Dynamic Round-Robin Task Scheduling to Reduce Cache Misses for Embedded Systems

Ken W. Batcher
Kent State University, Cisco Systems
batcher@cisco.com

Robert A. Walker
Kent State University
walker@cs.kent.edu

Abstract

Modern embedded CPU systems rely on a growing number of software features, but this growth increases the memory footprint and increases the need for efficient instruction and data caches. The embedded operating system will often juggle a changing set tasks in a round-robin fashion, which inevitably results in cache misses due to conflicts between different tasks. Our technique reduces cache misses by continuously monitoring CPU cache misses to grade the performance of running tasks. Through a series of step-wise refinements, our software system tunes the round-robin ordering to find a better temporal sequence for the tasks. This tuning is done dynamically during program execution and hence can adapt to changes in work load or external input stimulus. The benefits of this technique are illustrated using an ARM processor running application benchmarks with different cache organizations and round-robin scheduling techniques.

1. Introduction

A classic method for scheduling tasks in a multi-tasking embedded system is to use a *round–robin CPU scheduler*. This is especially true in low-cost systems that lack an operating system (OS) or use a very simple OS. While a full-featured OS such as Linux may use more complex techniques, a round-robin scheduler offers a simple and straightforward way to balance different tasks and share the CPU fairly in a low cost embedded system [5].

In many of these systems, commercial processor cores using instruction and data caches are common. These cores typically keep the cache sizes small (e.g. 4KB or 16KB) to reduce cost and power in the embedded environment. As a result, cache misses are common in these environments, a problem that is exacerbated by the limited memory bandwidth in these embedded systems. Further, even if the cache performs well when a single task is running, the situation is more challenging in a multi-tasking environment. Here the previously running tasks leave memory "footprints" in the cache, so as new programs displace the ones currently in the cache, they can cause a set of conflict misses that degrade the performance of newly scheduled tasks [8].

In many embedded systems, predictability is also very important — since embedded software is generally stable for the life of the application, such systems are expected to have stable performance as well. Unfortunately, embedded systems are often more sensitive to performance disturbances than general purpose systems, making predictability and stable performance a challenge. Further, real time processing constraints are common in embedded systems where some or all tasks have processing deadlines, adding to this challenge.

Fortunately, it is possible to address these embedded system challenges by tuning the OS. Our technique adaptively adjusts the round robin scheduler's task sequence order to reduce cache misses and improve the processor performance.

2. Related Work

There are many ways to organize the software in an embedded system. How does one determine what code structure is best to avoid cache misses in a multi-tasking environment? Researchers have developed both hardware and software techniques to address this problem. Our work relies on live execution profiling of the software as it runs. *Profiling* — a classic method introduced by Knuth — allows the behavior of the running program to be analyzed and used for feedback to optimize the performance [6].

Pettis and Hansen created an effective software technique for reordering code at the basic block level, branch alignment and procedure reordering [2]. This can be used as a compile time optimization to effectively reduce cache misses. Procedure placement using temporal ordering was improved upon by Gloy and Smith [3], using profile-driven code-placement with a weighted call graph to derive an improved procedure ordering.

Kalamatianos and Kaeli introduced Temporal Based Procedure Reordering [13] which involves constructing a conflict miss graph and coloring algorithm to produce an improved code reordering for instruction caches. As input to their algorithm they use code analysis and trace driven simulation to discover an improved layout of a program in the available memory space. Ghosh uses cache miss equations to drive compiler optimization decisions for improving cache performance [12]. Samples uses profile driven compilation in his work to help optimize code layout [4]. He describes how the software is instrumented to collect profile data, which is used to create a more optimized layout to avoid cache misses.

These software techniques all improve code performance by code analysis and evaluation of memory access patterns. The software is physically recompiled and / or linked to create a more optimal memory layout to avoid cache misses.

