Presented at *CASES'99: Workshop on Compiler and Architecture Support for Embedded Systems*. Washington DC, October 1999.

Hardware/Software Architectures for Real-Time Caching

Bruce Jacob

Electrical & Computer Engineering
University of Maryland, College Park
blj@eng.umd.edu

*There are two fundamental problems in guaranteeing cache performance for real-time embedded systems: **conflict** and **capacity** misses. Though fully associative caches would solve conflict misses, they are too expensive to implement in embedded systems. There are two alternatives: a real-time cache (a software-managed fully associative cache with extremely large cache blocks) and a virtually addressed cache. To address capacity misses, one can dynamically (and predictably) manage the cache contents.*

Introduction

Real-time embedded systems require guaranteed performance behavior because they interact with the real world. Today's embedded microprocessors are yesterday's high-performance desktop processors; they were designed with instruction throughput in mind—not predictable real-time behavior. Therefore, they cannot guarantee performance behavior, especially for inherently probabilistic mechanisms such as caches, and so are unsuitable for use in real-time embedded systems. As a result, real-time embedded systems often run with caches disabled.

It is difficult for software to make up for hardware's inadequacy. There have been numerous studies rearranging code and data to better fit into a cache: examples include cache blocking, page coloring, cache-conscious data placement, loop interchange, unroll-and-jam, etc. [McFarling 1989, Carr 1993, Bershad et al. 1994, Carr et al. 1994, Calder et al. 1998]. Most of these mechanisms have the goal of *increasing* the number of cache hits, not *guaranteeing* the number of cache hits: not surprisingly, guaranteeing cache performance requires considerably more work.

This abstract shows that a combination of both hardware and software—variations on traditional cache and runtime system architectures—can solve the problem. The problem reduces to two components: (1) guaranteeing that there will be no ***conflict*** misses in the cache, and (2) guaranteeing that there will be no ***capacity*** misses in the cache. ***Compulsory*** misses do not present a problem because they are statically predictable. A fully associative cache can address conflict misses, and we present two relatively low-overhead alternative designs. To address capacity misses, we briefly describe a real-time cache management scheme.

Conflict Misses: Virtual Addressing = Full Associativity

Assume that through code inspection or application profiling or compiler optimization it is possible to identify a subset of the program (chosen at a cache-block granularity) that should be cached. The rest of the program will remain non-cached for the duration of application execution. To guarantee there will be no cache conflict problems during execution, it is necessary to arrange the cached items so that the maximum degree of overlap in the cache is less than the cache's degree of associativity. The intuitive picture is shown in Figure 1: there are a number of atomic objects in the program that cannot be broken up any further. Portions of these objects may be cached or uncached. Assuming the total size of the regions to be cached does not exceed the size of the cache, there are a number of potential arrangements of the objects in the memory space, each of which might yield a conflict-free cache arrangement. Finding such a conflict-free arrangement of code and data objects is a non-trivial problem, given in the following formal description:

CONFLICT-FREE CACHE PLACEMENT

INSTANCE: <memory size M, cache size C, cache associativity A, set O of memory objects, v:{block} $\rightarrow$ {0,1}>, where M and C are in units of cache blocks, $A \in Z^+ \leq C$, and an *object* is an ordered set of contiguous cache-block-sized regions. We write set O as follows: O = { $\{b_{11}, b_{12}, \ldots b_{1n}\}$, $\{b_{21}, b_{22}, \ldots b_{2n}\}$, ... $\{b_{Z1}, b_{Z2}, \ldots b_{Zz}\}$ }, where b_{ij} is the jth block-sized portion of object i (termed o_i) and has the associated value $v(b_{ij})$, which is 1 or 0 depending on whether block b_{ij} is to be *cached* or *non-cached*, respectively.

QUESTION: Does there exist a realistic mapping σ:$\{o_i\} \rightarrow Z_0^+$ such that the degree of overlap in the cache is consistent with the cache associativity (thereby producing a memory footprint that is without cache conflicts)? Each object o_i can be mapped via a function σ onto the memory space $Z_0^+ < M$. This function indicates the starting point of each object. Because the objects are collections of contiguous regions, the mapping also implicitly defines the memory location for each individual block within each object: the first block-sized region must lie at the object's memory location (i.e., $\sigma(b_{i1}) = \sigma(o_i)$). The second block-sized region must lie at the next sequential (also block-

