SMART (Strategic Memory Allocation for Real-Time) Cache Design

David B. Kirk

Department of Electrical and Computer Engineering
Carnegie Mellon University, Pittsburgh, Pennsylvania 15213
and IBM Systems Integration Division

Abstract

Caches have been bridging the gap between CPU speeds and main memory speeds since they were first introduced in the IBM 360/85 computer in 1969. Their absence in real-time system designs, however, has been noticeable. In the past, this was often the result of space, power and weight limitations imposed on many embedded real-time systems. However, even though significant progress in solid-state technology has provided small efficient cache structures, few real-time systems choose to implement hierarchical memory designs. The extremely efficient, but unpredictable, performance of cache architectures provides little benefit to real-time systems that must guarantee hard deadlines. This paper discusses why the present approach to cache architecture design results in unpredictable performance improvements in real-time systems with priority-based preemptive scheduling algorithms. The SMART cache design is then presented, and shown to be compatible with the goals of scheduling in a real-time system.

Keywords: *Real-Time Scheduling; Cache Memories; Performance Predictability; Cache Partitioning; Priority-Driven Preemptive Scheduling; Cache Coherence.*

1. Introduction

1.1. Real-Time Systems and Scheduling

Task scheduling in a time-shared system addresses different concerns than those of importance to real-time systems. Efficient use of system resources, and maximized system throughput are the critical performance measures in a time-shared system. Multiple job streams are active simultaneously, and are often serviced with a round-robin ordering. Execution fairness (i.e. starvation avoidance) predominates as a scheduling concern. On the other hand, real-time systems are directed at an environment that imposes tight timing constraints on system behavior. They typically handle large amounts of data for use in computations that have hard deadlines. These deadlines are often the result of periodic events that provide large quantities of data at regular intervals. The computations to be performed on one data sampling must be completed before the next sampling overwrites the data buffer. In real-time systems, the value of a result is a function of both its accuracy and the time at which the result is produced. This *time-value* of a result is expressed as a constraint which if missed can cause system failure. Thus, maximized throughput is subject to ensuring that all externally imposed deadlines are met. Furthermore, task starvation may be unavoidable under conditions of transient overload. Transient overload occurs when there is not enough computation time available to meet all task deadlines, and usually results from stochastic execution times and aperiodics [23].

Real-time systems are often composed of periodic and aperiodic tasks, where periodic tasks have regular arrival times and hard deadlines, and aperiodic tasks have random arrival times with hard or soft deadlines. Hard deadlines imply that the value (usefulness to the mission) of the computation is zero once the deadline is missed. Often the consequence of a missed hard deadline is system failure, and possibly catastrophic loss. Soft deadlines, however, are modelled by a value function which decreases once the deadline is missed, but does not go immediately to zero. Many systems are composed of both hard and soft deadlines.

In the past, the solution to meeting real-time system timing constraints was to provide an overabundance of processing power and schedule tasks using a cyclical executive. An example of this static time-slicing of system resources is shown in Figure 1-1a. Tasks are placed along the time line such that all deadlines are met. They are typically partitioned to fill in the available time slots. Asynchronous (aperiodic) tasks are processed by dedicating a time slice to a synchronous server task which polls for aperiodic service requests. Fine tuning of the time line is often performed during lab debug. This can result in a very *ad hoc* development of the schedule.

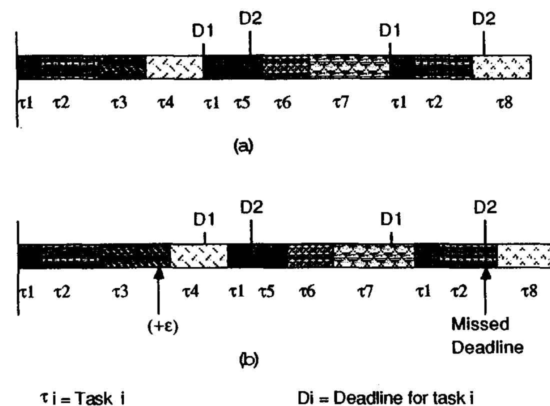

Figure 1-1: TIME LINES - a. original b. after modification

Changes in the task set or in any individual task can be devas-

CH2803-5/89/0000/0229/$01.00 © 1989 IEEE

tating to the completed time line. This phenomenon is depicted in Figure 1-1b where an alteration to task 3 has caused task 2 to miss a deadline. Correcting the time line usually involves shuffling task pieces around, repartitioning individual tasks, and may even result in a complete redesign of the time line sequence. For this reason, maintenance and system upgrade are particularly painful in a cyclical executive system. In addition, under conditions of transient overload, deadlines will be missed with no regard to the semantic importance of the task.

There is currently a trend to replace this *ad hoc* scheduling approach with scientifically-based algorithmic scheduling techniques. These scheduling policies use a static or dynamic priority assignment to guarantee that the system timing constraints are met. Two such algorithms are the *rate-monotonic* (static), and the *deadline driven* (dynamic) scheduling techniques [13]. Since the deadline driven algorithm misses deadlines unpredictably under conditions of transient overload [16], and often results in NP-hard problems when proving the schedulability of a task set [15], the remainder of this paper will focus on a modified rate-monotonic algorithm.

