Comparing Caching Techniques for Multitasking Real-Time Systems

Steve Dropsho
dropsho@cs.umass.edu

Chip Weems
weems@cs.umass.edu

Computer Science Department
University of Massachusetts-Amherst

April 30, 1997

Abstract

Correctness in real-time computing depends on the logical result and the *time* when it is available. Real-time operating systems need to know the timing behavior of applications to ensure correct real-time system behavior. Thus, *predictability* in the underlying hardware operation is required. Unfortunately, standard, embedded cache management policies in microprocessors are designed for excellent *probabilistic behavior* but lack predictability, especially in a multitasking environment. In this article we examine the two popular cache management policies that support predictable cache behavior in a multitasking environment and quantitatively compare them. Using a novel application of an existing analytical cache model we show that neither policy is best in general and delimit the system characteristics where each is most effective.

1 Introduction

In real-time computing, correct operation depends on both the logical result and *when* it is available. Real-time systems have the characteristic that missing a timing constraint, or **deadline**, can result in a catastrophic failure.

The real-time operating system **(RTOS)** must schedule a complex set of tasks so they meet their deadlines [15]. To do so, the RTOS must have estimates of the resources required by each task and, in particular, their execution times. In systems that can experience catastrophic failures, the latter must be **worst-case execution time (WCET)** estimates that the task is guaranteed not to exceed. Only with the upper bound provided by the WCET estimates is it possible to guarantee safe schedules.

Unfortunately, there is considerable variability in the execution time of code due to caching in processors.

Standard microprocessors implement a *probabilistic* cache management policy that generally has excellent *average-case* behavior, but lacks predictability, especially in a multitasking environment. The lack of predictability forces WCET estimates that are overly pessimistic, possibly by an order of magnitude or more [4]. One solution is to bypass the cache and access memory directly. While this removes caching variability, it also restricts processor performance to the speed of main memory and limits the ability of real-time systems to track performance gains due to new processor technology.

Current research shows significant progress in predicting cache behavior for a single task [11]. However, complex real-time systems can involve multitasking with processes sharing the CPU via timeslicing [8]. Within a single process a compiler might be expected to predetermine the code's behavior with respect to the cache. Across multiple tasks with interrupts on a dynamic and changing schedule, accurate prediction of cache behavior is intractable, in general. However, by constraining the use of the cache somewhat we can have predictable cache behavior while still exploiting its virtues to enhance processor performance. The benefit is a much improved WCET estimate at the cost of a higher *average-case* execution time. While such a tradeoff is generally unattractive in non-real-time systems, real-time designers worried about WCET view it as a significant improvement.

This article analyzes two fundamental methods of managing caches for predictable behavior. The two methods are software-based and have been presented previously in the literature [12, 9]. We quantitatively compare the policies by leveraging an existing analytical cache model in a novel manner that allows exploration of system performance across a wide range of designs. Although currently limited to systems with a single level of cache, this analysis reveals that cache policy effectiveness is highly system dependent, and delimits the conditions where each policy works well.

2 Related Work

Much of the research in real-time predictable cache behavior has focussed on analyzing an application's behavior with respect to a uniprocess system that does not suffer interprocess contention for the cache [13, 10, 14, 7, 5].

For a multitasking environment, two basic methods have been proposed for controlling interprocess cache interference for predictable cache behavior. The first method relies on dividing the cache into distinct partitions and the other restricts where context switches can occur. Kirk and Strosnider [6] detail a hardware design for the

MIPS R3000 that allows a cache to be partitioned among processes. Mueller [9] implemented a compiler that partitions the cache strictly through software, via positioning of code. On the other hand, the Spring Real-Time System [12] controls where context switches can occur using a software technique that defines regions of code during which interrupts are masked. By assuming the cache is empty upon entry into one of these regions, techniques can determine the WCET cache behavior between the entry and exit points because the cache behavior is equivalent to a uniprocess system during the interval. This article extends the previous work by directly comparing the relative effectiveness of these two methods to enhance system performance.

3 Cache Management Policies

The variety of fundamental cache management techniques is limited by the observation that in a multitasking environment processes can only share a cache resource in three ways: spatially, temporally, or both spatially and temporally. The key to predictable caching is in controlling contention for cache locations in a predictable manner between processes.

Spatial Only Sharing (SOS). Spatial only sharing processes use a common set of cache locations, but do not maintain **live** (i.e., active) entries simultaneously. This policy considers the cache empty of previous state between execution periods of a process. The cache may be physically flushed or only considered flushed for WCET estimation purposes. The advantage of this policy is the use of the full cache per process. The disadvantage is the overhead to warm-up the cache each time a process is swapped in.

