PA-8500 Processor. In *Hot Chips 10*, August 1998.

[18] T. L. Johnson and W. Hwu. Run-time Adaptive Cache Hierarchy Management via Reference Analysis. In *Proceedings of the 24st Annual International Symposium on Computer Architecture (ISCA)*, June 1997.

[19] N. P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Full-Associative Cache and Prefetch Buffers. In *The 17th Annual International Symposium on Computer Architecture*, pages 364–373, May 1990.

[20] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceno, R. Hunt, D. Mazieres, T. Pinckney, R. Grimm, J. Jannotti, and K. MacKenzie. Application Performance and Flexibility on Exokernel Systems. In *16th Symposium on Operating Systems Principles, Saint-Malo, France*, October 1997.

[21] B. Lynch and G. Lauterbach. UltraSPARC III: A 600 MHz 64-bit Superscalar Processor for 1000-Way Scalable Systems. In *Hot Chips 10*, 1998.

[22] Motorola. *MPC8240 Integrated Processor User's Manual*, July 1999.

[23] B. Nayfeh and Y. A. Khalidi. Us patent 5584014: Apparatus and method to preserve data in a set associative memory device, Dec. 1996.

[24] P. R. Panda, N. Dutt, and A. Nicolau. *Memory Issues in Embedded Systems-on-Chip: Optimizations and Exploration.* Kluwer Academic Publishers, 1999.

[25] Quantum Effect Devices. *RM7000 Family User Manual v2.0.*

[26] F. Sánchez, A. González, and M. Valero. Software Management of Selective and Dual Data Caches. In *IEEE Computer Society Technical Committee on Computer Architecture: Special Issue on Distributed Shared Memory and Related Issues*, pages 3–10, Mar. 1997.

[27] SGI. *R8000 Microprocessor Product Information.*

[28] T. Sherwood, B. Calder, and J. Emer. Reducing Cache Misses Using Hardware and Software Page Placement. In *Proceedings of the International Conference on Supercomputing, Rhodes, Greece*, June 1999.

[29] M. Tomasko, S. Hadjiyiannis, and W. Najjar. Experimental Evaluation of Array Caches. In *IEEE Computer Society Technical Committee on Computer Architecture: Special Issue on Distributed Shared Memory and Related Issues*, pages 11–16, Mar. 1997.

[30] G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun. A Modified Approach to Data Cache Management. In *Proceedings of the 28th Annual International Symposium on Microarchitecture Ann Arbor, MI*, November/December 1995.