Process Dependent Static Cache Partitioning for Real-Time Systems

David B. Kirk

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

Abstract

Cache architectures in real-time systems must have predictable performance if the resulting reduction in memory access time is to benefit task scheduling with guaranteed deadlines. However, priority-driven preemptible scheduling algorithms cause an unpredictable number of cache-reload transients resulting from task preemptions. A fixed task set in a real-time system allows intelligent caching schemes to significantly increase predictability at minimized cost to the hit ratio. This paper discusses a technique which statically partitions the cache for each task in the task set to allow maximum cache performance predictability, and often improved hit ratios.

This paper investigates the use of a priori knowledge of program behavior to partition an instruction cache of size C into a static partition of size S, and an LRU partition of size C-S. The value of S is task dependent, and is nonzero for most programs running on the system. Example programs are presented, and their behavior in various size caches is discussed. Cache partitions are generated and evaluated to determine the increase in cache performance and predictability. Finally, a high level hardware design is presented which provides the desired partitioning scheme.

1. Introduction

1.1. Overview

Caches have been bridging the gap between CPU speeds, and main memory speeds since they were first introduced in the IBM 360/85 computer in 1969. Their absence in real-time system designs, however, has been noticeable. This paper investigates one of the reasons for this phenomenon, and proposes an approach to cache designs which is more compatible with the scheduling algorithms used in real-time systems.

Cache memories are small, fast buffers that are used to temporarily hold recently used information, as well as information to be used in the near future. Although the specific design decisions may vary from one cache to another, the underlying success of the cache performance depends on two program properties: spatial and temporal locality. Temporal locality refers to the tendency of a program to revisit areas of memory that have recently been accessed. This time related behavior is exemplified by loops and repeated procedure calls. Spatial locality refers to the tendency of a program to access memory locations close to those locations that have recently been accessed. This space related behavior is exemplified by the execution of regions of sequential code, or access of sequentially stored data structures.

The cache attempts to ensure that information, local both in time and in space to the current information, is readily available [11]. It is important to note, however, that a cache attempts to ensure that the needed information is readily available. Unless a task is running in a single user environment with no interrupts, it is very difficult (if not impossible) to predict the cache hit ratio. This nondeterminism results from the cache-reload transient [14], and makes caches difficult to use in real-time systems striving for predictability in order to guarantee deadlines.

Unlike time-shared systems, however, real-time systems are often characterized by predefined task sets. For this reason, program behavior data is available at system configuration time, and can be used to enhance the predictability and sometimes the performance of a cache of size C. Specifically, this paper investigates the use of a priori knowledge of program behavior to logically partition a cache of size C into an static partition of size S, and an LRU partition of size C-S. The value of S is task dependent and would be nonzero for most processes running on the system. The benefit from this task based partitioning scheme is twofold. It will be shown that in some cases it is possible to improve the hit ratio for a given task, while at the same time guarantee a certain minimal hit count. The guaranteed minimal hit count is then used to reduce the worst case execution time. This reduced worst case execution time provides an improvement in the schedulability of a task set by reducing the utilization of each task. This has the same effect as increasing the schedulable utilization, which is the highest resource utilization attainable at or below which all task deadlines are guaranteed. Schedulable utilization is the performance measure used in conjunction with most priority-driven preemptive scheduling algorithms, such as the rate-monotonic and deadline-driven algorithms. These scheduling algorithms have been shown to be optimal for guaranteeing deadlines of periodic tasks in real-time systems [5].

In the remainder of this section, the design goals and requirements for batch and time-shared systems are reviewed and contrasted with the goals and requirements of real-time systems. The scheduling concerns and algorithms for each type of system are presented, using the rate-monotonic algorithm as an example for real-time systems. Section 2 presents and discusses the partitioning scheme in detail presenting cases that would benefit from such a scheme, as well as cases that would not. Section 3 reviews the cache model and tools used to evaluate the program trace data generated for this study. The tools and software routines used in this study are presented with details discussed when needed for clarification. Section 4 presents the LRU simulation results and discusses the results of implementing the recommended partitioning scheme. Section 5 describes the hardware approach to
designing such a logical partitioning scheme. Section 6 presents concluding remarks and a discussion of future work in this area.

1.2. Background: Performance Issues for Batch, Time-Shared, and Real-Time Systems

Effective system design strategies rely on an appropriate matching of the system's hardware, software, and environment [7]. In a similar manner, proper evaluation of the system performance must also take into consideration the system environment. The computing environments for batch, time-shared, and real-time systems have different requirements and performance measures. As a result, the appropriate CPU scheduling algorithms vary among them.

Batch systems typically schedule tasks using a shortest job first strategy [7]. Preemptive schedulers are used to allocate time-slices to each task. Issues such as job throughput, turnaround time, response time, and efficient resource utilization are the primary scheduling concerns of these types of systems. Correctness of the result is a function of accuracy and typically not the time at which the result was provided.