Under the rate monotonic algorithm, the priorities assigned to periodic tasks are directly proportional to their rate of requests. Assuming a task requests service each period, the task with the shortest period will have the highest priority. Liu and Layland [13] proved this algorithm guarantees that n independent periodic tasks can be scheduled to meet all task deadlines if the sum of the task utilizations (defined as the task computation time divided by the task period) for all n tasks is less than $n(2^{1/n}-1)$. This bound converges to ln 2 (= 0.69) for large n. It is, however, pessimistic, and represents the absolute worst case conditions. The average case scheduling bound is 88% [11]. In practice, the bound is often between 90 and 100% because task periods are often harmonic or near harmonic. Finally, through the period transformation method, the scheduling bound can be made arbitrarily close to 100% [18].

While the rate-monotonic algorithm works well with periodic tasks with fixed execution times, some modifications need to be incorporated to handle aperiodic task scheduling and task priorities during overload. Scheduling of aperiodic tasks can be done through methods such as the priority exchange algorithm, and the deferrable server as discussed in [12]. Stochastic execution times can lead to a required utilization greater than the scheduling bound. When this occurs, load shedding must take place to ensure that critical deadlines are met. Strategic load shedding can be incorporated into the static scheduler through techniques such as period transformation as discussed in [16]. Interprocess synchronization can lead to an unbounded number of priority inversions which can significantly reduce attainable utilization with guaranteed deadlines. Implementation of the priority inheritance and priority ceiling protocols [17] provides a bound on this type of priority inversion, as well as ensuring that deadlock can not occur as a result of synchronization.

1.2. Cache Memories - an Overview

Cache memories are small, fast buffers that are used to temporarily hold recently used information, as well as information that might be needed in the near future. These buffers are placed between the CPU and main memory and provide the functionality of main memory at the speed of the CPU. However, since the cache buffer is smaller than main memory, various techniques have been developed to determine what information should be held in cache. While the solutions to this problem are plentiful, they are all driven by two program properties: temporal and spatial locality. Temporal locality refers to the tendency of a program to revisit areas of memory that have recently been accessed. This time related behavior is exemplified by program loops and repeated procedure calls. Spatial locality refers to the tendency of a program to access memory locations close to those locations that have recently been accessed. This space related behavior is exemplified by the execution of regions of sequential code, or the access of sequentially stored data structures. The cache attempts to ensure that information, local both in time and in space to the current information, is readily available [21]. When this attempt is successful, the cache request results in a *cache hit*, and the access is performed at the speed of the cache. When the attempt fails, the cache request results in a *cache miss*, and the access is performed at the speed of main memory - often three or four times slower than cache.

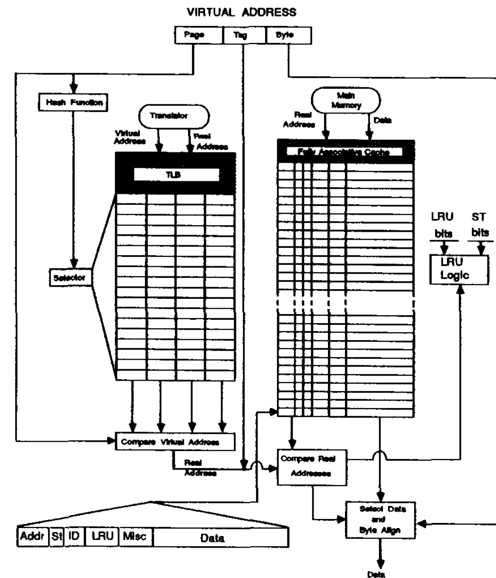

Figure 1-2: Virtual Address Cache Design

Figure 1-2 shows a high level cache design in a computer with virtual addressing. The CPU presents a virtual address to the memory management unit which in turn checks the entries in the translation lookaside buffer (TLB) for the translation to a real address. If there is no valid entry corresponding to that virtual address, it is passed to the slower translation logic which provides the real address to be saved in the TLB for future references. While the virtual page address is being translated by the TLB, the set address is used by the cache to identify a *set* of potential lines that may contain the desired information. The tag field of each line in the set is compared to the real address returned by the TLB or translation logic. If the tags match (a hit), the byte address is used to select the data for the CPU, and the cache status fields are updated to reflect the recent access. If the tags do not match (a miss), a main-memory access is started. The desired line of data is read into the cache, and the requested information is passed on to the CPU.

In a multi-tasking environment, it is very difficult (if not impossible) to predict the cache performance. This unpredictable behavior results from a phenomenon referred to as *cold start*

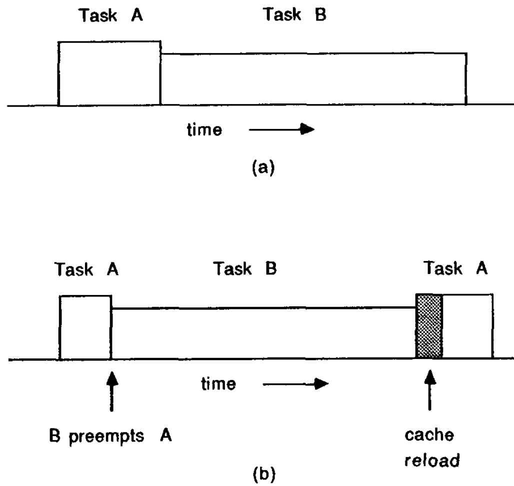

Figure 1-3: a. normal execution b. preemption & cache reload

[25] which occurs whenever there is a significant change in the active working set of the task in execution. Cold starts result in reload-transients while the cache loads the newly activated working set. During the reload-transient, the hit rate is significantly lower than during normal cache operation. The effect of this lower hit rate on execution times is depicted in Figure 1-3. The exact execution time for Task A in Figure 1-3b is dependent on the number of cache lines previously owned by Task A that were displaced by Task B. The shaded region indicates additional execution time caused by cache misses while these lines are reloaded. The relationship of interrupts preemptions and cold starts is discussed further in the next section.