Temporal Only Sharing (TOS). Temporal only sharing processes use independent sets of cache locations and maintain live entries simultaneously. In this policy the cache is partitioned and each process is allocated to one partition. The advantage is that state is maintained between execution periods of a process, eliminating the many cache warm-up phases suffered by SOS. The disadvantage is that each process is restricted to a smaller cache, thus, increasing the miss ratio.

Spatial and Temporal Sharing (STS). Allowing both spatial and temporal sharing is the standard cache management policy used in microprocessors. For real-time, achieving predictable behavior would require overhead to save and restore the cache state between consecutive runs of a process. We do not explore this option here.

3.1 Policy Implementations

3.1.1 SOS Implementation

An SOS policy assumes the cache is flushed between consecutive runs of a process. If the cache is physically flushed then overhead is incurred, however this may be as little as a single cycle if the processor supports fast reset of the valid bits in the cache tags. If the flush is only logical there there is no flushing overhead since the data between context swaps will remain. For our discussion we shall assume the overhead of flushing the cache is zero cycles.

Due to cache warm-up phases caused by flushing, an SOS policy is sensitive to the context switch frequency. Obviously, an SOS policy is most attractive when the context switch frequency is low. The rate of context switches can be controlled by defining regions of code in an application that will run without interruption. By turning off interrupts in such a code segment, a compiler can accurately estimate the cache behavior between the entry and exit points. The larger the code region, the more efficient is the cache usage. This cache policy is precisely that of the Real-Time Spring System [12]. The reader may note the similarity of the SOS policy to using critical code regions for protecting access to shared variables and resources. For this reason, we will use the more descriptive term **critical code regions (CCR)** synonymously with **SOS policy**.

3.1.2 TOS Implementation

A TOS policy requires partitioning of the cache to prevent processes from conflicting on cache lines. Mueller [9] presents details of a software only method that *logically* partitions the cache, thus avoiding the need of hardware support that physical partitioning would require [6]. In his scheme the compiler breaks application code into blocks and maps them into memory such that the application becomes restricted to only a portion of the cache via the normal cache mapping process. The compiler adds branches to skip over the gaps created by the code positioning. Figure 1 graphically depicts how code is partitioned. We will use the term **partitioning** interchangeably with **TOS policy**.

Partitioning only works if each application maps into a cache region without contention from another application. In a real-time system there may be hundreds of tasks, however, this does not necessitate hundreds of partitions. Real-time tasks are typically grouped by priority levels. Audsley and Tinsdell [2] show that tasks at the same priority level are scheduled *non-preemptively* relative to *each other* in an otherwise preemptive system with FIFO scheduling at the same priority level. Thus, there only needs to be one partition per priority level. We assume 32 priority levels as Mueller does.

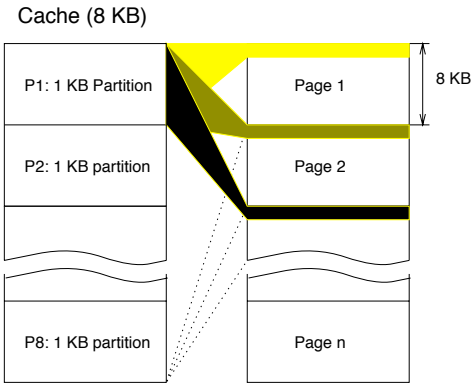

Figure 1: Memory and Cache Partitioning

4 Methodology

Analysis of the cache management policies depends on the applications being run, the scheduling of the tasks, and the cache size. Each of these three variables has a range of potential values resulting in a large number of combinations, each of which can be viewed as defining a particular system. We refer to this set of values as the **system space**.

4.1 Cache Model

Thiébaut et al. [17] developed an analytical model of fully associative caching behavior. The intended use of the model was to drive a synthetic address generator for cache simulations. They provide a method of measuring three parameter values- A, θ, and M_m- of actual applications that completely defines their cache behavior. They verify that the model generates synthetic traces that very closely mimic the cache behavior of the actual traces across the complete range of associativities from direct mapped to fully associative.

We make use of their cache model in a different manner, using the range of the three application-defining parameters to define the possible **application space**. Given a particular application definition, we then can generate cache miss rates for analytical system performance models. This allows easy exploration of thousands of design points in the range of two interesting system design parameters, cache size and context switch frequency, without costly detailed simulations. While there is no substitute for actual system simulation in analyzing a *particular* design, this analytical method provides first order approximations to system caching behavior and delimits regions of interest for more detailed study. We later support the analytical conclusions with results from detailed cache simulations of an MPEG compression application.

4.1.1 Cache Model Definition and Assumptions

The model gives the probability that a memory access will index an element not resident in a fully associative cache. In effect, this probability is the probability of a miss, commonly called the **miss rate**. Note, the miss rate is not equivalent to the **miss ratio**. The miss rate is the instantaneous probability that the next access will be a miss. Over an infinite number of accesses the miss ratio will asymptotically approach the lower miss rate.

