Cache Design for Embedded Real-Time Systems

Bruce Jacob

Electrical & Computer Engineering Department
University of Maryland at College Park

blj@eng.umd.edu
http://www.ee.umd.edu/~blj/

ABSTRACT

Caches have long been a mechanism for speeding memory access and are popular in embedded hardware architectures from microcontrollers to core-based ASIC designs. However, caches are considered ill-suited for embedded real-time systems because they provide a probabilistic performance boost—a cache may or may not contain the desired data at any given moment. Analysis that guarantees when an item will or will not be in the cache has proven difficult, so many real-time systems simply disable caching and schedule tasks based on worst-case memory access time.

Yet there are several cache organizations that provide the benefit of caching without the real-time drawbacks of hardware-managed caches. These are software-managed caches, and several different examples can be found, from DSP-style on-chip RAM to academic designs.

This paper compares the operation and organization of caches as found in general-purpose processors, microcontrollers, and DSPs; it also discusses designs for embedded real-time systems.

1 INTRODUCTION

It has long been recognized that, for good performance, applications require fast access to their data and instructions. Accordingly, general-purpose processors have offered caches to speed up computations in general-purpose applications. Caches hold only a small fraction of a program's total data or instructions, but they are designed to retain the most important items, so that at any given moment it is likely the cache holds the desired item. Cache designs work on a relatively simple principle—at any given moment, a program is likely to access data that it has accessed in the recent past, or data that is nearby data that it has accessed in the recent past—and this allows one to build correspondingly simple hardware controllers that achieve significant performance boosts for general-purpose applications. However, caching has been found to be detrimental to real-time applications, and as a result, real-time applications often disable all hardware caches on the processor.

Why is this so? The emphasis in general-purpose systems is typically speed, which is related to the *average-case* behavior of a system. In contrast, real-time designers are concerned with the accuracy and reliability of a system, which are related to the *worst-case* behavior of a system. When a real-time system is controlling critical equipment, execution time must lie within predesigned constraints, without fail. Variability in execution time is completely unacceptable when the function is a critical component, such as in the flight control system of an airplane or the antilock brake system of an automobile.

The problem with using traditional hardware-managed caches in real-time systems is that they provide a probabilistic performance boost; a cache may or may not contain the desired data at any given time. If the data is present in the cache, access is very fast. If the data is *not* present in the cache, access is very slow. Typically, the first time a memory item is requested, it is not in the cache. Further accesses to the item are likely to find the data in the cache, therefore access will be fast. However, later memory requests to other locations might displace this item from the cache. Analysis that guarantees when a particular item will or will not be in the cache has proven difficult, so many real-time systems simply disable caching to enable schedulability analysis based on worst-case execution time.

One solution is to pin down lines in the cache, for hardware systems that support it. System software can load data and instructions into the cache and instruct the cache to disable their replacement. The chief disadvantage of this approach is that it is not amenable to dynamic reorganization; once data and instructions have been pinned, it is often more overhead than it is worth to reorganize the cache contents. What is needed is a flexible, low-overhead mechanism that allows data and instructions to be pinned, and that also allows the contents of the cache to change quickly, under the supervision of the operating system.

2 TRADITIONAL CACHES

A cache is a device used to speed up accesses to storage devices, including tape drives, disk drives, and memory. It works on the principle of *locality of reference*, the tendency of

Presented at the *Embedded Systems Conference, Summer 1999.* Danvers MA, June 30, 1999.

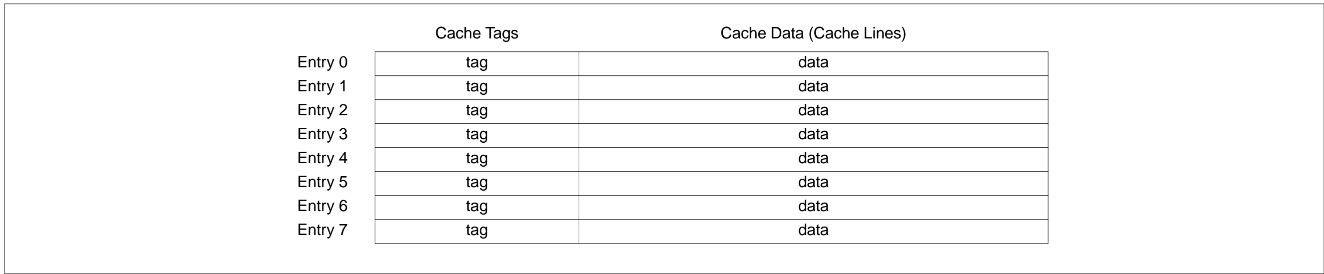

Figure 1. Basic cache structure. A cache is composed of two main parts; the cache tags and the cache data; each data entry is termed a *cache line* or *cache block*. The tag entries identify the contents of their corresponding data entries.