For more information on caches (line sizes, replacement policies, write policies, prefetch policies, and more), an excellent summary is provided in [20] and [4]. A good discussion on placement policies (associativity) is found in [8]. Cache coherence snooping and directory schemes are discussed in [1], and the write-once scheme is discussed in [7] and [26].

2. Real-Time Scheduling and Caches

Cache memory structures have enhanced system performance for generations of computers. The absence of such memory hierarchies in real-time computers can significantly reduce the potential system performance. Furthermore, when caches are present in real-time systems, the resulting performance improvement often leads to the underutilization of system resources. This is exactly the case in the Navy's AEGIS Combat System. The AN/UYK-43 computer, which provides the central computing power for the system, has a 32K-word cache partitioned for instructions and data. However, due to unpredictable cache performance, all module (task) utilizations are calculated as if the cache were turned off (cache bypass option). As a result, the theoretically overutilized CPU is often underutilized at run-time. This section will discuss the reasons this type of cache is unpredictable, and the benefits to task schedulability if the cache performance could be made predictable.

2.1. Scheduling Enhancements

An important evaluation criterion for real-time systems is the schedulable utilization, the highest attainable resource utilization at or below which all hard deadlines can be guaranteed. Schedulable utilization is the sum of all the task utilizations at a point when an increase in any one task utilization would cause the system to miss one of the deadlines. Clearly it would be ideal to design a system such that all deadlines were met, and all resources were utilized 100%. This, however, is not always possible when using static priority-driven preemptive scheduling algorithms. Consider the example shown in Figure 2-1 where the utilization of task 1 is 4/10 = .4 (in the introduction, we defined utilization to be the worst case execution time divided by the period), and the utilization of task 2 is 6/14 = .43, for a total of 83%. Furthermore, assume that a task's deadline is equal to its period. However, an increase in either task's computation requirement would cause task 2 to miss its deadline when scheduled using the rate-monotonic algorithm. Therefore, the schedulable utilization for this task set is 83%, which is equal to the Liu and Layland bound for n = 2, and the computation requirement for neither task can be increased.

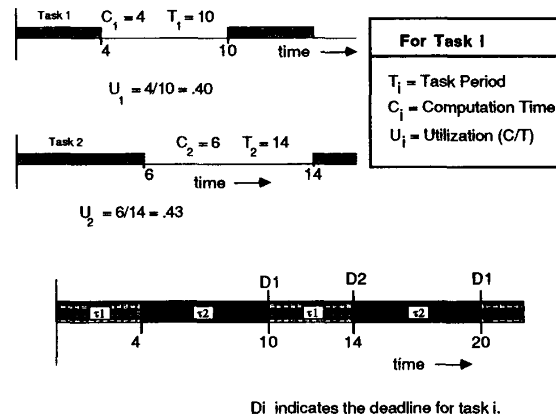

Figure 2-1: Scheduling 2 tasks with utilization of .83

It can be seen that task utilization is a limiting factor in increasing the schedulability (the task load that can be scheduled with guaranteed deadlines) of a system. Therefore, if individual task utilizations could be predictably reduced, additional computational responsibilities could be added to the system. For example, if the above two tasks were run uninterrupted on a system with a cache and executed with worst-case utilizations of 3/10 = .3 and 5/14 = .36 (the reduction in execution time results from decreased memory access times) the remaining processor utilization could no be used to schedule a third task. Thus, predictable cache behavior can indeed increase the task schedulability of a real-time system. Unfortunately, as the next section discusses, when the requirement that these tasks run uninterrupted is removed, the cache predictability is lost.

2.2. Predictability and Preemption

Having established the desirability of predictable cache behavior to achieve better schedulability, we must define the predictability required. For schedulability concerns, a cache is predictable if the performance achieved running the task uninterrupted can be guaranteed in a priority-driven preemptive scheduling environment. Thus, the worst case execution time (WCET) calculated

with the cache enabled will never be exceeded due to unusually low hit rates. This WCET can then be safely used to determine the schedulability of a task set. Unfortunately, conventional caches do not meet this requirement for predictability.

In the environment of priority-driven preemptible schedulers, the contents of a cache are virtually random for any task that is subject to preemption. During the preemption of a low priority task, the footprint ("which is the number of distinct cache lines touched by a program" [24]) of the preempting task overlays the footprint of the preempted task. As Stone points out in [24], even if the two programs could have fit in the cache simultaneously, because the cache lines each program occupies are not strategically distributed to avoid overlap, the probability for conflict is high. As a result of these conflicts, the preempting task displaces useful lines of the preempted task. When the preempted task resumes, it incurs an unusually high miss rate as it reloads its working set in cache. As depicted in Figure 2-2, a low priority task may run to completion without any preemptions, or it may be preempted one or more times by each of the tasks with higher priority. In addition, even the highest priority task can be subject to reload-transients caused by interrupt service routines which utilized the cache.

