

Memory System Architecture for Real-Time Multitasking Systems

by

Scott Rixner

Submitted to the Department of Electrical Engineering and
Computer Science in Partial Fulfillment of the Requirements for
the Degrees of

Bachelor of Science in Computer Science and Engineering and Master
of Engineering in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

May 1995

© Scott Rixner, 1995. All rights reserved.

The author hereby grants to MIT permission to reproduce and to
distribute publicly paper and electronic copies of this thesis docu-
ment in whole or in part, and to grant others the right to do so.

Author

Department of Electrical Engineering and Computer Science

May 26, 1995

Certified by

Jonathan Allen

Professor of Electrical Engineering and Computer Science

Director, Research Laboratory of Electronics

Thesis Supervisor

Certified by

C. Ross Ogilvie

Advisory Engineer, International Business Machines Corporation

Thesis Supervisor (Cooperating Company)

Accepted by

Frederic R. Morgenthaler

Chairman, Department Committee on Graduate Theses

Memory System Architecture for Real-Time Multitasking Systems

by

Scott Rixner

Submitted to the Department of Electrical Engineering and Computer Science on May 26, 1995, in partial fulfillment of the requirements for the Degrees of Bachelor of Science in Computer Science and Engineering and Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

Traditionally, caches have been used to reduce the average case memory latency in computer systems. However, real-time systems using preemptive scheduling algorithms cannot make use of this reduced latency, as it is not deterministic. A real-time system only benefits from an improvement of the worst case performance. Caches do little to improve worst case performance because preemptions allow cached data to be overwritten at unpredictable times, leading to nondeterministic behavior. Real-time systems using execution quantization can deterministically make use of a significant portion of the processor's resources despite the use of slow memory. Execution quantization allows deterministic caching by guaranteeing that tasks can execute for fixed length blocks of time once they start running, thereby controlling preemptions, and allowing cache restoration to occur predictably and efficiently.

Thesis Supervisor: Jonathan Allen

Title: Professor of Electrical Engineering and Computer Science

Table of Contents

1	Introduction.....	11
1.1	Real-Time Computer Systems	11
1.2	Research Goals	12
1.3	Organization.....	13
2	Real-Time Systems	17
2.1	Designing for Determinism	17
2.2	The Frame Manager.....	22
2.3	Caching	27
3	Designing a Real-Time Cache	33
3.1	Characteristics of the Target System	33
3.2	Design Parameters	35
4	Memory Bandwidth	41
4.1	Cache Performance	41
4.2	Decreasing Memory Latency.....	42
4.3	Reducing the Miss Rate	44
5	A Single Frame Manager System	47
5.1	The Single Frame Manager.....	47
5.2	The Loader and Unloader	49
5.3	Using DRAM in Page Mode.....	50
5.4	Task Timing.....	51
6	Execution Quantization.....	53
6.1	Preemptions	53
6.2	Quantization.....	54
6.3	Loading and Unloading the Cache.....	60
6.4	The Operating System	66
6.5	Hardware Implementation of Quantization	67
6.6	DRAM Refresh	69
6.7	Memory Bandwidth	70
6.8	Shared Memory.....	74
7	Expanding Execution Quantization	83
7.1	Task Timing.....	83
7.2	On Chip DRAM Example.....	90
7.3	Real-Time Cache Performance	95
8	Conclusions.....	115
8.1	Results.....	115
8.2	Future Work.....	116
	References	119

List of Figures

Figure 2.1: A Periodic Execution Frame	18
Figure 2.2: A Preemption by a Task with a Faster Frame Rate.....	24
Figure 2.3: Preemptions in Dynamic Deadline Scheduling.....	25
Figure 2.4: Conceptual LIFO Stack for the Schedule in Figure 2.3	26
Figure 3.1: A Code Fragment and its Associated Graph Representation.	39
Figure 5.1: Overview of a Real-Time Caching System.....	49
Figure 5.2: Task Execution with One Frame Manager.....	50
Figure 6.1: Quantized Execution Using Dynamic Deadline Scheduling.....	55
Figure 6.2: An Arbitrary Frame in the Schedule	58
Figure 6.3: Execution Quantization with Loading.....	62
Figure 6.4: Execution Quantization with Speculative Loading.....	62
Figure 6.5: Quantization Hardware	67
Figure 6.6: A Real-Time Caching System.....	68
Figure 6.7: A Data Read in a Cache with Virtual Partitions.....	78
Figure 7.1: Distribution of Cache Unload/Load Time in a Quantization Block.....	86
Figure 7.2: The Affect of Load Placement on Cache Line Contents.....	87
Figure 7.3: On Chip DRAM System Configuration	91
Figure 7.4: Utilized MIPS in Simulation of MIDI Tasks on a 50 MIPS DSP	108
Figure 7.5: Normalized Results of MIDI Synthesizer Simulation.....	109
Figure 7.6: Utilized MIPS in Simulation of Modem Tasks on a 50 MIPS DSP	110
Figure 7.7: Normalized Results of 14,400 Baud Modem Simulation	111

List of Tables

Table 6.1: Quantization Block Size for Various Cache Configurations.....	72
Table 6.2: Cache Actions For DSP Half Cycles.....	81
Table 7.1: Transfer Time for a 1 kword Cache.....	93
Table 7.2: Quantization Block Size for Caches with 32 Words per Cache Line.....	94
Table 7.3: Partial Output from Instruction Cache Simulation of a MIDI Task.....	100
Table 7.4: Comparison of Several 1 kword 4-Way Set Associative Data Caches	102
Table 7.5: MIDI Synthesizer Tasks	106
Table 7.6: 14,400 Baud Modem Tasks	107

Chapter 1

Introduction

1.1 Real-Time Computer Systems

Traditionally, most real-time computer systems have been used for process control. In order for a process control system to function correctly, it must be able to react to external events before a given deadline, as well as perform other specific tasks periodically. These control systems must be predictable, in order to guarantee that all of the necessary functions will be performed. In many process control systems, a system failure can be disastrous, causing the loss of equipment, or maybe even lives. As safety is a major concern, the cost of a process control system is typically less important than guaranteeing that the system will function properly and tasks will be performed predictably. If the control system does not deliver enough performance, then the expense of obtaining a bigger and faster system can easily be tolerated.

The emergence of multimedia computer systems has led to the design of real-time digital signal processor (DSP) systems that are much more versatile than real-time process control systems. These systems are used to perform a wide variety of functions, which will change as the user requires different multimedia functionality. Also, the cost of a multimedia computer system is important, as competitive fixed function solutions exist that do not use real-time DSPs. There is also certainly no expensive equipment or lives in jeopardy if a multimedia system fails, so much more importance is placed on performance than fault tolerance.

Process control systems generally run a fixed set of tasks, as the entire system is designed specifically to perform one function, whereas in a multimedia system, it is not always clear exactly what tasks will be run. The DSP may be used to perform many differ-

ent functions, and perhaps even some that were not anticipated when the system was originally designed. This difference further separates process control systems from multimedia DSP systems:

In a typical real-time system, you have awareness of the tasks the system must perform, how often the tasks must run, and the required completion times of the tasks. When designing the system, you use a method that meets the needs of the system. In many cases, a simple round-robin scheduler is enough. At other times, you select multiple priority levels and use a preemptive scheduler.

In the Windows/DSP [multimedia] case, on the other hand, you don't have this awareness. You can't predict which tasks the DSP will run and what the required priorities will be. For example, the DSP may be providing just a single function, such as a modem, or several simultaneous functions, such as a modem, compression, fax, and text to speech. You cannot predetermine the mix. [19]

A multimedia real-time system is much more versatile and cost constrained than a process control system. Users demand high performance, and are not willing to pay a premium for safety and customized software and hardware development as system requirements change.

1.2 Research Goals

Most real-time system memory architectures do not include caching. Either memory that is as fast as the processor is used, or the cost of using slower memory to the system is determined and accounted for, so as not to sacrifice the system's predictability. However, there are some exceptions, but no caching system exists that allows for general purpose real-time computing without using software control, and placing extreme restrictions on the system.

The introduction of a cache into a real-time system goes against the requirement that real-time systems must be predictable, as a cache, by its very nature, is unpredictable. Since cost is not an issue, process control systems do not need to make use of a cache to increase system performance while keeping costs low, but the advantages to a multimedia system in terms of cost and performance are enormous.

This is not to say that a process control system would not benefit from cheaper, slower memory in terms of cost, and a cache in terms of performance. Rather, the domain of multimedia computing is the driving force for this research, as cost and performance are much more important in that realm. Consumers are not as willing to pay large premiums for better multimedia performance as industry is for the required process control performance.

The primary goal of this research was to design a memory system, using memory that is slower than the processor's cycle time, for use in real-time systems that use priority driven preemptive scheduling. In order to be useful, the memory system must perform better than the trivial solution of just accounting for the slow memory's overhead in real-time resource allocation. Obviously, using slower memory will incur some sort of penalty, and this research is directed at minimizing that penalty. Furthermore, the details of the memory system should not alter the way software is run. The same software should be able to run on a system that uses fast memory that never stalls the processor, as well as on a system with slower memory and a cache. Automated tools may analyze the software to determine its behavior on either system, however, in order to ensure that the software will always be predictable.

1.3 Organization

Chapter 2 is an overview of real-time systems. The primary constraints of real-time systems are discussed, as well as some of the different methods that can be used to schedule

real-time programs in order to guarantee predictability. In addition, the difficulties of real-time caching are presented, along with a brief description of one existing real-time caching system design. This chapter presents the background that is needed to understand the requirements of a real-time caching system, and why it is difficult to make such a system effective.

Chapter 3 is a description of real-time systems as they affect the memory system. The characteristics of the particular system that was used as a model for the research is described. The major parameters that are allowed in the design of a real-time memory system are then explained, and their impact on the memory system is given. These parameters were used throughout the research to guide the design of a useful memory system.

Chapter 4 explains the importance of high memory bandwidth, and presents several methods of increasing memory bandwidth. The available bandwidth between the memory and the cache significantly affects the overall system performance, and sets a bound on the available real-time resources. In all caching systems, data must be transferred between the main memory and the cache, and the faster this can be done, the better the system's performance will be.

Chapter 5 presents a memory system design for a system in which there are no preemptions. This is quite similar to another existing real-time caching method, and is mainly used here as a precursor to execution quantization, which is presented in the succeeding chapter. Even though this kind of a system is of limited use, it is important to understand how it works, as some of the ideas used in the design are quite useful if they are expanded properly.

Chapter 6 presents execution quantization. Execution quantization is an extension to existing scheduling algorithms and a cache design that allows priority driven preemptive scheduling in a real-time caching environment. The fundamental idea of execution quanti-

zation is that cache restoration can successfully negate the adverse effects that context switches have on cache contents, if it can be done in a controlled manner. By allocating execution time to tasks in fixed sized blocks, preemptions can be controlled, and cache restoration can be done efficiently.

Chapter 7 extends the idea of execution quantization to ensure predictability in almost all real-time systems. By not only allowing cache restoration to occur predictably and efficiently, but also allowing cache contents to be dynamic, with a predictable method of calculating the worst case execution times of tasks, execution quantization becomes a predictable and versatile method of real-time cache management. Several examples are given to show the usefulness of execution quantization, and how it can be applied to a real system.

Chapter 8 presents the conclusions of the research and its applicability to a wide range of real-time systems. Several directions for future research are also suggested to explore real-time caching further.

Chapter 2

Real-Time Systems

2.1 Designing for Determinism

Real-time systems are unique in that they require deterministic execution. This precludes most real-time system architectures from taking advantage of nondeterministic components such as a cache. Without a cache, the use of slow memory, like dynamic random access memory (DRAM), significantly decreases the amount of processing power that can be devoted to real-time tasks. The use of static random access memory (SRAM) in the memory system allows increased processor utilization, but is clearly undesirable for cost reasons. Despite this, the IBM Mwave[†] DSPs do in fact use SRAM.

Halang and Stoyenko point out that,

The thinking in probabilistic or statistical terms, which is common in computer science with respect to questions of performance evaluation, is as inappropriate in the real time domain as is the notion of fairness for the handling of competing requests or the minimisation of average reaction times as an optimality criterion of system design. Instead, worst cases, deadlines, maximum run-times, and maximum delays need to be considered. For the realisation of predictable and dependable real time systems, reasoning in static terms and the acceptance of physical constraints is a must - all dynamic and “virtual” features are *considered harmful*. [7]

They further advocate that features such as “... caching, paging and swapping must either be disallowed or restricted.” [7] These arguments do not necessarily rule out the use of DRAM and other high latency memories entirely. However, in their view, attempts at reducing the memory latency, for the purposes of determining worst case execution times,

[†] Mwave is a trademark of International Business Machines Corporation.

are potentially harmful, and do not allow a systematic improvement in performance. This leads to decreased processor utilization for real-time tasks that is unacceptable in the Mwave system, which employs a pipelined RISC architecture that suffers heavily from slow memory access times. However, it is important to remember that in real-time systems, achieving deterministic behavior is far more important than the speed of execution. [6]

2.1.1 Execution Frames

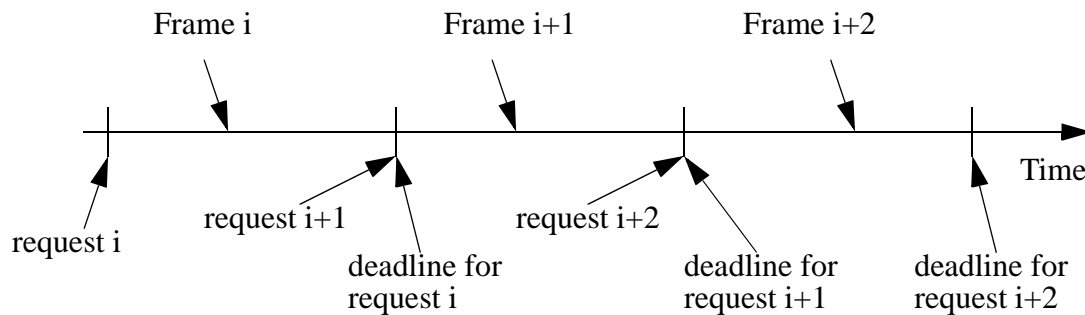


Figure 2.1: A Periodic Execution Frame

In a real-time system, each real-time task executes in an execution frame, which is a fixed length period of time. A frame starts when there is a request for service by a task and ends at the deadline by which execution must be completed. In general, there can be both periodic and aperiodic execution frames in a real-time system. A periodic execution frame is one which has a periodic request time, and typically the deadline for each request is the time at which the next request occurs, as shown in Figure 2.1. An aperiodic execution frame is one in which the requests for service come from an external aperiodic source and the deadline is defined to be a set amount of time after the request, regardless of when the request is made. The IBM Mwave system only allows tasks that run in periodic execution frames, and those frames are defined as a multiple of some periodic interrupt source. For instance, if a frame is given as 4 interrupts from an 8 kHz interrupt source, then there are 2000 frames per second, each lasting half a millisecond. A task must execute once in each

frame. This means that a task running in this particular frame has a hard deadline (one that cannot be missed) every half a millisecond. The frame rate is defined as the frequency of the execution frame, which in the above example is 2 kHz.

Each task must further provide its worst case instruction count. This is the maximum possible number of instructions that the task could ever execute in any one frame. Since the Mwave DSP has a pipeline that results in a throughput of one instruction per cycle, the worst case instruction count uniquely determines the worst case execution time of a task. A metric that is used to gauge how much of the processor's computation time must be devoted to a task is the number of millions of instructions per second (MIPS) needed for the task. Using the execution frame's frequency and the worst case execution time, the necessary MIPS for a task can easily be computed. [17]

2.1.2 Scheduling Real-Time Tasks

Scheduling real-time tasks is an extremely important part of any real-time system. If tasks are scheduled improperly, then tasks will miss deadlines, which can be catastrophic. If any task exceeds its worst case instruction count in the Mwave system, which could cause tasks to miss deadlines, then the whole system stops and signals an error. This prevents transient overloads and allows detection of the offending task, thereby discouraging programmers from falsely declaring worst case execution times. The two predominant scheduling algorithms for real-time systems that guarantee that all tasks will meet their deadlines are dynamic deadline scheduling (DDS) and rate monotonic scheduling (RMS). Both scheduling algorithms require that the worst case execution time of all tasks are known before they are scheduled. Otherwise, deterministic scheduling cannot be achieved.

As suggested by the name, DDS is a dynamic scheduling algorithm, meaning that task priorities are assigned dynamically. It can be shown that if any given task set can be suc-

cessfully scheduled on a processor, then DDS can also schedule that task set successfully. In this context, success denotes that all tasks will always meet their deadlines. DDS uses the proximity of a task's deadline as that task's priority, with higher priority given to tasks with closer deadlines. At any of the periodic interrupts, the scheduler schedules the task with the highest priority. It can be shown that theoretically DDS allows one hundred percent of the processor to be utilized. In practice, the actual processor utilization must be less than one hundred percent, because the scheduler uses some of the processor's time, and task switching takes some time as well. [10] [16] [20]

RMS is a static scheduling algorithm. RMS is an optimal scheduling algorithm in the sense that it can be shown that if any given task set can be successfully scheduled with a static algorithm, then that task set can also be successfully scheduled by RMS. RMS uses the frequency of execution of a task as that task's priority, with higher priority given to tasks with higher frequency frame rates. To construct the schedule, the highest priority task is scheduled first, and then the next highest priority task and so on until all tasks have been scheduled. Theoretically, the maximum possible utilization for a task set can be as low as sixty nine percent using RMS. In practice, however, usually much more than sixty nine percent of the processor can be utilized, and it is shown, using stochastic analysis, in [15] that the average scheduling bound for RMS is around eighty eight percent. As in DDS, however, the overhead of the scheduler and task switching must be taken into account. [16]

Task preemption is a significant event in a real-time system. When a task is preempted it is not allowed to complete its execution in the current frame. A preemption occurs when a task with a higher priority than the currently running task requests service. At that point there is a context switch and the higher priority task begins execution. When using DDS, preemptions are unpredictable and can occur at any time. In RMS, all preemptions are

known ahead of time, since task priorities are fixed. The IBM Mwave system currently uses DDS to schedule real-time tasks because it allows higher processor utilization.

2.1.3 Memory Systems

Real-time memory systems must also be deterministic. Memory latency must be predictable and repeatable. It is not beneficial to a real-time system for the memory system to have variable latencies depending on uncontrollable factors, which means that nondeterministic methods of decreasing memory latency are not useful. Caching serves to decrease the total time waiting for memory, but it does not ease the burden of the real-time environment, as the system must be prepared for everything to miss in the cache, as it well might for isolated sections of code. A cache monopolizes on significantly improving average case performance, whereas in a real-time system the only metric is worst case performance.

Currently, the IBM Mwave DSP uses SRAM in its memory system. The memory can be accessed in a single cycle so that the pipeline never stalls due to memory latency. This makes all execution times easily predictable from the worst case instruction count. The introduction of DRAM into the system causes problems. DRAM takes multiple cycles to access, so that the pipeline will stall on every memory access. Since the DSP must fetch the next instruction every cycle, this means that the pipeline will be stalled for every instruction. If there is a data memory access, it can go on concurrently with the instruction access because the Mwave system uses a Harvard architecture. This means that the worst case execution time of a task is still deterministically based on the worst case instruction count. However, if the DRAM takes four cycles to access, for example, then seventy five percent of the time, the processor will be stalled waiting for the memory system to return the data. The addition of a conventional cache would only lead to nondeterministic behavior, and is therefore not useful for decreasing worst case execution times.

Another minor issue is DRAM refresh. In a real-time system, the system cannot afford to have to wait while refresh occurs at unpredictable times. Therefore, the simplest and most effective solution is to have the operating system schedule a real-time task that refreshes the DRAM. That way, the refreshes will take place periodically, as they need to be, and they will only occur during the refresh task's execution so as not to interfere with other tasks.

2.2 The Frame Manager

Each real-time task must specify an execution frame in which it is going to run. In the Mwave system, a corresponding frame manager exists for each such execution frame that is being used in the system. There is only one frame manager for each execution frame being used in the system, so all tasks that use the same execution frame are assigned to the same frame manager.

2.2.1 Scheduling Tasks

The Mwave system makes use of these frame managers to schedule the real-time tasks using DDS. Each frame manager is responsible for scheduling the tasks in its frame. Since all of the tasks in any given frame manager have the same deadline, and therefore the same priority, they cannot preempt each other. Effectively, this means that once a frame manager is allowed to run on the DSP, it just runs through the list of tasks it is responsible for, and schedules the next one in the list until there are no tasks left. The frame manager then becomes idle until its next frame begins.

However, frame managers can preempt other frame managers, as each frame manager has a different priority based on its current deadline. A frame manager becomes active when its corresponding frame begins. At that point, if there is a lower priority frame manager, one with a later deadline, executing tasks on the machine then the newly activated frame manager will preempt the processor and begin executing its tasks. If the newly acti-

vated frame manager does not preempt another frame manager at the start of its frame, it must wait until all active frame managers with closer deadlines have run to completion before it will be allowed to run. Since the newly activated frame manager is now waiting to be run, a frame manager with a later deadline will not be allowed to run on the machine before the newly activated frame manager completes its execution. Since a lower priority frame manager will not be able to run, there is no chance for the newly activated frame manager to preempt the processor after the start of its corresponding frame. Therefore, the start of a frame is the only time at which a frame manager can preempt another frame manager.

2.2.2 Interaction Among Frame Managers

There are two important theorems that can be shown about the scheduling of frame managers using DDS. Both theorems deal with the order in which frame managers can run on the machine.

Theorem 2.1: In order for a frame manager to preempt another frame manager, the preempting frame manager must have a higher frequency frame rate than the preempted frame manager.