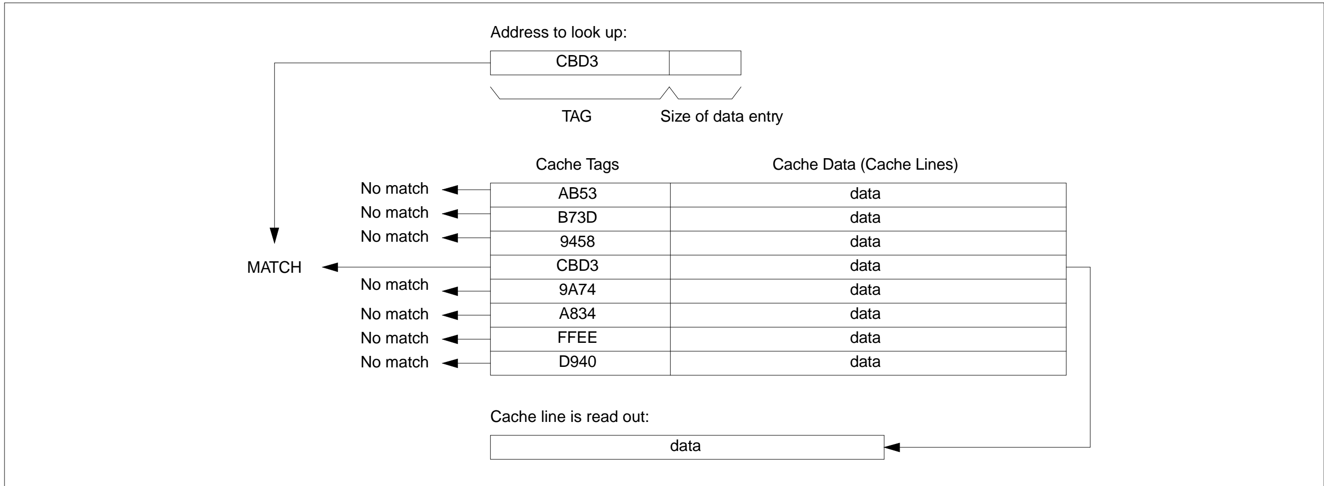

Figure 2. Fully associative lookup mechanism. This organization is also called a CAM, for content-addressable memory. It is similar to a database in that any entry that has the same tag as the lookup address matches, no matter where it is in the cache. This organization reduces cache contention but the lookup can be expenseive, since the tag of every entry is matched against the lookup address. Once the line is read out, the lower bits of the address that were not used in the tag match determine what portion of the cache line is to be sent out to the requestor.

applications to reference a predictably small amount of data within a given window of time. A storage device is built of a technology that has a certain access time and a certain cost, where faster technologies have a lower access time and typically cost more per storage unit than slower technologies. A cache for a given storage device is built from a technology that is faster than that of the storage device in question and only needs to be large enough to hold the application's *working set*—the set of instructions and data items the application is currently using to perform its computations—to be effective [2]. Most of the application accesses will be satisfied out of the cache, and so most of the time the overall access time will be that of the cache: far faster than the larger storage device.

2.1 Basic Cache Mechanics

A cache is usually composed of two parts; the *cache data* and the *cache tags*. The basic structure is illustrated in Figure 1. Since a cache is typically smaller than an entire address space, there is a possibility that any particular requested datum is not in the cache. Therefore there must be some mechanism to determine whether any particular datum is present in the cache or not. The tags fill this purpose; the tags are a list of valid entries in the cache, one per data entry. Therefore every tag entry identifies the contents in its associated data entry.

Virtually all hardware caches operate this way; one indexes the cache using the appropriate method (a cache can be *direct-mapped*, *set-associative*, or *fully associative*), and the associated tag entry indicates what datum is stored in the cache at that data entry. If the tag *matches*, i.e. if it corresponds to the requested datum, the datum in the data entry is read out. Figures 2 and 3 illustrate two different types of cache lookup procedure: fully associative and direct-mapped.

Figure 2 demonstrates a *fully associative* lookup. In this organization, the cache is essentially a small hardware database. A datum can be placed anywhere in the cache; the tag field identifies the data contents. A search checks the tag of every datum stored in the cache. If any one of the tags matches the tag of the requested address, it is a *cache hit:* the cache contains the requested data.

Figure 3 illustrates a *direct-mapped* lookup. In this organization, a given datum can only reside in one entry of the cache, usually determined by a subset of the datum's address bits. Though the most common index is the low-order bits of the tag field, other indexing schemes exist that use a different set of bits or hash the address to compute an index value bearing no