$\tau_1 : C_1 = 1 \quad T_1 = 5 \quad U_1 = .2 \qquad \tau_2 : C_2 = 1 \quad T_2 = 7 \quad U_2 = .14$

$\tau_3 : C_3 = 1 \quad T_3 = 9 \quad U_3 = .11 \qquad \tau_4 : C_4 = 3 \quad T_4 = 20 \quad U_4 = .15$

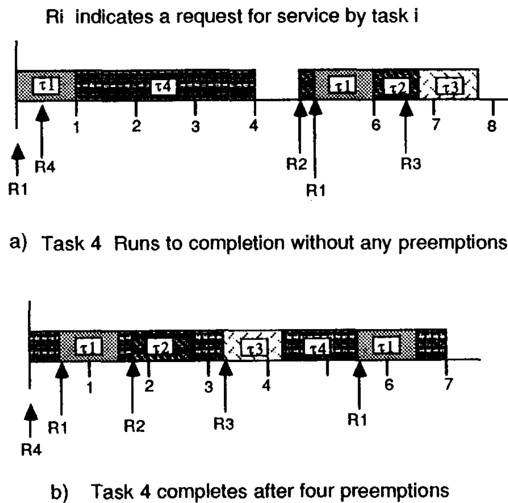

Figure 2-2: Variation in preemption occurrences

Therefore the problem of predictable caches is reduced to hiding the effects of preemption and interrupts for all tasks presently in execution. Hiding the effects of preemption in a cache can be achieved through one of two mechanisms: protection or restoration.

Protecting the cache contents involves implementing a scheme that prevents the preempting task from being able to destroy the information in cache that belongs to the preempted task. If the preempting task is not allowed to overlay information owned by the preempted task, then once the preempted task is allowed to continue, the cache appears undisturbed by the preemption. In reality, the cache contents changed, but not in areas seen by the preempted task. Approaches to cache predictability using protection are achieved through cache partitioning. The analysis of cache partitioning costs, benefits, and techniques are discussed in [20], [2], [19], and [27].

Restoration of cache contents involves allowing the preempting task to overwrite the information in cache owned by the preempted task. However, before the preempted task resumes execution, the cache is reloaded to provide the information resident before preemption occurred. The overhead involved with restoring the cache can lead to a significant reduction in performance, and could easily negate the benefits of a cache.

Caches in real-time systems must implement design strategies which in some way address the effects of preemption on predictable cache performance. During this research, various approaches involving both restoration and protection were examined. These approaches are summarized below, and further discussed in [9]. Each approach led to design goals used in developing the SMART cache design technique.

3. Partitioning

Predictable cache performance requires knowledge about the contents of the cache in an environment that allows preemptions. This section discusses various techniques for meeting this requirement in order to guarantee a certain minimal cache hit count. Three cache partitioning approaches are presented. The pros and cons of each approach are used to establish the design goals of the SMART cache partitioning approach.

3.1. N-Way Partitioning

The N-Way Partitioning (NWP) scheme divides a cache into N separate partitions and provides predictability through protection. These partitions can then be statically or dynamically allocated to various tasks. A task is forced to use only the partition which it owns, and therefore all other cache data is protected. A task which is preempted in the middle of its execution resumes execution with the exact cache partition contents that it had established before being preempted, and therefore experiences no reload-transients due to multitasking. Figure 3-1 shows N tasks statically bound to partitions comprising a cache of size C. The cache partitioning is accomplished by mapping the cache address to a specific area of cache defined by the User ID.

The ELXSI System 6460 used a technique similar to N-way partitioning, with a cache size of 1-MByte divided into separate 512-Kbyte Data and Instruction caches [5]. The cache can be further divided into halves, quarters, eighths, or a combination thereof. Each partition is then dynamically allocated to one real-time task. If the cache requests exceed the available cache, tasks with outstanding requests wait for enough cache to be released. This approach provides cache data protection, but can not be used in a system with hard deadlines due to the potential stall while waiting for a cache partition. Techniques for maintaining cache coherence within the partitions are provided [5].

The N-way partitioning scheme is easy to implement, requiring only a hashing function, and provides complete protection of task specific data across preemptions. However, to prevent unpredictable stalls while tasks wait for cache partitions, only static binding of partitions can be used. Therefore, a maximum of N tasks can be active at any time. Tasks are limited to using only one partition, and shared data must not be cached. The SMART cache design will provide increased cache partition sizes, and areas to cache shared data structures.

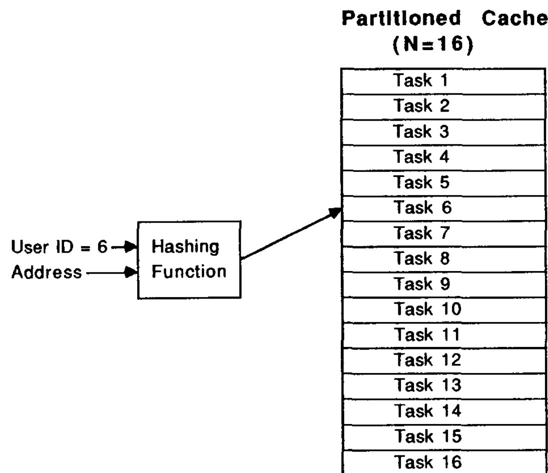

Figure 3-1: N-Way Partitioning

3.2. Static-Locked Partition

The Static-Locked Partitioning (SLP) approach, discussed in [9], divides a fully-associative cache into two partitions (one small and one large), each of which is available to the task now executing. The smaller partition is loaded with frequently accessed subroutines, and is restored in the event that the task is preempted. Routines to be loaded in this partition are determined at compile time. Since the contents of this partition are always known, any access to this region of the cache is predictable. However, this approach requires an intelligent cache controller to perform the reload, and returns a limited number of cache hits due to the fixed contents (determined at compile time) of the smaller cache partition.