Equation 1 defines the model. The value C_c is called the **critical cache size** and is defined in equation 2, x is the cache size. C_c marks the transition from the cache warm-up phase to steady state behavior. Equations 1 and 2 are motivated by previous work of Gillis and Weiss [1] on *hyperbolic random walks* and supporting empirical observations by Kobayashi and MacDougall [3] and Thiбault [16]. For caches smaller than the critical cache size ($x < C_c$) the miss rate equation is derived by assuming a uniform probability of $p = P/C_c$ for indexing any item in the cache, where P is the probability that the index is less than C_c, or $1 - P[hit\ index\ >=\ C_c]$. The reader is referred back to the original paper [17] for additional details.

It is important to note that the cache size is measured in *lines* rather than words. The effects of the cache line size are embodied in values of the application definition parameters A, θ, and M_m.

$$Pr[hit\ index > x] = MissRate(A, \theta, x) = \begin{cases} \frac{1 - \frac{x}{C_c}(1 - \frac{1}{\theta}) - K}{1 - K} & \text{if } x \leq C_c \\ \frac{\frac{A^\theta}{\theta} x^{(1-\theta)} - K}{1 - K} & \text{if } C_c < x \leq M \\ 0 & \text{if } M_m < x \end{cases} \tag{1}$$

$$C_c = A^{\theta/(\theta-1)} \tag{2}$$

$$K = \frac{A^\theta}{\theta} M_m^{(1-\theta)} \tag{3}$$

- θ descibes the spatial locality of the memory reference pattern and determines the probability of making large jumps. As θ increases, the probability of visiting a new cache line diminishes, i.e., its locality increases.
- A determines the average size of the addressing neighborhood. A smaller value of A decreases the size of the neighborhood, or working set, i.e., its locality increases.
- M_m represents the size of an application's address space. A cache greater than M_m lines will have a zero miss

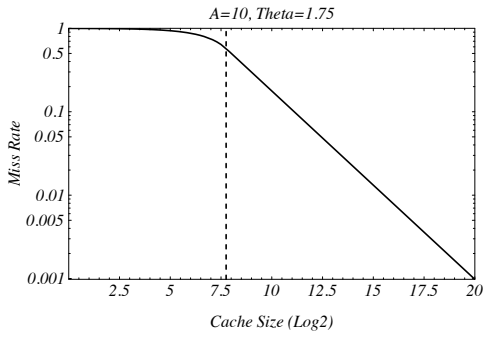

Figure 2: Log Log Plot of Miss Rate to Cache Size

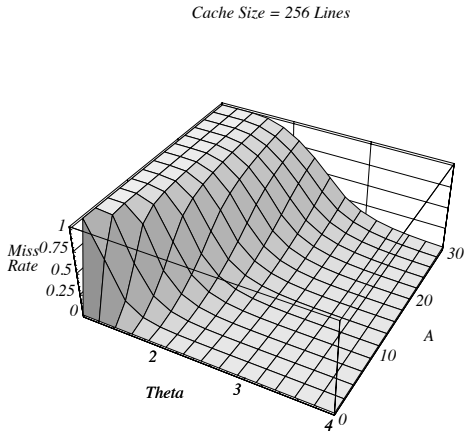

Figure 3: Miss Rate vs. the Parameters A and θ

rate in the steady-state since the entire application fits in cache. For simplicity, our discussion will initially assume M_m is infinite and, thus, $K = 0$ (because $\theta > 1$). Later we will include the effects of a finite M_m.

A log-log plot of the miss rate versus cache size is shown in figure 2. The vertical line marks the critical cache size. The basic shape of this plot is seen in the cache behavior of actual codes.

Figure 3 shows how the miss rate varies with A and θ. Thiébault et al. measured typical values in the range of 2 to 17 for A and 1.3 to 3.1 for θ. For worst-case paths in real-time applications we should expect small values of θ and large values of A. For the initial discussion we arbitrarily use a value of 1.75 for θ and 10 for A which translates to a pessimistic miss rate of 17.8% for a cache size of 1024 lines. The effects of varying A and θ are explored in the second half of the results section.

4.2 System Performance Equations

Equation 4 describes the performance in cycles per instruction of a simple single-issue processor accounting only for steady-state cache miss overhead. The first term is the base issue rate, 1 instruction per cycle. The second term accounts for cache miss overhead. We assume a 20 cycle miss penalty to access main memory and calculate the cache miss rate from equation 1 given the parameters A, θ, and M_m.

$$RP = 1 + MissRate(A, \theta, M_m, CS) \times MissPenalty \tag{4}$$

If traces are sufficiently long the the steady-state equation is a good approximation to the overall miss ratio.