Hardware methods include creating cache structures more suited for reducing cache conflicts. Lee, et. al. use a temporal and spatial cache system based on time interval to optimize performance [1]. Their technique uses a small block size direct-mapped cache more suited for temporal locality and a fully associative buffer with large block size to address spatial locality.

Xia and Torrellas also use a clever means of improving cache performance by leaving software hints in the compiled code to help guide the hardware prefetching [7]. They create a code layout more suited for locality based on dataflow analysis, address traces, and frequency of routine invocation.

After an embedded system's software or firmware is compiled and linked, instruction addresses remain static unless the program is upgraded in the field. As a result, the methods described above can be used to organize the code in order to avoid cache misses.

3. Dynamic Round-Robin Scheduling

Our technique, which we call *Dynamic Round-Robin Scheduling* (DRRS), complements the techniques described in the previous section by monitoring and then improving the performance of a system that uses round-robin task scheduling. DRRS simply reorders the round-robin scheduler's task execution sequence. This reordering is done dynamically during program execution, and hence is a run-time improvement done in software, with no special hardware required. Further, DRRS uses live execution profiling of cache miss information, so no post processing of data is required and the system can adapt dynamically as it runs.

Note that the goal of this task reordering is not to find the optimal schedule. DRRS simply tries to find a schedule that reduces the cache misses below a user-specified level. The goal of our work was to find a <u>low-overhead technique</u> to provide <u>moderate improvement</u>, not a high-overhead technique that provides the best possible improvement. Like Gloy, we also realized that even small changes can make a difference in performance [9].

Thus DRRS is a <u>software technique to directly support the hardware</u>. No changes are required to the code layout, which remains as generated by the compiler. However the round-robin schedule **sequence** is modified to reduce the cache misses experienced.

For example, consider a round-robin schedule consisting of six tasks where the round-robin scheduler proceeds in task execution order A,B,C,D,E,F, and then repeats continuously in that sequence. During execution, each task replaces cache data, and a series of cache misses occur, harming performance.

Now suppose the execution order of tasks B and C are swapped. Using a classic round-robin scheduler, the <u>order</u> in which the tasks are processed in each round does not matter to the scheduler — all that is required of the scheduler is that all tasks be given an equal time slice in each round.

However, the new execution order (A,C,B,D,E,F) results in a different cache usage, potentially a reduction in the number of cache misses, and if so, overall better performance. This is the key to DRSS — simply changing the execution order of the tasks in the round-robin CPU scheduler to improve cache performance. It is also important to understand that we are **not physically changing the location** of the tasks in memory, which are still fixed at compile and link time.

Changes can be common at run-time with many systems that can disturb the state of the instruction and data caches, leading to periods of cache misses. For example, activities such the data processed by the embedded CPU or interrupts experienced can alter the execution paths of a collection of tasks, even if the software is fixed. Without a dynamic method, one cannot adapt to such changes. Thus our problem statement is as follows:

Given a set of N tasks scheduled by a round-robin CPU scheduler, find a schedule order that reduces the number of cache misses below some user-specified threshold. Adapt to system changes by adjusting the schedule order dynamically during run-time to keep the number of cache misses below this threshold.

3.1 Implementation

Dynamic Round-Robin Scheduling (DRRS) begins in the *running* state with some initial round-robin task ordering. This initial ordering can be arbitrary, or based on static analysis or some other profiling technique. In the *running* state, there is no difference in activity compared to a regular system, except that the amount of cache misses experienced are monitored at regular intervals. Thus after a set number of round robins have completed, a checkpoint is triggered to see if the number of cache misses have increased above some user-defined threshold. If so, the system enters *tuning* state as described below and the round-robin ordering is changed to reduce the number of cache misses.

This *threshold* sets a user-defined bound on the number of cache misses experienced, resulting in more predictable performance. Thresholds can be set by any one of a variety of methods, such as performance profiling of the running system (as described later in Section 4), or trace analysis of system execution under various data sets. For real time systems, it might be desirable to set the threshold when the round-robin performance degrades to a point where deadlines risk getting missed.