3.3. Dynamic-Locked Partitioning

The Dynamic-Locked Partitioning (DLP) approach, discussed in [9], uses an approach very similar to the SLP approach with the cache divided into two partitions. However, unlike the SLP approach, the contents of the smaller partition can change when the task reaches certain checkpoints. The new contents of partition are determined by which checkpoint has been reached. A table of checkpoints and cache contents is built at compile time and referenced by the cache controller as checkpoints are reached. This smaller cache partition contains a minimized working set whose lifetime is defined as the *activation interval*, or the time between the two bounding checkpoints. In the event of a preemption (or interrupt), when the preempted task is swapped back in, the contents of the cache partition are restored to the latest checkpoint.

Although the DLP approach provides more flexibility in the partition contents, it requires an intelligent cache controller to handle the checkpoints, and introduces contention between the cache controller and the CPU for cache resources.

4. SMART Cache Partitioning

The three previous partitioning techniques have introduced important desirable features of predictable cache designs. The SMART (Strategic Memory Allocation for Real-Time) cache partitioning strategy attempts to incorporate the beneficial characteristics of the previous three approaches, while eliminating the undesirable characteristics. It will be shown how the SMART partitioning scheme:

- is implemented with minimal hardware overhead;
- allows each task to utilize a large fraction of the total cache;
- results in no overhead during context switch;
- utilizes compile time information to improve predictability;
- provides overlay protection of frequently accessed information by infrequently accessed information;
- does not compete with the CPU for system or cache resources;
- maps readily onto n-way associative memory;
- allows an unlimited number of tasks to be active;
- allows shared data structures to be cache.

4.1. The Partitioning

Under the SMART cache partitioning scheme, a cache of size C is partitioned into M+1 segments. These segments are then allocated to the N tasks in an active task set, where N may be greater than, equal to, or less than M. The partitioning is performed such that a large fraction of the cache is contained in one partition referred to as the *shared pool*, and the remainder of the cache is divided into M segments as shown in Figure 4-1. The M segments are allocated to those tasks considered performance critical. These tasks, due to the frequency of their occurrence or their semantic importance, require predictable cache hits. Each such task is allocated one or more of the segments. The shared pool is used to service tasks that are considered performance non-critical, usually due to low service frequency. For example, the Navy's Inertial Navigation System (INS) [3] task set contained tasks such as the Update Ship Attitude which occurred every 2.5 ms, as well as the Update Ship Position which occurred every 1.25 sec. In this task set, the position updater would be considered performance non-critical and allocated to the shared pool, while the attitude updater would be granted one or more of the remaining M segments. It is assumed that hits generated by the task running every 1.25 sec are not as beneficial to those hits generated by the task running every 2.5 ms.

Cache segments can also be allocated to a group of tasks if these tasks share the same preemption level, and are not allowed to preempt one another. The AEGIS Tactical Executive System (ATES) implements such a scheduling scheme. ATES supports four preemption levels, but prevents preemptions of tasks within the same level regardless of the scheduling priorities.

Each of the M segments can only be accessed by the task (or preemption group) to which it was allocated, and is consequently protected across preemptions. This logical grouping of one or more cache segments is referred to as the task's cache partition. Each task owning a cache partition is also free to use the shared pool to store infrequently accessed, and shared memory locations. Since the shared pool is accessed by multiple tasks, it is not protected across preemptions, and resulting cache hits are not predictable. Assume for example, a 64K 2-way set-associative cache, and a task set of 8 performance critical tasks and 4 performance non-critical background tasks. For simplicity assume that each of the 8 tasks is granted a 4K cache partition, and 32K is allocated to the shared pool. Each of the critical tasks is capable

of using 36K of the cache, 4K of which is protected through partitioning. As each task runs, it is permitted to selectively store information in the private partition, or the shared pool. This allows a task to determine *a priori* just how the 4K partition should be used at run-time.

4.2. Partitioning and Hit Rates

As discussed in [22], 16K caches can easily attain hit rates above 90 percent, while 4K caches typically achieve hit rates around 90 percent for a range of program traces including database management, scientific applications and batch jobs. In addition, [10] and [22] have shown that caches as small as 256 words typically achieve hit rates of 75%. When the SMART cache partitioning strategy is applied to a 32K cache with a 16K shared pool, then all tasks can utilize at least 16K of cache, and even more for those tasks allocated private segments. Therefore, both total and predictable cache hits rates are expected to be high when the SMART cache partitioning scheme is applied to caches of size greater than or equal to 32K.

Returning to the 64K cache example with eight tasks evenly sharing the private segments, each task would own 4K of predictable cache. As an example of the potential benefit to schedulability, we now consider a 1 ms task executing on a 2 MIP processor with one instruction fetch and .15 data fetches per instruction (a number sometimes used for RISC machines), a 50 ns cache, and a main memory capable of performing a 200 ns read through access when a cache miss occurs. The 1 ms computation requirement assumes no cache hits. With the above requirements, the task requires 2300 memory accesses, or .460 ms memory access time. If the 4K private partition results in a 90 percent hit rate, the 2070 accesses result in cache hits, each of which saves 150 ns, for a total reduction of .31 ms. The original task which required 1 ms now requires only .69 ms, for a 31% improvement in execution time. Since the period did not change, this results in a 31% reduction in the original task utilization. This utilization reduction is predictable, and can be used directly to increase the schedulability of a specific task set. In addition, hits occurring in the shared pool reduce execution time and result in increased service time for aperiodics.