Proof: The nature of DDS is such that the highest priority task is always the task that is currently running. Priority is determined by the proximity of deadlines, so therefore the currently running task must have the closest deadline in the system. This means that all tasks with closer deadlines have already completed in their current frame. This is true in a theoretical DDS algorithm, but it is also true in the Mwave system. In the Mwave system, scheduling decisions are made at every interrupt and whenever a task completes execution. As previously stated, a preemption can only occur at the start of a frame. Also, frame boundaries always correspond to one of the periodic interrupts, as they are defined in relation to one of those interrupt sources. Since the only way that a task switch can occur is by

preemption or when a task completes, the Mwave DDS algorithm always schedules the same tasks that would be scheduled in the theoretical algorithm, as the scheduler is active in both of those situations.

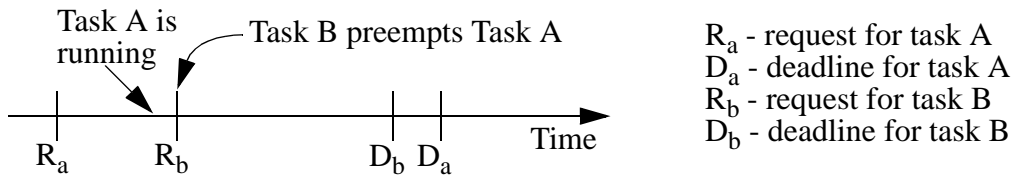


Figure 2.2: A Preemption by a Task with a Faster Frame Rate

Given that the currently running task must have the closest deadline in the system, it must belong to the frame manager that controls the frame with the closest deadline in the system. The only way that the currently running frame manager can be preempted is if another frame begins, since preemption only occurs on frame boundaries. It is also necessary that the preempting frame manager has a closer deadline than the current frame manager. Since the preempting frame manager controls a frame that begins after the current frame manager's frame and ends before it, this frame has a shorter period, and therefore must have a higher frequency. This can be seen in Figure 2.2. Task A is running prior to R_b , because it is the task that has the closest deadline on the machine that has not completed its execution in its current frame. At time R_b , task B becomes the highest priority task as D_b is sooner than D_a , and task B preempts task A. If D_b were after D_a , then task B would have lower priority, and no preemption would occur, which shows that the deadline of the preempting task must be sooner than the deadline of the task that is being preempted.

Some important consequences of Theorem 2.1 include the fact that the frame manager with the highest frequency can never be preempted, and the frame manager with the low-

est frequency can never preempt another frame manager. This can be helpful in understanding certain aspects of real-time scheduling.

Theorem 2.2: Preempted frame managers resume execution on the processor in a last-in first-out order.

Proof: If a frame manager is preempted, it must be because another frame manager has a closer deadline. Therefore, the preempted frame manager cannot run on the DSP until the preempting frame manager completes its execution because the preempting frame manager has a higher priority. Also, any frame managers that are allowed to run between the time that a frame manager is preempted and resumed, must have a closer deadline than the preempted frame manager. Therefore, all of those frame managers must also complete execution, whether they are preempted themselves or not, before the preempted frame manager can be allowed to resume execution.

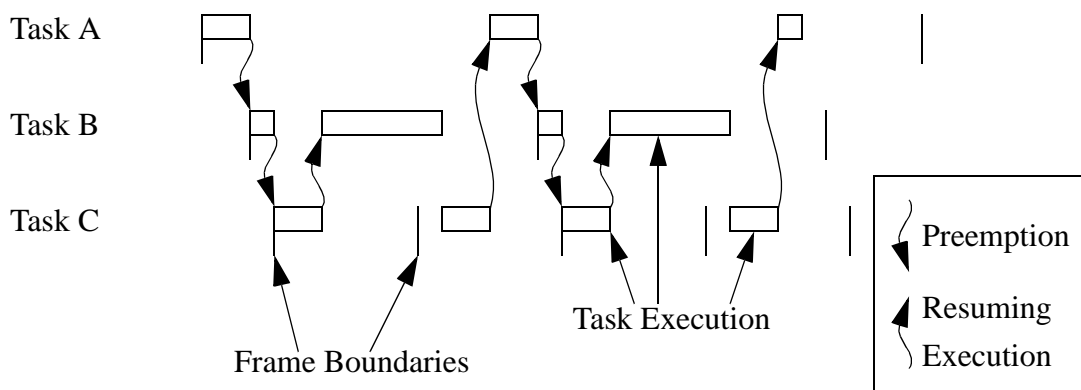


Figure 2.3: Preemptions in Dynamic Deadline Scheduling

Figure 2.3 illustrates how tasks are preempted and resumed. In the figure, downward crooked arrows represent preemptions and upward crooked arrows represent a preempted task resuming execution. Context switches that result from a task completing execution and a new task starting execution are not marked. From the figure, it can easily be seen that once a task is preempted, it does not resume execution until the preempting task completes execution. Furthermore, if other tasks commence execution before the preempted

task resumes, then those tasks will also complete their execution before the preempted task is resumed.

Thus, any preempted frame manager is effectively put on a LIFO stack and cannot resume execution until such time as it is on the top of the stack. This is true because all frame managers below a given frame manager on the conceptual LIFO stack must have later deadlines than the given frame manager, as it was allowed to run on the machine more recently than they were.

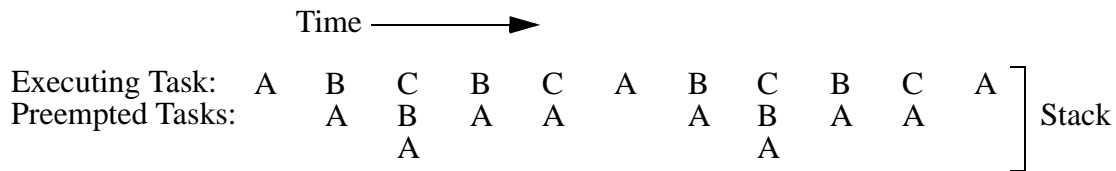


Figure 2.4: Conceptual LIFO Stack for the Schedule in Figure 2.3

Figure 2.4 shows the contents of the conceptual LIFO stack for the schedule in Figure 2.3. The stack is shown each time it changes, so the first entry just has task A running on the machine. Then, when task B preempts task A, task B is at the top of the stack, which corresponds to running on the machine, and task A is pushed down on the stack. This continues for the rest of the schedule that is shown in the figure. From Figure 2.4, it is very easy to see that tasks never reorder themselves on the stack. Once a task takes a position on the stack, it can only be pushed further down on the stack as new tasks with higher priorities execute on the machine, or be raised on the stack as tasks complete execution. Either way, the task always maintains its relative position on the stack.

The most important consequence of Theorem 2.2 is that there is a predefined order in which preempted frame managers resume execution. There may be other frame managers that run in between them, and frame managers could be preempted and resumed multiple times, but preempted frame managers will always leave the stack to resume execution in a last-in first-out order.

2.3 Caching

Conventional caches are driven by program memory accesses. This means that the data in the cache at any given time is completely dependant on the history of the current execution of a given process. This is assuming a non-multitasking system, so that processes do not interfere with each other in the cache. Once multitasking is allowed, however, it is impossible to predict what is contained in the cache.

The problem with multitasking is that preemptions are allowed, so different processes can overwrite each other's cache lines, thereby making the contents of the cache dependant on the history of the execution of all the processes on the system, as well as the order in which they were scheduled. Essentially this means that it cannot be predicted whether any given memory access will hit or miss in the cache. This is an unacceptable situation in real-time systems. A conventional caching scheme would only be useful if it were possible to predict exactly which memory accesses would hit in the cache and which ones would miss. Then the execution time of a task could accurately be predicted.

2.3.1 Real-Time Caching

Either protection or restoration can be used to prevent interference amongst tasks in the cache. This would allow each task to behave as if it exclusively owned the entire cache. To protect data, each task's data must be preserved across preemptions. This can be done by allowing cache lines to be locked in place so that they cannot be evicted. A unique cache management scheme would have to be developed in order to handle locked cache lines properly. Also, provisions would have to be made for the possibility that the entire cache could be locked, rendering the cache essentially useless if there is data that needs to be accessed which is not already in the cache.

In a restoration scheme, each task would be allowed to use the entire cache, and then when a context switch occurs, the cache would be copied to memory and the cache image

of the new task would be copied back into the cache. Restoration must occur before a task is allowed to execute on the machine. To restore data before execution without stalling the processor, the next task to run must be predicted. Using DDS, this can be quite difficult, if not impossible. If RMS were used, the next task to run could be easily determined by checking the schedule, which is completely predictable, as all frame managers have static priority. Another option is to delay a task's execution until the task's data can be completely loaded. This involves tampering with the real-time scheduling algorithm which could cause tasks to miss their deadlines unless extreme care is taken.

One of the best solutions to the real-time caching problem so far is the Strategic Memory Allocation for Real-Time (SMART) Cache. The SMART cache was developed for use in real-time systems that use RMS, however the SMART cache should work in systems using any preemptive scheduling algorithm. A system using the SMART cache can achieve processor utilization within ten percent of a system using a normal cache, and the SMART cache is deterministic. The SMART caching scheme uses protection to prevent task interference by dividing the cache into partitions. These partitions are then allocated among the tasks that are to be run on the processor. One larger partition is reserved as the shared pool, and acts as a normal cache. The shared pool allows shared memory to be cached without complicated cache coherence schemes. The shared pool can also be used to cache tasks when there are no available private partitions left. [12]

The allocation of partitions is static, since if it were dynamic some tasks could miss their deadlines while the cache was being reallocated. This means that the task mix to be run on the processor must be known ahead of time, so that a static analysis can be made to determine which tasks can use which partitions. Partitions can be reallocated dynamically, but it is time consuming. In his thesis, Kirk estimates the time to reallocate the partitions for a sample task mix running on a 25 MHz RISC processor to be around 0.15 seconds.

[13] In that amount of time, almost every task will miss at least one deadline. However, the allocation scheme is quite clever in that it allocates partitions in such a manner that the utility of each cache partition is maximized. Unfortunately, to accomplish this it is necessary to have detailed information about each task's hit and miss rates given an arbitrary sized private cache. Once the partitions are allocated, the tasks own their partitions. Essentially, this means that only the task to which a partition is allocated can use that partition, so that data is preserved across preemptions. [14]

As well as the fact that a static task mix is preferable, another major drawback to the SMART cache is that it requires additional instructions in the instruction set to control which partitions are being accessed, either the private ones or the shared pool. This requires that the programmer or the compiler know too much about the actual hardware implementation, which may change. This is not desirable in the Mwave system, as software must be able to run on all Mwave systems, not just those that have DRAM in the memory system.

Using either a protection or a restoration scheme eliminates the problem of conflict between tasks, but does not eliminate the difficulty in timing individual tasks.

2.3.2 Task Timing

The process of determining the worst case execution time of a task is called timing. In the current Mwave system, which uses SRAM so that there are no pipeline stalls due to memory accesses, all instructions take one cycle to complete because of the pipeline. In this system, timing tasks is reduced to finding the longest possible execution path through the code. With the introduction of DRAM and caching, timing becomes much more complicated. Now the longest execution path through the code might take less time than a shorter path that has significantly more cache misses.

Timing in this new situation requires keeping track of the state of the cache at all times, so that it is possible to determine if a given memory access will hit or miss in the cache. If the state of the cache is always known, then any execution path can be timed by simply adding the miss penalty for each cache miss to the time that it would have taken to execute the given execution path if there were no cache misses. This is in agreement with the assumption made by Halang and Stoyenko:

The fundamental assumption we make is that only software and hardware that can be checked (i.e. analysed for schedulability) for adherence to the constraints *before the application is run* fit the requirements of predictable real time applications. [7]

Since it is not always possible to determine the address of a given memory access before run time, it becomes a very difficult problem to incorporate a cache into a real-time system and still be able to time all of the tasks. However, in order to adhere to the above assumption, a method must be devised to allow task timing prior to run time, or the use of a cache in a real time system may be impossible. A likely solution is to use a pessimistic algorithm in which unpredictable memory accesses are assumed to miss in the cache. That will provide an upper bound on execution time, but the pessimistic bound may not be tight enough to allow sufficient processor utilization.

2.3.3 Task Size

One way to eliminate the timing problem is to make the cache large enough so that an entire task can fit into the cache, or a partition of the cache if partitioning is being used. Similarly, a restriction could be placed on task size to limit the tasks so that they fit in the cache. This is undesirable because cache sizes may be different on different processor implementations. However, if this is done, and a cache replacement scheme is used that does not allow data to be displaced from the cache, or optimally, the entire task is pre-

loaded, then the task can simply be timed by noting that in the worst case, the task will miss in the cache exactly once for each memory location.

Chapter 3

Designing a Real-Time Cache

3.1 Characteristics of the Target System

The specific system at which this research is targeted is the IBM Mwave digital signal processor for multimedia use. However, the results can be applied to other DSP systems and largely to real-time systems in general. Some of the important characteristics of the target system include having a flexible task set that can be changed dynamically by the user as the system is running, allowing the introduction of newly written software without redesigning the system, not requiring software to be aware of the hardware in the system, and allowing the system to use dynamic deadline scheduling.

The most important characteristic of the target system is that it is able to run arbitrary task sets. The user should be able to run whatever task set he wants, which may even include tasks he wrote himself. This is an important distinction, as many real-time systems only run fixed task sets, or dynamic subsets of a larger fixed set. By allowing arbitrary task sets, the memory system can be used in *all* real-time systems, specifically including DSPs for use in multimedia computer systems. Adding a cache, or other enhancement to allow the use of slow memory, should not prohibit the use of arbitrary task sets, if the system can otherwise do so.

Furthermore, users should be able to add and remove tasks from the task set while the system is running without disrupting *any* of the active tasks. This is important in a multimedia DSP system, as users do not want the music they are playing to be disrupted when they start up their fax or modem application, for example. Once again, adding a cache cannot enable the system to change the task set dynamically, but if the system can otherwise handle this, then the addition of a cache should not inhibit this ability.

Another important restriction is that modifying the memory system should not prevent existing software from being used on the system. The introduction of a cache should not force all existing real-time tasks to be rewritten, but knowledge of how the caching system works may make it desirable to rewrite existing software for performance gains.

Using a cache should not preclude the use of dynamic deadline scheduling. Using DDS is desirable for the increased real-time utilization that it allows, as well as the simplicity of the scheduling feasibility checks. Also, if possible, it is desirable to allow as many tasks to run on the system as computation and memory resources allow using the given scheduling algorithm. Adding a cache should keep further restrictions placed on the task set to a minimum.

Most importantly, the addition of a cache should not lead to nondeterministic behavior, as discussed in Section 2.3 on page 27. If the addition of a cache makes the system nondeterministic, then the system may not function properly. Typically, in order to achieve deterministic behavior, real-time caching systems ignore the nondeterministic behavior associated with caches and assume that in the worst case all memory accesses will miss in the cache. By introducing a deterministic cache, this worst case performance can be significantly improved.

The target system described here is a little more flexible than many real-time systems. The fact that the DSP can be used by a wide variety of users to perform various multimedia functions makes the problem of adding a cache more difficult than in a controlled real-time system where the software and hardware are developed together, usually for one specific purpose. Therefore, a solution that works on this general target system should be easily adaptable to many real-time systems, including those with more specific purposes.

3.2 Design Parameters

There are many factors that influence the design of a real-time caching system. However, five of these factors are more significant than the rest:

- the bandwidth between the memory and the cache,
- the number of frame managers allowed on the system,
- the use of shared memory,
- the size of typical tasks in relation to the size of the cache, and
- the method used to determine the time each task requires to run.

The bandwidth between the memory and the cache is the most significant factor in determining how fast cache pages can be swapped in and out. The number of frame managers running on the system can also have a significant impact on determining how to contain preemptions. The final three design parameters are dependent on the choice of the allowable number of frame managers, so they will be explored as they affect the design once that decision has been made. Whether or not shared memory is allowed can have an influence on the cache design if it is possible for the same address to be cached in more than one cache partition. Finally, the overall size of each task, and how each task will be timed are important in determining the usefulness of any real-time caching system.

3.2.1 Memory to Cache Bandwidth

Obviously, the higher the bandwidth, the less penalty there will be for using DRAM. A system that uses large cache blocks that are swapped in and out infrequently, rather than smaller cache blocks that are swapped quite frequently, is desirable, as sequential access offers the possibility for higher bandwidth.

If the bandwidth is sufficiently high, it may be possible to use a cache restoration scheme in which the cache is restored at context switch time by the operating system without much additional overhead. If the bandwidth is too low for this to occur, then the operating system will probably have to take more of the processor's resources to manage the cache in some less transparent manner.

The bandwidth also affects the way that individual tasks and frame managers will manage their cache space. Higher available bandwidth gives more flexibility in cache management, and allows frequent changes in the data in the cache. Memory that has lower available bandwidth requires that more attention be paid to carefully organizing data changes in the cache, and opens the opportunity for pipelining the stages of moving data into the cache, executing the task, and writing data back out to memory.

3.2.2 Number of Frame Managers

The number of frame managers that are allowed on the system has a significant affect on the system design. There are essentially two cases. The first case is that only one frame manager can be running on the system. The second case is that an arbitrary number of frame managers can be running on the system simultaneously.

Handling the single frame manager case is obviously much easier than handling an unknown number of frame managers. If there is only one frame manager, then that frame manager completely owns the entire cache. Since only frame managers can preempt each other, there will be no preemptions. The problem is reduced to the problem of scheduling the cache for the tasks contained within the single frame manager. Since the tasks within any frame manager are always executed in a predetermined, sequential order, the problem is relatively straightforward, as much of the work can be done prior to run time. An extension of the single frame manager case is the case in which there are only a small bounded number of frame managers allowed to run concurrently. In that case, there can be an independent cache, or more likely an independent cache partition, dedicated to each frame manager, so that all of the frame managers on the machine act as if they were operating in the single frame manager case. In this way, preemptions simply reduce to changing which cache, or cache partition, is connected to the processor.

The case in which the number of active frame managers is unpredictable is much more difficult. In this case, frame managers can preempt each other, and it is not guaranteed that there will be enough cache space to support all of the running frame managers. This leads to contention for cache space among the frame managers. The key to this situation is to make each frame manager think that it is running in the single frame manager case. This is likely to require cache partitioning and a nontrivial cache management scheme.

3.2.3 Shared Memory

If shared memory is allowed, cache coherence can become an issue. In any real-time caching system, each task will have its own cache image. If these images are separate caches or partitions, rather than sharing a single cache amongst all the tasks, then cache coherency must be maintained. Without shared memory, each frame manager can schedule its tasks and associated cache space without regard for other frame managers. However, once shared memory is allowed, there must be some method to prevent tasks from accessing stale data or updating data out of order. This is similar to the situation in multiprocessing systems where methods such as snoopy caching are used. [9]

There are many options to maintain cache coherence. One method is to require all shared data to be cached in a special shared partition which can be accessed by all tasks. This, however, brings about new problems of its own, as all shared data on the machine must be able to fit into this shared partition, and if it is not large enough, then cache scheduling problems will arise. Another option is to restrict shared data to reside in the cache in only one place at a time. That way, only the task that is currently using the data can have it in its cache space. Then when another task accesses that shared data, it will invalidate the existing copy and bring in its own copy. Once again, this will create timing problems, as it will almost never be possible to determine if shared data is resident in a particular frame manager's cache space in the face of preemptions.

3.2.4 Size of Tasks

The maximum size of tasks is an important factor to consider in the design. Each frame manager must be able to function correctly given a finite sized cache or cache partition. If all the tasks can fit entirely into this cache space, then there is no problem as long as a task's data is loaded before it is needed and written back to memory once the task has completed execution. However, once the task becomes too large to fit into the available cache space, the problem becomes significantly more difficult.

When a task is too large to fit entirely in the cache, it becomes important to decide what data should be in the cache at what time. This is quite a significant problem when it is coupled with the fact that the task's execution must be deterministic. There must be a method of determining the maximum number of cache lines that will be swapped in and out by the task before the task is run. That would allow the cache overhead to be included in the determination of the task's worst case execution time.

3.2.5 Task Timing

Task timing becomes necessary whenever there is a situation in which it is unclear whether or not a task's data will be resident in the cache. This encompasses just about all situations when DRAM and caching are components of the memory system. However, this is not a problem, as timing tasks is quite possible. Halang and Stoyenko present one such method, which they call a "schedulability analyser," in [7]. This method is part of the compiler, so it is completely automated, and is easily generalizable to any high level language that meets certain constraints, which they enumerate, that are necessary for use in a real-time system. The following is a brief description of such a method.

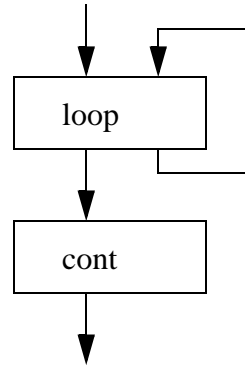
The heart of any task timing tool will involve constructing a weighted directed graph from the task's code. In the graph, the nodes represent straight line sections of code such that if the first instruction in the section is executed, then all of the instructions in the sec-

tion will be executed sequentially. The edges represent possible control flow paths. In other words, each node will have edges leaving it to all of the possible other nodes to which the program could branch. An example is given in Figure 3.1, which shows how a simple program fragment can be turned into a program graph.

```

loop:  R0 <- R0 - 1
      R1 <- R2 + R3
      R4 <- Mem[R1]
      R2 <- R2 + 4
      BNE R0, loop
cont:  ...

```



Sample Code Fragment

Associated Program Graph

Figure 3.1: A Code Fragment and its Associated Graph Representation.