However, real applications have finite run-times and the warm-up phase may have significant impact on system performance. Certainly with a CCR policy this will be the case. Incorporating cache warm up behavior prevents us from directly using the Thiébault et al. cache model. We can, however, leverage the model indirectly.

Equation 4 still gives the relative processor performance, but the function to calculate the miss ratio must change. Instead of an equation, we use a look up table of miss ratios derived as follows.

The number of instruction references required to experience I total misses is given by equation 5. The inverse of the miss rate is the number of references on average that will be required before the next miss is generated. Summing these values gives the total references required to generate I misses. As shown in equation 6, we can use equation 1 for the instantaneous miss rate by using the current number of unique cache entries (which is equivalent to the number of misses at that point) as the instantaneous effective cache size and then capping the cache size at its full value if $I > CacheSize$. Thus, after each miss the effective cache size grows by one line and this new value is used to determine the number of references before the next miss.

$$References(A, \theta, M_m, CacheSize, I) = \sum_{i=1}^{I} 1/MissRate_i(A, \theta, M_m, CacheSize, i) \tag{5}$$

$$MissRate_i(A, \theta, M_m, CacheSize, i) = \begin{cases} MissRate(A, \theta, M_m, i) & \text{if } i < CacheSize \\ MissRate(A, \theta, M_m, CacheSize) & \text{otherwise} \end{cases} \tag{6}$$

Since it wouldn't be practical to generate a table for all potential numbers of references, we create a small thirty-two entry table of total reference counts by calculating $References(A,\ \theta,\ M_m,\ CacheSize,\ I)$ using miss counts of $I = 2^x$ where x ranges from [0..31]. The miss ratios are then easily calculated by dividing I by $References(A,\ \theta,\ M_m,\ CacheSize,\ I)$. Interpolating between the two entries in the reference count table that bound the number of memory references gives a good approximation to the miss ratio. In the next section we compare analytical results modeling an MPEG application to the simulation data.

System Parameters	
A	10
θ	1.75
M_m	infinite
Full Cache Size	$32\times$ Partition Size
Cache Partition Size	**Vary:** $[1..2^{15}]$ **lines**
References Between Context Switches	**Vary:** $[1..2^{30}]$ **references/switch**
Cache Miss Penalty	20 CPU cycles

Figure 4: System Design Parameters

5 Results

5.1 Effects of Cache Partition Size and Context Switch Rate

To explore fully the system design space we have the parameters shown in figure 4. The first two parameters, A and θ, define the cache behavior of the software application being run. The values are arbitrarily fixed at $A = 10$ and $\theta = 1.75$ but reflect code with relatively low locality of reference as might be expected along an application's worst-case execution path.

Varying only these two parameters we can compare the cache policies to each other as shown in figure 5. One axis represents the cache *partition size* in log_2 scale with the full cache size being 32 times the partition size. The context switch rate is defined relative to the number of references between switches. The switch rate varies from 1 switch per reference to 1 switch per 2^{30} references, also given in a log_2 scale. Standard context switch overhead (e.g., maintaining process queues, etc.) is ignored. The dark patch marks a reasonable design space for actual systems. In this and all subsequent graphs this patch covers a rectangular region bounded by system interrupt rates from 1 per 128 memory references to 1 per 128 K memory references and cache sizes from 2^7 lines to 2^{12} lines. For a cache line size of 32 bytes this translates to a cache size range from 4 KB to 128 KB.

Figure 5 graphs the ratio of the processor's performance using a partitioning policy relative to using CCR. A value greater than one implies partitioning is less efficient. A somewhat clearer pictorial of the same information is shown in figure 6 which delimits where a each policy is most effective. The intermediate bands define performance advantages of less than 2%.

Figure 7 refines the previous figure by delimiting regions where the system outperforms the other by 33% or more. The lightest region is where a CCR system is at least 33% better and the darkest region greatly favors partitioning.