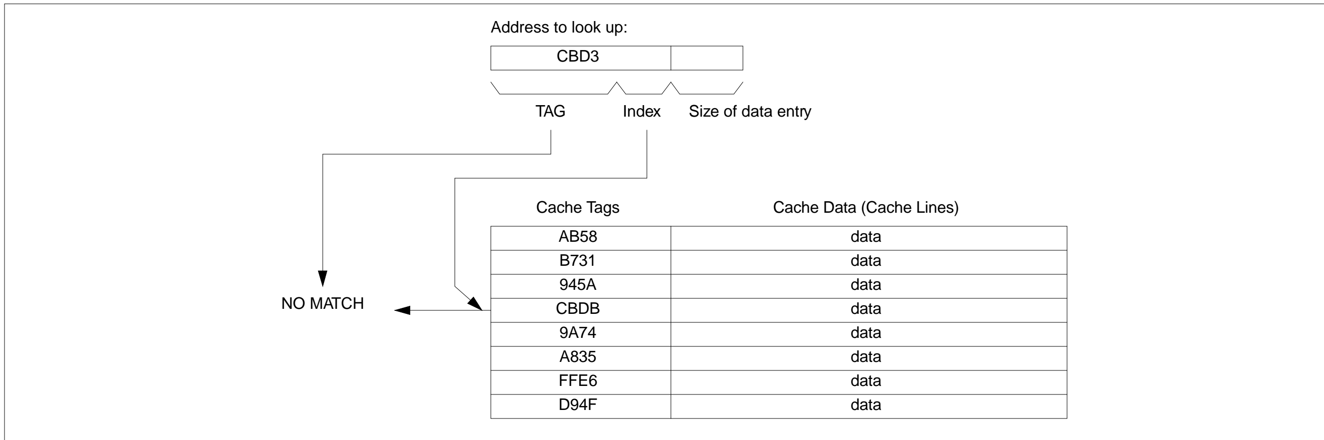

Figure 3. Direct-mapped lookup mechanism. Unlike a fully-associative organization, a particular item can only be found in one entry in the cache, if it is in the cache at all. For instance, in this cache of eight entries, the bottom three bits of the tag in the first entry must be 000, the bottom three bits of the tag in the second entry must be 001, the bottom three bits of the tag in the third entry must be 010, etc. Since they will always match, these bits are not explicitly present in the cache tags—this saves space in the cache for more useful data. The bits determine which entry in the cache will be checked. The direct-mapped organization makes the lookup faster, but cache contention is higher than in a fully associative cache. If the tag matches and the line is read out, the lower bits of the address that were not used in the tag match determine what portion of the cache line is to be sent out to the requestor.

obvious correspondence to the original address. Whereas in the associative scheme there are *n* tag matches, where *n* is the number of cache lines, in this scheme there is only one tag match because the requested datum can only be found at one location: it is either there or nowhere in the cache.

The benefit of a direct-mapped cache is that it is extremely quick to search, since there can only be one place that any particular datum can be found. However, this introduces the possibility that several different data might need to reside in the cache at the same place, causing what is known as *contention* for the desired data entry. This results in poor performance, as entries in the cache are frequently replaced. The problem is solved by a fully associative cache, which allows any datum to reside in any data entry in the cache. The advantage is that this reduces contention as much as is possible, but the disadvantage is that every single tag must be checked against the data one is looking for. If they can be checked in a short amount of time, for example all in parallel, the design works well. A set associative cache lies in between the two in the design continuum and often reaps the benefits of both designs—fast lookup and lower contention.

2.2 Virtual Memory Primer

Before we get into different virtual/physical organizations of caches, we must first define *virtual addresses*. Most general-purpose systems support the abstraction of *virtual memory*, a mechanism that provides a mapping from a perceived address space (perceived by the running application) to the physical memory system [1, 7]. The addresses that a running process uses are relative to this perceived address space and bear no correspondence to where the data actually reside in main memory. They are thus called *virtual* addresses.

Virtual memory is one of the few interfaces through which the architecture and system software directly interact. It was developed to automate the movement of program code and data between main memory and secondary storage to give the appearance of a single large store. This greatly simplified the job of the programmer, particularly when program code and data exceeded the size of main memory. The basic idea proved readily adaptable to additional requirements including address space protection, the execution of processes only partially resident in memory, and a user-friendly programming paradigm in which a process may assume that it owns all available hardware resources, including memory resources beyond the storage capabilities of the machine. Consequently, virtual memory has become widely used and most modern processors have hardware *memory-management units* (MMUs) to support it [6]. The MMU component that does the actual translation is called the *translation lookaside buffer* (TLB).

Dynamic memory management has recently made the transition from general-purpose systems to embedded systems, in part to facilitate the rapid development of embedded applications. It is playing an increasingly significant role in embedded systems as more designers take advantage of low-overhead embedded operating systems that provide virtual memory (for example, Windows CE or Inferno), and as more designers choose object-oriented software platforms in which run-time garbage collection is pervasive (for example, Sun's Java Virtual Machine). It is interesting to note that National Semiconductor recently abandoned several years' worth of development on their Pentium-class embedded processor, largely due to customer dissatisfaction with the processor's lack of a memory-management unit [12].