Once this graph is constructed, the backward edges must be removed. In order for a task to be bounded in time, which is a necessary condition for any real-time task which must have an inviolable worst case execution time, all backward branches must have an upper bound on the number of times they can be taken. If a backward branch can be taken n times, then the graph can be unrolled to produce n copies of the loop. The last thing to do is to assign the edge weights. The weight of any given edge is the time it will take to execute the code represented by the node to which the edge points. This must take into account cache contents and reload times. Once this is completed, it is a simple matter to find the longest path through the graph, and there are many algorithms in existence that do exactly that. A few algorithms that find the shortest path in a weighted directed graph are given in [2]. They can easily be modified to find the longest path of the graph instead by changing the inequalities and making other minor modifications.

Timing tasks, therefore, while not a simple proposition, is entirely possible. The fact that all code used in the Mwave system is written in assembly language makes it a little more difficult to time. The branch targets and loop bounds are not always clearly defined in such a way that they could automatically be extracted from the code. This would require an interactive timing tool that requests input from the programmers, which, while not as elegant as the automatic tool of Halang and Stoyenko, is still possible.

Chapter 4

Memory Bandwidth

4.1 Cache Performance

In any system that makes use of a cache, the memory latency of the slower memory limits the achievable processor utilization. If there is a cache miss, then the processor *must* wait until the slower memory returns the required data. Obviously, the faster the data can be transferred, the less time will be spent waiting on a cache miss. In a real-time caching system, as in other systems, it is important for cache misses not to degrade the system performance severely. The best way to decrease the penalty of a slow memory access is to group large blocks of sequential accesses together, as the additional overhead is significantly less than that for isolated accesses. This serves to reduce the average access times for all of the data within the block.

However, reducing the average memory access time by transferring large blocks of data does not necessarily decrease the cache miss penalty. In fact, it probably increases the cache miss penalty, as the processor may have to wait for the entire transfer to complete, depending on where the required data is located in the block and the cache miss protocol. Although, if the rest of the data in the block is accessed by the processor later, then the overall system performance will improve. There is a trade-off between the average access time and the largest useful data block. The optimal block size must be determined by the locality of reference in the system. If the block size is too large, then too much time will be spent transferring unneeded data. If the block size is too small, then the miss rate will be too high. This problem is not unique to real-time systems, rather it is fundamental to good cache design.

The following is an overview of many techniques that can be used to increase memory bandwidth, which can directly increase processor utilization, as time spent waiting for data is minimized.

4.2 Decreasing Memory Latency

There are many ways to decrease the memory latency of DRAM. The problem is that none of the methods are deterministic. Almost all of the current research in the area of reducing memory latency is targeted at reducing the average case memory latency. Unfortunately, this is not effective for real-time systems, as the worst case memory latency must be reduced. However, by controlling the way the cache accesses the memory system, it may be possible to use some of these methods deterministically.

The use of static-column, or page mode, DRAM can help improve system performance. When data is accessed from static-column DRAM, the entire column that is being accessed is buffered. This allows subsequent accesses to the same column to be much quicker. If each task's data can be stored in one column of the DRAM, then worst case execution times are dependent on the column access time, rather than the DRAM's normal access time. This could easily increase processor utilization. If the data takes up multiple columns, even a small number of them, then a problem arises, as the task could access the columns alternately, and thus render the column buffer useless. However, for sequential block accesses within a column, the average memory access time is much lower when static-column DRAM is used instead of normal memory. [5]

Interleaved memory is a set of memory banks in which sequential addresses are spread across memory banks. The use of interleaved memory banks would allow multiple concurrent memory accesses. However, this is too easily defeated by simply using the data immediately after it is loaded. Interleaving can also be defeated by having consecutive

memory accesses to the same bank. This problem can be avoided to some extent by making use of several techniques designed to decrease the likelihood that consecutive memory accesses will be to the same memory bank. Interleaved memory works best for sequential accesses, as it is guaranteed that consecutive accesses will be to different banks. As long as consecutive accesses are to different banks, multiple memory accesses can occur concurrently, which dramatically increases the bandwidth of the DRAM. [8]

The use of video random access memory (VRAM) would allow quick instruction access for straight line sections of code. VRAM is dual ported DRAM in which one port is for random access, and the other is for sequential access. The sequential access port has very high throughput, so that the VRAM would be beneficial for sequential blocks of code, but whenever there is a branch that is taken, the full latency of the DRAM is incurred. [5]

Similarly, the use of a stream buffer would also help for straight line sections of code. A stream buffer is essentially a FIFO queue, in which instructions are prefetched sequentially after the current address and placed into the queue. If the DRAM has enough bandwidth to keep the stream buffer full, and there is a straight line section of code, the processor would be able to fetch instructions from the fast queue, instead of the DRAM. As with the use of VRAM, though, whenever there is a branch, the full memory latency of the DRAM will be incurred. [11]

Another possible option is to try to prefetch data and instructions. The stream buffer is really a method of prefetching instructions. Prefetching data requires looking ahead in the instruction stream to anticipate load instructions. Anticipating load instructions can be difficult due to branching, but when it is possible, a prefetch can be initiated. However, it may not be possible to determine the correct memory address if there is a register offset, which may change between the time of the prefetch and the time that the load is executed.

A better method of prefetching would be to add instructions to the instruction set to allow code driven preloading. [1]

When used to access sequential blocks of data, many of these methods deterministically decrease the memory access time. The use of static-column DRAM, interleaved memory, and VRAM can easily be used to access sequential blocks of data quicker than if each data word were accessed individually.

4.3 Reducing the Miss Rate

The most useful approaches to reducing memory latency are those that combine caching and some other methods for reducing memory latency. Some of the methods with the most potential are backing up a cache with static-column DRAM so that miss penalties would be decreased, or using intelligent prefetching into the cache to increase the hit rate.

Since digital signal processing code typically has access patterns similar to vector code, data prefetching can be driven by anticipating vector accesses. If the vector access stride can be determined, then prefetching can occur without looking for load instructions in the instruction stream. [3] One possible method to determine the vector access stride as a task is running is to use a stride prediction table, as described in [4]. This would allow speculative prefetching to be done, with the prefetched data stored in the cache, hopefully increasing the hit rate. This is not likely to lead to deterministic behavior. However, if the stride is specified in the code, then prefetching can deterministically decrease execution times.

For instruction memory, a cache could be combined with stream buffers or VRAM. This would allow straight line sections of code to benefit from the sequential access capabilities of the stream buffer or VRAM, and branches to be handled by the cache.

Interleaved memory could also be used in conjunction with a cache. If the cache is used to hold large blocks of data instead of small lines, then memory interleaving can dramatically increase the memory bandwidth from the DRAM to the cache. In loading a large block of data, all memory accesses are sequential, which would allow the interleaved memory to make full use of all of the memory banks in the system simultaneously.

The methods mentioned here work best with a cache that holds large blocks of data rather than small lines. The larger the data block, the more the memory latency can be amortized across a long sequential access. Of course, there are limits, and the block size can be made too large to take advantage of the memory configuration successfully. Also, there is a point at which increasing the block size will also increase the miss rate, and therefore be detrimental to the system, as explained in Section 4.1 on page 41.

Chapter 5

A Single Frame Manager System

5.1 The Single Frame Manager

As mentioned in Section 3.2.2 on page 36, handling a single frame manager is much simpler than handling an unknown number of frame managers. There can be no preemptions with a single frame manager, as all tasks have the same priority, and tasks are always executed in a predetermined, sequential order. Therefore, the problem of dealing with the cache reduces to the problem of scheduling the cache for a predictable execution order of tasks.

A system in which there is only a single frame manager can be a useful system. That is effectively what is done in a system that uses the timeline scheduling algorithm, such as AT&T's VCOS[†] real-time operating system. [19] [20] [21] VCOS, which stands for Visible Caching Operating System, is intended for use in a real-time system with a visible cache. A visible cache is simply a cache that is visible to the users of the system, unlike conventional hidden caches, which are usually only visible to the operating system. In VCOS, there is only one execution frame, and in that frame, tasks are executed sequentially.

Functioning systems that use VCOS typically have small caches that contain about 1 or 2 kwords and store program code and data in the host system's memory, as the system does not have its own memory beyond the cache. The caches are small so that data can easily be transferred from the host memory to the cache without significantly degrading the system's real-time performance. All tasks must fit within the cache, as the miss penalty would be unbearable if data had to be fetched from the host memory. The cache load and

[†] VCOS is a trademark of AT&T.

unload takes place sequentially before and after a task's execution. This is moderated by the fact that the frame size is rather large (usually 10 ms) so the tasks are correspondingly longer, as they must operate on a larger number of samples. The load and unload time is therefore not as large a percentage of the execution time as it otherwise would be.

The tasks that can execute on this type of system are limited by both the cache size and the frame size. Tasks must be able to function in the given execution frame. If they need a shorter frame, they must do multiple frames' worth of work in the one larger frame, and if they need a larger frame, they must either split up the work among multiple frames, or do nothing in some frames. Having tasks that are idle in some frames is clearly a waste of real-time resources. Also, unless a task is executing in its desired frame, or possibly a multiple thereof, it will be inherently inefficient. However, this solution leads to a functioning caching system, which is not trivial. Tasks are limited to the cache size, because the DSP executes out of host memory space, so there is no way to effectively service a cache miss without forfeiting nearly all of the system's real-time capabilities.

In this framework, task writers do not have the flexibility to choose a frame rate that coincides with a useful number of samples for an efficient signal processing algorithm. Instead sample rate conversion must be done to simulate a particular sampling rate that is more efficient for the purpose of coding an algorithm. There are also other limitations to this type of system. If there can be only one frame rate in the system, it probably will be a slow one, so that all tasks can have useful work to do in each frame. This brings about issues of response time, as each task has to wait an entire frame before it can respond to an external event, which could be a severe constraint. Despite these problems, and possibly others that are related to digital signal processing issues, a usable system can be constructed that hides most of the memory latency with a single frame manager.

Ideally, a real-time system will be able to handle whatever frames the programmer desires. Obviously, full generality cannot be obtained, but hopefully more than one frame will be supported, as a system that only supports a single frame places a large burden on the programmer. However, a system that uses only a single frame manager can be used as a basis for constructing a more general system.

5.2 The Loader and Unloader

In order to schedule the cache effectively, there needs to be a notion of a loader and an unloader for the cache. The loader is something that loads a task's data from memory into the cache, and likewise the unloader copies modified data in the cache back into memory upon completion of execution. The loader and unloader may be conceptual constructs or actual hardware. Figure 5.1 shows a simplistic design of how the loader and unloader would fit into a caching system.

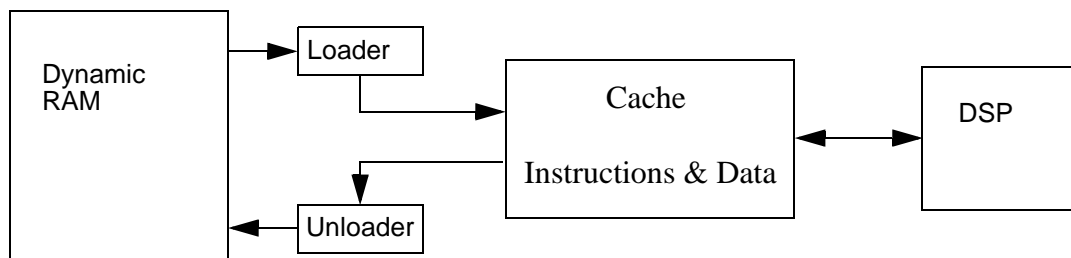


Figure 5.1: Overview of a Real-Time Caching System

The stages of a task's execution must be broken up into blocks of loading, execution, and unloading. Each task can specify multiple load and execution blocks, and unloads would occur automatically after each execution block on all modified data. If the loader and unloader cannot be active simultaneously with the DSP, then task execution would occur sequentially, in the following order: load, execute, and unload. However, if all three systems can be active simultaneously, then the next task in the schedule can be loaded and

the previous task unloaded at the same time that the current task is executing. In this case, the length of each pipeline stage is the longest of loading, executing, or unloading.

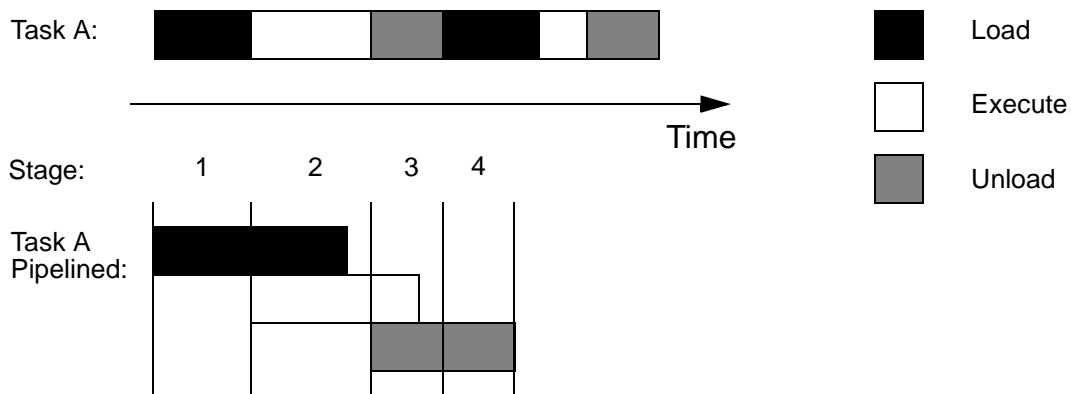


Figure 5.2: Task Execution with One Frame Manager

Figure 5.2 shows an example of sequential and pipelined execution. If loading, execution, and unloading occur simultaneously, then there can be three tasks' data in the cache simultaneously. This means that either the size of the cache will seem one third as big to each task, or the cache size must be tripled in order to maintain the same effective cache size as in the sequential case. Also, cache coherency issues must be addressed if three tasks are allowed to reside in the cache simultaneously. A method of handling cache coherency in a partitioned cache will be presented in Chapter 6.

Using the pipelined system, tasks can be pipelined amongst themselves, and similarly, all of the tasks in the frame manager can be put into the pipeline one after another, with no pipeline stalls. Since everything is known in advance, the length of each stage can be predetermined, and the entire time of execution from the first load to the last unload can be calculated ahead of time for the frame. This allows deterministic execution, so the system can effectively operate in real-time without missing deadlines.

5.3 Using DRAM in Page Mode

The above model also applies to DRAM that can be used in page mode without a cache.

Page mode DRAM, as described in Section 4.2 on page 42, is memory that allows a page to be accessed in the normal DRAM access time, but once that page has been accessed, subsequent accesses to the same page can be serviced quickly. When another page is accessed, the old page is lost. So effectively in this arrangement the active page can be considered the cache. The load time then becomes the time to bring up a page, and there is no unload time, as writes update the DRAM directly.

However, loading and execution cannot occur simultaneously, so paging DRAM can only be used in the sequential model. This is not too much of a problem, as the time to bring up a page is typically quite short, perhaps on the order of four or five machine cycles. The use of page mode does require timing on a finer granularity than using a cache, as a page in a DRAM is typically quite small, as it is really just the column that is selected in the DRAM which is stored in a fast buffer.

5.4 Task Timing

As discussed in Section 3.2.5 on page 38, task timing will be necessary if the cache is not large enough for the entire task to fit into the cache. However, in this system, timing can be simplified by separating a task into sections. If each section only accesses a block of data that is smaller than the cache size, then these sections can be pipelined in the loader, DSP, and unloader. Furthermore, if the execution time of each section is about the same or greater than the load time for the next section, and the unload time for the previous section, then there will be almost no overhead for loading and unloading the cache, except the initial priming of the pipe and the final flushing of the pipe in each frame.

If tasks cannot be broken down into small enough sections, then timing becomes much more difficult. In that case, the methods described in Section 3.2.5 on page 38 can be used

to time the tasks. However, the sequential model must be used, as a cache miss cannot be serviced if the loader is busy loading up another cache partition.

Chapter 6

Execution Quantization

6.1 Preemptions

Preemptions occur when more than one frame manager are running on the system. It is impossible to predict when these preemptions will take place. As a result, it is necessary to manage the data that is stored in the cache carefully. When a task resumes execution after being preempted, the cache must contain the same data that it did when the task was preempted. The time taken to ensure that the cache contents are correct in spite of preemptions must be deterministic, and it must be guaranteed that there are sufficient resources available to manage the cache. These restrictions are important, because if they are not met, then it is conceivable that many tasks will not complete before their deadlines because the cache was not ready for them to use.

One way to manage the cache is to make each frame manager save the cache before it executes and restore the cache when it completes. This would guarantee that the cache contents are always correct, because after a preemption they will always be returned to the state they were in before the preempting frame manager started to execute. To guarantee that there are enough resources to make this possible, each frame manager must increase its worst case execution time to include the time needed to save and restore the cache. Since frame managers can only be preempted by other frame managers with a faster frame rate, as shown in Theorem 2.1, the slowest frame manager does not have to increase its worst case execution time, as it will never preempt another frame manager. This is a very simplistic method of cache management, and it is not very efficient. Clearly frame managers do not preempt each other at the start of every frame, but this method requires that they be prepared to do so anyway. This may be necessary because of the unpredictable nature

of preemptions, but it is very costly. A significant amount of time is being reserved in case it is needed to save and restore the cache, and if it is not used, then it cannot be reclaimed for real-time use.

6.2 Quantization

A better way to overcome the problem of preemptions is to guarantee that each task can execute for a certain length of time once it starts running. By providing a guarantee, the possibility of loading the cache only to execute a few instructions is eliminated. This is the basis for execution quantization (EQ). In EQ, time is broken up into equal sized quantization blocks. Each quantization block is a period of time in which only one task can execute, so once a task begins execution it is guaranteed that it can execute for the entire quantization block. These quantization blocks are separated by quantization boundaries. During the quantization boundaries, the scheduler can become active and decide which task will execute in the next quantization block. Effectively, the scheduler sees time discretely, as it only makes decisions on the equally spaced quantization boundaries.

There are two major drawbacks to EQ. The first is that one hundred percent processor utilization cannot be achieved. This occurs because the scheduler cannot make decisions on a fine enough granularity, so the processor is sometimes idle even though there are tasks waiting to be run. However, this penalty is minimal, and does not prevent the system from being useful. The second drawback is that tasks must execute for the duration of an integral number of quantization blocks. This problem arises because the scheduler can only make decisions on quantization boundaries, so if a task completes execution in the middle of a quantization block, a new task cannot be switched in until the next quantization boundary.

In spite of these problems, EQ deterministically allows arbitrary task sets to achieve processor utilizations that are very close to the maximum utilization of the scheduling algorithm. In addition, EQ allows tasks to be guaranteed that they will execute for a given length of time before they can be preempted. Most importantly, with the use of EQ and dynamic deadline scheduling, a cache that is transparent to the programmer can be incorporated into a real-time system, which has been impossible without succumbing to very low processor utilizations.

6.2.1 Dynamic Deadline Scheduling

When using execution quantization in conjunction with dynamic deadline scheduling, the dynamic deadline scheduling algorithm fundamentally stays the same as described in Section 2.1.2 on page 19. The only real difference is that scheduling decisions are only made on quantization boundaries.

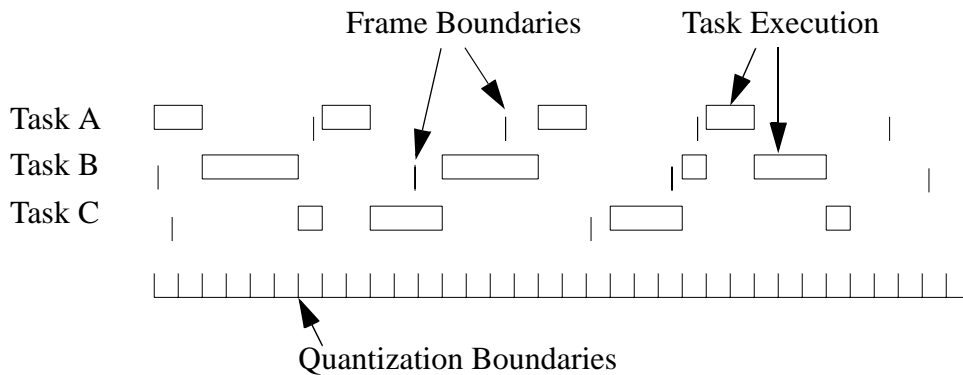


Figure 6.1: Quantized Execution Using Dynamic Deadline Scheduling

Figure 6.1 shows how quantized execution affects dynamic deadline scheduling. By forcing the scheduler to act only at discrete time intervals, some preemptions that would have happened earlier are held off until the next quantization boundary. It is also clear from the figure that each task is allowed to execute for an entire quantization block once it begins execution in that block, but at the end of that block, the task is subject to preemption. If the task is not preempted, then it can continue to execute in the next block, and it

cannot be preempted until the next subsequent quantization boundary. Looking at the figure, it is also easy to see that each task *must* execute for the entire quantization block once it begins executing in that block. This phenomenon occurs because scheduling decisions are only made on quantization boundaries, so once a task is executing on the system no other task can begin execution until the next time the scheduler becomes active. In reality, the task does not actually have to continue execution; however, that time is not available for real-time use! Therefore, each task must round up its worst case execution time so that it is an integral number of quantization blocks.

6.2.2 Utilization Restrictions for Execution Quantization

It is important to notice that in regular DDS each task executes as early as possible in each of its frames. If the processor utilization is below one hundred percent, then there is some time between the completion of execution of each task and its deadline in every frame. A task's utilization is a measurement of the amount of the processor's resources that the task requires. Utilization for an individual task is defined to be the ratio of the task's worst case execution time to the period of the task's frame. The total processor utilization is the sum of the utilizations of each task that is running on the processor. The following theorem shows that there is a direct relation between the total processor utilization and the minimum time between the completion of a task and that task's deadline in every frame.