4.3. Controlling the Partitions

SMART cache design allows both real-time caching as well as conventional cache operation. A single hardware flag is used to toggle between the two modes. The implementation of the SMART cache partitioning scheme for real-time is straightforward. This discussion assumes a set-associative cache, but the same conclusions hold for a direct-mapped cache (a comprehensive discussion of the differences appears in [8]). The set address for the cache is combined with the cache ID and a one bit hardware flag as shown in Figure 4-1. The cache ID is loaded in the ID register during the context swap. The ID is used to identify how many segments are owned by the task, and which segments they are. The hardware flag is used to determine if the shared pool or private partition is to be used. All performance non-critical tasks would set this flag to indicate the shared pool at the start of the task and it would not change until a performance critical task gained control. Performance critical tasks on the other hand would be allowed to set and reset this flag during the program execution, thereby utilizing both the shared pool as well as the private segments. Instructions to set and reset this flag would be embedded in the execution code either as additional instructions (vertical insertion), or as additional bits in the present instructions (horizontal insertion).

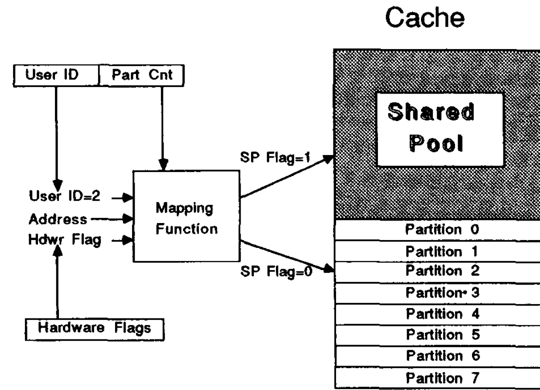

Figure 4-1: Hardware for SMART Partitioning

Software control of the partitions give the programmer (or compiler) the ability to determine exactly what data should be placed in the private partitions. These decisions are made based on information about semantic importance, code frequency, branch decisions, inter-reference locality [14], and any other information useful in determining access patterns and the effects of resulting cache hits on schedulability. As in the DLP approach, compile time information about variables and loop counts can be used in this process, or direct compiler commands can be inserted by the programmer. In addition, separate data and instruction hardware flags are supported to allow instructions to be mapped into the private partition while data is being mapped into the shared pool (see Section 4.5).

4.4. Allocating Segments to Tasks

The SMART cache partitioning scheme divides the cache into finer grain segments than the N-way partitioning scheme, and then allocates one or more segments to each task based on its performance criticality. Once again we will use the 64K cache example in which eight tasks divide 32K for private partitions, and 32K is used for the shared pool. However, now we relax the requirement that the tasks evenly share the available cache. If the cache to be allocated is divided into 1K segments rather than 4K segments, the finer resolution allows the high priority tasks to utilize 8K for example, while lower priority tasks may use 1K or 2K of private cache. Cache segments are allocated in a manner which results in the highest change in the total utilization of the task set. If we assume that initially no segments are allocated, we assign the segments one at a time to task$_i$ if and only if

$$\frac{\delta U_i}{\delta \rho_i} = Max\ Decrease\left(\frac{\delta U_1}{\delta \rho_1}, \frac{\delta U_2}{\delta \rho_2}, \ldots, \frac{\delta U_N}{\delta \rho_N}\right),$$

where ρ_i represents the partition size for task$_i$. The allocation scheme attempts to:

$$Minimize\ U_T = \sum_{i=1}^{N} U_i\ , \quad where \quad U_i = \frac{WCET_i}{T_i}\ ,$$

and $WCET_i$ is the worst case execution time for task$_i$, T_i is the period of task$_i$, and N is the number of tasks. The subscript i will be used to refer to task$_i$ throughout the remainder of this analysis. Then, looking at the components of the execution time we see that

$$WCET_i = t_{mem_i} + t_{ex_i} + t_{misc_i} ,$$

where t_{mem} is time spent accessing memory, t_{ex} is time spent executing instructions, t_{misc} is time spent performing miscellaneous operations such as task synchronization and I/O. For this exercise, we can consider t_{ex} and t_{misc} fixed, and

$$t_{mem_i} = t_{c_i} + t_{m_i}$$

where t_c is the time expended accessing cache memory and t_m is the time expended accessing main memory. Both these parameters are dependent on the number of memory accesses by $task_i$, the hit rate for the cache partition, and the characteristic access time of the memory. Therefore we have

$$t_{c_i} = A_i \mu_i T_c , \quad and \quad t_{m_i} = A_i (1-\mu_i) T_m ,$$

where A is the number of memory references, μ is the hit rate, T_c is the access time for the cache, and T_m is the access time for the main memory.

Now if we examine how the utilization varies with the partition size we have

$$\frac{\delta U_i}{\delta \rho_i} = \frac{\delta\left(\frac{WCET_i}{T_i}\right)}{\delta \rho_i} = \frac{1}{T_i} \cdot \frac{\delta\,(t_{mem_i} + t_{ex_i} + t_{misc_i})}{\delta \rho_i} .$$

But since t_{ex} and t_{misc} are constant with respect to ρ_i, their derivatives are zero. Therefore, if we substitute for t_{mem} we get

$$\frac{\delta U_i}{\delta \rho_i} = \frac{1}{T_i} \cdot \frac{\delta}{\delta \rho_i} (t_{c_i} + t_{m_i}) =$$

$$\frac{1}{T_i} \cdot \left[\frac{\delta}{\delta \rho_i} \{A_i \cdot \mu_i \cdot T_c\} + \frac{\delta}{\delta \rho_i} \{A_i \cdot (1-\mu_i) \cdot T_m\} \right] ,$$

and since A_i, T_c and T_m are constant with respect to ρ_i, this reduces to

$$\frac{\delta U_i}{\delta \rho_i} = \frac{A_i \cdot T_c}{T_i} \cdot \frac{\delta \mu_i}{\delta \rho_i} - \frac{A_i \cdot T_m}{T_i} \cdot \frac{\delta \mu_i}{\delta \rho_i} =$$