Real-time systems are designed for an environment that presents tight timing constraints on the system behavior. They typically handle large amounts of data for use in computations that are bounded by hard and soft deadlines. These deadlines are often the result of periodic events that provide large quantities of data at regular intervals. It is necessary to perform computations on the present iteration of data before the next arrives. Real-time systems correctness of a result is a function of both precision and time. This time-value of a result is expressed as a constraint which if missed, can cause system failure. Examples of real-time systems include nuclear power plant controllers and avionic systems.

Schedulers for a real-time system are concerned with maximizing resource utilization subject to meeting system timing constraints. The rate-monotonic algorithm has been shown to be the optimal static priority scheduling algorithm for independent periodic tasks [5]. Under the rate-monotonic scheduling algorithm, the priorities assigned to periodic tasks are directly related to their rate of requests. Liu and Layland proved this algorithm guarantees that n periodic tasks can always be scheduled to meet all task deadlines provided that the processor utilization is less than n(2^(1/m) - 1). This bound converges to ln 2 (≈ 0.69) for large n. It is, however, pessimistic, and represents the absolute worst case conditions. The average case scheduling bound is typically 88% [3].

While the rate-monotonic algorithm works well with periodic tasks with fixed execution times, some modifications need to be made to handle stochastic execution times and aperiodic tasks. Scheduling of aperiodic tasks can be performed through methods such as the priority exchange and deferrable server algorithms as discussed in [4]. Stochastic execution times can lead to a required utilization greater than the schedulable utilization bound of the task set. In these cases, the period transformation algorithm [9], can be used to guarantee that the deadlines of critical tasks will be met.

2. Static Partitioning Scheme

Unlike batch or time-shared schedulers, the priority-driven preemptive schedulers used in real-time systems require predictable cache performance to calculate worst-case execution times. Given a fixed task set, use of program behavior information can lead to an efficient as well as predictable approach to cache design.

As mentioned earlier in this paper, the success of cache performance depends on temporal and spatial locality. Consequently, a cache of size C might be used very efficiently by one task with certain characteristic locality, and very inefficiently by another task with a different measure of locality. For example, a cache of size 100 might be used very efficiently by a task with a loop of 100 instructions, and very inefficiently by a task with a set of 10 instruction loops (90% of the cache will be holding unused information). This study will show that some tasks will run very efficiently in a cache of size C-W, and literally waste W lines of cache. The remainder of this paper discusses a technique for optimizing cache designs on a task specific basis to increase performance and predictability.

Programs which run efficiently in a cache of size C-W typically have a significant number of loops of instruction length less than or equal to C-W. The first pass through the loop initializes the cache, and the next n-1 passes experience cache hits (for a loop iteration length n). If, however, the remaining loops all have instruction length C+m (where m is any positive integer), it is possible that the remaining portion of the cache is used very inefficiently. This results in C-W lines having high hit ratios, and W lines being used to hold data that is typically accessed once, and then replaced by new data before being used again. In reality, C-W lines are used effectively for some time, and then all C lines are used very inefficiently. For simplification, we assume a model such as the one in Figure 2-1 which shows a convenient split between the lines used efficiently, and those that are not.

![Fully Associative Cache](image)

Figure 2-1: Inefficient Cache Model

Instruction address traces and compiler techniques can be used to detect good candidates for loading in the static partition. Partitioning is then performed to maximize predictability, and in some cases improve the overall hit ratio. The cache is logically partitioned into two regions: a static partition of size S (where S is set equal to W) and an LRU partition of size C-S. The LRU region would function as an ordinary cache architecture with an LRU replacement algorithm. The static partition would be preloaded when the task is swapped in. Ideal candidates would have a high frequency of access and a high miss count in a cache of size C-S. This would be an instruction which occurs often, but has a distance which is typically greater than the cache size of C-S. The distance of instruction I, is the number of unique instructions accessed between two successive accesses of instruction I. The majority of the instructions chosen to be preloaded in static partition will have a distance of greater than C. However, it is possible to benefit from choosing a partition that causes instruction I to miss in the cache of size C-S, even if I hit in a cache of size C.
For example, suppose a process runs very efficiently in a cache of size 100, but the actual cache size is 200. Assume the cache is fully associative with one instruction per line. Suppose further that no hits occur for instruction distances greater than 100, and less than 195, but that 5 instructions hit in the cache at distance of 195 or greater. If each of these instructions hits 1000 times, they contribute to the overall hit count by 5000 hits. Now suppose that we cause these instructions to miss in the LRU partition by setting the size equal to 100. However, at the same time, we preload these 5 instructions into the static partition of size S (=100). The original hit count is maintained by 105 lines of the 200 line cache. This leaves 95 lines to "profit" from. Suppose that there is a loop with an instruction length greater than 200, and an iteration count of 100. If 95 of these instructions are preloaded in the static cache, there is an improvement in the hit count by 9500 hits. The portion of the cache that was contributing only 9000 hits, now contributes 14500 predictable hits. If we assume a main memory access time of 150 nsec, and a cache access time of 30 nsec, the extra 9500 hits results in a savings of 1.115 msec. On a 4 MIPS processor, this would result in approximately 4460 instructions that can be used for servicing access requests with a scheduling algorithm such as the Extended Priority Exchange (EPE) discussed in [13].