Theorem 6.1: In a system that uses DDS to schedule a task set with total utilization U , each task completes its execution in every frame of period T seconds at least $(1 - U)T$ seconds before the end of the frame.

Proof: Let S be a schedule of some arbitrary task set, τ . Let U be the utilization of τ . In S , each task must execute as soon as it possibly can. This is a direct result of the DDS algorithm, as a task is scheduled as soon as it is the highest priority task that still needs service.

Let τ' be a new task set in which all tasks in τ are also in τ' , and all tasks in τ' have the same or greater utilization than the corresponding tasks in τ . Let S' be the schedule of the new task set, τ' . Since, in the schedule S , all tasks execute as soon as they can, and all tasks in τ' take as long or longer to execute than the corresponding tasks in τ , no task will execute earlier in S' than the corresponding task did in S .

Since DDS allows any task set with utilization of one hundred percent or less to be scheduled, the utilization of any task in τ can be increased by $1 - U$, and τ will still be schedulable. No task will execute any earlier once this task's utilization has been increased, which was shown above, as no task's utilization has been decreased. Furthermore, the task with the increased execution time completes execution before its deadline in every frame. In the schedule S , the task must have completed at least $(1 - U)T$ seconds before the end of each of its frames of period T seconds, in order for the schedule S' to be feasible. Therefore in S , every frame of period T seconds, has $(1 - U)T$ seconds left between the completion of that frame's task, and the end of the frame, since any task's utilization can be increased by $(1 - U)T$ seconds without causing an overflow to occur.

6.2.3 Proof of Correctness for Execution Quantization

In order for execution quantization to work, the utilization of each task must be quantized. A task's utilization is quantized by forcing the task to have a worst case execution time that is an integral multiple of the length of a quantization block. This is necessary because the scheduler can only make scheduling decisions on quantization boundaries, so in the worst case a task would have to execute until the completion of the final quantization block in which it executes. The following theorem will be used to show that any task set can be scheduled using dynamic deadline scheduling and execution quantization, as long as the task set's quantized utilization is less than $\frac{T-Q}{T}$, where T is the length of the shortest frame period in the task set, and Q is the length of a quantization block.

Theorem 6.2: Let Q be the length of a quantization block. If every task has a worst case execution time that is an integral multiple of Q , then, in the worst case, using execution quantization causes all tasks to complete on the quantization boundary immediately following the time when they would have completed if dynamic deadline scheduling without execution quantization were used.

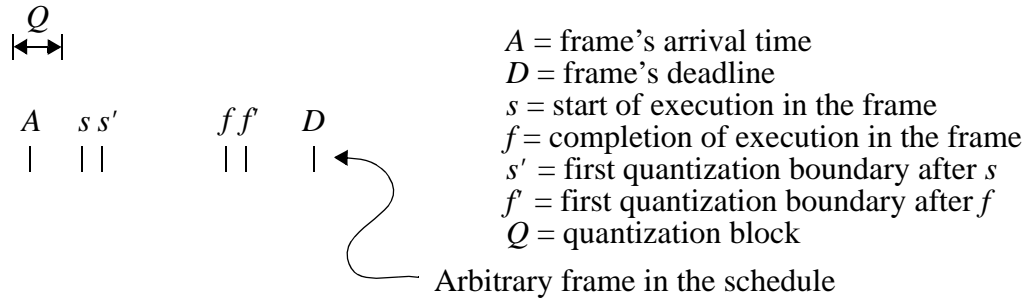


Figure 6.2: An Arbitrary Frame in the Schedule

Proof: Let S be a schedule of a task set, in which each task has a worst case execution time that is an integral multiple of Q , that was scheduled using dynamic deadline scheduling without execution quantization. Figure 6.2 shows an arbitrary frame in S .

In the figure, s is the time at which the task first executes in the frame, and f is the time that the task completes execution in the frame. Let τ be the task that runs in this frame, and let C be the worst case execution time of τ . The first thing to note about this frame is that the time interval $[s, f]$ must contain an integral number of quantization blocks. This follows directly from the fact that the execution times for all tasks are an integral number of quantization blocks. Clearly, at time s , τ has the highest priority of any task that is ready to run. Therefore, until τ has completed its execution, only tasks with a higher priority will be allowed to run. Furthermore, if a task with a higher priority begins execution, it must also complete execution. This was shown in Theorem 2.2, as the preempted task cannot resume execution until all tasks that preempt it have completed execution. So clearly the sum of the execution times of tasks other than τ running in the interval $[s, f]$ is the sum of

the entire execution times of some number of tasks. Since each term in that sum is an integral number of quantization blocks, the sum will also be an integral number of quantization blocks. Since τ must also run in $[s, f]$ and C is an integral number of quantization blocks, the total time spent on computation in $[s, f]$ is an integral number of quantization blocks, and there is no idle time in $[s, f]$, as τ is ready to run, and does not complete its execution until f .

For the rest of the proof, mathematical induction will be used. Assume, for all time before t , that execution quantization forces all tasks in the schedule S to complete on the quantization boundary immediately following their completion in S . Therefore, for every frame that occurs before t , the completion time, f , is moved back to f' , in the worst case. The base case is the first task that executes. In the beginning of any schedule, every frame begins at the same time. Therefore, the task with the fastest frame rate will execute first, and, as Theorem 2.1 shows, it cannot be preempted. Since this task cannot begin execution until the first quantization boundary after the start of the schedule, it will complete on the quantization boundary immediately after the time that it completed in S .

To show that if this property holds until time t , it will hold for the task that starts execution at time t , the points of interest are when t is the start of execution in some frame. In this frame, s is forced to move to s' . There are two cases to consider. The first is if $s = A$. In this case, s must move to s' , as τ cannot preempt until a quantization boundary. The second is if $s > A$. In that case, s is also the completion time of some task of higher priority than τ . From the induction hypothesis, that completion time is moved to s' , so the start of τ must also be moved to s' . It may no longer be true that τ starts at s' , but it can certainly start no earlier.

Examining the time interval $[s', f']$ more closely reveals that the total computation in that region is identical to the computation in the schedule S in the time interval $[s, f]$. Since

all frame boundaries that occur in $[s, s']$ will be delayed until s' , and all frame boundaries that occur in $[f, f']$ will be delayed until f' , the frame boundaries in $[s', f']$ in the new schedule are exactly the same frame boundaries that were in $[s, f]$ in the schedule S . Therefore, the same tasks that executed in $[s, f]$ in the original schedule, will execute in $[s', f']$ in the quantized schedule. Therefore, τ will have completed by f' , which completes the proof by induction.

Theorem 6.2 shows that, in the worst case, execution quantization delays the completion of all tasks to occur on the quantization boundary that immediately follows the time that each task would have completed using dynamic deadline scheduling without execution quantization. This delay can be no greater than one quantization block.

Let T be the length of the shortest frame's period in the task set, and Q be the length of a quantization block. As proved in Theorem 6.1, for any task set whose utilization is less than $\frac{T-Q}{T}$ that is scheduled using dynamic deadline scheduling, there will be at least Q seconds left at the end of each frame between the task's completion and the deadline of the frame. Since Theorem 6.2 proves that execution quantization will delay the completion of each task by at most Q seconds, there can be no overflows if the task set's quantized utilization is less than $\frac{T-Q}{T}$. Therefore, execution quantization is feasible for all task sets whose quantized utilization is less than $\frac{T-Q}{T}$.

6.3 Loading and Unloading the Cache

To ensure that tasks will not be preempted shortly after they begin execution, execution quantization guarantees that tasks will run for blocks of time. Once the cache is loaded and a task begins execution, the task is guaranteed to be able to do some useful work. When time can be spent loading the cache only to execute a very small number of instructions it is very difficult to ensure that tasks will complete before their deadlines, as it is quite pos-

sible that a task will be preempted before it can accomplish anything. The time spent reloading a task over and over can quickly dominate a task's execution time if it is not guaranteed that the task will execute for some length of time before being preempted. EQ eliminates this problem by guaranteeing that a task will be able to execute without interruption for a given length of time. Even though EQ guarantees that tasks will execute for entire quantization blocks, it is still necessary to load the cache with the appropriate task's data at the correct time.

EQ also facilitates cache management. If quantization blocks are long enough to unload the cache and reload it with new data, then a task's data can be loaded in the quantization block preceding its execution, and unloaded in the quantization block succeeding its execution. In the IBM Mwave system, instructions are immutable, so they do not have to be unloaded back into memory, and coefficients and other immutable data do not have to be unloaded either. Also, if there is enough hardware to allow simultaneous loading and unloading, then they can obviously go on concurrently. The loader and unloader should still work as described in Section 5.2 on page 49, and pictured in Figure 5.1. The major difference in the functionality of the loader and unloader is that they are controlled by the operating system for use by all tasks on the machine, rather than by a single frame manager for the exclusive use of that frame manager's tasks.

The remaining problem is the determination of which task to load into the cache at any given time. Ideally, an omniscient scheduler would be used. However, it is unlikely that the scheduler will always know which task to load next, and such speculation could result in a task's data being loaded into the cache before the start of its execution frame. Typically, the start of an execution frame denotes the time at which it can be guaranteed that a task's data will be ready, so it is not advisable to attempt to load that data early. A useful

system requires the scheduling algorithm to schedule the cache loads as well as task execution.

6.3.1 Dynamic Deadline Scheduling and Loading

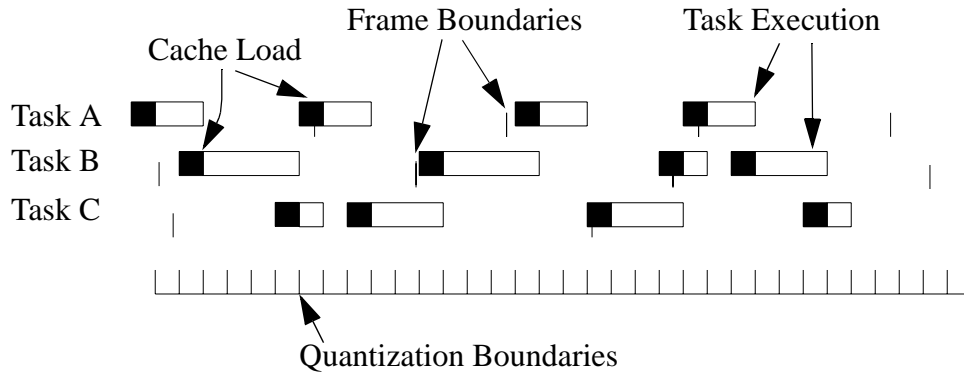


Figure 6.3: Execution Quantization with Loading

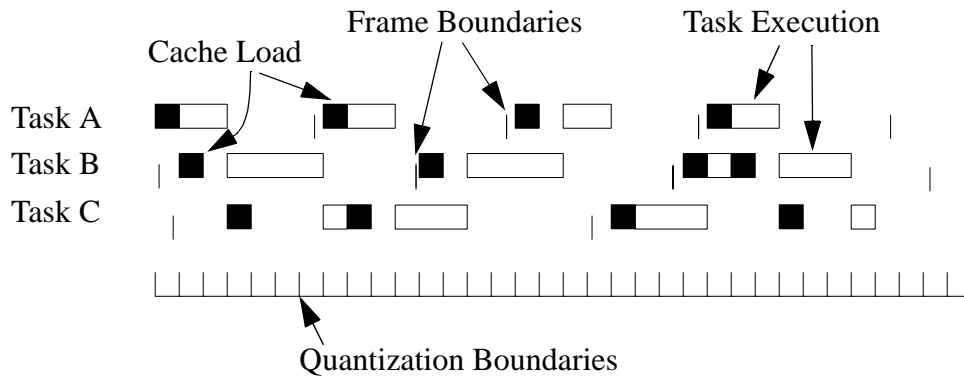


Figure 6.4: Execution Quantization with Speculative Loading

The schedules shown in Figure 6.3 and Figure 6.4 are modifications to the schedule originally presented in Figure 6.1. These modified schedules include the time that is used to load the cache, which is represented by the black boxes. The actions that occur in the cache load time are to copy the mutable data that is currently in the cache back to memory and then to copy the new task's data and instructions from memory to the cache. In order not to disrupt the schedule, the cache must be loaded at the same time that another task is running. There must be two caches, one that can be used by the processor, and one that can be used by the loader and unloader. The scheduler must be able to switch which cache the

processor and cache loader have access to, in order to perform a context switch. Rather than having two separate caches, a single cache with two partitions could be used, but it must be possible to access both partitions in the same machine cycle. This can be accomplished by using either a dual ported cache or by using a cache with a sufficiently fast access time that multiple accesses can occur each machine cycle.

Figure 6.3 shows a modified dynamic deadline scheduling algorithm that loads the cache in the quantization block immediately preceding execution. This requires prediction of which task will execute next. Also, if a task completes execution before its worst case execution time, then the cache will not be ready for another task to run. However, the scheduler is capable of scheduling all of the tasks if they all take their worst case execution times, so the next task does not need to be scheduled until the current task would have completed in the worst case. Requiring the scheduling algorithm to predict which task will execute next, however, is quite difficult. It is not always possible to predict exactly when an interrupt will occur, and therefore any possible strategy for predicting the schedule is prone to error. If the scheduler is not always able to predict which task to load, then there will be time spent waiting for a task to load when that task should be executing. If this occurs at unpredictable times, the system will not be deterministic.

In order to make Figure 6.3 clearer, unloading is not pictured. It is implicit that a task's data is copied back to memory in the quantization block immediately following the task's execution. Since a quantization block is long enough to perform the longest possible unload and the longest possible load, it is guaranteed that it will always be possible to unload the cache at that time. Figure 6.4 also does not show unloading for the sake of clarity. However, it is also assumed that when speculative loading is used, a task's data will be copied back to memory in the quantization block immediately after a task completes its execution.

Speculative loading is a better loading method, as it does not require prediction. The idea of speculative loading is to have the highest priority task's data always in one partition of the cache and the second highest priority task's data always in the other partition. If they are not, then they should be loaded, with preference given to the higher priority task, if they both need to be loaded. This method is illustrated in Figure 6.4. In this variant of dynamic deadline scheduling, a new task is loaded as soon as there is a free cache partition, assuming that there is a task that is ready to execute. This ensures that unloading will take place as soon as possible, and if there are no available tasks to load, then unloading can occur by itself. This method alleviates both problems from the previous method. First, if a task finishes early, then another task is available to start running at the next quantization boundary, unless there is no task that is ready to be loaded by that time. Also there is no prediction involved, as the cache simply contains the data for the two highest priority tasks. There is one significant drawback, however, which is that data may be loaded and then never used before it is unloaded. This occurs when a task's data is speculatively loaded, and then a higher priority task becomes active. The speculatively loaded task no longer has the second highest priority, so it will be unloaded. In this case, the cache has effectively been preempted. This wastes bandwidth between the cache and memory, as data was transferred that was never used. However, if that data had not been transferred, the memory, the loader, and the empty cache partition would have been idle, so the resources were available. Using available resources for a potentially unnecessary data transfer is outweighed by the usefulness of the speculative loading method.

6.3.2 Utilization Constraints

Loading and unloading the cache require that the maximum processor utilization be restricted even further than the restrictions required for execution quantization to work. The use of speculative loading delays each task's execution by exactly one quantization

block. This is true because of the latency required to load the cache. Since prediction is not used, any task that runs at the start of its frame must be delayed by one quantization block, as that first quantization block must be used to load the cache. By induction, it is clear that if all tasks prior to a given task execute one quantization block late, then the given task will also be delayed by one quantization block, waiting for the previous task to complete its execution and simultaneously loading its cache partition, if it is not already loaded. Other than the delay of one quantization block relative to the frame boundaries, loading does not modify the schedule. This effect can be seen by comparing Figure 6.1 with Figure 6.4.

As the cache is unloaded in the quantization block immediately following the completion of execution, each task is complete exactly two quantization blocks after it would have been completed using execution quantization without loading or unloading. This means that if it can be guaranteed that there are two free quantization blocks at the end of *every* frame, then loading and unloading can be added to the schedule, and no task will overflow. Theorem 6.1 states that if the utilization of a task set is U , then there will be $(1 - U)T$ seconds free at the end of each frame of period T seconds. In order to guarantee that there will be two quantization blocks free at the end of each frame, the processor utilization must be restricted by two quantization blocks out of the shortest execution frame. So in order for execution quantization to work with loading and unloading, one quantization block out of the shortest execution frame must be unused, as stated in Section 6.2.3 on page 57, in addition to the two required for loading and unloading. Therefore, any task set whose processor utilization is less than $\frac{T-3Q}{T}$, where T is the shortest execution frame period and Q is the length of a quantization block, can be scheduled in a system with a cache using DDS and EQ.

6.4 The Operating System

Every time there is an interrupt from one of the periodic interrupt sources or a task completes its execution, the scheduler must decide which task should execute next. Since interrupts occur quite often, the operating system is frequently executing small sections of code. The ideal system would store the operating system in its own cache partition that is locked in place. This would always allow the operating system to execute immediately without waiting for its data to be loaded.

The operating system does not fit nicely into execution quantization. Since the operating system only executes small sections of code, it is not practical to make the operating system use an entire quantization block every time it needs to make a scheduling decision. If the operating system is allowed to execute within a quantization block without using the whole block, then the task that uses the rest of that block is not guaranteed that it will be able to execute for an entire quantization block.

One solution is to guarantee that only the time in the quantization block minus the longest path in the operating system is available for real-time use. This is not an ideal solution for two reasons. First, it is very pessimistic, because the operating system will only need to execute in some fraction of the quantization blocks, so there will be a lot of time that is not available for real-time use. Second, the real-time resources that must be devoted to the operating system are dependant on the size of the quantization block. This is extremely undesirable, as it will drive the quantization blocks to be larger in order to reduce the overhead of the operating system, whereas it is advantageous to have small quantization blocks in order to allow the scheduler to make decisions on as fine a granularity as possible.

A better solution is for the operating system to execute between quantization blocks in the time denoted as the quantization boundary. The size of the boundaries is only

increased when the operating system must execute. Since the operating system only executes for very short periods of time, subsequent quantization blocks will not be delayed very long. If the operating system's utilization is u , then EQ will be feasible if the total task set utilization does not exceed $\frac{T-3Q}{T} - u$. This follows simply from the fact that tasks cannot make use of resources that are needed for the operating system. Once those resources are reserved, removing three quantization blocks out of the fastest frame on the system still guarantees that tasks will have sufficient slack time to allow loading, quantization, and unloading.

6.5 Hardware Implementation of Quantization

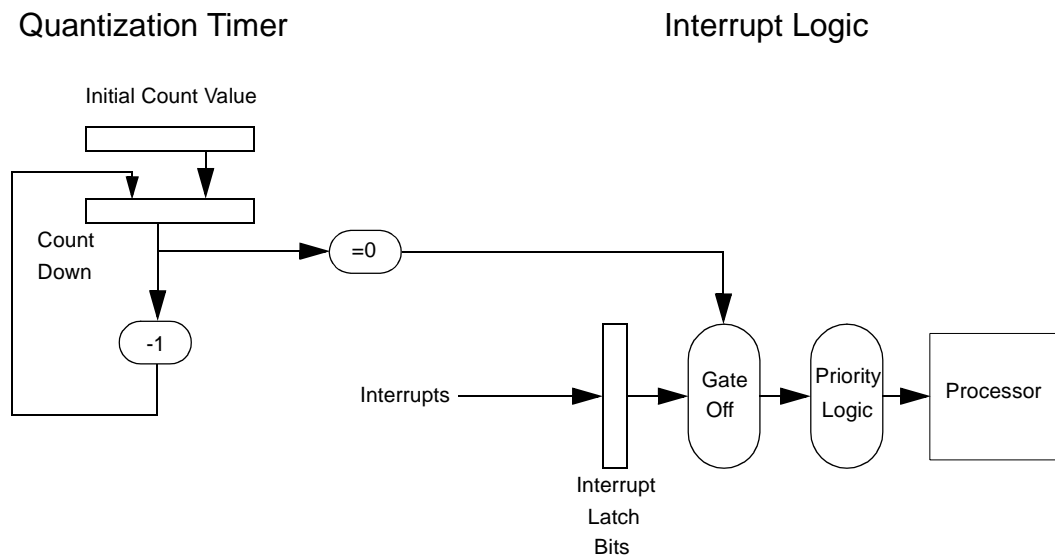


Figure 6.5: Quantization Hardware

Interrupts are used to define frame boundaries, as explained in Section 2.1.1 on page 18, so in order for execution quantization to work, interrupts must be delayed until quantization boundaries. The interrupts still need to be latched and time stamped when they occur, to ensure that the interrupts are eventually able to reach the processor with accurate timing information. The time stamp is used to determine an estimate of when a frame's deadline will occur. By using hardware to delay the arrival of interrupts to fall on a quantization

boundary, the regular dynamic deadline scheduling algorithm can be used with only minor modifications to control the loader. Quantization is enforced by not allowing an interrupt to interfere with a task's execution in the middle of a quantization block.

All of the interrupts can be delayed using a simple counter, as shown in Figure 6.5. The counter must be initialized with the number of instructions that are in one quantization block, which is stored in the register labelled "Initial Count Value" in the figure. The counter is then clocked with the system clock and is decremented by one each cycle. When the counter becomes zero, it is a quantization boundary. At that point, the interrupts become visible to the processor and the counter is reset to its initial value. While the operating system is running, the counter is not clocked, as that time is not part of a quantization block.