Virtual memory, and dynamic memory management in general, is becoming increasingly important to embedded sys-

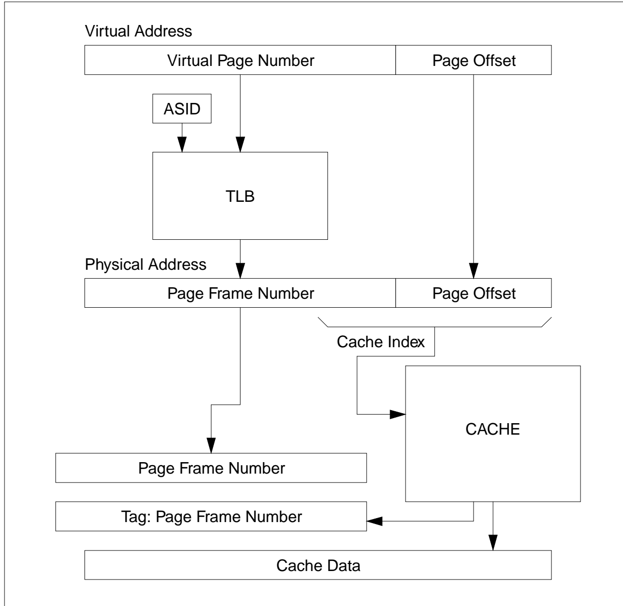

Figure 4. A physically indexed, physically tagged cache organization.

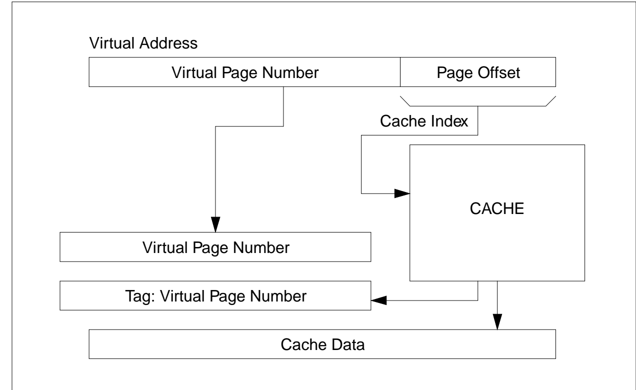

Figure 5. A physically indexed, virtually tagged cache organization.

tems today. The reason that virtual memory is an issue in cache design is because many embedded OSes are supporting it, and the existence of virtual addresses increases the choices for cache organization.

2.3 Physical and Virtual Cache Organizations

There are a number of different cache organizations that a microarchitecture designer can choose from, depending on how the cache is indexed and what kind of information it uses for the tags. The cache is a small database, and the address of the datum, either physical or virtual, is a datum's corresponding database key. Most data and instruction caches are direct-mapped or set-associative. Full associativity is typically reserved for specialized cache structures such as translation lookaside buffers or special-purpose memories. In direct-mapped and set-associative caches, a portion of the key is used to index into the cache to choose a data set—a cache line. If the cache is direct-mapped there is only one cache line at a given index; in set-associative caches there are more than one cache lines. All of the corresponding tag entries (one for a direct-mapped cache, or more than one for a set-associative cache) are read out, and if one of the tags matches the key, the appropriate cache line is read out.

Since the key can be a virtual address or a physical address we have four choices for cache organizations:

Physically indexed, physically tagged. The cache is indexed and tagged by its physical address, therefore the virtual address must be translated before the cache can be accessed. The advantage of the design is that since the cache uses the same namespace as physical memory, it can be entirely controlled by hardware, and the operating system need not concern itself with managing the cache. The disadvantage is that address translation is in the critical path. This becomes a problem as clock speeds increase. Application data sets are increasing with increasing memory sizes, and larger TLBs are needed to map the larger data sets. Larger TLBs are required, and it is both difficult and expensive to make a large TLB fast. The physically indexed, physically tagged organization is illustrated in Figure 4.

Physically indexed, virtually tagged. The cache is indexed by its physical address but the tag contains the virtual address. This is traditionally considered an odd combination, as the tag is read out as a result of indexing the cache; if the physical address is available at the start of the procedure (implying that address translation has been performed), then why not use the physical address as a tag? If a single physical page is being shared at two different virtual addresses, then there will be contention for the cache line, since both arrangements cannot be satisfied at the same time. However, if the cache is the size of a virtual page, or if the size of the *cache column* (the cache size divided by its associativity: the span of the cache index) is no more than a virtual page, the cache is effectively indexed by both the virtual and physical address since the page offset is identical in the virtual and physical address. This organization is pictured in Figure 5. The advantage of the physically indexed, virtually tagged cache organization is that, in addition to the speed advantage of moving address translation off the critical path, like the physically-indexed/physically-tagged organization, the operating system need not perform explicit cache management.