There are, however, two scenarios in which a static partition will show little to no increase in the cache hit count. The first of these cases is due to programs that have a high number of misses due to instructions which are executed very infrequently. In the previous example, this would be true if the loop of instruction length greater than 200 had had an iteration count of 2, and furthermore, there were no other instructions that occurred more often than these. In this case, the last 95 lines of the cache would only provide an increase of 190 hits to the original hit count. The second case that would not benefit from the static partitioning scheme is a task which has a large number of hits contributed by instructions that have a distance of C - e, where e is a very small integer. In such a scenario, any reasonable partition of C would cause a decrease in the total number of hits that the new partition could not match. In both of these cases, it would not be possible to improve the overall hit rate of the cache. However, it is still possible to gain predictability at the cost of a reduced hit rate. The reduced hit rate results from a cache size which is now C - S rather than C. If cache performance is more of a concern than predictability, then the static partition size S can be set to zero.

3. Software, Tools, and Models

The study of selected program behaviors, and their performance in various size caches used a simplified model of a cache. The cache of size C is a fully-associative cache with one entry per line. Small cache sizes were chosen for this study due to the size of the tasks that were being modelled. Cache sizes were chosen small enough to prevent the entire program from being resident in cache, and large enough such that reasonable hit counts were experienced. Typical cache sizes were between 128 and 256. Larger programs and caches will be modelled in future tests.

3.1. Software - Pass 1

Software routines representative of typical real-time applications were chosen for modelling. Modules which provided reasonable trace capabilities were favored. Therefore, code which performs basically the same operations whether there are 100 inputs or 1000, such as matrix operations, were good candidates. The routines selected included a Fast Fourier Transform (FFT) routine, a matrix factorization routine, and an eigenvalue calculation routine.

The FFT routine chosen computes the Discrete Fourier Transform (DFT) for a complex array of size N, and outputs the results to a different complex array of size N. The Cooley and Tukey butterfly computation is used to calculate the FFT with \( (\frac{N}{2}) \log_2 N \) multiplications. The code is written in C, compiled on a MicroVax workstation. Trace data were obtained using an input array size of 128.

The matrix factorization routine factors a double precision band matrix by Gaussian elimination. The routine, DGBCO.F, is part of the LINPACK software library, and is written in Fortran. A front end calling routine, written in C, was used to initialize the array and input parameters. The code was compiled on a MicroVax using the F77 compiler package. Trace data were obtained using a square matrix of size 8.

The eigenvalue calculation routine finds the eigenvalues for an upper Hessenberg matrix using QR factorization. The routine, HQR.F, is part of the EISPACK software library, and is written in Fortran. A front end calling routine was written in C to initialize the matrix and other input parameters. The code was compiled on a MicroVax using the F77 compiler package. Trace data were obtained for a square matrix of size 4.

3.2. Software - Pass 2

The three software modules discussed in the previous section resulted in a significant amount of I/O in the C front end calling routine generated by the scarf function used to initialize the input data. This was particularly true in the FFT routine which required an input of 128 values. Consequently, a high proportion of the addresses in the address traces were due to the I/O routines. Since most of this code is related to data formatting that would not be necessary in the actual real-time system with DMA facilities, a second simulation was performed which eliminated the I/O trace addresses. Once again, three software modules were tested. The three software modules were compiled using the F77 compiler package, and run on a DEC 8800.

The FFT routine used in the second simulation was selected from the FFTPACK library. The routine, CFFT1.F is written in Fortran, and computes the complex discrete Fourier transform, and Fourier coefficients for the complex input array of size N. The Fortran routine CFFT1.F is used to initialize the working array. Trace data were obtained for an input array of size 150.

The matrix factorization and eigenvalue routines were identical to those used in the first simulation pass. However, larger square matrices (20 and 8, respectively) were used to increase the address trace which had been reduced by eliminating the I/O portion.

3.3. Tools

The software compiled as discussed above was executed on a DEC MicroVax workstation, or DEC 8800, under the dca symbolic debugger. The tracei option provided an instruction level address trace for each routine.