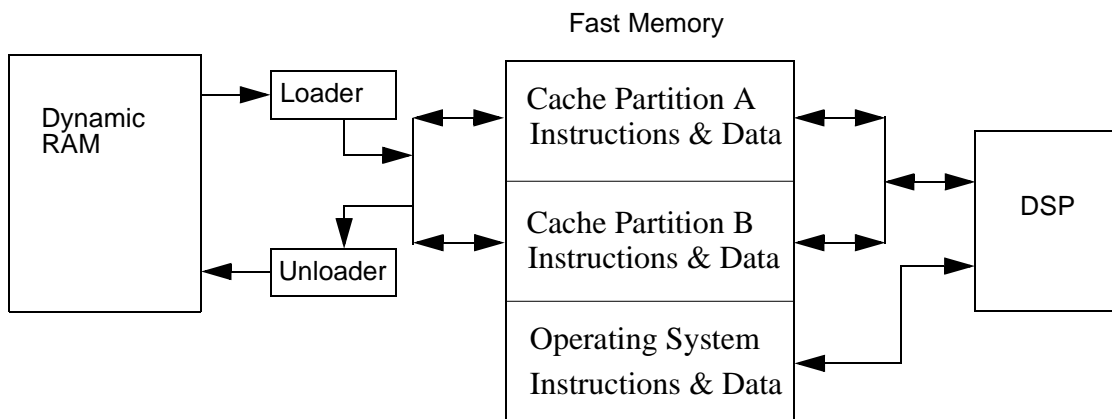


Figure 6.6: A Real-Time Caching System

Figure 6.6 shows how the cache actually fits into the overall system. There are two cache partitions that can be used for tasks, and one locked partition that is always used for the operating system. The loader and unloader have access to both cache partitions, as does the DSP. However, for the duration of any given quantization block, the loader and unloader can only access one of the partitions, and the DSP can only access the other partition. By restricting access to different partitions, the loading hardware and the processor

cannot interfere with each other in the cache. The operating system has control of which partition is available to the DSP and which one is available to the loader and the unloader. The loader and unloader can be the same or separate hardware, depending on the specifics of a given implementation. In the IBM Mwave system, the instructions are immutable so they do not need to be copied back to memory and could be overwritten while the data is being unloaded. It is also possible either to use one cache for both instructions and data, or to have separate caches, one for instructions and one for data. The IBM Mwave system would probably need two caches, as instruction and data accesses by the processor can occur simultaneously.

The operating system is loaded into its partition once, and then is completely self contained. The instructions are always available for use and there must be enough room in the partition to store all of the possible data that the operating system may need. That way, the operating system never has to wait for the cache to be loaded.

6.6 DRAM Refresh

In a real-time system, special consideration must be given to DRAM refresh, as explained in Section 2.1.3 on page 21. EQ further alters the notion of how refresh can be performed. In EQ, control of the memory system is not guaranteed for the entire length of execution of the task. However, every task is in control of the memory system for at least one load time and one unload time in each frame. Therefore, the simplest solution is to schedule as many refresh tasks as it takes to guarantee control of the memory system for long enough to refresh the entire DRAM. These tasks should be as short as possible, which is one quantization block, since they will not be doing any useful computation. By scheduling these tasks at the period that refresh needs to occur, and refreshing the DRAM instead of loading the cache during these tasks' load and unload times, the DRAM is guaranteed to be refreshed correctly without interfering with any other task's execution.

6.7 Memory Bandwidth

The available bandwidth between the cache and main memory is extremely important in EQ, as it determines the minimum size of a quantization block. For given cache sizes, the quantization block must be long enough to accommodate unloading and loading the entire data cache, and loading the entire instruction cache, assuming instructions are immutable. Obviously, using memory with higher bandwidth is desirable, as it decreases the length of a quantization block, and therefore decreases the loss of available resources for real-time computation because of EQ.

In a system that only runs tasks that entirely fit in the cache, each task's data will consist of a small number of large blocks of memory and it is likely that each task's instructions will be located in one contiguous block. Therefore, the memory system should be designed to transfer blocks of data at as high a rate as possible, which means more emphasis should be placed on peak transfer rate than on the latency that is incurred when a transfer is started. Memory interleaving and other techniques designed to increase peak transfer rate, as described in Chapter 4, can be used to maximize the transfer rate of the memory system. Memory latency becomes more important in systems in which all tasks do not entirely fit into the cache. In these systems, the quantization block size is no longer the most important design feature, and main memory is not always accessed in such a way that allows the use of the memory system's peak bandwidth.

For example, consider a system that has a separate instruction and data cache, and instructions and data are held in separate DRAM memory spaces. The example system has a 20 ns clock cycle, and assume that the DRAM's cycle time is 80 ns. In the DRAM's 80 ns cycle time, assume that an entire word can be retrieved, where a word is 2 bytes. This can be achieved either by using DRAM that is 16 bits wide, or by using several DRAMs to obtain 16 bits per access. Also, assume that the pipeline allows a throughput of one

instruction per cycle, which means that the processor can execute 50 million instructions per second (MIPS) if there are no constraints on the processor by the memory system. Simply introducing the 80 ns DRAMs without a cache, or with a nondeterministic cache, reduces the available real-time resources to 12.5 MIPS. This occurs because an instruction must be fetched each cycle, so the clock cycle of the machine has effectively dropped to 80 ns. Data accesses can occur at the same time as instruction accesses, so they do not degrade the machine's performance any further.

Assume that all instruction words are immutable, so they never have to be written back to memory, they can just be overwritten in the cache. By interleaving 8 of the 16 bit wide DRAMs, the memory system can transfer sequential blocks of data at a rate of 100 Mwords per second. The minimum length of a quantization block in this system is determined by the amount of time necessary to load the instruction cache, and the amount of time necessary to unload and load the data cache. If I is the size of the instruction cache, in kwords, then for the given system, the amount of time needed to load the instruction cache is:

$$\frac{I\text{kwords}}{100\text{Mwords/s}} = I \times 10\mu\text{s} \quad (6.1)$$

And if D is the size of the data cache, in kwords, then the amount of time needed to unload and load the data cache is:

$$2 \times \frac{D\text{kwords}}{100\text{Mwords/s}} = D \times 20\mu\text{s} \quad (6.2)$$

Once the time necessary to load the instruction cache and unload and load the data cache are known, the length of time necessary for each quantization block is just the maximum of the instruction cache and data cache transfer times, since they both need to be transferred, and the transfers can occur simultaneously.

Instruction Cache Size	Inst. Cache Transfer Time	Data Cache Size	Data Cache Transfer Time	Quantization Block Size
2k words	20 μ s	2k words	40 μ s	40 μ s
4k words	40 μ s	2k words	40 μ s	40 μ s
6k words	60 μ s	2k words	40 μ s	60 μ s
2k words	20 μ s	4k words	80 μ s	80 μ s

Table 6.1: Quantization Block Size for Various Cache Configurations

Table 6.1 shows the amount of time necessary in each quantization block for a few different cache configurations. The transfer times for the instruction and data caches were calculated using Equations 6.1 and 6.2. As can be seen from the table, unless the instruction cache is more than twice the size of the data cache, the size of a quantization block is dependant only on the time it takes to unload and reload the data cache.

At first it may seem that between 2 and 4 kwords of cache space is not enough to accommodate any reasonable task. However, digital signal processing tasks are typically quite small, as they must execute frequently and they must be quick. The typical digital signal processing task will have an execution frame on the order of milliseconds, and it will process some small number of samples each frame. When compared with the total memory size of the IBM Mwave DSP of 32 kwords of instructions and 32 kwords of data, it is clearer that a few kwords of cache can be quite sufficient to hold an entire task.

For the cache configurations in Table 6.1 the quantization block time ranges from 40 μ s to 80 μ s. The next step is to determine what this means in terms of availability of real-time resources. The fastest frame allowed in the IBM Mwave system has the frequency 44.1/32 kHz, which corresponds to 32 samples of compact disc quality sound. This is actually a relatively fast frame, and it is not likely that a real-time system will allow the use of frames faster than this. In the following calculations, the fastest allowable frame in the IBM Mwave system will be used as the fastest execution frame in the system.

The maximum real-time utilization of a system using EQ is $\frac{T-3Q}{T}$, as shown in Section 6.3.2 on page 64, where T is the shortest execution frame period, and Q is the length of a quantization block. Therefore, for systems with Q equal to 40 μ s, the maximum real-time utilization is:

$$\frac{(32/44.1\text{kHz})-(3 \times 40\mu\text{s})}{32/44.1\text{kHz}} = \frac{725.6\mu\text{s} - 120\mu\text{s}}{725.6\mu\text{s}} = 0.835$$

So, 83.5% of the available time can be used for real-time computation. On a processor with 50 MIPS, this translates to 41.75 MIPS available for real-time use. If Q is 80 μ s instead, then the maximum real-time utilization is:

$$\frac{(32/44.1\text{kHz})-(3 \times 80\mu\text{s})}{32/44.1\text{kHz}} = \frac{725.6\mu\text{s} - 240\mu\text{s}}{725.6\mu\text{s}} = 0.669$$

So, on a machine that can sustain 50 MIPS when the memory system can keep up, using EQ allows 66.9% of the available MIPS to be used, which is 33.45 MIPS.

Clearly, page mode DRAM can be effectively used in this kind of a system. Since all of the transfers to and from the memory in this system are sequential blocks, the latency to bring up a page is quickly outweighed by the increased transfer rate of data within the page. If, for instance, the same DRAM as in the above examples has a page mode cycle time of 40 ns, then only 4 DRAMs, instead of 8, would need to be interleaved to achieve roughly the same performance, and using more would significantly increase the amount of time available for real-time computation.

There is a lot of flexibility in how systems can be configured, and the above examples show how the available real-time MIPS can be calculated for a few configurations. In practice, however, these numbers are probably slightly optimistic, as there is no time allowed for any overhead associated with the memory system. In a real design, these factors must be added into the calculation of the quantization block size.

6.8 Shared Memory

Since a system using EQ must have two caches, there is a cache coherency problem. If the same address is cached in both caches, then there is potential for the use of stale data and for data to be updated out of order. This is similar to the problem that occurs in multiprocessor systems, where a single piece of data may be cached for multiple processors. When a write occurs, cache coherency methods generally involve invalidating or updating all of the other cached copies of the data. Since the number of possible invalidations or updates is unpredictable, it is not feasible to allow that to happen in a real-time system. Any action that is external to a particular task that can affect that task leads to nondeterministic behavior, unless an upper bound can be placed on the amount of disruption. Therefore, a method must be devised that does not involve an arbitrary number of invalidations or updates to cache lines that still maintains cache coherency.

Furthermore, checking all writes to see if they affect the other cache partition is time consuming. The partition that is connected to the DSP is heavily used, as most DSP algorithms are data intensive and therefore have a very high percentage of data accesses in the instruction mix. Similarly, if a new task is being loaded into the other partition, then the majority of the time in the current quantization block is needed to transfer data between the memory and the cache. Snooping on the other partition's activities would slow down data transfer rates, thereby increasing the quantization block size, which adversely affects the system's performance.

The problem of shared memory is eliminated entirely in a single cache system where all tasks share the same address space. Since virtual memory is cumbersome and unnecessary for small real-time DSP systems, all tasks inherently share the same address space. This means that there can be no naming confusion, as all addresses are physical addresses.

The real problem is to make execution quantization work with a single cache, rather than with two caches or two cache partitions.

6.8.1 Using a Separate Shared Cache

An alternative to making EQ work with a single cache is to separate shared memory from private memory. Effectively, this would require three caches or cache partitions instead of two. Two private caches would work as described previously, and a third shared cache would hold all shared memory that needs to be cached. That way, the two caches required by EQ only contain private data, allowing them to function without regard for cache coherence. The shared cache can then be used to maintain coherence by not allowing the same address to be cached multiple times. Unfortunately, by having a shared cache that is common to both of the EQ caches, it may no longer be possible to guarantee that the data for the two tasks will not interfere with each other. Different tasks with different shared cache footprints can reside in the shared cache together at different times, making it almost impossible to predict if data will overlap in the shared cache. This means that it will be difficult, if not impossible, to determine what data will be available to the task in the shared cache. By making the shared cache fully associative, conflicts can be avoided, as tasks will not compete for cache sets. A fully associative cache, however, requires a large number of comparators to determine if there is a cache hit, this can become quite expensive in terms of chip area for all but the smallest of caches. However, the shared cache must be large enough so that it can hold all of the shared data for any two tasks that are running on the system. The third factor to be considered is the mixture of shared and private memory in different tasks. It is quite likely that there will be some tasks that primarily communicate with other tasks and have a large amount of shared memory to deal with, as well as tasks that have hardly any shared memory at all. It becomes quite difficult to trade-off the size of the shared cache and the size of the private caches. Worst of all, there

must be enough time in a quantization block to unload and load both the entire private cache and as much of the shared cache that can be used by any one task.

It also may be difficult to know which locations of memory are shared. The memory locations in the DRAM that are shared would somehow have to be marked, so that the loader could check to see if the location should go into the shared cache or the private cache. This would increase the memory requirements and it would also increase the transfer time, as this would have to be checked for every transfer that the loader makes. Also, the most insidious form of shared memory is not really shared memory at all, but rather it is a shared cache line. Potentially, different addresses that fall in the same cache line can be loaded with data from different tasks. In this situation the memory is not really shared, but since the data belongs to the same cache line, the data could potentially be cached in both private caches unless it is somehow marked. It may be extremely difficult to identify the regions of memory that are shared cache lines. Even if all of these problems can be overcome, the fatal flaw with the shared cache is that it would be extremely difficult to determine which cache lines belong to which task, and which lines belong to both for that matter, in order for the loader and unloader to know what to do. The unloader would not easily be able to determine which cache lines need to be written back to memory, and it may not be clear which cache lines the loader can overwrite with new data.

6.8.2 Virtual Cache Partitioning

Returning to the single cache idea, a viable system could use a single cache, as long as it can be guaranteed that the data from the two tasks will not interfere with each other. By using virtual partitions, a cache with a single partition can be made to function as if it had two partitions. The idea behind virtual partitioning is to use a normal set associative cache, but to allow only half of each set to be used by either partition. The key is that the half of the set that is used can change and it can be different for every set. In order to enforce the

virtual partitioning, it is necessary to be able to identify which partition any particular cache line belongs to. By using two valid bits per cache line, it can be determined if the cache line belongs to both partitions, only one partition, or neither partition. In order to guarantee that each partition is no larger than half the cache, a given partition may only have half of its valid bits set in any given cache set. That way, it is impossible for either partition to grow beyond half of the cache size, and each partition acts exactly as if it were an independent cache with half of the set associativity, and half the size, of the actual cache.

Virtual partitioning allows shared memory to reside in the cache without conflict. Assume that a cache line that contains shared memory is in the cache and has its valid bit set for one of the partitions. Then the same cache line is loaded into the other partition. When the loader hardware tries to write the line into the cache, it will see that the line already resides in the cache, and instead of loading the line into the cache, it will just set that partition's valid bit so that both valid bits are set for that particular cache line. When one of the partitions gives up that cache line in the future, it will just clear its valid bit, and the data will remain cached and valid in the other partition. Furthermore, no cache line can be overwritten if the valid bit for the other partition is set. That way, the partitions will not invalidate each other's data.

If, in a particular cache set, all of the data is shared between both partitions, then half of the lines in the cache set are invalid, and it would seem that it should be possible to use these lines. A given partition cannot be expanded into the empty lines without invalidating one of its other lines. However, once data is moved into one of the empty lines, the invalidated line is still valid in the other partition. This restricts each partition to continue to use only half of the lines in the set. Sharing cache lines does not allow a partition's size to increase, even though the resources are available. This is necessary to achieve determinis-

tic caching. If a partition were allowed to grow beyond its given size simply because there are available resources at the time, then it may not fit back into the cache after the task is preempted and resumed, as no assumptions can be made about the data that will be in the other partition in the future.

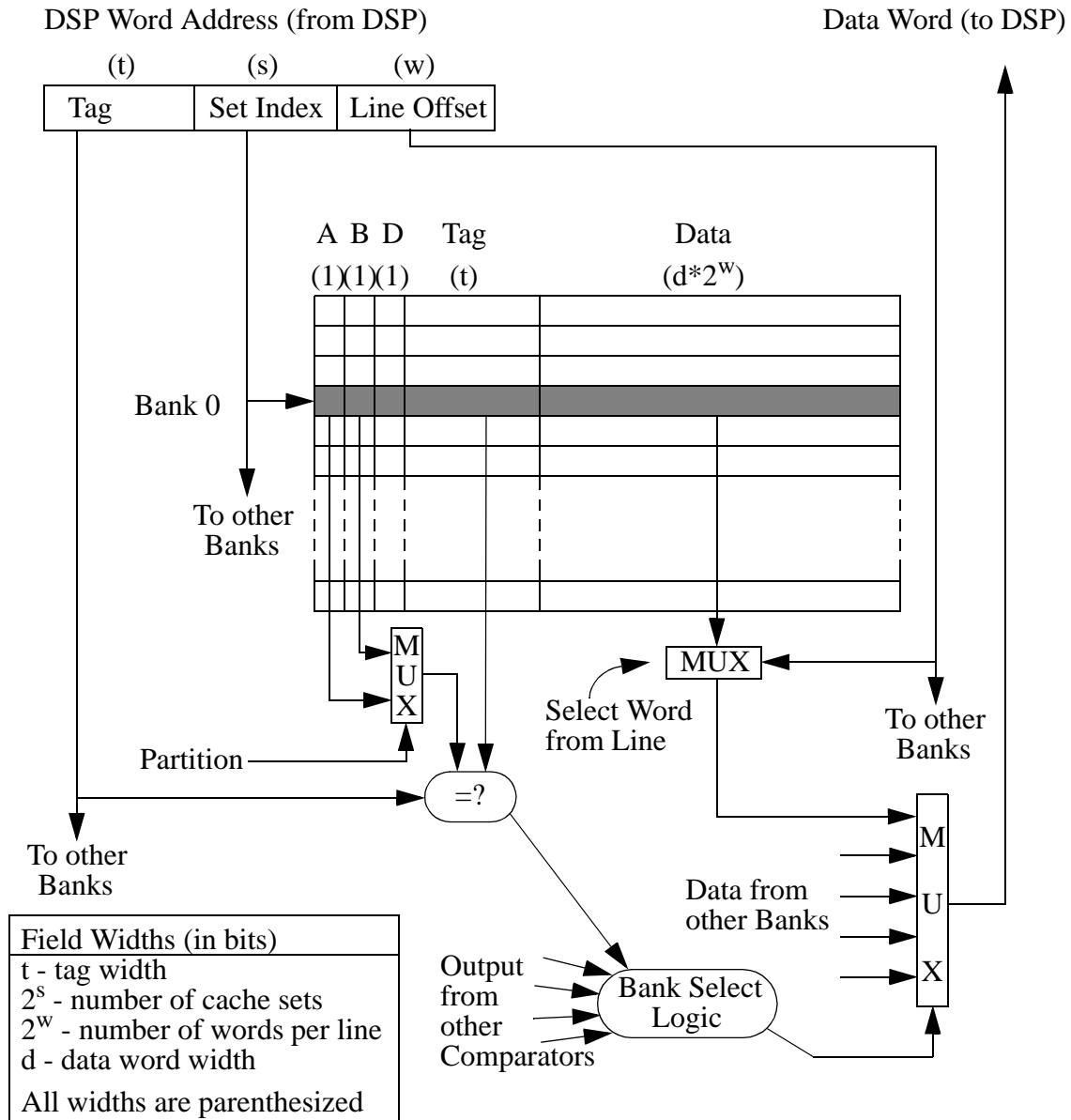


Figure 6.7: A Data Read in a Cache with Virtual Partitions

Figure 6.7 shows how a data read would take place in a cache with virtual partitions. In the figure, the partitions are labelled with an A and a B bit. If the A bit is set, then the

cache line is valid for Partition A, and similarly, if the B bit is set, then the cache line is valid for Partition B. Note that these are exclusive conditions, whether the A bit is set or cleared has no bearing on whether the cache line is valid for Partition B. The third status bit is the D bit, which is the dirty bit, indicating that a write has occurred to this cache line, and it will need to be written back to memory.

The dirty bit is necessary because any cache that is used in conjunction with EQ is probably a write back cache, rather than a write through cache. This is because the memory system is frequently tied up with moving data between the memory and the cache partition that is not connected to the DSP, so if there is a cache hit on a write, it is much more expedient to update the cache and continue execution, rather than wait for access to the memory.

In general, the actions that occur on a data read are almost identical to those that would occur in any set associative cache. First, the set index is taken from the DSP address, and used to index the appropriate set in each bank. For a k -way set associative cache, there are k banks. The tag from this set in each bank is then compared against the tag from the DSP address. The comparisons occur simultaneously, in order to speed up the cache access. The partition variable, which controls which partition the read is accessing, then chooses to latch the comparator output with either the A or the B bit. As the comparisons are taking place, the data from the chosen set for each bank is also accessed. Either the desired word is accessed singly out of the array, or the correct word is selected from the cache line immediately after it is accessed. The results from the comparator for each bank are then run through some logic, to determine which line of data, if any, matches the given address. If there is a cache hit, the appropriate data word is then passed through a multiplexer and sent to the DSP. If there is a miss, which will never occur if the task fits entirely into the cache, then the appropriate cache line must be loaded with the correct data. A write is very

similar. If there is a cache hit on a write, then the data is written to the appropriate word of the cache line, and the D bit is set, so that the line will be transferred back to memory when the task is preempted or completes execution.

At the same time that the cache is servicing DSP memory accesses in one partition, the cache loader must be able to load or unload the other partition, in order to support EQ. This means that in one machine cycle, the cache must be accessed twice, once in each partition. Since there really are no partitions, this means that the cache must be able to support two accesses each machine cycle. This is quite possible, if the tags and data are kept in separate arrays for each bank. That way, access to the tag arrays and access to the data arrays can be split apart, forming a two cycle access period. If each of these cycles can be made to take exactly half of the machine cycle, then the cache can be accessed twice each machine cycle. An effective way to split up the hardware is to group the A and B bits with the tags from each bank and allow those resources to be accessed each half cycle. This leaves the D bits and the data from each bank to be grouped together, and these can also be accessed each half cycle. The comparators and other multiplexing logic can be used concurrently with the data access each half cycle. Alternatively, a dual ported cache could be used, if the resources are available.