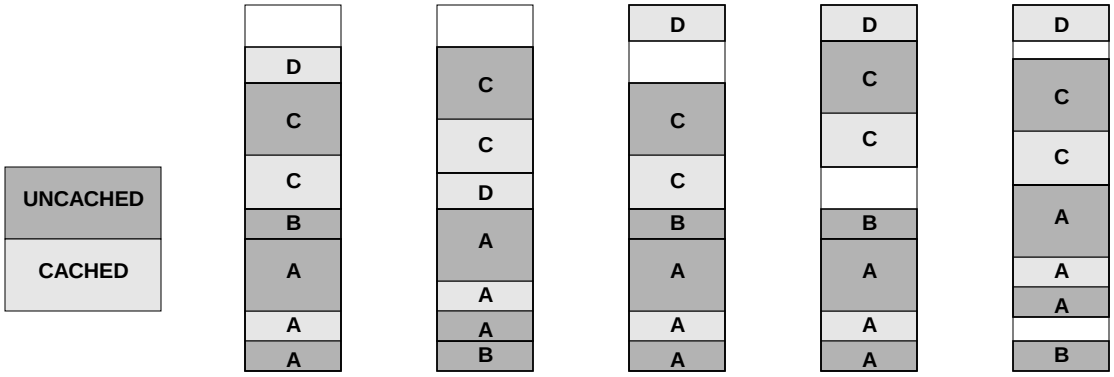

Figure 1: Possible layouts of cached and uncached objects in the memory space.

sized) memory location (i.e., $\sigma(b_{i2}) = \sigma(b_{i1}) + 1 = \sigma(o_i) + 1$). The third block-sized region must lie at the next sequential memory location (i.e., $\sigma(b_{i3}) = \sigma(b_{i2}) + 1 = \sigma(b_{i1}) + 2 = \sigma(o_i) + 2$), etc. This places the block-sized regions of each object into *C/A* different equivalence classes $\{ E_0, E_1, \ldots E_{C/A-1} \}$ which correspond to cache sets: a region's equivalence class is the cache set to which it maps, determined by its memory location modulo the number of sets in the cache (*C/A*). Therefore, by definition, $b_{ij} \in E_{\sigma(b_{ij}) \bmod C/A}$.

To have a realistic mapping σ with a conflict-free cache layout, we must satisfy the following. First, the program must fit into the memory space.

$$\left(\sum_{o_i \in O} |o_i| \le M \right) \wedge (\max(\sigma(b_{ij})) < M) \tag{1}$$

Second, there must be no overlap among objects in the memory space (informally, $\forall a \neq b$, $o_a \cap o_b = \emptyset$).

$$\sigma(b_{ij}) = \sigma(b_{xy}) \Rightarrow (i = x) \wedge (j = y) \tag{2}$$

Last, for the cached objects, the number of overlaps in the cache must be consistent with the cache's degree of associativity (e.g. for a direct-mapped cache, there should be at most one cached element in each equivalence class; for a two-way set associative cache, there should be at most two cached elements in each equivalence class, etc.).

$$\forall k \sum_{b_{ij} \in E_k} v(b_{ij}) \le A \tag{3}$$

This decision problem is NP-complete for A=1. Notice that for A=C (a fully associative cache) we have the result that all cached memory blocks map to the same equivalence class, but there can be no more than A such cached blocks, assuming that the cache is large enough. Therefore, for a fully associative cache, the decision problem is trivial: if the sum of the sizes of the objects is less than or equal to the size of the cache, Eq. (3) is trivially satisfiable, and all memory arrangements are conflict-free.

A quick conclusion is that we should use fully associative caches in embedded systems. However, fully associative caches burn too much power to be useful at large sizes (several kilobytes or larger). One alternative is a design that disassociates naming from storage in the same vein as virtual memory [Jacob & Mudge 1998a, Jacob & Mudge 1998b, Jacob & Mudge 1998c]. This design allows one to freely locate objects in a tagless SRAM without having to change their location in physical memory. Objects are relocated at the granularity of pages, which are on the order of 100 bytes (128 bytes, 256 bytes, etc.). To provide real-time guarantees, every page of the SRAM is mapped in hardware.

The architecture is illustrated in Figure 2. A translation lookaside buffer (TLB) maps the contents of the SRAM: if a chunk of data is in the SRAM, its mapping is held in the TLB. As in normal TLBs, the search is fully-associative—the TLB is a content-addressable memory (set-associative TLBs are possible and simply limit one's flexibility in placing items in the memory space but are still more flexible than an ordinary cache). The difference is that the equivalent of the page frame number (the "SRAM offset") is not actually held in the TLB. It is inferred from the slot number. Therefore, if the matching page number is found in slot 0, it refers to the first page within the SRAM, etc. The bottommost bits of the physical address are an offset into this page. If the page number is not found in the TLB, it is assumed that it is a valid physical address, and the address is sent to the primary memory system as-is. Thus, every physical address that an application generates may or may not reference an item held in the SRAM, and the caching is transparent.