Program behavior in an LRU cache was determined by simulating the hit/miss success of each address in the address trace for a cache of variable size C using the priority-stack method [6]. Cache sizes ranging from 1 to 200 were selected for the first simulation run. The output of this simulator consisted of each instruction address with its hit/miss count, as well as the overall hit rate for the specific cache size.

The cache simulator was then modified to allow a static partition. Input to this simulator included: the total cache size (C), the static
partition size (S); and the data to preload in the static partition. The address trace was analyzed using a (C-S) partition as a fully associative cache with an LRU replacement algorithm. The static partition was preloaded and never altered for the life of the task. Hits in either partition counted as cache hits, but misses only considered the LRU portion for replacement. Output from this routine included individual hit/miss counts for each address, the total hit/miss ratio, and the total hits in the static partition.

Finally, the cache simulator was modified to gradually increment the size of the cache to C. For each cache size (c), a hit and miss count was recorded, as well as the (C-c) addresses with the highest miss counts. These (C-c) addresses are the best candidates to be loaded in the static partition of size (C-c). These data could then be used to calculate the exact hit count for any size LRU and static partition of total size less than or equal to C.

3.4. Program Behavior Modelling

Although this research performed the cache partitioning by examining address traces, there are basically two models required to perform this cache partitioning by analytical means. One model is needed to predict the expected hit count for an n line static partition for a specified program. This involves predicting the current miss count for the n lines selected to reside in the static partition. Secondly, it is necessary to find an appropriate model for program memory access patterns. This model would then be used to predict the success of the program executing with an LRU based cache of size C. During this study, three models were considered for program memory access patterns. The models showed no significant correlation to the access patterns. We feel this is due to their application to program pieces rather than the full-size programs they were developed for. The models examined are discussed below.

The first model was proposed by Saltzer in [8]. This approach defined the headway in a multi-level memory hierarchy as the number of references at level one before a request is generated to level two. In a cache system, this would be the number of requests issued to the cache before a miss occurred, and there was a request to main memory. The model proposed that the headway increases linearly with the size of memory level one. In our test cases, although the headway did increase with the size of the cache, it did not increase in a linear manner. Instead, the headway would remain the same for long intervals of cache size, and then suddenly increase rapidly in one step. We could not get the linear model to hold.

The second model was proposed by Coffman and Ryan in [1]. This approach used the working set model defined by Denning in [2], in which the working set at time t is the set of distinct pages referenced in the last I memory references. Coffman proposed that over n time samples, the size of the working set would be normally distributed. Using this model in the cache environment, as the cache size is increased from C to C+c, the change in the hit ratio is reflective of the success of working set size C+c. Therefore, a plot of the hit ratio for a cache of size one to a cache of size C should appear to be normally distributed. However, for our test cases, it did not. Once again, the hit ratio did increase with the size of the cache, however, it was not a normal distribution. The hit ratio remained constant for long intervals of cache sizes, and then showed sudden increases.

The final model was proposed by Thiebaut, Stone, and Wolf in [15]. In this paper, the program behavior is modelled as a fractal-random walk on a one-dimensional lattice. A log/log plot of the unique memory references to total memory references is used to extract four parameters needed to model the program behavior. The four parameters are derived from the slope and y-intercept of two distinct linear regions of the graph. This fractal model is then used to predict the miss ratio in a fully-associative cache of size C with an LRU replacement policy. The fractal dimension, which is inversely proportional to the slope on the log/log plot of the region where the cache is no longer suffering cold-start, is a measure of locality of reference. Typical numbers are between 1 and 2, where 1 is considered to have very little locality, and 2 is considered very high locality. However, when this model was applied to the programs in our study, the fractal dimension was extremely higher. We believe this was due to the fact that we are tracing only instruction addresses, whereas Thiebaut traced all memory accesses. Once the loops were executed once, very few unique addresses were issued. This results in extremely high locality. Once again, regions that the model predicted should be linear showed areas of constant unique memory references followed by intervals of sharp increases. It is our intention to reexamine this model when our future research considers both instruction and data addresses for larger program samples.

4. Simulation Results

4.1. Traces with I/O

Address traces used in this section were generated by the method discussed in the section on tools. The addresses which implement the SCANNF.C function used in each of the C front end calling routines to read the data for the input array are included in these traces.

4.1.1. FFT Algorithm

Nine cache sizes were selected for the initial cache LRU simulation. The set of (1,5,10,20,30,40,50,100,200) was inserted into the simulator, and a total of 122,759 addresses were used. A summary of the results is shown by the solid line in Figure 4-1. As can be seen in the figure, the hit count makes significant progress for caches less than 40, but at 40 it begins to level off. While the hit count increases up to size 100, very little progress is made after that. Based on these data certain partitionings were selected. For some cache sizes the static partition size was easily chosen. For example, since a cache of size 200 did just a fraction better than a cache of size 100, it implies that half of this cache is "wasted", so a partition of 100/100 is selected. In a similar manner, a cache of size 50 was only marginally better than that of a cache of size 40. Hence a cache partition of 40/10 (LRU/Static) is chosen. In about half the cases, however, the partition boundary was not as clearly defined. For these cases, trial and error was used until a better procedure was adapted in later simulations. In all cases, it was possible to improve the overall hit count while now providing guaranteed hits and predictable cache performance. The results are summarized by the dashed line in Figure 4-1. The most significant data is recorded for the case of a cache of size 200.