Table 6.2 shows the resources needed for the four different types of cache accesses. DSP reads and writes must only take one machine cycle, as otherwise the cache will stall the DSP. Cache loading and unloading are not as restricted, and can span several machine cycles, if need be, but the transfer of any one cache line must be an atomic action, in that the cache loader must be able to load or unload an entire cache line in consecutive machine cycles. For a DSP access, it is clear that the A and B bits and the tag from the appropriate set of each bank are needed first. So, for the first half of the machine cycle, the DSP address will be used to obtain this data. The second half of the machine cycle is then

DSP Cycle	DSP Read	DSP Write	Cache Load	Cache Unload
1a	Access Tags & A, B Bits	Access Tags & A, B Bits		
1b	Use Comparators & Access Data	Use Comparators, Write Data & Set D Bit	Access Tags & A, B Bits	Access Tags & A, B Bits
2a	Access Tags & A, B Bits	Access Tags & A, B Bits	Use Comparators & Write Data	Use Comparators & Access Data and D Bit
2b	Use Comparators & Access Data	Use Comparators, Write Data & Set D Bit	Set A or B Bit	Clear A or B Bit
3a	Access Tags & A, B Bits	Access Tags & A, B Bits	Write Data	Access Data & Clear D Bit
3b	Use Comparators & Access Data	Use Comparators, Write Data & Set D Bit	-----	-----

Notes:

1. For each DSP cycle in the table (1, 2, and 3), cycle *Na* is the first half of DSP cycle *N*, and similarly cycle *Nb* is the second half of the same DSP Cycle.

2. For each a/b cycle, either a DSP Read or Write action can occur, and for each b/a cycle, either a Cache Load or Unload action can occur.

3. A DSP Read or Write action only takes one a/b cycle - 3 Read/Write actions are pictured in the table, occurring one each in cycles 1, 2, and 3.

4. A Cache Load or Unload action takes multiple b/a cycles, depending on the length of the cache line compared with the width of the bus - 1 Load/Unload action is pictured in the table, starting at cycle 1b, and continuing on past the end of the table until the entire cache line is read or written (only the 'a' half cycles will be used, as there is no further need to access the tags or the A and B bits).

Table 6.2: Cache Actions For DSP Half Cycles

spent simultaneously accessing the data from the appropriate set of each bank and performing the tag comparison for each bank. The tag comparisons are latched with the valid bits that correspond to the partition the DSP is accessing. In the case of a DSP read, the output of the comparators are then used to select the appropriate data to send back to the

processor. For a DSP write action, the comparators are used to select which bank will be updated with the new data, and the D bit for that bank's selected cache line will be set.

The loader and the unloader are able access the cache similarly. However, a complete machine cycle is skewed by half a cycle. In the second half of a machine cycle, the loader or unloader can access the tags and the A and B bits from each bank. Then in the first half of the next machine cycle, the loader or unloader can use the comparators and the data array either to load or to unload the appropriate data. During an unload access, the D bit will also be accessed during this half cycle to determine if the data needs to be written back to memory. For an unload, the comparators are not used. Instead, the partition bits are used to select the next valid data to unload. For a load, the comparators are used on the other partition's valid cache lines in the set to determine if the data will be shared. In the second half of this machine cycle, the A or B bit is set or cleared for a load or unload respectively. If a cache line is larger than the available bus width to the memory, then the first half of subsequent machine cycles will be used to continue to transfer the cache line to or from memory.

The nature of EQ guarantees that the DSP will never access data in the partition that the cache loader is working on, so there is no problem with changing the valid and dirty bits before the entire transfer is complete. However, since the valid and dirty bits are associated with the entire cache line, and not just the block that can be transferred, the cache line transfer cannot be interrupted. For example, if the DSP misses in the cache, then the processor must be stalled until the cache loader has completely finished transferring a cache line. Therefore, as long as cache line transfers are always atomic actions, and each partition is restricted to use only half of each cache set, virtual partitioning allows deterministic caching of shared memory.

Chapter 7

Expanding Execution Quantization

7.1 Task Timing

Execution quantization is a useful method for allowing a cache to be used in a real-time system where each task can fit entirely into one cache partition. However, it is unlikely that *every* task will fit into a cache partition, and even if they all do, restricting task sizes severely inhibits future software development. Execution quantization needs to be extended to allow real-time tasks that are larger than the available cache space. Upon reexamination, it becomes clear that the most important feature of EQ is that it preserves the cache for each task across preemptions, since preemptions are the major obstacle in a real-time caching system. The only necessary modification to EQ is to allow the cache contents to change with program execution, rather than requiring them to be fixed.

As long as the loader used in EQ is able to restore the cache contents to their exact state upon a task's resumption after a preemption, then EQ is still useful, even if the cache contents are allowed to change during a task's execution. The real problem, however, lies in determining how to service a cache miss. Whenever a task is running, it is always possible that the cache loader is simultaneously loading the other cache partition. It is impossible to predict whether or not the cache loader will be active in any given quantization block, so it must be assumed that the DRAM and the cache loader are always being used to load the other cache partition. Therefore, these resources are not available to reload a cache line for the DSP.

This problem can be alleviated by increasing the quantization block size. If the length of a quantization block is longer than the time it takes to unload and reload a cache partition, then it can be guaranteed that there will be time in every quantization block when the

DRAM is not being used by the loader. This time can then effectively be used to service cache misses for the DSP. This introduces a very important trade-off in the size of the quantization block. If the quantization block is made extremely large, so as to allow plenty of time to service cache misses for the DSP, then the penalty in real-time resources to implement EQ becomes prohibitive. Conversely, if the quantization block is made just slightly longer than the minimum required to unload and load a cache partition, then the cache miss penalty will be so large that each task's worst case execution time will grow to unacceptable levels.

The overhead required by EQ can easily be computed from the quantization block size, as was shown in Section 6.7 on page 70. However, the effect of the quantization block size on each task's worst case execution time is more subtle. Even determining the cache miss penalty once given the quantization block size is not completely straightforward.

In order to determine the worst case execution time of each task, a timing tool is obviously necessary. A general overview of how such a tool would work is given in Section 3.2.5 on page 38. The timing tool must be able to determine when a cache miss will occur. More likely it needs to pinpoint memory accesses which *may* miss in the cache, as it is probably extremely prohibitive to search all possible program paths to determine the exact state of the cache at all times. Presumably, all instruction accesses should be completely predictable for any given flow of control. However, it may be difficult to decipher an indirect data reference prior to run time. Of course, the more accurate the timing tool, the tighter the worst case execution time will be. Even if the timing tool can only identify memory accesses that may miss, any memory access that it determines will hit in the cache, *must* hit in the cache in order to guarantee that the task will not execute for longer than its reported worst case execution time.

Determining a task's worst case execution time is a two part process. First, the cache miss penalty must be determined, and second, the timing tool uses this miss penalty to determine the task's worst case execution time. The worst case cache miss penalty is the longest amount of time that the DSP could be forced to wait, after missing in the cache, for the required data to be available. Obviously, the cache miss penalty is dependant on the quantization block size, cache loading requirements, and memory bandwidth. Once the worst case cache miss penalty is known, then the timing tool can determine the worst case path through the task's code. In finding the worst case path, each memory access that will miss in the cache is presumed to stall the DSP for the worst case cache miss penalty. The timing tool will return an upper bound for the actual worst case execution time of the task, since many cache misses will be serviced in less than the worst case cache miss penalty. As not all cache misses incur the worst case miss penalty, a more complicated timing tool could be designed to use a more accurate miss penalty for each cache miss, rather than always using the worst case penalty. This would lead to a more accurate timing tool, but it is more difficult, and may not be worth the effort involved.

7.1.1 Determining the Cache Miss Penalty

The cache miss penalty is dependant on how time is allocated in each quantization block between loading and unloading one cache partition and servicing cache misses for the DSP in the other partition. To achieve deterministic behavior, the loader and unloader should have control of the memory at regular intervals. In that case, the worst case miss penalty can be easily determined. By evenly distributing the required cache line transfers, the available cache miss service time is also evenly distributed.

The smallest unit that should be transferred is a cache line. Therefore, blocks of time that are allocated to unloading or loading the cache should be an integral multiple of the amount of time necessary to transfer one cache line. Similarly, time that is set aside for the

service of cache misses should be an even integral multiple of the amount of time necessary to transfer one cache line, as a cache miss requires unloading one cache line, if necessary, and then loading the new cache line. Figure 7.1 shows how the time in a quantization block could be evenly distributed among loading and unloading a cache partition, and servicing cache misses for the other partition. Once this distribution is established, then it is an easy task to determine the longest possible time that could be spent waiting for a memory access by the DSP to complete.

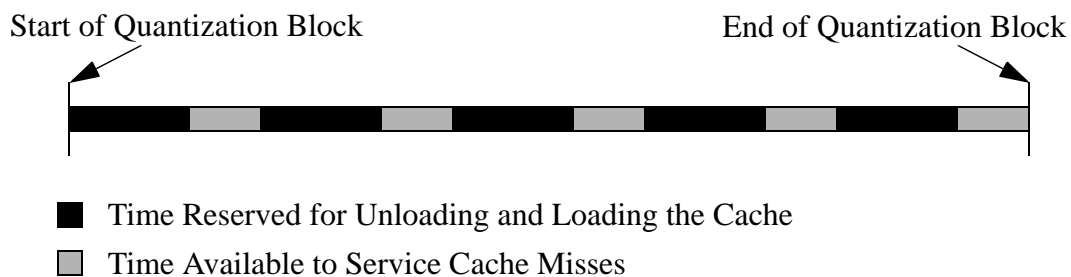


Figure 7.1: Distribution of Cache Unload/Load Time in a Quantization Block

The worst time that a cache miss could occur is immediately after the start of one of the blocks scheduled for loading or unloading the cache. In that case, the processor must wait for the entire length of the load or unload block and then for the time that it takes to service the actual cache miss. If a cache miss occurs right before the start of an unload or load block, then the miss should be serviced and the load or unload block pushed back in time. This will not lead to nondeterministic behavior, as this will just push the load block into the next period of time that was left available to service cache misses. Therefore, the worst possible cache miss penalty is the length of one load or unload block plus the time it takes to service a cache miss. Once this number is calculated, it can be used as a worst case cache miss penalty by the timing tool, without needing to worry about the workings of the system any further. A more sophisticated timing tool could make use of the fact that the load and unload blocks are fixed in time relative to a quantization block. If the code's

location is known in the quantization block, the actual cache miss penalty could be calculated, instead of relying on the worst case miss penalty.

By distributing the cache load and unload blocks, the advantages of transferring large blocks of data may be lost. Since new cache lines can be loaded from almost anywhere on a cache miss, the only guarantee is that each cache line will be a sequential block of data. As the cache line is now the unit of transfer between the memory and the cache, the size of a cache line, and the resulting cache line transfer time, is a major factor in determining the quantization block size.

7.1.2 Difficulties in Building a Timing Tool

Since it is impossible to predict whether or not the loader or unloader will be active during any given quantization block, any timing tool must act as if they will be active during *every* quantization block. Otherwise, the timing tool will report worst case execution times that are dependant on when the cache loader will and will not be active. Since this will vary from frame to frame, it is extremely likely that the timing information will be incorrect for some execution frames, and the task may exceed its worst case execution time in those frames, which is unacceptable in a real-time system.

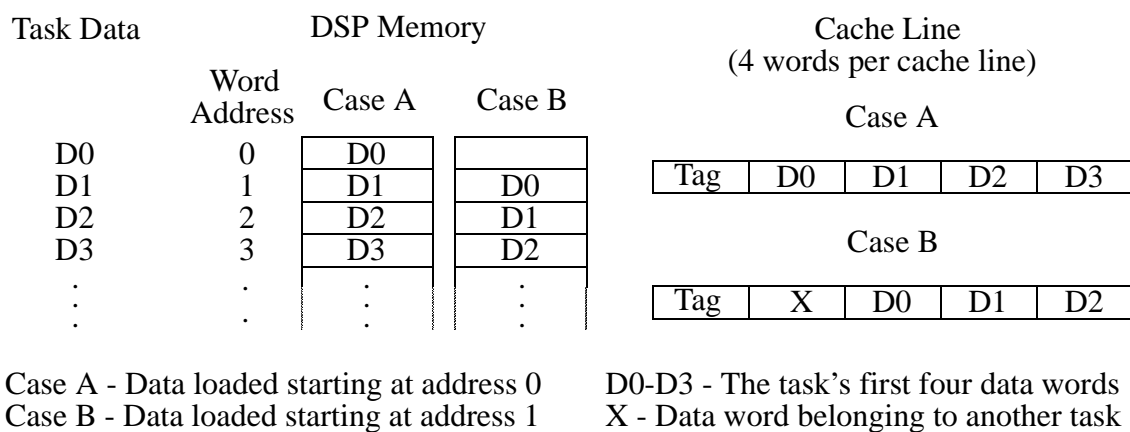


Figure 7.2: The Affect of Load Placement on Cache Line Contents

The placement of a task's code and data in memory also affects the task's timing, since it affects which data will be grouped together in particular cache lines. Figure 7.2 illustrates this point. Consider a cache with four words per line. If a task is loaded starting at address zero, which corresponds to case A in the figure, then the first four words will always be grouped together in a cache line, as will the next four words, and so on. However, if that same task is loaded starting at address one, which corresponds to case B in the figure, then the first three words will always be together in a cache line, as will the next four, and so on. Clearly, these load placements will lead to different timing situations, as a miss on the second word (D1 in the figure) in the first case will load the cache with words zero through three (D0-D3), whereas a miss on the second word (D1) in the second case will load the cache with words zero through two (D0-D2), plus another word that does not belong to this particular task at all (X). If the next memory access is to the fourth word (D3), then depending on where the data was loaded, this access could either hit or miss in the cache, since it is already in the cache in case A, but it is not in case B.

The timing tool must account for this situation. One possible solution is for the timing tool to require the loader to place code and data at certain offsets. This would allow the timing tool to time each task for only one particular load placement, and a sophisticated timing tool could choose the optimal load placement. Another solution would be for the timing tool to time the code for all possible load placements, allowing the program loader to place the task anywhere. This might take a considerable amount of work, but probably not much more than a timing tool that finds the best load placement. The number of placements that need to be evaluated is the same as the number of words in each cache line. This follows from the fact that in a cache with four words per line, the same words will share a cache line for any placement that is equivalent modulo four.

As discussed in Section 3.2.5 on page 38, a timing tool will need to construct a graph representing the flow of control of a task. The graph can normally be constructed from the task's code, but the addition of a cache further complicates the timing process by adding another state variable into the graph. The addition of a cache requires knowledge of the cache contents at all points in the graph, since it is impossible to evaluate edge weights without knowing whether or not particular memory accesses will hit or miss in the cache. This also complicates the construction of the graph in that a particular edge may have different weights depending on the particular path that was taken to get to that edge. This ambiguity can be resolved by turning the graph into a tree where each path through the tree is a unique path through the code. Many of the nodes in the tree would represent the same sections of code, however, the edge weights would accurately reflect the time it would take to execute that code with the given cache state. Clearly, turning the graph into a tree increases the work that must be done to determine the longest path through the graph, but it is still possible. Some algorithms are presented in [2] to find the shortest path from a source node to all other nodes in a graph. These algorithms can easily be modified to find the longest path from a source node to all other nodes, and then the distance to all the leaf nodes can be compared to determine the worst case execution time.

7.1.3 Timing Tool Requirements

There are several requirements that any timing tool must adhere to. The primary input to the timing tool must be the task's code and data. The timing tool must also be aware of the specific cache configuration and quantization block size of the system for which the task is to be timed. The tool may also need some input from the programmer if there are ambiguities in the code. The tool will need the task's code to construct the program graph, and it may also need the task's data if the data influences the control flow, or if it influences which memory locations are accessed. Programmer input will be required to ascer-

tain loop bounds, branch targets, and possibly data access targets. The programmer input could be in many forms, such as the programmer answering questions posed by the timing tool, or placing comments or directives in the code to eliminate ambiguities. Programmer input can be completely eliminated if all code is forced to be structured in such a manner that the timing tool does not encounter any ambiguities.

The output of the timing tool must be the worst case execution time for the task. This number must be an upper bound for the actual worst case execution time, but it does not necessarily have to be exact. Obviously, the tighter the bound is, the better, as this number will be used to determine the amount of real-time resources the task needs, and the difference between the actual and reported execution time will be wasted. However, it may not be possible to create a timing tool that is so precise that it can accurately report any task's worst case execution time. As long as the worst case execution time that the tool reports is always an upper bound, the task will always be able to run deterministically on the system. This worst case execution time must also account for load placement. It can either report the worst case execution time for every placement, or for one placement and force the loader to use that placement.

7.2 On Chip DRAM Example

The Mwave DSP is rather small, as are most DSPs. This coupled with the fact that there is a very small amount of memory allows the DSP, cache, and memory to be placed on one chip. This is extremely advantageous, as the access times and transfer bandwidths of the DRAM are much better when the data does not have to leave the chip. This means that the DRAM and the cache can be connected with a bus that may be on the order of 64 bytes, as opposed to 2 or 3 bytes if the DRAM were external. Placing the DSP, cache and memory all on the same chip probably will limit the cache size, as chip area will quickly become scarce. However, the cache would probably be quite small anyway, as the memory

requirements of DSPs are typically quite low.

By being able to move one large block of data in the same time or less than it would take to move a couple of bytes from off chip, a tremendous advantage is gained. The minimum quantization block size decreases dramatically. Also, the time to service a cache miss is reduced. These factors make the use of EQ even more attractive. Once cache misses are allowed, the fundamental transfer unit is the cache line. For optimal performance, it makes sense to make the cache line size an integral multiple of the bus width. That way, data can efficiently be moved to and from the cache without having bus bandwidth that goes unused.

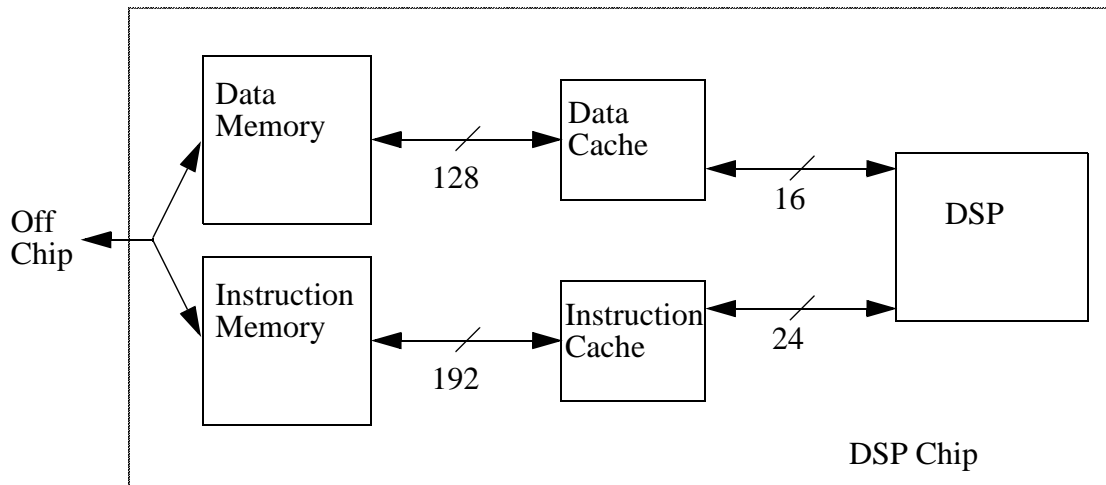


Figure 7.3: On Chip DRAM System Configuration

In the Mwave DSP system, data words are 2 bytes and instruction words are 3 bytes. In order to achieve uniformity, although that is not necessary, consider buses from the DRAM to the caches that can each transfer 8 words at a time. This means the bus from the data memory to the data cache will be 128 bits wide, and the bus from the instruction memory to the instruction cache will be 192 bits wide. This configuration is shown in Figure 7.3.

Furthering this example, assume that the clock cycle of the DSP is 20 ns. Since the DRAM is on the same chip as the caches, memory latencies and transfer times will be greatly reduced. For this example, assume that when a memory access is initiated there is a one cycle latency, and then after that, 8 words, which is the width of the bus, can be transferred each consecutive cycle. If enough data is accessed to cross a page boundary in the DRAM, then the initial one cycle latency is incurred again, but for this example, assume that all accesses are properly aligned and small enough so that any continuous access will only be to one page. If the page size is an integral multiple of the cache line size and the bus width, then no cache line will ever contain data that crosses page boundaries. Furthermore, if the cache line size is also an integral multiple of the bus width, then cache lines can be transferred without wasted bandwidth. These relations will always hold if the page size is larger than a cache line, a cache line is larger than the bus width, and all three are powers of two.

When tasks can be larger than the cache size, there is a new decision to be made in EQ. That decision is whether or not the cache should be loaded before execution each frame. It would seem that performance could be increased if the cache were loaded, but that may not be the case. The problem is, that it may not be possible to determine the optimal data (or instructions) to place in the cache originally. If the data and instructions that are preloaded are of little use, then the effort of preloading the cache was a waste. It may seem that nothing is lost by doing so, since EQ is designed to do exactly that. However, in order to do this, the amount of tag memory must be doubled. In EQ, the tags of all the data that belongs in the cache for each task must be stored somewhere. This is a record of the data that the cache loader should load when the operating system tells the cache loader that a task should be reloaded into the cache. If there must also be a record of the data that should be preloaded into the cache before a task's execution in each frame, then this tag

memory doubles. Continuing the current example, assume that the task is not preloaded into the cache, thereby reducing the necessary amount of tag memory. The overhead of EQ is now only two quantization blocks out of the fastest frame rate, as time does not need to be reserved to load the cache before tasks start to execute in each frame. If there is available memory to hold the extra tags, it is likely that the system's performance would be improved by preloading the tasks before their execution. This would certainly be a tremendous advantage for all tasks that can fit entirely into the cache, because they would never have to wait for the memory system to service a cache miss.