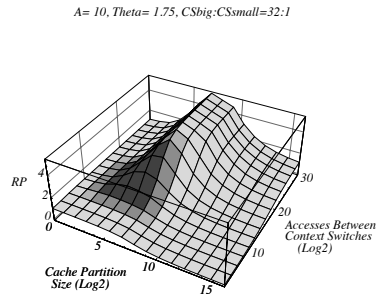

Figure 5: Performance ratio Partitioning:Critical Code Regions

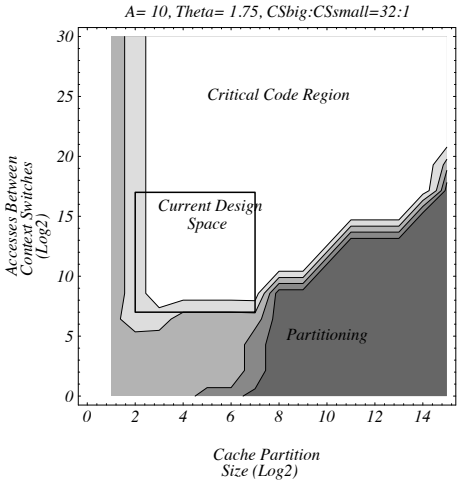

Figure 6: Regions delimiting best policy

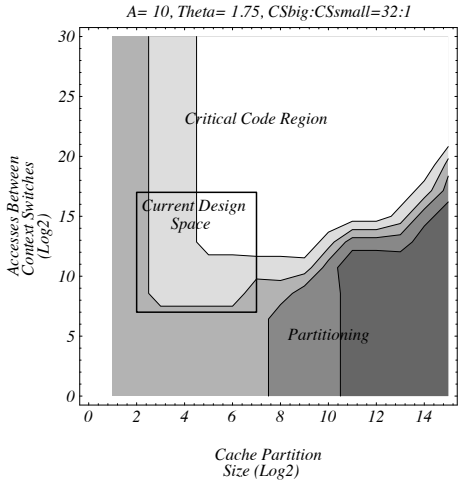

Figure 7: Regions defining 33+% advantage

The middle grey band defines a performance advantage of less than 5%. The remaining two bands define greater than 5% but less than 33% advantage.

From the plots, where the context switch frequency is high relative to the cache size partitioning is more efficient than CCR. However, a CCR policy can have a significant performance advantage at lower context swap frequencies.

5.2 Verification

We demonstrate the accuracy of the model by comparing analytical and simulation results on an MPEG compression application written in C. The parameters of the instruction stream for the MPEG application were measured to be $A = 9.37, \theta = 2.82$, and $M_m = 3963$ lines. Figure 8 is the analytically derived contour plot comparing the two cache policies. Figure 9 is a similar plot using data from simulation. To improve simulation speed, the associativity

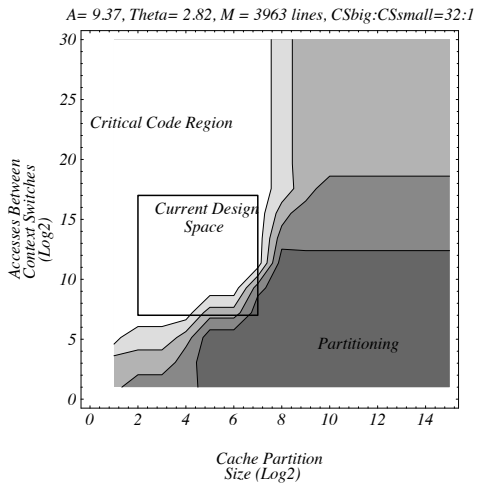

Figure 8: MPEG example, analytical

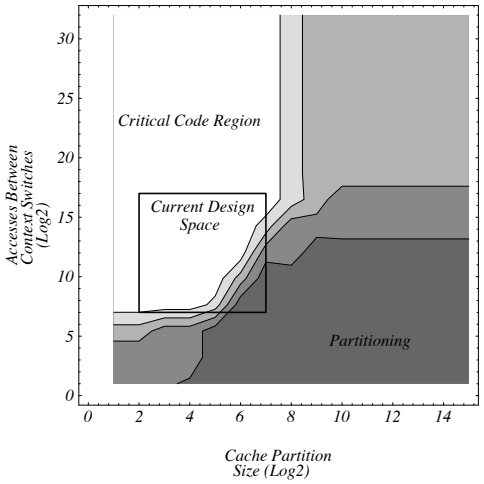

Figure 9: MPEG example, simulation (8-way)

was limited to 8-way to approximate a fully-associative cache.

By inspecting the graphs we can see that the simulation data correlates well with the analytical data except at the smallest cache sizes. Thus, the analytical model accurately portrays the behavior of the cache policies and greatly reduces the time for estimating system performance (the analytical models run in a couple of seconds versus 10 hours of cache simulations for this example on a two processor 100 MHz SGI Onyx).

The effect of a finite M_m is seen in the upper right corner of the plot in figure 8 where the advantages of either policy abruptly diminish just around where the partition size equals M_m. The apparent clipping of the performance advantages after this point is because the miss rate falls dramatically for a partitioning policy and likewise for a CCR policy with a sufficiently low interrupt frequency to allow the entire application to be brought into cache.

The figures 8 and 9 reflect caches with high associativity. Figure 10 is a slight blowup of figure 9 for comparison with figure 11 that graphs simulation data of the direct-mapped case. While the miss rates differ between the direct-mapped and 8-way data the relative performance of the two cache policies remains the same.

We correlate the calculated miss ratios to those of simulation in Figure 12 which plots the miss ratio of the MPEG simulation at various cache sizes as a dotted line and the analytical equivalent as a solid line. Except for very small cache sizes the match is good. The mean error is 16.2%, standard deviation is 32.1%, and the maximum error point has a 107% error. This relatively large deviation at the smallest (and largely uninteresting) cache sizes accounts for some of the differences between figures 10 and 8.