To determine the absolute best partitioning boundary for a cache of size 200 or less, the third cache simulator was used. A summary of the results for C equal 200 is presented in Figure 4-2. Note that the same data used to form this summary could be used to form a summary for any cache size less than 200. The dashed line in Figure 4-2 represents the hit count for an LRU cache partition of the size (L). The dotted line represents the hit count for the static partition of size (C-L), where C is 200 for this simulation pass. Hit count information for any size static partition up to and including (C-L) is available from the output data.
Figure 4-1: FFT Cache Hits: Partitioned vs. Nonpartitioned

Figure 4-2 reveals that the optimum partitioning for cache hit count performance with the FFT routine is 93/107. This partition results in an increase in total hits of 16%, and an increase in the hit ratio of 11%. However, the predictability is increased significantly as the LRU partition is reduced below 30. This results from the guaranteed hits in the 170+ line static partition.

A look at the savings in utilization for the above task shows significant advantages to partitioning. If we once again assume a 4 MIPS processor with a 150 nsec main memory access time, and a 30 nsec cache access time, the original task would require approximately 30.7 msec utilization over period P due to memory access. With a cache partitioning scheme, 13,702 hits are guaranteed for a savings of 1.644 msec or 6,577 instructions every period P. Thus by reducing the utilization of this task, 6,577 instructions are now available for servicing aperiodic tasks, or other periodic tasks.

4.1.2. Eigenvalue Calculation

The same nine cache sizes were used to evaluate 107,951 addresses from the eigenvalue computation. The results were very similar to those of the FFT simulation. An initial guess for a partition of 100/100 was made for the same reasons discussed in the previous section. Data similar to that depicted in Figure 4-2 showed that the optimum cache performance could be obtained with a partition of 121/79 (LRU/Static), resulting in an increase in the hit ratio of 6%. However, significant increases in predictability occur for LRU partitions less than 50.

4.1.3. Matrix Factorization

The same nine cache sizes were used to evaluate 61,119 addresses from the matrix factorization. A summary of the results is shown in Figure 4-3. As can be seen in the figure, the hit count makes significant progress for caches of all sizes. From these data, it is unlikely that any partitioning of cache of size 200 will be of much benefit.

Figure 4-3: Matrix Fact: Cache Hits for Size C

Once again, partitioning simulation was only performed for a cache size of 200. A summary of the results for cache size C=200 is given in Figure 4-4. The optimal partitioning division is 191,9. However, this scheme provides a negligible improvement in the hit count and ratio. This phenomenon results from a fairly even distribution of instruction distances throughout the range of 1 to 200. This explains the continuous increase in the hit count as the LRU cache partition size is increased. This program fits into the category of programs that have a significant number of instructions with distance of C-ε, where ε is a small integer. Although there is no significant performance increase through the use of partitioning, a minimal reduction in the hit rate allows very good predictability. This is shown by the large number of hits in the fully static cache (i.e. LRU size = 0) in Figure 4-4.

4.2. Traces without I/O

As a general statement, address traces in this section indicated fairly compact code, with higher hit ratios. This results from the elimination of a significant amount of straight-line code implementing the SCANF/C routine which was used to perform the I/O in the previous section. Although this I/O code was executed frequently, the instruction distance was large enough to result in very few hits. With the increased hit ratio due to program trace behavior, increases in hit ratio through partitioning are generally
lower (with the exception of very small caches). However, significant improvement in predictability is achieved.

4.2.1. FFT Algorithm
Eight cache sizes were selected for the second cache LRU simulation. The set of (2, 4, 8, 16, 32, 64, 128, 256) was inserted into the simulator, and a total of 47,347 addresses were used. A summary of the results is shown in Figure 4-5. The cache sizes were changed to more closely model realistic cache designs. Unlike the previous FFT simulation, this simulation shows steady improvement in the hit count as the cache size increases. Partitioning schemes were selected as shown in Figure 4-5, and the associated increase in the hit count is shown by the dashed line. This FFT module made more efficient use of the larger cache sizes. Recall that the first simulation showed very little improvement in hit count from cache size 100 to cache size 200. However, this routine has an increase of 9000 hits, implying less "wastefulness" of cache. This is seen by the reduced gain in hit count for the partitioned 256-line cache.