Line Size	Transfer Time for Each Line	Lines in a 1 k Cache	Transfer Time for the Cache
8 words	2 cycles	128	256 cycles
16 words	3 cycles	64	192 cycles
32 words	5 cycles	32	160 cycles
64 words	9 cycles	16	144 cycles
128 words	17 cycles	8	136 cycles

Table 7.1: Transfer Time for a 1 kword Cache

With the above specification of the DRAM on a machine with a 20 ns cycle time, Table 7.1 shows the number of cycles needed to transfer a 1 kword cache for various cache line sizes. The table can be used to determine the number of cycles necessary to transfer a cache of any size that is a multiple of 1 kword with one of the given line sizes. For instance, a cache of size n kwords and a line size of 32 words would take $160n$ cycles to transfer.

Using Table 7.1, the minimum quantization block size for different cache configurations can be calculated. Table 7.2 shows the minimum quantization block size for a small number of cache configurations. When looking at Table 7.2, it is important to remember that the data cache must first be copied back to memory and then loaded, as data is muta-

ble, whereas instructions can just be overwritten. By comparing Table 7.2 with Table 6.1 on page 72, it is obvious that there is more than a threefold improvement in transfer times by moving the DRAM on to the chip. And that improvement comes even though lines of 32 words are being transferred instead of as many consecutive words as possible.

Instruction Cache Size	Inst. Cache Transfer Time	Data Cache Size	Data Cache Transfer Time	Quantization Block Length
2k words	6.4 μ s	2k words	12.8 μ s	12.8 μ s
4k words	12.8 μ s	2k words	12.8 μ s	12.8 μ s
6k words	19.2 μ s	2k words	12.8 μ s	19.2 μ s
2k words	6.4 μ s	4k words	25.6 μ s	25.6 μ s

Table 7.2: Quantization Block Size for Caches with 32 Words per Cache Line

As was shown in Section 6.3.2 on page 64, the maximum real-time utilization of a system using EQ is $\frac{T-3Q}{T}$, where T is the shortest execution frame period, and Q is the length of a quantization block. By eliminating the original cache load in each frame, that utilization can be increased to $\frac{T-2Q}{T}$. As in Section 6.7 on page 70, assume that the fastest frame allowed in the system has the frequency 44.1/32 kHz. Therefore, for systems with Q equal to 12.8 μ s, the maximum real-time utilization is:

$$\frac{(32/44.1\text{kHz})-(2 \times 12.8\mu\text{s})}{32/44.1\text{kHz}} = \frac{725.6\mu\text{s} - 25.6\mu\text{s}}{725.6\mu\text{s}} = 0.965$$

So, 96.5% of the available time can be used for real-time computation. For the processor in this example with a cycle time of 20 ns, meaning there are a total of 50 MIPS available, this translates to 48.25 MIPS available for real-time use.

In practice, however, if the tasks can be larger than the cache size, it will not be possible to use the full 48.25 MIPS for real-time. First, the quantization block must be made larger in order to accommodate cache misses. Second, the time spent servicing cache misses will be lost, and therefore unusable for real-time. These are the problems that were

first introduced in Section 7.1 on page 83. However, high real-time utilizations can still be achieved. By doubling the quantization block size, half of the time is now available for servicing cache misses, and EQ still allows a utilization of:

$$\frac{(32/44.1\text{kHz}) - (2 \times 25.6\mu\text{s})}{32/44.1\text{kHz}} = \frac{725.6\mu\text{s} - 51.2\mu\text{s}}{725.6\mu\text{s}} = 0.929$$

Therefore, 92.9% of the available time, or 46.45 MIPS, can be utilized by real-time tasks. However, that will be further decreased by servicing cache misses, which will be shown in the following section.

7.3 Real-Time Cache Performance

The quantization block size affects not only the real-time overhead required by EQ to work properly, but also the cache miss penalty. The cache miss penalty has a significant affect on system performance, as there will be many cache misses, and the ability to service these misses in a timely manner will make the difference between an effective system and a useless one. Section 7.1.1 on page 85 shows how the cache miss penalty can be calculated for a given cache size and quantization block size. Using this information and a simulator, the performance of the system can be evaluated for various cache sizes and quantization block sizes, in order to evaluate EQ as a method of real-time caching.

The time a task spends executing can be divided up into two distinct sections: the time spent running on the machine, and the time spent waiting for cache misses to be serviced. Obviously, there is a minimum time that a task must execute, and this time is the worst case execution time of a task in a system with no memory stalls. Therefore, the goal of real-time caching is to bring each task's execution time as close to this minimum as possible. The only way to do that is to reduce the amount of time spent waiting for cache misses to be serviced. Therefore, it is important to remember that the goal of real-time cache design is not to minimize the number of cache misses, but to minimize the total time that a

task must be stalled waiting for the memory. This is actually an important point in any cache design, and a thorough discussion of the art is given by Przybylski in [18]. As the cache in a system using EQ is essentially a regular set associative cache with minor modifications, most all of the considerations that Przybylski raises are applicable in this case.

Since those issues are not fundamental to real-time caching, they will be largely ignored here, but in any real system design it would be foolish not to attempt to optimize the cache design to minimize execution times. In the following sections, some simple optimization is performed solely to yield reasonable estimates of the performance of a system using EQ.

7.3.1 Simulation

Since a timing tool is not currently available for the Mwave system, simulation was used to show the performance of a real-time system with a cache using EQ. Since EQ eliminates all conflicts in the cache due to preemptions, as long as the total quantized task set utilization is below the appropriate bound, each task's total execution time is independent of how many times the task is preempted. In the simulations, tasks were allowed to run to completion without preemptions to determine their behavior in the cache. This is identical to the task's performance if it were preempted and resumed an arbitrary number of times, since when the task is resumed, EQ guarantees that the cache is in exactly the same state as it was when the task was preempted. Since simulation was used instead of actually timing the tasks, the results are not necessarily from the worst case paths through the tasks.

In order to generate address traces for each task's execution in one frame, an architectural simulator was used. The simulator accurately reflects the behavior of the IBM Mwave DSP at the level of registers and memory on a cycle by cycle basis. In order to more accurately simulate the tasks as they would execute under normal circumstances, the

instructions and data were captured while the DSP was performing the actual functions for which the tasks were intended. For example, in order to get an accurate simulation of the five tasks that make up the 16 bit MIDI synthesizer function, the code and data space of the DSP was captured using a debugger while the DSP was actually playing music. It was necessary to capture the data while the DSP was actually performing the desired function because each task checks pointers to its data buffers and other status information to determine if it has any useful work to do each frame. If the data that is originally loaded for each task were used, then the tasks would believe that the function they were performing was idle, and therefore do almost nothing. Clearly, the simulation is more representative of actual execution if the tasks perform their functions rather than quit prematurely.

The debugger for the Mwave system allows instructions and data to be transferred between the DSP and the host system without disrupting other tasks that are running on the DSP. Using these memory images, the simulator can then run each task from start to completion for one frame, and produce address traces that are very close to those that would have been produced had the traces been made from execution on the actual hardware. The simulations were done allowing each task to execute for an entire frame without being preempted. This is acceptable, as execution quantization will preserve the cache for each task across preemptions, so the hit rates would be exactly the same. However, if the memory was captured in the middle of execution of some tasks, which is almost certainly the case, then those tasks will be simulated starting at the beginning of their code. However, they have already executed some of their code during the current frame, so data might be processed twice, or the task might only have to process part of the data that it would have had to if it ran for the whole frame. Ignoring that discrepancy, the code that is simulated is the actual task's code, and the only difference is that the data is captured at a

fixed point in time. So the address trace generated by the simulator is a valid address trace from actual DSP code that runs on the IBM Mwave DSP.

Since the cache contents for each task will be undisturbed by other tasks throughout the task's execution, each task's address trace can then be run through a cache simulator to determine which memory accesses would hit or miss for a given cache configuration. The cache simulator runs through the address trace and handles each access in order. First the simulator determines if the access was an instruction or data access, as there are separate instruction and data caches. Then the simulator decides if the access is a hit or a miss, and updates the least recently used information for the accessed cache set. A cache set is modelled as a stack, so that many different cache configurations can be evaluated simultaneously. For example, an access is a hit in a 4-way set associative cache if the address resides in the top four entries in the set. If the address happens to be the fifth entry in the set, then it would be a miss in a 4-way set associative cache, but it would be a hit in an 8-way set associative cache. When an address is accessed it moves to the top of the stack, which is the position of the most recently used address, whereas the bottom of the stack is the least recently used address. This models a cache using a least recently used replacement policy. In this fashion, the cache simulator determines the number of hits and misses for various cache configurations of various sized caches. Since the miss penalty for each miss depends heavily on the quantization block size, the cache simulator does not output execution times, only the number of accesses, and the number of those accesses that missed in the cache for the given cache configurations.

This simulation strategy does not deal with the worst case execution path of the tasks. Since it is extremely difficult to force a program to follow its worst case execution path, the simulations were done with an arbitrary path through the code. Also, the tasks were not timed by a timing program, so the worst case path was not even known. Regardless,

the simulations represent sample execution traces through actual real-time tasks, and as such give a feel for the access patterns of general tasks that run on the IBM Mwave system.

The data on cache accesses and cache misses must be interpreted in the context of EQ. In order to do this, the cache miss penalty must be determined, using the method described in Section 7.1.1 on page 85. The cache miss penalty is a function of both the cache configuration and the quantization block size, so it must be determined for each such system configuration that is to be analyzed. For a given system configuration, the total execution time of a task can then be calculated by adding the time the task takes to run with no memory stalls to the amount of time the task is stalled, which is simply the product of the number of cache misses and the cache miss penalty. By doing this for a wide range of quantization block sizes, an optimal quantization block size and cache configuration can be found for each task. The utility of a given system configuration for a task is the percentage of the task's execution time that is actually spent doing computation. This utility is the ratio of the amount of time the task spends doing actual work to the total execution time of the task, including memory stalls, which is a good means of comparison among tasks and system configurations.

Two full DSP functions were simulated. The first was the Mwave 16 bit MIDI synthesizer while it was playing a MIDI file, and the second was the Mwave 14,400 baud modem while it was transferring a file. Each of these functions includes multiple tasks.

Table 7.3 shows an excerpt from the output created by the cache simulator program. This particular simulation was run on an address trace generated by one of the MIDI synthesizer tasks, and the table shows the results for some of the possible instruction cache configurations. The first column is the number of sets in the cache. A set is one or more cache lines to which an address can be mapped. Once an address is mapped to a particular

Sets	Line Size (words)	Associativity	Cache Size (words)	Accesses	Misses
128	8	1	1024	4533	93
64	8	2	1024	4533	93
32	8	4	1024	4533	93
16	8	8	1024	4533	93
64	16	1	1024	4533	50
32	16	2	1024	4533	50
16	16	4	1024	4533	50
8	16	8	1024	4533	50
32	32	1	1024	4533	27
16	32	2	1024	4533	27
8	32	4	1024	4533	27
4	32	8	1024	4533	27
16	64	1	1024	4533	15
8	64	2	1024	4533	15
4	64	4	1024	4533	15
2	64	8	1024	4533	15
8	128	1	1024	4533	8
4	128	2	1024	4533	8
2	128	4	1024	4533	8
1	128	8	1024	4533	8

Table 7.3: Partial Output from Instruction Cache Simulation of a MIDI Task

set, it can be stored in any cache line in the set, but there is generally some replacement policy to determine which cache line will be overwritten. All of the cache configurations presented here use a least recently used policy of cache line replacement. The second column is the line size of the cache. A cache line is the smallest unit of data that can be transferred to and from the main memory. The third column is the cache's associativity. The

associativity is the number of cache lines in each set, and therefore the number of cache lines that any given address could be cached in. An associativity of one denotes a direct mapped cache, as there is only one choice for caching any particular address. The next column, cache size, is mainly for clarity, as the cache size is just the product of the number of sets, the line size, and the associativity. The fifth column is the number of memory accesses present in the address trace. The sixth and last column is the number of memory accesses that missed in the cache for each cache configuration.

At first glance it may seem that the cache should be organized with a line size of 128 words, as that minimizes the number of cache misses down to eight. Of course, the time necessary to transfer the larger cache line can easily offset the quick transfer times of the smaller line sizes if much of the data in the larger lines goes unused. If the object of this simulation was to construct an optimal conventional cache, then the data in the table would be used to calculate the total time spent waiting for memory for each configuration based on the number of misses and the miss penalty for each configuration. Of course, this is a rather small simulation to base those results on, and in practice much larger traces would be used. However, the object is to design an optimal real-time cache for use with EQ, which means that the quantization block size must also be considered. For a given quantization block size, the cache miss penalty can be calculated for each configuration, and an execution time can be calculated from that. The problem is to solve the optimization problem across two dimensions, the cache configuration and the quantization block size.

The affects of the quantization block size on the execution time of one task is shown for several cache configurations in Table 7.4. Table 7.4 was generated using the cache simulator program on the address trace from the same task that was used to generate Table 7.3. In addition, Table 7.4 shows the miss penalty and total time spent waiting for

Quantization Block Size (ns)	Cache Configuration		Cache Misses	Miss Penalty (ns)	Memory Stall Time (ns)
	Line Size (words)	Sets			
9,000	16	16	88	420	36,960
	8	32	52	400	20,800
	4	64	31	540	16,740
	2	128	19	1,020	19,380
10,000	16	16	88	300	26,400
	8	32	52	300	15,600
	4	64	31	540	16,740
	2	128	19	1,020	19,380
11,000	16	16	88	240	21,120
	8	32	52	300	15,600
	4	64	31	540	16,740
	2	128	19	680	12,920
12,000	16	16	88	180	15,840
	8	32	52	300	15,600
	4	64	31	360	11,160
	2	128	19	680	12,920

Notes:

1. All table entries are from cache simulation using the same address trace.
2. There were 2,683 total data accesses in the simulation.
3. The shaded row is the optimal system configuration in the table.

Table 7.4: Comparison of Several 1 kword 4-Way Set Associative Data Caches

the data memory by the task for quantization block sizes ranging from 9,000 ns to 12,000 ns. The four cache configurations shown in the table are all 1 kword 4-way set associative caches. The only difference between the four different cache configurations is the line size. The optimal system configuration for the sample space depicted in the table is a 4-way set associative 1 kword cache that has 64 words per line and a quantization block size

of 12,000 ns. In practice, a wider range of cache configurations and quantization block sizes would be used and instructions and data would be analyzed simultaneously. The domain of the problem was restricted here simply to allow the solutions to be presentable in a small table that is easily readable. For this task, the actual optimal configuration is a quantization block of size 11,520 ns, a 1 kword instruction cache having a line size of 64 (associativities of 1, 2, 4, and 8 produce equivalent results), and a 1 kword data cache having a line size of 64 and associativity of either 4 or 8. With that configuration, the task takes 5,361 cycles to execute. With no memory stalls, the task would take 4,533 cycles to execute, so the real-time utility of this configuration for this task is 84.6%.

Section 7.1.1 on page 85 explains the theory behind calculating the cache miss penalty, but no equations were presented. The following analysis will yield some simple equations that can be used to determine the cache miss penalty for a given system configuration. First, define the following variables:

$$\begin{aligned}
 Q &= \text{the quantization block size} \\
 l &= \text{the total number of cache lines} \\
 t_l &= \text{the line transfer time}
 \end{aligned}$$

The above variables can apply to either an instruction or a data cache, and the meaning should be clear from the context. It should be obvious that for an instruction cache, the time that it takes to transfer the entire cache is $(l)(t_l)$. A data cache can be transferred entirely in $2(l)(t_l)$, since the old data must be copied back to memory before the cache can be loaded. Similarly, once the cache loader begins to service a cache miss, it takes t_l to service an instruction cache miss and $2t_l$ to service a data cache miss, once again because the old data must be copied back to memory. Using these values, the maximum number of cache misses that can be serviced in a quantization block in the instruction and data caches, N_i and N_d respectively, can be calculated as follows:

$$N_i = \frac{Q - lt_l}{t_l}, N_d = \frac{Q - 2lt_l}{2t_l}$$

The maximum miss penalty is directly related to N_i and N_d in that the time reserved to service these cache misses will be evenly distributed throughout the quantization block, as shown in Figure 7.1. All line transfers must be allowed to run to completion, because if they are not, it is possible to have cache lines that are internally inconsistent. Also, if cache line transfers can be interrupted, the line transfer time will increase, since the original memory latency will be incurred each time the line transfer is resumed. It follows, then, that the maximum cache miss penalty will be the sum of the most line transfers that can occur before the cache loader will service a cache miss plus the time it takes to service the actual cache miss. Let M_i and M_d denote the instruction cache miss penalty and data cache miss penalty. M_i and M_d can be calculated as follows:

$$M_i = \left\lceil \frac{L}{N_i} \right\rceil t_l + t_l \quad (7.1)$$

$$M_d = \left\lceil \frac{2L}{N_d} \right\rceil t_l + 2t_l \quad (7.2)$$

Equations 7.1 and 7.2 will always be used to calculate the worst case miss penalty throughout the rest of this chapter.

The quantization block size also impacts the percentage of real-time resources that must be reserved to allow EQ to function properly. As the quantization block size is increased, the maximum possible utilization using EQ decreases. When calculating the required overhead for EQ, 105% of the quantization block size was used. This allows for overhead which was not considered in this simple analysis, including, but not limited to, the overhead of making the cache loader switch between transferring one cache partition to servicing misses in the other partition. The actual required amount of overhead is specific to the implementation, and as such is not considered here. However, in order for a specific implementation to function properly, the required quantization block size must be calculated precisely to allow for any extra time that may be necessary beyond transferring

data. The 5% that is used here is an estimate of what that overhead might be. For example, the task that was considered earlier had an optimal quantization block size of 11,520 ns. In order to determine the maximum utilization that EQ will allow, 105% of this number, 12,096 ns, would be used. However, since the quantization block is measured in terms of machine cycles, this number must be rounded up to an integral number of cycles, so the actual quantization block size would be 12,100 ns, or 605 cycles. Using 44.1/32 kHz as the fastest frame rate in the system and removing two quantization blocks from this frame, 96.7% of the processor's resources are available. If all tasks behave in generally the same way as the above task, then the maximum total real-time utilization would be:

$$0.846 \times 0.967 = 0.818 = 81.8\%$$

Obviously, the more simulation that is carried out, the closer this number will be to the actual achievable utilization. Once a timing tool is available, though, all tasks would be timed for use on a specific system with a given cache configuration and quantization block size. Then the actual real-time utilization for any given task set would be the number of MIPS the task set would use if there were no memory stalls divided by the actual number of MIPS that are being used by the task set, including those reserved for EQ.

All five tasks that comprise the MIDI synthesizer function as well as the seven tasks that comprise the 14,400 baud modem function were simulated to determine an estimate of the achievable processor utilization using execution quantization. The results are presented in the next section.

7.3.2 Simulation Results

Most digital signal processing tasks are very small. They execute briefly but frequently, and generally operate on large amounts of data that are passed from one task to the next until the desired result is achieved, which could range from playing music on a speaker to displaying an image on the screen. The tasks used in the cache simulation are

all actual tasks taken from the IBM Mwave DSP system, so the results of the simulation should be useful in that system, and perhaps others. However, since the tasks are so small, the actual number of simulated cycles is also small, which implies that the results may not be universally applicable outside of the Mwave system.

Task Name	Simulation Cycles	Data Accesses	Code Size (words)	Data Size (words)
INT2244	566	526	331	101
MIXER22	749	496	198	6
PLY22M08	83	41	125	12
Q22	535	488	155	87
SYN22K16	4533	2683	944	1535
Total	6466	4234	1753	1741
Average	1293	847	351	348

Table 7.5: MIDI Synthesizer Tasks

Table 7.5 shows the five MIDI synthesizer tasks that were simulated. Since the Mwave system has a pipeline that allows a throughput of one instruction per cycle, each cycle corresponds to one instruction access. Since all of the MIDI tasks are quite small, and every task but one would entirely fit within 1 kword instruction and data caches, this function could be done using only EQ as described in Chapter 6, without worrying about cache misses. In order to do that, either a 2 kword data cache would be needed or the SYN22K16 task would need to be modified to decrease its data size. Despite this, if the system is set up to handle tasks that may not fit in the cache, and the required tag memory is not available to allow preloading in each frame, then no task will be able to have the cache loaded at the start of each frame. Therefore, simulating the MIDI synthesizer tasks produces useful information about how small tasks will behave in a system using EQ. Another noticeable fact about the table is that the average task only executes for about 1000 cycles per

frame. In this function, every task executes in the same frame, which is 44.1/32 kHz, to allow the generation of high quality sound. The fact that these tasks only execute for a short period of time is another characteristic of digital signal processing, and this makes caching quite difficult, since tasks do not always execute long enough to be able to repeatedly make use of cached data.