Clearly, this organization supports the static management of data. One need only choose the subset of code and data

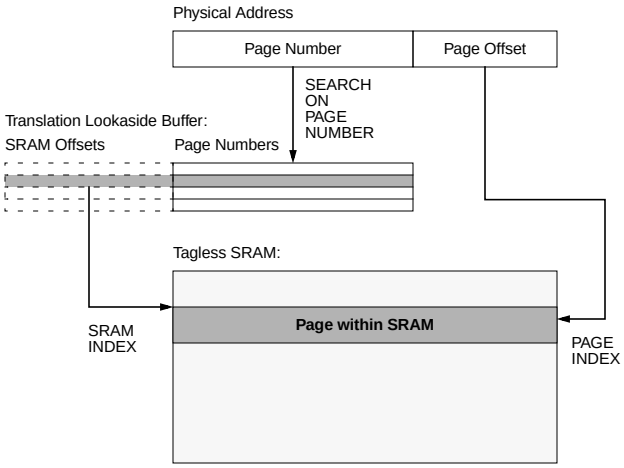

Figure 2: A real-time cache architecture.

that is to be held in the SRAM, copy that code and/or data into the SRAM, and initialize the TLB appropriately. From that moment on, all references to the cached information would go to the SRAM, not main memory. For an SRAM of 8Kbytes and a granularity of 256 bytes (the page size), the TLB would have 32 entries. Larger page sizes can be used to reduce the size of the TLB, at the expense of flexibility. This organization is also much more flexible than a scratch-pad RAM (i.e. the cache organization typically found in DSP architectures, in which an item's address determines the memory structure in which it is held [Lapsley et al. 1994]) in that it allows the arbitrary arrangement of data at a page-sized granularity, without regards to contiguity or inter-object distances. For example, items that are adjacent in the memory space can be cached or not cached without creating any cache conflicts or addressing problems.

Note that this architecture is logically equivalent to a fully associative cache (a CAM) with an unusually large block size. The differences are that the system uses physical addresses, not virtual addresses, and the cache tags in this organization are loaded by software, not by hardware (much like a software-managed TLB).

Note also that there is another way to make the decision problem solvable in polynomial time, assuming that the cached/non-cached regions have a simple organization (for example, one cached region per atomic object): if M is large enough, then we can simply choose mappings from $\{o_i\}$ to Z_0^+ such that each cached region begins exactly one cache block beyond the previous object's cached region. This fails to work when we have odd organizations, such as two arrays, one in which every other block should be cached, the other in which every third block should be cached. However, if the application has well-behaved cached/non-cached regions, M can be increased to (almost) arbitrary size by using virtual addressing. Hardware support includes a software-managed cache [Jacob & Mudge 1998a] or a traditional TLB+cache, provided that the TLB is software-managed and fully maps the cache with at least one slot to spare, using the uppermost TLB slots for cached data and the remaining TLB slot/s for translating the rest of the application in memory. This scheme requires translation for *all* memory locations, not just *cached* memory locations, which means we need a small memory space, a large TLB, a large page size, or a well-thought-out translation scheme.

Capacity Misses: An Architecture for Real-Time Cache Management

If the cache is statically managed—that is, the decision of what to cache or not cache is made statically, and the contents of the cache do not change during program execution—then all addresses are either cached or non-cached, and both types of access have absolutely deterministic access times. The challenge is to determine at compile time (or just-in-time before program execution) what will be the most important page-sized regions of memory, and to rearrange code and data so that a page's contents hold either "hot" data or "cold" data, but not a combination of both, otherwise the cache would contain cold data that is referenced infrequently. These are jobs for a compiler and code profiler, along the lines of work done on DSP dataflow models and memory buffer usage [Lee & Messerschmitt 1987, Bhattacharyya et al. 1996].