During this simulation pass, all of the optimal partitioning schemes were determined by data similar to that presented in Figure 4-6. This is a summary of all possible partitions for a cache of total size 256. Similar data were compiled for cache sizes 2, 4, 8, 16, 32, 64, and 128. Figure 4-6 shows the optimum cache performance can be achieved with a partitioning of 170/86 resulting in an increase in total hits of 5.3%, and an increase in the hit ratio of 4%. However, the predictability is extremely low for this partition, and can be greatly improved with LRU partitions of less than 100.

4.2.2. Eigenvalue Calculation
The same eight cache sizes were used to evaluate 75,051 addresses from the eigenvalue computation. Once again, the results were very similar to those presented in the FFT section. Unlike the FFT simulation, optimal cache performance for the eigenvalue computation in a cache of size 256 was achieved with a partition of 256/0. This results from a gain of almost 4000 hits when the LRU partition increases from 235 to 256 lines.

Figure 4-7 is a summary of the possible partitioning of a cache of size 128. The steep rise in the LRU partition hit count at 52 is due to a loop of length 52 which now fits into the LRU partition. There is a corresponding drop in the static partition hit count. Once again, although the optimal cache performance partition is 52/76 resulting in an increase in total hits of 4.3%, cache predictability increases greatly for LRU partitions which are less than 52 (shown by the hits indicated by the dotted line).

4.2.3. Matrix Factorization
The same eight cache sizes were used to evaluate 52,818 addresses from the matrix factorization. A summary of the results is shown in Figure 4-8. Once again, the hit count is higher than in pass one due to the straight-line code of the SCANFC routine. Partitioning schemes were selected as shown in Figure 4-8, and the associated increase in the hit count is shown by the dashed line. As with the previous matrix factorization simulation, there is very little performance gain from partitioning, but the cache predictability is significantly increased.

As Figure 4-8 indicates, there was negligible gain for cache sizes 128 and 256. This results both from a well distributed hit count in the LRU partition, as well as a lack of high frequency instructions for the static partition. Most of the high frequency instruc-
4.3. Partitioning Cache in a Real-Time System

Optimal cache performance is obtained by partitioning cache at the peak hit count for each task in the task set, and clearly if cache performance was the only concern in the system design this would be the approach taken. However, guaranteed deadlines is the principal concern for most real-time systems. As mentioned in the introduction, this requires keeping the total utilization below a specific bound. If the task set utilization is above that bound, the cache partitioning scheme can be used to lower the utilization.

Recall that task utilization is defined as the worst-case execution

time (WCET) divided by the period (P) of the task. Therefore, a reduction in the (WCET) of t reduces the utilization by t/P, and it can be seen that the improvement in utilization is both a function of the time saved, and the period of the task. The WCET is reduced by inserting instructions along the worst-case execution path into the static partition of the cache. As mentioned in the introduction, these instructions are preloaded in the cache as the task is swapped in, and are therefore guaranteed to be resident across preemptions. The new WCET can be calculated by replacing the main memory access time with the cache access time for each of these instructions. Hence, by strategically choosing tasks with a high frequency (1/P), and instructions with a high access rate along the worst-case execution path to load into the static partition, it is possible to significantly reduce the total utilization.

The cache partitioning scheme is therefore a two step approach. First the optimal cache performance partition is identified for each task. As a result of this partitioning, there will be some number of predictable cache hits due to static partitions. These cache hits are then used in determining the initial WCET. The task set can then be checked to see if it is schedulable with guaranteed deadlines. If the task set is not schedulable, the partitions of the highest frequency tasks are then altered to increase predictability by moving more instructions along the worst-case path into the static partition. Each increase in the static hit count is weighted by the frequency of the task. This process is continued until the reduction in WCETs is significant enough to make the task set schedulable.

5. Design of a Logically Partitioned Cache

5.1. Overview

Traditionally, cache partitioning schemes have been used to physically partition caches into data and instruction subsections. The two cache subsections are designed as physically separate cache systems. Physical partitioning allows parallel access to the two smaller, and hence faster, caches. This provides an improvement in cache bandwidth, as well as cache access time. Some systems implement the instruction cache as a read-only cache, and prohibit self-modifying code. This approach simplifies the instruction cache design by eliminating the write-back logic. Ac-
companioning the data/instruction partitioning scheme, there have been debates about the inflexibility of the resulting cache structure. Due to the physical division, it is impossible to tune the partition to the currently executing task.

Logical partitioning, on the other hand, is meant to create cache divisions which can readily be tuned to the task currently executing. Based on the specific task's program behavior parameters, the cache is partitioned into static and LRU regions. This partitioning requires very little hardware assistance.

Cache coherence can be a concern when designing partitioned caches. Coherence involves having multiple copies of the same data in different partitions. If one of these copies is updated, and the other is not, there is a coherence problem and one task is using old data. However, in this cache partitioning scheme, coherence will not be a problem, because while the cache is logically partitioned into two regions, the entire cache is examined to determine when a hit has occurred. Therefore, there will never be more than one copy of an instruction in cache.