During *tuning* state, a new round-robin task order is selected by the software. This is done by reconfiguring the contents of a *scheduling* array, which contains an ID number for each active task and represents a trial ordering of the tasks to be executed in a round robin. The scheduler will reference this array each time before a task is selected for CPU execution. By using a jump table indexed by the array, DRRS can be implemented with a low instruction count overhead.

The system then uses the new round-robin schedule over the same checkpoint interval and compares the new performance measure to the previous best value. If there is an improvement in performance, this becomes the new best performance value and the new round-robin schedule ordering is selected. This new performance value is also compared to the threshold, and when the threshold is reached, the system moves to the *running* state and the new round-robin sequence is used until the next *tuning* cycle. The new threshold is also stored as the result of the tuning operation. Figure 1 summarizes the activities of the *tuning* phase.

DRRS is a run-time heuristic that attempts to iteratively improve system performance by altering the round-robin schedule order and then grading the effect of this alteration. Through a series of stepwise refinements, the number of cache misses is eventually reduced and the overall performance improves. This method is similar to many hill climbing / branch and bound heuristics; so if a change to the round-robin schedule is selected that does NOT improve performance, the change is rejected.

During the tuning phase, a new schedule order must be selected to test. We used a simple swapping of the schedule array elements or randomly juggling the array to accomplish this task with little execution overhead. We found in our sample experiments (see Section 4) that after a dozen or so trials, the performance usually improved enough to dip below the threshold and get back into the *running* state, and the main additional code overhead (the ~40 instruction cycles to run a pass of the rescheduling algorithm, executed once per interval) did not

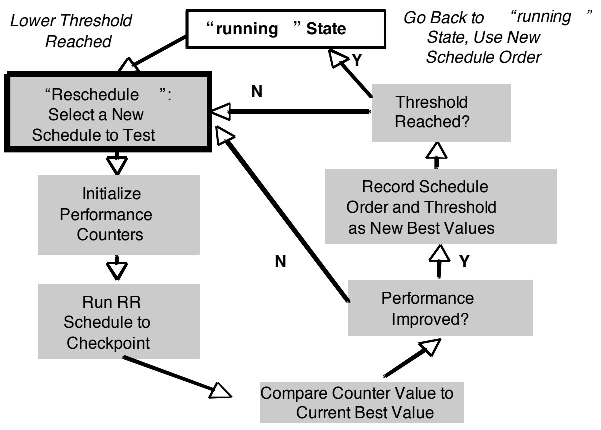

Lower Threshold Reached
"running " State
Go Back to "running " State, Use New Schedule Order
"Reschedule ": Select a New Schedule to Test
N
Y
Threshold Reached?
Initialize Performance Counters
Record Schedule Order and Threshold as New Best Values
N
Y
Run RR Schedule to Checkpoint
Performance Improved?
Compare Counter Value to Current Best Value

Figure 1. Tuning the System's Schedule

significantly harm the performance. As in any hill-climbing algorithm, performance occasionally got worse during a test but those effects were temporary and the performance quickly improved. Overall, we met our goal of developing a low-overhead algorithm that could make minor changes to the round-robin schedule and yet give a worthwhile performance improvement.

4. Experimental Validation

In order to explore our DRRS technique, we created a test workload consisting of the Dhrystone benchmark and 5 tasks selected from Mibench, an embedded systems benchmark [11]. These tasks (see Table 1) were arranged in a classic round-robin schedule where each would be selected to run for one time interval in turn. The tasks had different total run-times, and hence required a different amount of CPU time to complete. The total executable code size including shared libraries was 40KB.

Table 2 shows the summary of 10 simulations created by swapping array elements in a sequential fashion for two different cache sizes. It shows the task ordering when the tuning phase is entered (above the threshold), the starting number of cache misses, the average number of cache misses experienced during the tuning, the ending task ordering and number of cache misses, and the improvement.