The mismatch at small cache sizes stems from a fundamental assumption in the model that the misrate for one line is quite high, very near 1.0. For real caches that have large cache lines this is not true in practice. For example,

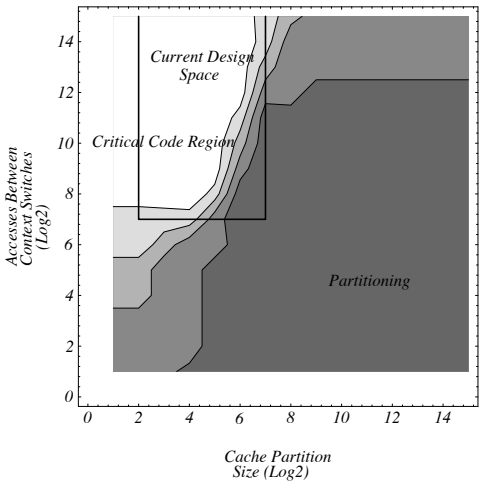

Figure 10: MPEG example, simulation (8-way)

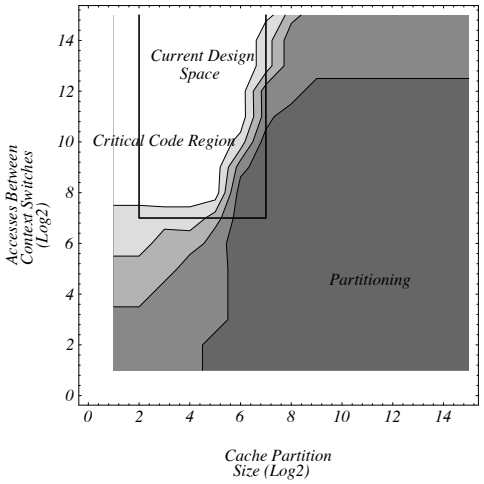

Figure 11: MPEG example, simulation (direct-mapped)

the MPEG instruction cache simulation with a single 8-byte cache line experienced only a 60.1% miss ratio rather than the 97.0% calculated by the model. To correct the model for cache line effects we introduce a normalizing factor shown in Eqn 7 using the miss ratio of a single line as a boundary condition for its value. For this example the factor is 0.62 (0.601/0.970). The resulting analytical miss ratios are shown in figure 13. A and θ remain the same. Regraphing the contour graph comparing the normalized model to the simulation data is shown in figure 14. The model matches better at the smaller caches. The mean error is -9.4%, standard deviation 16.9%, and the maximum error point is 47%. While normalization appears to help the model and adjusting A and θ for it can create a tighter fitting curve ($A = 13.5, \theta = 3.15$ gives mean error -7.1%, standard deviation 12.8%), we have not yet explored its ramifications to the underlying mathematical theory, so all subsequent discussion will use a normalization factor of 1.0, i.e., the original model.

$$MissRateNormalized(A, \theta, M_m, CS) = ScaleFactor * MissRate(A, \theta, M_m, CS) \quad (7)$$

5.3 Effects of Varying Application Parameters A and θ

In this section we consider the effects of varying the two application parameters A and θ to explore how the policies fare across a wide range of application types.

As mentioned before, the parameters A and θ describe an application's memory reference behavior. Figure 16 is an array of figures generated with various values for A and θ with M_m again infinite. A varies in the graphs