5.2. Hardware

Logical partitioning provides a partition based on the memory access requirements of each task executing in the multi-tasking environment. The partition is chosen to maximize the task hit count and predictability. The partitioning scheme is then superimposed on a single cache structure by modification of the cache line. Specifically, a two bit "static field" is added to the cache line, and set when that line is to be included in the static partition. Use of the second bit is discussed in the section on context swaps.

The high-level cache design is shown in Figure 5-1, with the control flow chart in Figure 5-2.

![Figure 5-1: Partitionable Cache Design](image)

The cache design only differs from the design presented in [10] by the extra flag bits on each cache line. This flag acts like a lock to the LRU replacement hardware. If the flag is set, the line is not considered for replacement. However, on every hit, the LRU bits are maintained as they would be in the LRU section, regardless of the flag setting.

![Figure 5-2: Partitioned Cache Control Flow Chart](image)

5.3. The Effects of Context Swaps

So far, we have discussed the program behavior in an environment of a single cache user. We now consider how the cache should behave when a context swap occurs before the task has completed. Assume for this discussion that task-2 has been running, and task-1 preempts task-2. Figure 5-3 shows the possible configurations of the cache at the time that task-1 starts to execute. The question to answer is how the cache should behave in each of these scenarios.

![Figure 5-3: Possible Cache Configurations at a Context Swap](image)

In each case in Figure 5-3, the cache is either fully occupied by task-1 instructions, or task-1 occupies part of the cache, and the remainder is uncommitted. If there is an uncommitted region (cases a, b, and c), the LRU replacement algorithm will replace these lines first, simply because they are the oldest lines in cache. This is expected, and desired. However, the question remains as to how the cache should handle the remaining partitions. The choices are simple: treat all lines owned by task-1 equally; or give priority to one task-1 partition over the other.

Clearly there is no foolproof method of knowing which lines should be replaced first. However, there is some intrinsic knowledge available to aid in the decision. Lines of cache in the
static partition have been placed there because of their high frequency of occurrence. Lines in the LRU section, may or may not have a high frequency of occurrence. Indeed, some of these lines will only be needed once. For this reason, the static partition of task-1 should be given higher priority than the LRU partition. This is done during the context swap. When task-1 is swapped out, all lines which were flagged as foreground static, are converted to background static. Lines that were foreground LRU are converted to background LRU, and lines that were background static are converted to background LRU. Once the categories have been changed, the LRU algorithm assures the tasks are handled in the correct order. A summary of this transition is given in Figure 5-4.

<table>
<thead>
<tr>
<th>Before Context Swap</th>
<th>After Context Swap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static Field</td>
<td>Static Field</td>
</tr>
<tr>
<td>BGFG LS State</td>
<td>BGFG LS State</td>
</tr>
<tr>
<td>0 0 0 Background LRU 0 0 Background LRU</td>
<td></td>
</tr>
<tr>
<td>0 1 Background Static 0 0 Background LRU</td>
<td></td>
</tr>
<tr>
<td>1 0 Foreground LRU 0 0 Background LRU</td>
<td></td>
</tr>
<tr>
<td>1 1 Foreground Static 0 1 Background Static</td>
<td></td>
</tr>
</tbody>
</table>

* States 10 and 11 will be occupied by the new task.

**Figure 5-4:** Static Bit Transition at Preemption Context Swap

There is, however, a difference between the scheduler swapping out a task due to task completion or a task entering the wait state, and a task that is swapped out due to preemption. When task 1 is swapped out due to preemption, it is assumed the task will resume at the completion of the preempting task (unless preempted by another task). For this reason, it is desirable to keep as much information in the cache as possible. A task which runs to completion is not expected back in the immediate future. For this reason, when a task is swapped out for reasons other than preemption, its static field bits are all set to background LRU (lowest priority). In this case, fields that were background static remain background static. A summary of this transition is given in Figure 5-5. To understand the reasoning behind this, consider the following example.

<table>
<thead>
<tr>
<th>Before Context Swap</th>
<th>After Context Swap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static Field</td>
<td>Static Field</td>
</tr>
<tr>
<td>BGFG LS State</td>
<td>BGFG LS State</td>
</tr>
<tr>
<td>0 0 0 Background LRU 0 0 Background LRU</td>
<td></td>
</tr>
<tr>
<td>0 1 Background Static 0 1 Background Static</td>
<td></td>
</tr>
<tr>
<td>1 0 Foreground LRU 0 0 Background LRU</td>
<td></td>
</tr>
<tr>
<td>1 1 Foreground Static 0 0 Background LRU</td>
<td></td>
</tr>
</tbody>
</table>

* States 10 and 11 will be occupied by the new task.