$$\frac{A_i}{T_i} \cdot \frac{\delta \mu_i}{\delta \rho_i} \cdot (T_c - T_m) .$$

These findings are consistent with expectations. Since cache is faster than main memory, T_c will be less than T_m, and the quantity $(T_c - T_m)$ will be negative. Both A_i and T_i are positive, and the hit rate increases monotonically with the partition size. Therefore, the derivative of the utilization with respect to the partition size is negative, and the utilization will decrease with increasing partition size. Furthermore, the rate of decrease is proportional to the number of memory references, the frequency of occurrence of the task $(1/T_i)$, the difference between the access times of the two memories, and the rate with which the hit rate changes with the partition size. The hit rate is the only parameter that varies with the partition size.

While cache designs with small segments provide more resolution during the allocation phase, there is overhead associated with finer grain segments. Finer grain segments require larger tags to be stored in the cache directory. When the segments are small, fewer of the original address bits are being used to address the cache. As a result, additional tag bits must be stored to provide the comparison needed to determine a cache hit. Since there is a cost associated with smaller cache segments, the cost of finer granularity should be weighed against the benefits.

4.5. Shared Data Structures

As with any partitioning scheme or multiple cache system, cache coherence is a concern. Cache coherence deals with maintaining the contents of the cache in such a way that valid lines always reflects the latest information [26]. Different techniques, such as directory schemes and snoopy caches are used to assure cache coherence in multiple cache systems.

The SMART cache partitioning scheme must assure internal cache coherence between the multiple partitions when data structures are shared. If such data structures are allowed to be contained in the private partitions, problems with predictability arise. If the structures are moved to different partitions as they are needed, then invalidation causes unpredicted cache misses. On the other hand, if each data structure is allocated to the private partition of some *parent* process, problems arise when another process accesses that structure and experiences a cache miss. To bring the data structure into the cache requires replacing a line in the parent's cache partition, leading to unexpected misses by the parent. An alternative is for shared data structures to be kept in the shared pool. This eliminates the possibility for multiple copies of the same data, and removes the risk of stale data when two tasks share a common data structure. For reasons discussed earlier, data accesses to these structures do not have predictable cache hits.

Maintaining cache coherence in multi-cache systems (i.e. multiprocessor systems) can also be solved by mapping shared data structures to the common pool. This approach is compatible with the proposed IEEE Futurebus coherence protocol [6]. Depending on the memory and I/O configuration, the Futurebus coherence protocol can be maintained across the entire cache, or just the shared pool. Various memory hierarchies and the associated SMART cache design approach are currently under investigation.

Unpredictable cache behavior resulting from shared data structures in both uniprocessors and multiprocessors can be avoided if we assume that such data structures are protected by read and write locks. Once write access is granted to a process, no other process can read or write the structure until it has been released. This process could then access the data structure through its private partition without concern for cache coherence while the write lock is active. Before releasing the lock, these cache lines would either be flushed and/or invalidated depending on the write policy (copy-back or write-through). This prevents aliasing problems in the SMART cache caused by mapping identical addresses to different partitions determined by the requesting task. If the flush were not performed, and the most recent copy was maintained in the private partition for process P1, any reference by process P2 would result in old data since P2 can not "see" the private partition for P1. If the invalidate were not performed, it is possible that task P1 might read an old version of the data from its own private partition following modification by P2 elsewhere.

4.6. Partitioning for Aperiodics and Interrupts

Aperiodics are often serviced using some type of pseudo periodic service routine such as the deferrable server and priority exchange algorithms [12]. This periodic server, or any other aperiodic task, can be allocated a partition using the same type of decision scheme used for periodic tasks. An average arrival time could be used for the task period. All aperiodics could be assigned to the shared pool, assigned separate partitions, or groups of aperiodics could share a single partition. A similar approach can be used to handle interrupts. A good approach for interrupts might be to allocate a private partition for each interrupt level.

5. Conclusions and Future Work

Present cache architectures do not support priorities and predictability. Consequently it is impossible to use their resulting performance enhancement to ease the scheduling burden in real-time systems which guarantee hard deadlines. The SMART cache partitioning scheme extends scheduling concerns into the system hardware architecture resulting in a new approach to cache design. This approach allows a single cache design to support both time-share and batch applications with a conventional cache architecture, as well as provide a predictable cache management scheme for use in priority-driven preemptive real-time scheduling environments. Switching between the two modes is accomplished with one instruction to set/reset a hardware indicator specifying the mode.

While operating in the real-time mode, the cache is divided into private partitions allocated to individual tasks or preemption priority levels. Unlike other schemes, partitions are not granted on a first come first serve basis, but rather based on a static analysis of the task priorities, semantic content, and contribution to schedulability. Furthermore, static assignment of partitions assures that no task is ever forced to wait for a cache partition to become available. In addition, a large fraction of cache is dedicated as a common pool used primarily by low priority tasks. The introduction of the common pool allows the number of tasks to exceed the number of partitions. The common pool is also utilized by critical tasks via embedded cache instructions which provide software control over the placement of cache contents. Thus, the SMART cache partitioning scheme continues the current trend of allowing compilers to make decisions related to the use of system hardware resources. Use of compile time information to provide more efficient utilization of system resources is very feasible in real-time systems which have minimal compilation requirements due to a fixed task set.