Virtually indexed, physically tagged. The cache is indexed by the virtual address, which is available immediately since it needs no translation, and the cache is tagged by the

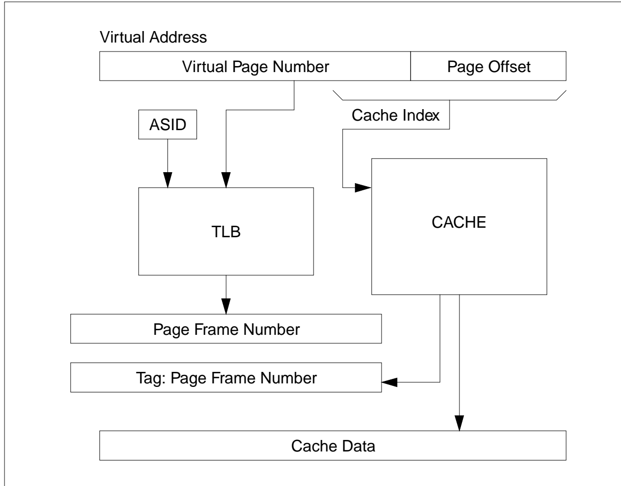

Figure 6. A virtually indexed, physically tagged cache organization.

physical address which does require translation. Since the tag compare happens after indexing the cache, the translation of the virtual address can happen at the same time as the cache index; it can be done in parallel. Though address translation is necessary, it is not in the critical path. The advantages of the scheme are a much faster access time than a physically indexed cache, and a reduced need for management as compared to a virtually indexed, virtually tagged cache—though management is still necessary, as opposed to physically indexed caches. The virtually indexed, physically tagged cache organization is illustrated in Figure 6.

Virtually indexed, virtually tagged. The cache is indexed and tagged by the virtual address. The advantage of this design is that address translation is not needed anywhere in the process. A translation lookaside buffer is not needed, and if one is used, it only needs to be accessed when a requested datum is not in the cache. On such a cache miss, the virtual address must be translated to load the datum from physical memory. A TLB would speed up the translation if the mapping were found in the TLB. Since the TLB is not in the critical path, it could be very large; though this would imply a slow access time, a larger TLB would hold much more mapping information, and its slow access time could be offset by its infrequency of access. The virtually indexed, virtually tagged cache organization is illustrated in Figure 7.

There are, of course, many more components to the choice of cache organization. For instance, the cache can be set associative, which looks like several direct-mapped caches all searched in parallel, and the degree of associativity is a design choice: higher associativity approaches the hit rate of a fully associative design, but slows the processor. The size of the cache line is a design choice: larger line sizes tend to yield better hit rates, but only up to a point [11]. There are also protocols for how write misses are handled [9] and mechanisms such as victim caches that attempt to identify important data items that are thrown out prematurely [8]. For the sake of this paper, these are all beyond the scope of our discussion.

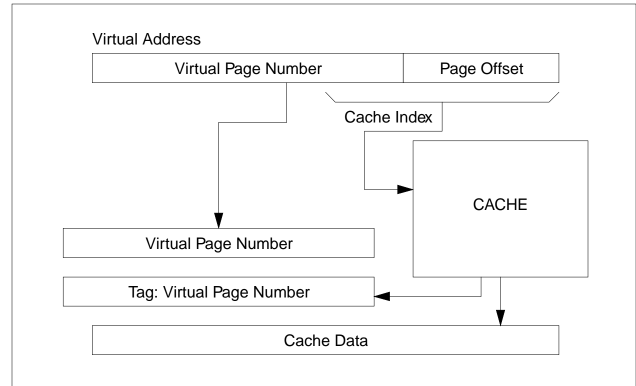

Figure 7. A virtually indexed, virtually tagged cache organization.

2.4 The Advantage of Caching

The benefit of caches is that they hold data close to the processor, and the closer the data, the faster its access. Caches take advantage of locality in two ways:

- When an item is referenced, it is brought into the cache. Therefore, if the item is referenced again in the near future, it is likely still within the cache, and its access will be fast. This exploits *temporal locality*—the tendency of a program to reuse data in the near future.
- When an item is referenced, the surrounding data is also brought into the cache: a cache line or cache block is usually larger than a single datum. Therefore, if the program uses data near the original datum, the data is likely already in the cache. This exploits *spatial locality*—the tendency of a program to reference items that are nearby those it has used in the recent past.

In general, data tends to be found in the cache: items are rarely displaced immediately, and thus a program can easily get good performance in cases like tight loops that re-reference instructions and data.

2.5 The Disadvantage of Caching

The problem with this arrangement is that, in the steady-state, the cache is full of important data, and any reference to an object that is not already in the cache displaces something that has been referenced in the past and is therefore important. Because the execution path in a typical program is based partly on the input data, it is difficult to predict exactly which instruc-