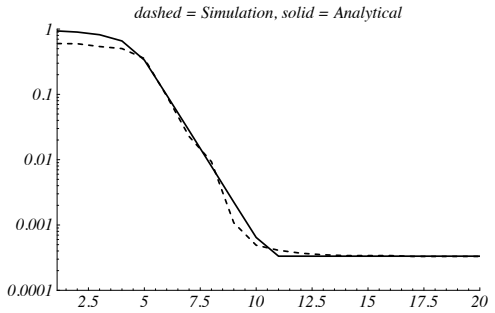

Figure 12: MPEG Miss Ratios

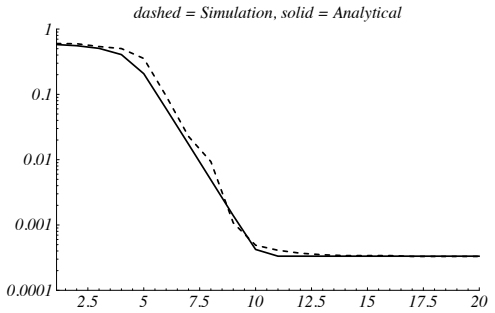

Figure 13: MPEG Miss Ratios, normalized model

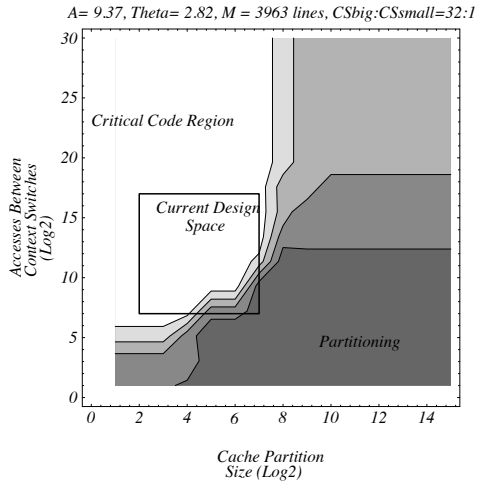

Figure 14: MPEG Analytical, using normalized model

Miss Rates: Cache Size of 1024 Lines			
A	1	10	19
θ			
1.50	2.08%	65.88%	95.02%
1.75	0.32%	17.75%	54.58%
2.00	0.05%	4.88%	17.63%

Figure 15: Application Miss Rates

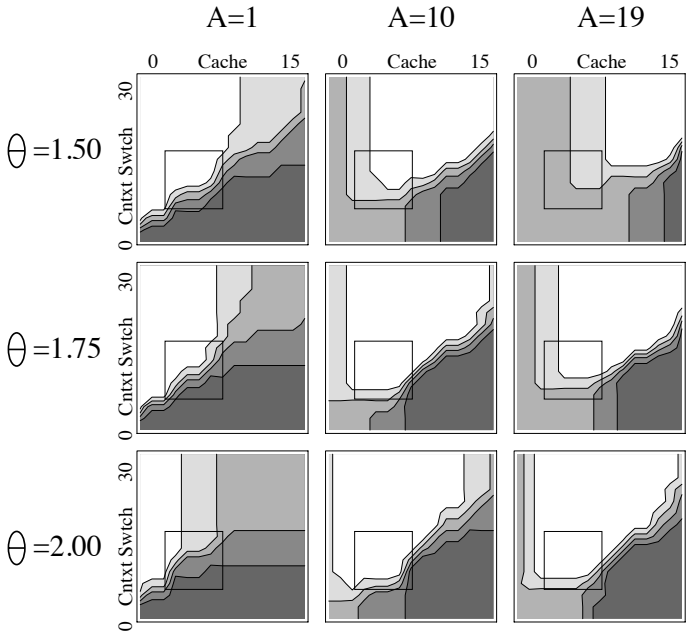

Figure 16: Varying A and θ

horizontally from left to right taking on the values 1, 10, 19. θ varies vertically from top to bottom taking on the values 1.5, 1.75, 2.0. Although axis markings are not shown, for reference the middle graph is a reprint of figure 7 at the same scale and the same application parameters $A = 10, \theta = 1.75$.

Figure 15 shows the miss rates for each combination of parameters at a total cache size of 1024 lines (equivalently, a partition size of 32 lines in the graphs). The data shows that quite a wide range of memory reference behavior is being represented.

From the nine graphs, as A grows large (left to right) indicating *less* locality of reference in applications, the differences between the two policies lessens in the delimited region. With less locality the application cannot exploit either policy effectively in that region. Conversely, as θ increases (top to bottom) indicating *greater* locality of reference, the policies strongly differentiate themselves.

Figure 17 is the same set of plots but now M_m is set to 2048 (2^{11}) lines. Note the similarity between this figure and figure 16 in the lower left quarters of each of the plots. The effect of a finite address space is most visible at larger cache sizes and low context swap rates, hence at the upper right corners. Notice how the effects of A and $theta$ are restricted in the finite memory plots. This is due to the mathematical effects of K in equation 1 that effectively squeezes the infinite range of cache sizes into the range $[0..M_m]$ lines with the compression increasing nonlinearly as the cache size approaches M_m. Thus, The plots in figure 16 using an infinite M_m are accurate for a broader range