We ran our experiments on simulations of the commercially popular ARM11 series CPU core. We used this CPU with two different cache sizes for instruction and data (4K and 16K), and with 4-way set associativity and a selectable random or LRU replacement policy. We performed simulations using all combinations of these cache options on a cycle accurate hardware simulator. Our software code was compiled with default optimizations using the ARM Realview ^{TM} compiler tools [10].

We began our experiments by running our system using various arbitrary schedule orderings, and studied the number of cache misses experienced. Based on these initial tests, we selected a checkpoint interval of 10 round-robins and accumulated the cache miss information for both data and instruction caches. Since this was a test system without real requirements, we arbitrary set the high threshold at the worst case observed cache misses, and the low threshold at 15% below this number, for both 4K and 16K cache sizes.

Table 1. Test Simulation Software

#	Name	Exec.Time %	Description
1	bitcounts	2%	Count bits in a vector
2	Djikstra	4%	Shortest path
3	Dhystone	21%	Popular Benchmark
4	aes	23%	Encryption of data
5	fft	40%	Fourier Transform
6	qsort	10%	Sort a set of numbers

Table 2. Simulation Results

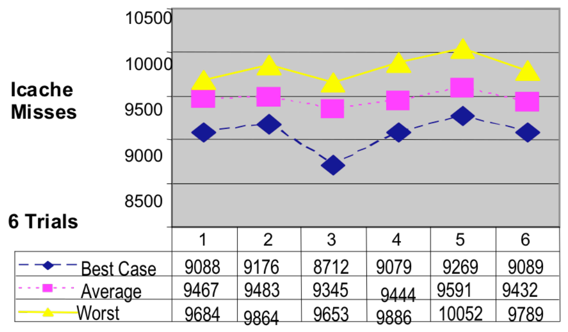
#	Start/End Order	Start End	Avg	Imp-rove	Inter -vals	Cache Size
1	1,2,3,4,5,6 4,2,3,1,5,6	8600 7542	7979	13%	4	16K
2	6,1,3,4,5,2 2,5,1,4,3,6	8878 7572	8072	15%	16	16K
3	4,2,6,1,5,3 2,4,5,1,6,3	8360 7533	7989	10%	7	16K
4	1,5,2,3,4,6 3,2,5,1,4,6	8614 7658	8086	11%	12	16K
5	6,5,4,3,2,1 1,4,3,5,2,6	9003 7663	8081	15%	20	16K
6	1,2,3,4,5,6 3,2,1,4,5,6	28850 26532	27625	8%	3	4K
7	6,5,4,3,2,1 4,3,5,6,2,1	27934 26695	27907	4.5%	19	4K
8	4,1,6,2,5,3 1,4,6,2,5,3	27691 26380	27470	5%	2	4K
9	2,5,3,6,4,1 4,5,3,6,2,1	27686 26575	27769	4%	15	4K
10	4,2,1,5,6,3 2,4,1,5,3,6	27983 26837	27586	4%	2	4K

With our 6 tasks, we ran many simulations using arbitrary starting points that resulted in a high degree of cache misses above the threshold. Compulsory misses were screened out by delaying the cache miss counts until a few round-robin iterations have elapsed. In this way, the effect of the cache footprint due to the previous running combinations and due to cold start misses was largely flushed out before measurements were taken.

As can be seen in Table 2, the number of instruction cache misses was reduced by just changing the task ordering. The end result was achieved after running the indicated number of rescheduling intervals. However, this was not the case for data cache misses — there seemed to be no pattern of data cache miss that was influenced by our positional rescheduling. Overall, our test system was clearly dominated by instruction cache misses, which accounted for >95% of all cache misses. Of course, this consequence could have been due to the benchmark programs being tested, so further experimentation is needed to confirm whether or not this is a general effect.