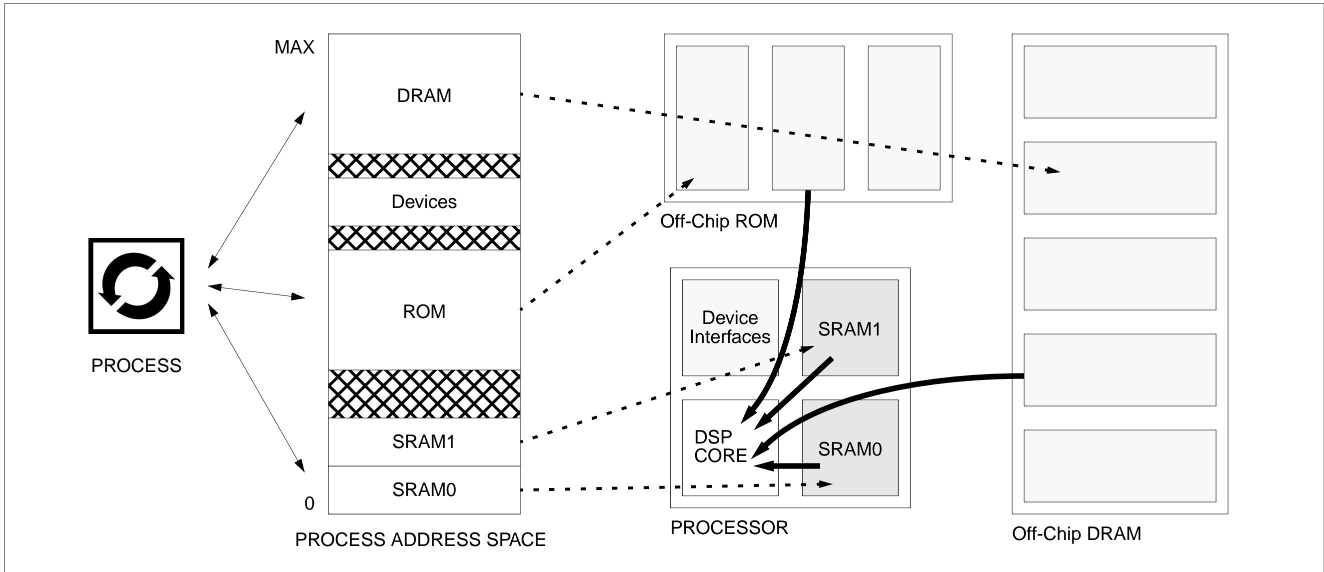

Figure 8. DSP-style SRAM in a distinct namespace separate from main memory.

tions a program will execute in the far future. Similarly, a program's data reference pattern is based in part on the data values; thus it is difficult to predict exactly which data the program will touch in the far future. Because of this inherent lack of predictability, it is virtually impossible to tell how long any given datum will last in the cache before it is displaced by some other datum—the end result is that it is very difficult to predict what the steady-state cache contents will be, and therefore it is very difficult to predict the future execution time of any given instruction or collection of instructions. This lack of precision is inimical to real-time systems, which require absolute confidence in the execution times of their operations—it is no wonder that many of these systems execute without caches.

3 SOFTWARE-MANAGED ORGANIZATIONS

There are two primary cache organizations that lend themselves to real-time processing. The first is the kind used in *digital signal processors (DSPs)* [10]. DSPs typically use on-chip SRAMs that form a separate namespace from main memory. Instructions and data only appear in these memories if software explicit moves them there.

Another organization is the *software-managed virtual cache* that has been discussed in the academic literature for high-performance systems [4, 5] and has recently made the transition to real-time embedded systems [3].

Software-managed cache organizations allow the software to determine, even on a cacheline-by-cacheline basis, whether or not to cache instructions and data; they are especially valuable in real-time systems. For example, initialization code of a real-time process would never be cached, while the periodic body of the code would always be cached. Since initialization code only executes once, the loss in performance by not caching the code is amortized over a long execution time. The periodic loop, however, is cached, and results in significantly increased performance during the entire execution, since an RTOS managing the cache can guarantee that the code remains cached for the lifetime of the process.

3.1 SRAM in a Separate Namespace

Figure 8 illustrates this organization: executing software sees a namespace that spans several distinct storage types. The software is thus completely aware of the storage types available and can make intelligent choices (both statically and dynamically) about where each function or data object should reside for maximum performance. In this particular memory map, there are two on-chip SRAM arrays, found in the low region of the address space. At the top of the address space is the DRAM array, which in this example is located off-chip. In the middle of the memory map are the devices and the ROM array. Suppose, that the memory areas have the following sizes and correspond to the following ranges in the address space:

Address Range	Size	Storage Device
0x0000–0x0FFF	4 KBytes	SRAM-0
0x1000–0x1FFF	4 KBytes	SRAM-1
0x2000–0x3FFF	8 KBytes	invalid
0x4000–0x5FFF	8 KBytes	ROM
0x6000–0x6FFF	4 KBytes	invalid
0x7000–0x77FF	2 KBytes	Device interfaces
0x7800–0x7FFF	2 KBytes	invalid
0x8000–0xFFFF	32 KBytes	RAM

Then if the system designer wants a certain function that is initially held in ROM to be located instead in the very beginning

of the SRAM-1 array (as in the previous example given in which initialization code is never cached but the periodic loop of the main program is always cached), the software could perform the following operation:

```
void function();
char *from = function;  /* in range 4000-5FFF */
char *to = 0x1000;   /* start of SRAM-1 array */
memcpy(to, from, FUNCTION_SIZE);
```

This would copy `FUNCTION_SIZE` bytes from the ROM array to the SRAM-1 array. From that point on, the static value of `function` could never be used to call the function. Future invocations of `function()` would have to use the address 0x1000 instead of wherever `function()` is located in ROM, otherwise those invocations would call the ROM version of the function, not the cached version.