Task Name	Simulation Cycles	Data Accesses	Code Size (words)	Data Size (words)
ASync	343	229	631	99
C96	387	231	56	19
MCTL	218	98	2519	807
MEIO	9	3	44	10
MNP	11948	4525	3688	3419
V22	5916	4126	3771	2896
V32	22043	15442	7600	7375
Total	40864	24654	18309	14625
Average	5838	3522	2616	2089

Table 7.6: 14,400 Baud Modem Tasks

Table 7.6 shows the seven modem tasks that were simulated. These tasks are slightly different than the MIDI tasks, as most of them are much larger. Clearly, this function will not be able to execute solely out of a 1 or 2 kword cache. Therefore, EQ requires the ability to service cache misses in order to execute this function. Even though the required code and data space of these tasks are significantly larger than the MIDI tasks, there are still small tasks that execute for very few cycles. However, there are also tasks that execute for tens of thousands of cycles, which is significantly longer than any of the tasks in the MIDI synthesizer function.

Even though the total number of simulated cycles between the two functions is only about 50,000, these two functions are typical of the IBM Mwave system, and therefore are likely to have memory access patterns that are similar to many tasks in that system. However, since the tasks are small and do not execute for long periods of time, small perturbations in the access patterns could potentially skew the results. With that in mind, the following results are used to depict the potential usefulness of EQ in the IBM Mwave system.

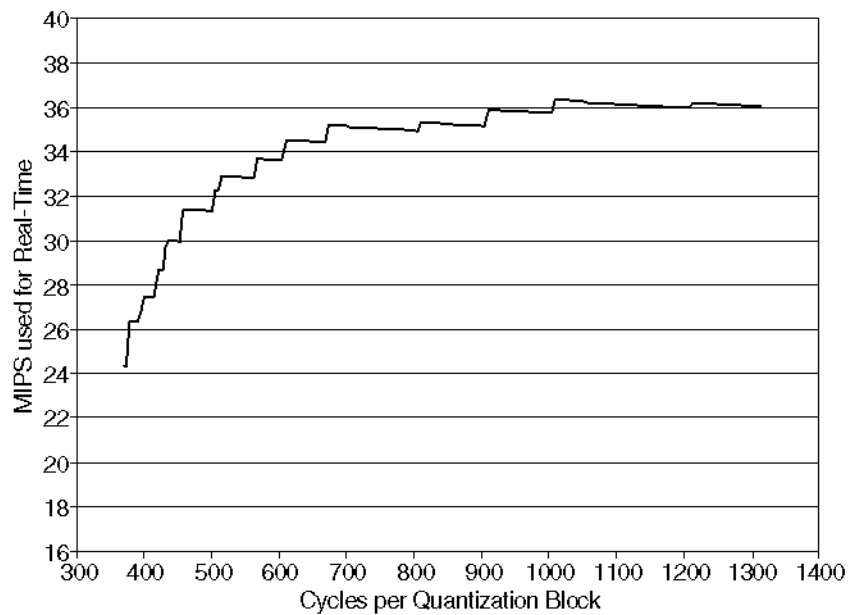


Figure 7.4: Utilized MIPS in Simulation of MIDI Tasks on a 50 MIPS DSP

Figure 7.4 shows the results of simulating all of the MIDI tasks to determine how much of the processor could be used for real-time computation. The number of memory accesses and cache misses were determined for a range of cache configurations by the cache simulator. The cache miss penalty for each cache configuration was then calculated for a range of quantization block sizes. These cache miss penalties were used to calculate the total execution time of the tasks for each cache configuration and quantization block size. The length of the quantization block represents 105% of the actual time that is avail-

able for memory transfers, once again allowing for implementation overhead. The graph accounts for the lost MIPS due to the overhead required by EQ and represents the percentage of the MIPS that would have actually been used for computation. At each quantization block size, the utilization that is graphed is the utilization that can be achieved with the optimal cache configurations, so the graph shows the results of varying the quantization block size and the cache configurations at the same time. Looking at the graph, it is clear that the real-time utilization of the system is maximized when the quantization block is either about 1000 cycles or 1200 cycles. Obviously, there can be only one cache configuration and quantization block size in an actual system, so this data would have to be used to choose a single system configuration, but it gives an idea of roughly what length the quantization block should be.

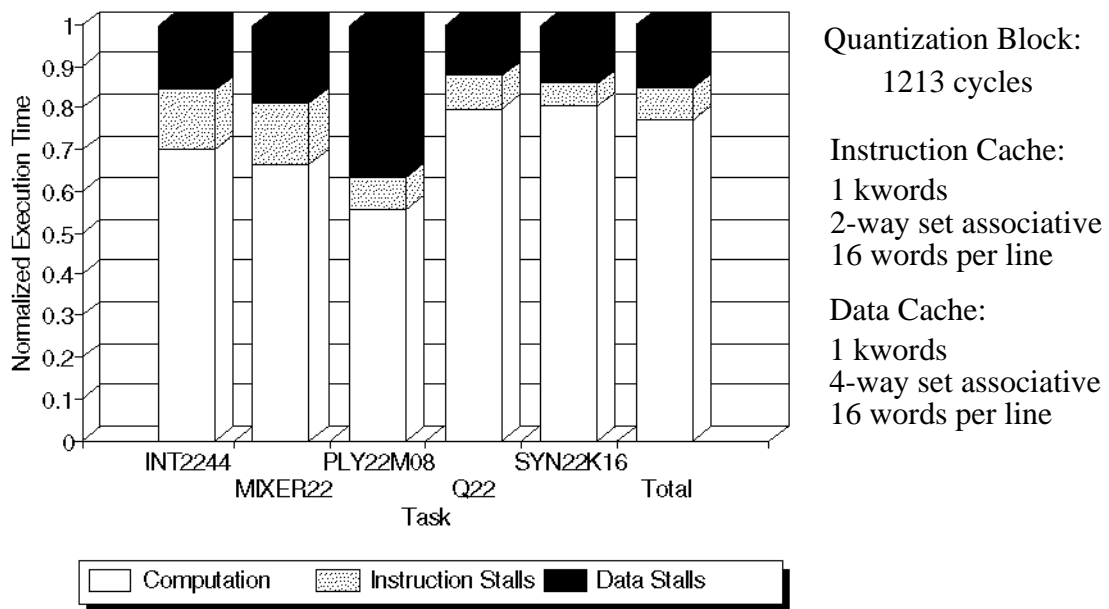


Figure 7.5: Normalized Results of MIDI Synthesizer Simulation

Figure 7.5 shows the simulation results for each task in the MIDI synthesizer function for a specific system configuration. The system configuration that is listed in the figure is the configuration that allows the MIDI synthesizer function and the 14,400 baud modem

function to achieve the highest total utilization. By analyzing both functions with the same system configuration comparisons can be easily made, and the results are more realistic, since in an actual system, the configuration cannot change for each function. In the figure, the total execution times were normalized to show percentages of time that are spent doing computation and waiting for the memory. This function uses the cache effectively, as seventy seven percent of the tasks' total execution time was used for computation, wasting only a little over twenty percent of the time waiting for cache misses to be serviced. Since the quantization block size is 1213 cycles, 93.3 percent of the processor's 50 MIPS, or 46.65 MIPS, are available for real-time use. Therefore, if all tasks behave similarly to the MIDI synthesizer tasks, then a total of 71.8 percent of the processor's resources can be utilized for real-time, which amounts to 35.9 MIPS.

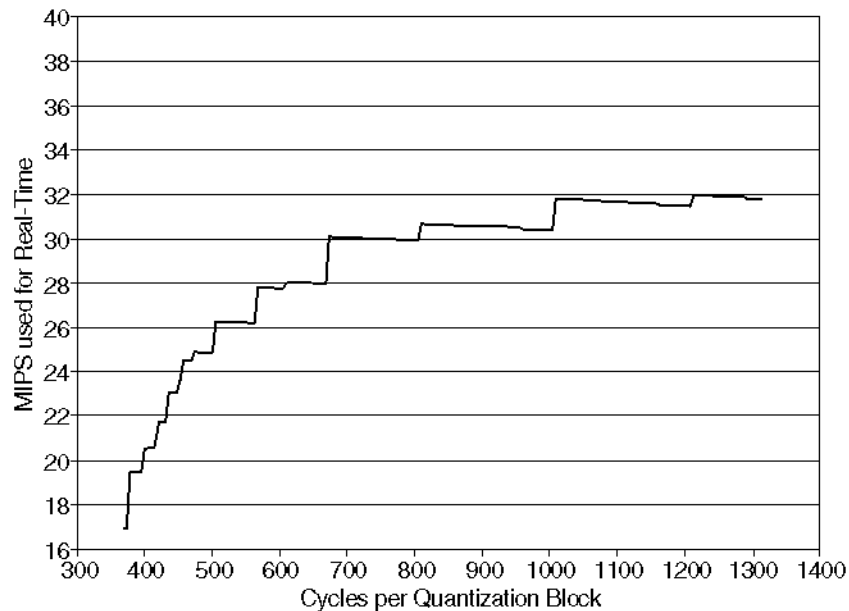


Figure 7.6: Utilized MIPS in Simulation of Modem Tasks on a 50 MIPS DSP

Figure 7.6 shows the results of the 14,400 baud modem simulation. The graph was generated in exactly the same way as Figure 7.4, except the address traces used by the cache simulator came from the modem tasks, rather than the MIDI tasks. The results are

quite similar to those of the MIDI synthesizer simulation in that the most real-time computation can be performed when the quantization block size is either about 1000 cycles or about 1200 cycles. This would seem to indicate that many of the Mwave tasks behave similarly and benefit from roughly the same sized quantization block as each other. This is good news, as only one quantization block size can be used in a real system. It would be difficult to design a good system if all of the tasks do not behave well using roughly the same design.

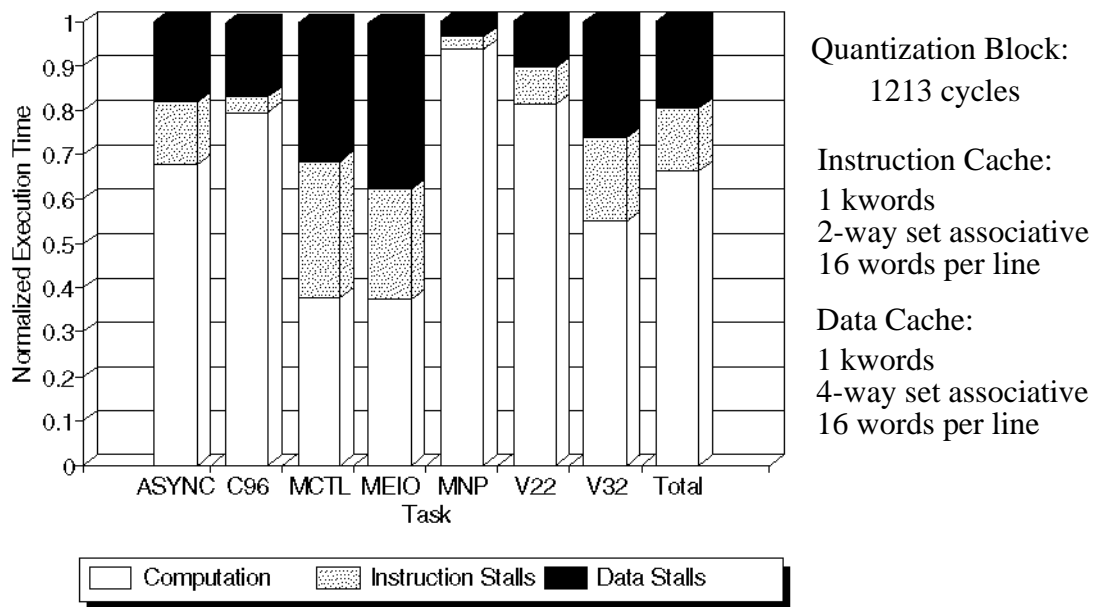


Figure 7.7: Normalized Results of 14,400 Baud Modem Simulation

Figure 7.7 shows the simulation results for each task in the 14,400 baud modem function for the same system configuration that was used to analyze the MIDI synthesizer function. This function does not perform as well as the MIDI synthesizer, as the time spent doing real-time computation amounts to only about 66 percent of the total execution time. Once again, using a quantization block size of 1213 cycles only allows 93.3 percent of the 50 MIPS processor's resources to be used for real-time, meaning that a total of 61.6 per-

cent of the processor's resources, or 30.8 MIPS, can be used for real-time by tasks behaving like the 14,400 baud modem tasks.

When the totals for both functions are combined, using the same system configuration, then the time spent doing real-time computation amounts to 67.5% of the total execution time. When this is derated by the time reserved for EQ, 63% of the processor's total resources can be used for real-time computation. On a machine with a total of 50 MIPS available, this amounts to 31.5 MIPS being available for actual real-time computation.

The two functions that were simulated give an idea of the kind of performance that can be achieved by using EQ in a real-time caching system. Depending on the memory access patterns of the tasks, utilizations between sixty and seventy five percent can be achieved. However, the numbers given in the simulation results do not take into account the fact that a task's actual worst case execution time must be quantized to be an integral number of quantization blocks. The utilizations that were found in the simulations can be used to extrapolate what percentage of time a given task will spend doing real-time computation and what percentage of time that task will be stalled waiting for memory. These numbers should remain roughly the same regardless of the length of execution of the task. If the execution times were quantized, then the percentages would change with the length of the task, depending on how much time is lost because the task does not execute for an integral number of quantization blocks. Also, the only execution time that really would be quantized is the worst case execution time, and these simulations do not necessarily measure the performance of a task's worst case, but rather of a typical case. Therefore, the percentages that were derived from the task simulation should provide a guideline that can be used to evaluate EQ, but in practice, worst case execution times must be quantized in order to allow EQ to function properly.

Another deficiency of the simulation results is that overlapping instruction and data cache misses were not accounted for. Since the instruction and data memories and caches are completely separate, overlapping instruction and data misses can be handled simultaneously. Since the simulator was not able to recognize overlapping misses, the results of the simulation are slightly pessimistic. Therefore, the actual achievable performance using EQ could be slightly higher than the simulation results indicate.

Chapter 8

Conclusions

8.1 Results

Even though caches have become a part of almost every computer system, they are generally not used in real-time systems. Even if a cache is used in a real-time system, the system usually does not take advantage of the available performance improvement that the cache allows. Execution quantization enables a cache to be deterministically incorporated into a real-time system that uses priority driven preemptive scheduling.

For a fixed overhead, which can be determined ahead of time, EQ ensures that the cache contents are always restored for each task when it resumes execution after being preempted. This eliminates interference among tasks in the cache, and allows each task to be analyzed individually, without regard for the rest of the system. In order to achieve this restoration without severely degrading performance, tasks must execute for fixed length blocks of time.

EQ allows the deterministic use of a slightly modified set associative cache with two partitions by extending the dynamic deadline scheduling algorithm. The major modifications to a caching system to allow it to use EQ are to modify the operating system slightly to make it aware of the cache loader and unloader, to add hardware to gate off the scheduling interrupts to occur only at discrete times, and to add hardware to implement a cache loader and unloader.

With these modifications, simulation shows that the IBM Mwave system can make use of EQ and still be able to use about 63 percent of the processor's resources for deterministic real-time computation, and one function was able to utilize almost 72 percent of the processor's total resources. This is a distinct improvement over the use of DRAM without

a cache, which would allow very little of the processor's resources to be used for real-time computation.

The addition of a deterministic cache to any real-time system will undoubtedly place restrictions on the use of that system. However, EQ has the following properties:

1. there is little hardware overhead,
2. each task is allowed to use half of the total cache,
3. there is no additional context switch overhead,
4. programmers and the compiler do not need to be aware of the cache hardware,
5. a conventional n-way set associative cache can be used with minor changes,
6. there is protection of each task's data, and
7. deterministic caching of shared memory is allowed.

These features should make EQ extremely useful in most any real-time system, as EQ does not place restrictions on the tasks or the programmer. As long as a task can report its worst case execution time, it can be used with EQ, and in any real-time system all tasks must report their worst case execution times anyway.

8.2 Future Work

This research was targeted at allowing the use of a cache in a real-time system using dynamic deadline scheduling. There are many ways in which the research could be expanded and explored more thoroughly. The specific goal of the research was to be able to use a cache in the IBM Mwave DSP system. As such, there is room for execution quantization to be evaluated and expanded for use in other systems.

8.2.1 Rate Monotonic Scheduling

As many real-time systems use rate monotonic scheduling, an important extension to EQ would be to extend it for use in a system that uses RMS. The basic concept of EQ would remain unchanged, as the main focus is still to negate the impact of preemptions on cache contents, which EQ will perform in any priority driven scheduling environment.

RMS allows many scheduling extensions, such as transient overloading and aperiodic task scheduling. These features would have to be incorporated into EQ, and a scheduling bound would need to be established to allow for quantized scheduling, as well as loading and unloading the cache.

8.2.2 Non-Real-Time Tasks

Many real-time systems allow the time that is not used by real-time tasks to be claimed and used by non-real-time tasks. Non-real-time tasks do not have deadlines, and as such are similar to regular programs that execute on general purpose processors. EQ has no provision for scheduling non-real-time tasks into the unused portions of quantization blocks after tasks have completed execution. It would be extremely beneficial to allow this otherwise unusable time to be used by non-real-time tasks.

In order to schedule non-real-time tasks, the scheduler would have to become active when a task completes in the middle of a quantization block to schedule the non-real-time tasks. A method would then have to be devised to either allocate the non-real-time tasks some cache space, or to allow them to execute directly out of the DRAM. Either way, the non-real-time tasks' execution cannot interfere with the real-time tasks in any way.

8.2.3 Task Timing

Although the requirements and specifications of a timing tool were largely described, such a tool was not constructed. Instead, simulation was used to analyze the potential usefulness of EQ. However, the simulation done in this research was confined to the Mwave system. Clearly, this could be extended to other processor architectures to show the performance of EQ on a wider scale. Although, the construction of an accurate timing tool far outweighs the usefulness of task simulation, since EQ does not allow tasks to interfere with each other in the cache.

Having such a timing tool would certainly be advantageous to the IBM Mwave system, since currently the programmers must time their tasks by hand, and the addition of a cache is likely to make this a near impossible task. However, such timing tools do exist for systems without a cache, such as Halang and Stoyenko's schedulability analyzer which is described in [7]. Future research in this area should include the extension of such a tool to include task timing in a caching system that uses EQ.

References

- [1] William Y. Chen, Roger A. Bringmann, Scott A. Mahlke, Richard E. Hank, and James E. Siculo. "An Efficient Architecture for Loop Based Data Preloading," *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pp. 92-101, 1992.
- [2] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*, Cambridge, MA: The MIT Press, 1990.
- [3] John W.C. Fu and Janak H. Patel. "Data Prefetching in Multiprocessor Vector Cache Memories," *Proceedings from the 18th International Symposium on Computer Architecture*, pp. 54-63, 1991.
- [4] John W.C. Fu, Janak H. Patel, and Bob L. Janssens. "Stride Directed Prefetching in Scalar Processors," *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pp. 102-110, 1992.
- [5] Dan Hafeman and Bob Zeidman. "Memory Architectures Compound RISC's Gains," *Electronic Design*, Vol. 39, pp. 71-82, July 11, 1991.
- [6] Wolfgang A. Halang and Matjaz Colnaric. "Determination of Tight Upper Execution Time Bounds for Tasks in Hard Real-Time Environments," *Proceedings from the 6th Mediterranean Electrotechnical Conference*, pp. 1073-1076, May 1991.
- [7] Wolfgang A. Halang and Alexander D. Stoyenko. *Constructing Predictable Real Time Systems*, Boston: Kluwer Academic Publishers, 1991.
- [8] David T. Harper III and Darel A. Linebarger. "Conflict-Free Vector Access Using a Dynamic Storage Scheme," *IEEE Transactions on Computers*, Vol. 40, No. 3, pp. 276-282, March 1991.
- [9] John L. Hennessy and David A. Patterson. *Computer Architecture A Quantitative Approach*, San Mateo, CA: Morgan Kaufmann Publishers, Inc., 1990.
- [10] Elizabeth A. Hinzelman-Fortino. "The Mwave Operating System: Support for Hard Real-Time Multitasking," *DSP Applications*, Vol. 3, No. 3, March 1994.
- [11] Norman P. Jouppi. "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," *Proceedings from the 17th Annual International Symposium on Computer Architecture*, pp. 364-373, 1990.
- [12] David B. Kirk. "SMART (Strategic Memory Allocation for Real-Time) Cache Design," *Proceedings of the IEEE Real-Time Systems Symposium, CA*, pp. 229-237, December 1989.
- [13] David B. Kirk. *Predictable Cache Design for Real-Time Systems*, Ph.D. Thesis, Carnegie Mellon University, Pittsburgh, PA, December 1990.
- [14] David B. Kirk, Jay K. Strosnider, and John E. Sasinowski. "Allocating SMART Cache Segments for Schedulability," *Proceedings from EUROMICRO '91 Workshop on Real Time Systems*, pp. 41-50, 1991.
- [15] John Lehoczky, Lui Sha and Ye Ding. "The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior," *Proceedings of the IEEE Real-Time Systems Symposium*, pp. 166-171, 1989.

- [16] C.L. Liu and James W. Layland. "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *Journal of the Association for Computing Machinery*, Vol. 20, No. 1, pp 46-61, January 1973.
- [17] Virgil Mitchell. *Mwave Developer's Toolkit DSP Task Programming Guide*, Intermetrics Inc., 1993.
- [18] Stephen A. Przybylski. *Cache and Memory Hierarchy Design: A Performance-Directed Approach*, San Mateo, CA: Morgan Kaufmann Publishers, Inc., 1990.
- [19] David Shear. "Three DSP RTOSs Are Ready To Merge With Windows," *EDN*, pp. 29-34, June 23, 1994.
- [20] Jay K. Strosnider and Daniel I. Katcher. "Mwave/OS: A Predictable Real-Time DSP Operating System," *DSP Applications*, Vol. 3, No. 3, pp. 27-32, March 1994.
- [21] Dave Wilson. "Real-Time DSPs Target Multimedia Motherboards," *Computer Design*, pp. 36-38, February 1992.