**Figure 5-5:** Static Bit Transition at Completion Context Swap

Three tasks with priorities ranging from 1-3, with 1 the highest, are competing for the CPU, and task-3 is being executed. Task-1 preempts task-3, and runs to completion. However, before task-1 relinquishes the CPU, task-2 becomes ready. Since task-3 is expected to resume when task-2 completes, it should have priority over the cache lines owned by task-1. This priority is generated by telling the cache that task-1 has completed, at which point all lines owned by task-1 are made background LRU, and the LRU bits are set to the oldest value. In the rare case that task-1 has such a short period that it is expected to resume before task-3 does, the scheduler can tell the cache that task-1 has been "preempted", at which point task-3 lines are converted to background LRU with the oldest possible LRU setting, and task-1 lines are set to background static and LRU with no alteration of the LRU bits. This provides task-1 priority over task-3.

5.A. Compiler Assistance

In earlier sections, we have presented a model for program behavior, and discussed a means of determining the cache partition based on this program behavior. In the last section, we presented the cache structure necessary to perform the partition. However, what is missing, is the information to instruct the cache as to which instructions are static, and which instructions are LRU. Clearly, it is necessary to have some type of communication to the cache. There are basically two ways in which this can be accomplished: vertically and horizontally.

The vertical approach requires in-line commands to the cache structure. A cache command would be issued at the start of the load to the LRU partition. The cache would continue to load lines of instructions flagging each as LRU, until a second cache command was received. At this time, the cache would start loading lines into the static partition. Hence, two additional cache commands are needed (in addition to those issued by the system scheduler).

The horizontal approach does not require additional commands in the instruction string, but rather would tag each instruction as LRU or static. One could think of this as being a one bit extension to the opcode field. As each instruction was brought into the cache, the cache would tag it according to this bit. Obviously, in the case where there were multiple instructions per line of cache, there would only be one tag per line. However, in this research, a cache line equal to one instruction was used.

The vertical approach is preferred when the program is very easily divided between the LRU portion, and the static portion. In this case, very few new instructions are added to the instruction stream, and the increase in memory used and execution time is minimal. However, if the instructions are not well behaved, it is possible that every line of cache could require a cache command to redirect the load. This is obviously an extreme case, and the typical case would lie somewhere in between these two extremes.

6. Conclusions and Future Work

This paper has demonstrated that cache design strategies for real-time systems have more information available to them than time-shared and batch processing systems. Due to the fixed task set and periodic nature of the tasks, program behavior information can be used to better the performance of the system. Specifically, this paper discussed a technique for logically partitioning a cache into two regions: static and LRU. The LRU region functions in the normal manner. Instructions loaded into the static region, however, remain here for the life of the task. This technique minimizes the cost to the overall hit ratio, while maximizing guaranteed cache hits which can be used to increase schedulability.

Simulations were performed for three software modules: an FFT routine; an eigenvalue routine; and a matrix factorization routine. The FFT routine experienced a 10% improvement in the hit ratio, while the eigenvalue routine saw a 6% increase. However, practically all combinations of code and cache size provided a significant increase in predictability with little or no loss in cache.
performance. Address traces for the same three routines, with all I/O related instruction addresses deleted, were also tested in simulated cache partitions, and the results were similar.

The high-level cache structure necessary to perform logical partitioning was presented. It differs from a standard cache design by a single two bit field referred to as the static field. A description of the use of this field is given, as well as the techniques involved in handling context swaps. Cache replacement priorities are assigned to lines in the cache based on ownership and expected scheduler decisions.

Future work in this area will focus on many aspects of improving the approach and application of this partitioning scheme. It is necessary to develop program behavior models that can be used directly to determine the partitioning scheme analytically. This study also limited the trace addresses to instruction addresses. Future studies will incorporate data addresses into the analysis as well.

As portions of the program are statically maintained in cache, the worst case execution time for a task decreases due to guaranteed cache hits. This reduced worst case execution time will result in better task schedulability. However, it is necessary to overcome the effects of cache reload-transients resulting from preemptions. This can be accomplished if the active region of the static partition is maintained across context swaps. Future work will examine techniques for identifying and preserving the active static partition, as well as determining the impact of static partitioning on scheduleable utilization. Current research is investigating a dynamic partitioning scheme which minimizes the overhead during context swaps. In general, the future work in this area will be exploring techniques for using information available about real-time task sets to develop more intelligent and predictable cache architectures.

Acknowledgments

The author would like to thank John Lehoczky, Lui Sha, Hide Takuda, Jay Strosnider, Ragunathan Rajkumar, Brinkley Sprunt and the symposium referees for their comments and insight on this work, as well as the IBM Systems Integration Division for their financial support through a doctoral fellowship.

References