Note that this software-managed cache organization works because DSPs typically do not use virtual memory: in the DSP world, it is not considered a system error to allow applications direct access to the memory system. This is perhaps considered an "unsafe" design because any process could read or write the code or data of any other process in the system. Whereas that is an important issue in general-purpose time-sharing systems, it is a non-issue in most embedded systems. However, it appears that the trend is for embedded systems to look increasingly like desktop systems, in which case address-space protection *will* be a future issue.

One obvious question is *how can this scheme be extended to support address-space protection?* One solution is to simply provide access to the memory arrays via a virtual memory mechanism similar to those used in general-purpose systems: a memory-management unit with a translation lookaside buffer. The arrangement would require something a little different from general-purpose systems, for several reasons: first, the management of the memory system would become more complex because there is more than simply a DRAM array involved; second, the management of the TLB would have to be more deterministic than it is in typical general-purpose systems (which often use random replacement strategies, etc.).

For the memory-management interface, a DSP-based operating system might want to provide several variations on hte function `malloc()`, each of which allocates to the process a virtual region that maps to a separate area of the physical namespace. For example, here is a set of functions that a DSP-based OS could export to its processes to provide such access:

```
void *sram0_malloc( size_t size );
void *sram1_malloc( size_t size );
void *dram_malloc( size_t size );
void *rom_malloc( addr_t start, size_t size );
```

Note that the `rom_malloc()` function does not actually provide heap space in the ROM array; it simply allocates a region within the process address space that is mapped to a region in the ROM array. All of the functions behave this way, but the `rom_malloc()` function is the only that would require the software to specify a region within the storage device, for obvious reasons.

This memory-allocation interface would have the desired effect of allowing a process to make device-specific optimizations, but it would also hide from the process the particulars of how big each region is, where in the address space it is located, etc. It would also protect the virtual address spaces of processes from each other.

For a deterministic TLB mechanism, the operating system software would want to avoid random replacement policies and perhaps have several classes of page table entries: those that are held in the TLB, those that are held in fast memory (SRAM arrays), and those that are held in the DRAM system. Provided that a program can predict its future memory usage with a reasonable degree of accuracy, the program could assign regions of memory to be mapped by page-table entries with statically-known access times.

3.2 Software-Managed Virtual Caches

Another way to give software control over the memory system is to make software responsible for cache-fill and decouple the translation hardware, such as the MMU and TLB, if they are part of the system. In general-purpose systems, hardware handles cache-fill: if a memory access misses the cache, hardware fetches the datum from memory and places it in the cache transparently. Note that if the system uses virtual-address translation hardware (a TLB), the hardware must translate the address before sending it to main memory.

However, one can make software responsible for all cache-fill activity by *upcalls* to the software that happen on cache misses: every cache miss would interrupt the software and vector to a handler that fetches the referenced datum and places it into the cache (transparently to the rest of the software). If the software determines that a particular datum should not be cached, it fetches the datum but does not place it in the cache. To avoid constant interruptions for access to intentionally uncached data, a separate region of the virtual space would refer directly to physical memory—the hardware must interpret addresses in this region as physical, non-cached addresses. It would send these addresses directly to the DRAM array.

This is in fact similar to the preceding example, in that it also allows software to determine, at either compile time or run-time, to cache or not cache portions of the address space. This example differs in that it works with general-purpose style caches that hold copies of data held in the DRAM system; whereas the preceding example the caches are disjoint with the DRAM system.

Figure 9 illustrates this organization. Figure 9(a) depicts a typical real-time system that runs without caches; access is slow to every location in the system's address space. Figure 9(b) shows the effect of adding a hardware-managed cache; in the steady-state, each item in the address space has a statistical