However, what if we cannot fit everything we want cached into the cache? The previous section solves conflict issues for us, but what about capacity issues? The cache can also be dynamically managed—Figure 3 illustrates an idea similar to virtual memory. All code and data in a real-time application (as well as the RTOS itself) is categorized:

1. Always to be cached
2. Never to be cached
3. Exhibits periodic locality

The first two categories speak for themselves. Hopefully, the size of category #1 is smaller than the SRAM itself. If so,

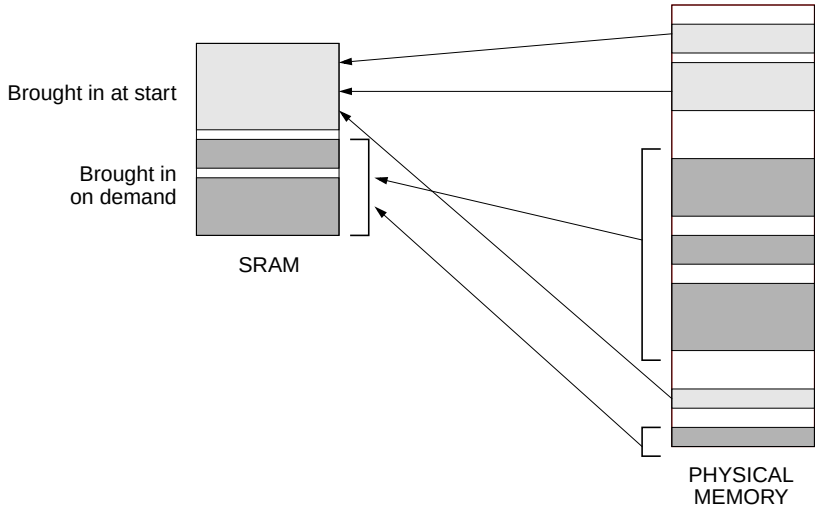

Figure 3: Dynamic management of the real-time cache architecture.

the third category represents items that exhibit predictable burstiness in their locality—items that are used repeatedly for a short duration and then not referenced at all for a long period. Examples that immediately come to mind include loop code and data. These items should be cached, but only while the loop is active—at other times the items should not occupy cache space.

The real-time management of these items is effected by placing code at their beginning and endpoints. We will call these code blocks the start-block and the end-block. The start-block brings the loop code and data (at least, that of the loop code and data that will be cached) into the cache, and sets the TLB appropriately. The end-block unmaps the cached items from the TLB and writes out any data to the memory space that requires it.

Since the size of loops is usually known in advance for most DSP algorithms, the execution time for the entire block (including start-block and end-block regions) can be calculated statically. Thus, this cache management routine is deterministic and offers real-time memory management. The issues to explore are how well one can statically identify blocks of code that are disjoint in their execution and small enough to fit into the cache at the same time as the always-cached code and data. Current DSP analysis [Bhattacharyya et al. 1998, Bhattacharyya et al. 1996] has identified such blocks; the challenge will be to do the same for general embedded systems.

References

B. N. Bershad, D. Lee, T. H. Romer, and J. B. Chen. 1994. “Avoiding conflict misses dynamically in large direct-mapped caches.” In *Proc. Sixth Int’l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS’94)*, pages 158–170, San Jose CA.

S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. 1996. *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers.

S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. 1998. “Synthesis of embedded software from synchronous dataflow specifications (invited paper).” *Journal of VLSI Signal Processing*.

B. Calder, C. Krintz, S. John, and T. Austin. 1998. “Cache-conscious data placement.” In *Proc. Eighth Int’l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS’98)*, pages 139–149, San Jose CA.

S. Carr. 1993. *Memory-Hierarchy Management*. PhD thesis, Rice University.

S. Carr, K. S. McKinley, and C. Tseng. 1994. “Compiler optimizations for improving data locality.” In *Proc. Sixth Int’l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS’94)*, pages 252–262, San Jose CA.

B. L. Jacob and T. N. Mudge. 1998a. “A look at several memory-management units, TLB-refill mechanisms, and page table organizations.” In *Proc. Eighth Int’l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS’98)*, pages 295–306, San Jose CA.

B. L. Jacob and T. N. Mudge. 1998b. “Virtual memory in contemporary microprocessors.” *IEEE Micro*, 18(4):60–75.

B. L. Jacob and T. N. Mudge. 1998c. “Virtual memory: Issues of implementation.” *IEEE Computer*, 31(6):33–43.

P. Lapsley, J. Bier, A. Shoham, and E. A. Lee. 1994. *DSP Processor Fundamentals*. Berkeley Design Technology, Inc.

E. A. Lee and D. G. Messerschmitt. 1987. “Synchronous dataflow.” *Proceedings of the IEEE*, 75(9):1235–1245.

S. McFarling. 1989. “Program optimization for instruction caches.” In *Proc. Third Int’l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS’89)*, pages 183–191.