In summary, the results of this research not only provide a scheme for utilizing the performance enhancement provided by hierarchical memory designs, but also for fine tuning these enhancements to provide increased benefit to the desired scheduling goal. In the past, hierarchical memory design costs made their use impractical for many real-time applications where the performance benefits only served to reduce risk of transient overload. The SMART cache partitioning approach provides a technique for using hierarchical memory designs to increase functionality and/or decrease design costs of the processor.

Future work in SMART caching will focus on investigating models of memory access patterns which can be used to represent the tasks competing for cache segments. These models will be used to predict the expected benefit of allocating individual segments to the various tasks. Algorithms are presently being developed which allocate the cache segments based on weighted scores achieved by each task.

References

1. A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz. An Evaluation of Directory Schemes for Cache Coherence. The 15th Annual International Symposium on Computer Architecture Conference Proceedings, IEEE, Hoholulu, Hawaii, May, 1988, pp. 280-289.

2. J. Bell, D. Casasent, C. G. Bell. "An investigation of alternative cache organizations". *IEEE Transactions on Computers TC-23*, 4 (April 1974), 346-351.

3. M. W. Borger. VAXELN Experimentation: Programming a Real-time Periodic Task Dispatcher using VAXELN Ada 1.1. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, September, 1987.

4. D. W. Clark, B. W. Lampson, and K. A. Pier. "The memory system of a high performance personal computer". *IEEE Transactions on Computers TC-30*, 10 (October 1981), 715-733.

5. *ELXSI System 6400 - The System Foundation Guide*. ELXSI, 2334 Lundy Place, San Jose, CA, 95131, 1988. Order No. D9512.

6. *Futurebus P896.2 Specification, Draft 1.0*. IEEE, 345 East 47th St., New York, NY 10017, 1988. Prepared by the P896.2 Working Group of the Microprocessor Standards Committee.

7. James R. Goodman. Using Cache Memory to Reduce Processor-Memory Traffic. The 10th Annual International Symposium on Computer Architecture Conference Proceedings, IEEE, Hoholulu, Hawaii, June, 1983, pp. 124-131.

8. Mark Hill. "A Case for Direct Mapped Caches". *IEEE Computer 21*, 12 (December 1988), 25-40.

9. David B. Kirk. SMART (Strategic Memory Allocation for Real-Time) Cache Design. PhD Thesis Proposal. Carnegie-Mellon University.

10. David B. Kirk. Process Dependent Static Cache Partitioning for Real-Time Systems. Proceedings of the Real-Time Systems Symposium, IEEE, Huntsville, AL, December, 1988, pp. 181-190.

11. J. P. Lehoczky, L. Sha, and Y. Ding. The Rate Monotonic Scheduling Algorithm --- Exact Characterization and Average Case Behavior. Department of Statistics, Carnegie Mellon University, 1987.

12. John P. Lehoczky, Lui Sha, Jay K. Strosnider. Enhanced Aperiodic Scheduling In Hard Real-Time Environments. Proceedings of the Real-Time Systems Symposium, IEEE, San Jose, CA, December, 1987, pp. 261-270.

13. C. L. Liu and James W. Layland. "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment". *Journal of the Association for Computing Machinery 20*, 1 (January 1973), 46-61.

14. Carl McCrosky. An Analytical Model For Cache Memories. Deptartment of Computational Science, University of Saskatchewan, Saskatchewan, Canada, November, 1986.

15. A. K. Mok. *Fundamental Design Problems of Distributed Systems For The Hard Real Time Environment*. Ph.D. Th., Massachusetts Institute of Technology, 1983.

16. Lui Sha, John P. Lehoczky, and Ragunathan Rajkumar. Solutions for Some Practical Problems in Prioritized Preemptive Scheduling. Proceedings of the Real-Time Systems Symposium, IEEE, New Orleans, Louisiana, December, 1986, pp. 181-191.

17. Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, 1987.

18. Lui Sha, John Goodenough. Real-Time Scheduling Theory and Ada. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, to be published, 1988.

19. G. S. Shedler, D. R. Slutz. "Derivation of miss ratios for merged access streams". *IBM Journal of Research and Development 20*, 5 (Sept 1976), 505-517.

20. Alan J. Smith. "Cache Memories". *ACM Computing Surveys 14*, 3 (September 1982), 473-530.

21. Alan J. Smith. "Cache memory design: an evolving art". *IEEE Spectrum 24*, 12 (December 1987), 40-44.

22. Kimming So, Rudolph N. Rechtschaffen. "Cache Operations by MRU Change". *IEEE Transactions on Computers 37*, 6 (June 1988), 700-709.

23. Brinkley Sprunt, David B. Kirk, and Lui Sha. Priority-Driven, Preemptive I/O Controllers for Real-Time Systems. Proceedings of the International Symposium on Computer Architecture, IEEE, Honolulu, Hawaii, 1988, pp. 152-159.

24. Harold S. Stone, Dominique F. Thiebaut. Footprints in the Cache. Proceedings of the ACM Sigmetrics Conf. on Meas. Mod. of Comp. Sys., ACM, May, 1986, pp. 4-8.

25. Harold S. Stone. *High-Performance Computer Architecture*. Addison-Wesley, Reading, Massachusetts, 1987.

26. Paul Sweazy and Alan Smith. A Class of Compatible Cache Consistency Protocols and their Support by the IEEE Futurebus. The 13th International Symposium on Computer Architecture Conference Proceedings, IEEE, Tokyo, Japan, June, 1986, pp. 414-423.

27. D. F. Thiebaut, H. S. Stone, and J. L. Wolf. A Theory of Cache Behavior. IBM Research Division, Yorktown, NY, 1987.