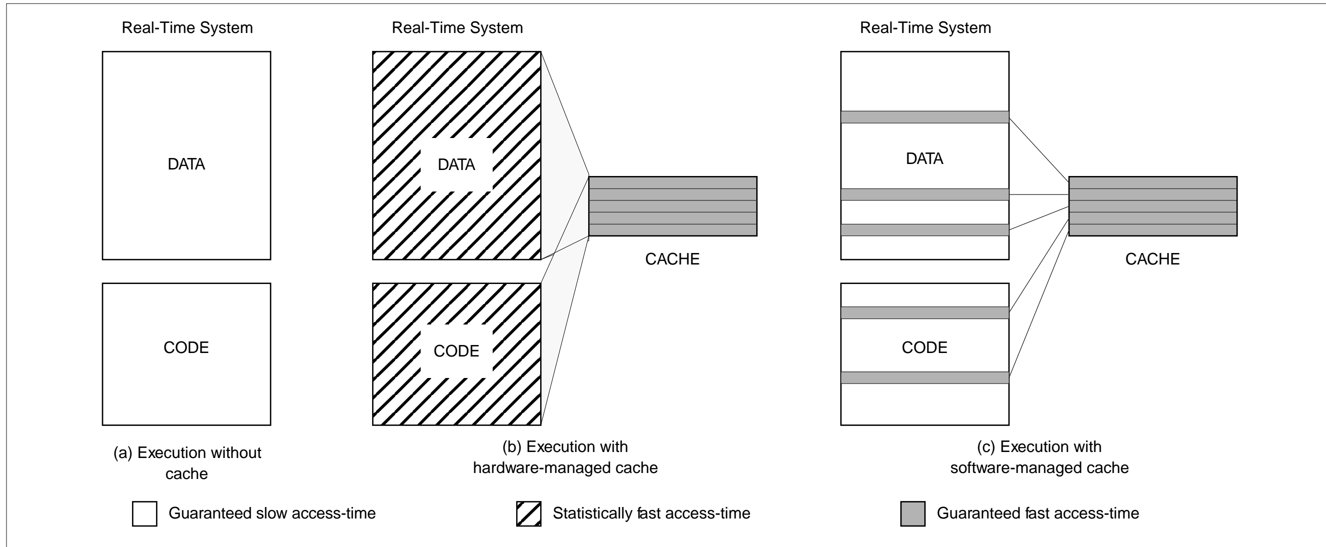

Figure 9. The use of software-managed virtual caches in a real-time system.

likelihood of currently existing in the cache—it may or may not be in the cache at any given point in time. Figure 9(c) shows the effect of adding a software-managed cache; the software determines what can and cannot be cached, therefore the software can ensure (if so desired) that certain portions of the address space will always be cached. As compared to a traditional hardware-managed cache, timing analysis is as simple as in the non-cached case, because access to any specific memory is consistent, either always in cache, or never in cache. Compared to a processor with no cache, selected data accesses and instructions execute 10-100 times faster.

Address translation hardware can still be used in this scheme. It can speed up the translation of addresses that miss the cache, and it can also be used to create a third class of memory access beyond *cached* and *uncached*. The uncached accesses would be divided into two sub-classes: those that are translated to physical addresses by software (which has significant overhead because the operating system must load page-table entries from memory to perform the translation), and those that are translated by the TLB.

4 CONCLUSIONS

This paper describes several software-oriented cache management schemes that allow real-time systems to make use of on-chip SRAM caches. The paper covers both DSP-style caches whose name spaces are disjoint with the DRAM system, and general-purpose microprocessor-style caches that share the same namespace as the DRAM system.

In both cases, address-space protection is provided by a virtual memory mechanism. As in general-purpose systems, implementing a layer of virtual memory and address translation adds an amount of overhead that would be absent if user-level processes executed directly on top of the raw memory system. However, if indeed embedded systems begin to look increasingly like general-purpose systems, with software written by multiple vendors, then perhaps this performance overhead is a small price to pay for process protection and modularity.

REFERENCES

[1] P. J. Denning. “Virtual memory.” *Computing Surveys*, vol. 2, no. 3, pp. 153–189, September 1970.

[2] P. J. Denning. “Working Sets Past and Present.” *IEEE Transactions on Software Engineering*, vol. 6, no. 1, pp. 64–84, January 1980.

[3] B. L. Jacob. “Software-managed caches: Architectural support for real-time embedded systems.” In *CASES98: Workshop on Compiler and Architecture Support for Embedded Systems*, Washington DC, December 1998.

[4] B. L. Jacob and T. N. Mudge. “Software-managed address translation.” In *Proc. Third International Symposium on High Performance Computer Architecture (HPCA-3)*, San Antonio TX, February 1997, pp. 156–167.

[5] B. L. Jacob and T. N. Mudge. “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-8)*, San Jose CA, October 1998, pp. 295–306.

[6] B. L. Jacob and T. N. Mudge. “Virtual memory in contemporary microprocessors.” *IEEE Micro*, vol. 18, no. 4, pp. 60–75, July/August 1998.

[7] B. L. Jacob and T. N. Mudge. “Virtual memory: Issues of implementation.” *IEEE Computer*, vol. 31, no. 6, pp. 33–43, June 1998.

[8] N. P. Jouppi. “Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch

buffers." In *Proc. 17th Annual International Symposium on Computer Architecture (ISCA-17)*, May 1990, pp. 364–373.

[9] N. P. Jouppi. "Cache write policies and performance." In *Proc. 20th Annual International Symposium on Computer Architecture (ISCA-20)*, May 1993, pp. 191–201.

[10] P. Lapsley, J. Bier, A. Shoham, and E. A. Lee. *DSP Processor Fundamentals: Architectures and Features*. Berkeley Design Technology, Inc., Berkeley CA, 1994.

[11] A. J. Smith. "Cache memories." *Computing Surveys*, vol. 14, no. 3, pp. 473–530, September 1982.

[12] J. Turley. "National kills in-house embedded x86 work." *Microprocessor Report*, vol. 12, no. 2, pp. 10, February 1998.