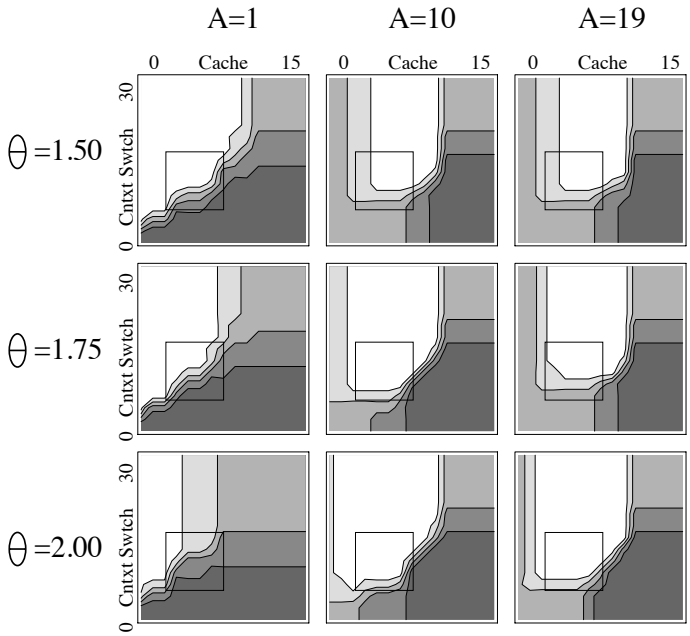

Figure 17: Varying A and θ with $M = 2048$

of applications, and we know the effect of a finite M_m is to clip the performance advantages at the upper right corner of the plots and to compress performance advantage regions around cache sizes of M_m.

6 Conclusion

We have compared two basic cache management techniques- partitioning and CCR- that eliminate the problem of interprocess cache contention to improve predictions of worst-case execution times in real-time systems.

Making novel use of an analytical cache model, we were able to analyze the effectiveness of each cache policy over a wide range of system parameters, revealing that each policy is most effective under particular system configurations with neither policy being superior overall. Thus, real-time systems designers must analyze the target set of applications and the anticipated schedule to determine which policy will provide higher overall system performance.

Future work includes extending the cache model to account for second level caches and applying the cache policies toward predictable TLB behavior. The similarities of TLBs to caches indicate similar tradeoffs will be encountered when tuning real-time system performance.

References

[1] J.E. Gillis an dG.H. Weiss. Expected Number of Distinct Sites Visited by a Random Walk with an Infinite Variance. *J. Math. Phys.*, pages 1307–1312, April 1970.

[2] N.C. Audsley and K.W. Tindell. On Priorities in Fixed Priority Scheduling. Technical Report TR 95-???, Dept of CS, Uppsala University of Sweden, May 1995.

[3] M. Dobayashi and M. MacDougall. The Stack Growth Function: Cache Line Refence Models. *IEEE Transactions on Computers*, pages 789–805, June 1989.

[4] Steve Dropsho. RISC Processor Worst-Case Execution Time Penalties. Technical Report TR-95-110, University of Massachusetts- Amherst, 1995.

[5] Christopher A. Healy, David B. Whalley, and Marion G. Harmon. Integrating the Timing Analysis of Pipelining and Instruction Caching. *Proc. of the IEEE Real-Time Systems Symposium*, December 1995.

[6] D.B. Kirk. SMART (Strategic Memory Allocation for Real-Time) Cache Design. *Proceedings of the Tenth IEEE Real-Time Systems Symposium*, pages 229–237, December 1989.

[7] Jyh-Charn Liu and Hung-Ju Lee. Deterministic Upperbounds of the Worst-Case Execution Times of Cached Programs. *IEEE Real-Time Systems Symposium*, 1994.

[8] J.J. Molini, S.K. Maimon, and P.H. Watson. Real-Time System Scenarios. *Real-Time Systems Symposium*, pages 214–225, December 1990.

[9] Frank Mueller. Compiler Support for Software-Based Cache Partitioning. *ACM Sigplan Workshop on Languages, Compilers and Tools for Real-Time Systems*, pages 125–133, June 1995.

[10] Frank Mueller and David B. Whalley. Fast Instruction Cache Analysis via Static Cache Simulation. Technical report, Florida State University, 1994. To appear in Proceedings of the 28th Annual Simulation Symposium, April 1995.

[11] Frank Mueller, David B. Whalley, and Marion Harmon. Predicting Instruction Cache Behavior. *ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems*, June 1994.

[12] D. Niehaus, E. Nahum, and J.A. Stankovic. Predictable Real-Time Caching in the Spring System. *IFAC Real-Time Programming*, pages 79–83, 1992.

[13] D. B. Whalley R. Arnold, F. Mueller and M. Harmon. Bounding Worst-Case Instruction Cache Performance. *IEEE Symposium on Real-Time Systems*, pages 172–181, Dec. 1994.

[14] Jai Rawat. Static Analysis of Cache Performance for Real-Time Programming. Technical Report TR93-19, Iowa State University of Science and Technology, November 1993.

[15] J.A. Stankovic. A Serious Problem for Next-Generation Systems. *Computer*, 1988.

[16] Dominique Thiebaut. On the Fractal Dimension of Computer Programs and its Application to the Computation of the Cache Miss-Ratio. *IEEE Transactions on Computers*, pages 1012–1026, July 1989.

[17] Dominique Thiebaut, Joel L. Wolf, and Harold S. Stone. Synthetic Traces for Trace-Driven Simulation of Cache Memories. *IEEE Transactions on Computers*, pages 388–410, April 1992.