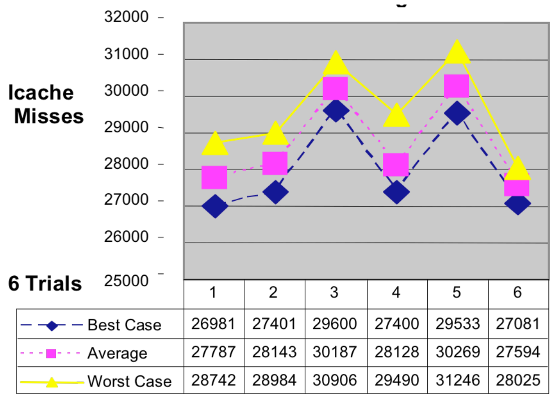
The table shows the average number of cache misses, which is important because it constitutes the cost of tuning phase. However, one can see that the cost of the rescheduling can easily be outweighed by the reduction in cache misses experienced.

Table 3. Random Rescheduling 16k Cache



	1	2	3	4	5	6
Best Case	9088	9176	8712	9079	9269	9089
Average	9467	9483	9345	9444	9591	9432
Worst	9684	9864	9653	9886	10052	9789

Table 4. Random Rescheduling 4k Cache



	1	2	3	4	5	6
Best Case	26981	27401	29600	27400	29533	27081
Average	27787	28143	30187	28128	30269	27594
Worst Case	28742	28984	30906	29490	31246	28025

Since on entry to the rescheduling, the system is already in a bad state, changes are likely to help get below the threshold

Our random rescheduling experiments arbitrarily juggled the elements in the scheduling array during each trial. This allowed a new round-robin schedule to be derived during each trial period with low instruction count overhead. With this trial, instead of using a threshold, all runs consisted of 20 intervals using an arbitrary random start point. The best/worst/average results achieved are shown in Tables 3 and 4. These tables show the variation in performance that one can expect just by changing the task ordering. By selecting the best out of 20 fixed runs, we could achieve as much as a 10% improvement in cache miss reduction compared to the worst case.

With 4K caches, the performance improvement was less dramatic compared to 16K caches. Most likely this difference was due to the high degree of capacity conflicts which involved frequent line flushes. This saturated the affect of cache conflict misses. The 16K caches thus represented a more reasonable choice for our application, so the task rescheduling resulted in more obvious results showing cache conflict avoidance.

5. Conclusion

This paper has introduced Dynamic Round-Robin Scheduling (DRRS) as a flexible framework for improving the run-time performance of multi-tasking embedded systems. Continuous monitoring of the system performance is used to monitor the effect of the task rescheduling. With our limited experimentation, we were able to achieve a 4% to 15% improvement in the reduction of instruction cache misses between tasks in the round-robin schedule, with low enough overhead to justify the algorithm.

Our results indicate that even small changes in the task ordering can result in dramatic changes in the cache performance. This can be important for multi-tasking embedded or real-time systems with delicate performance criteria. The code overhead is low and the implementation technique is flexible; which can be efficiently implemented using a task ordering array. Techniques such as DRRS can be helpful at augmenting other techniques for code layout improvements used in embedded systems.

Much research has already been done to optimize code layouts at compile and compile time to minimize cache conflict. However this improved layout may only be suitable for a given static workload, while systems follow different execution paths and workloads in the field. So, without a dynamic technique, one must live with what the compiler provides. The DRRS technique in contrast, monitors the cache misses and dynamically adjusts the task schedule to incrementally improve performance under changing system conditions.

6. References

[1] Lee, J., Lee, J., Kim, Selective temporal and aggressive spatial cache system based on time interval. International Conference on Computer Design; Sept. 2000. pp. 287-293.

[2] Pettis, K., Hansen, R. Profile Guided Code Positioning. Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation, pp. 16-27, June 1990.

[3] Gloy, N., Smith, M. 1999. Procedure placement using temporal-ordering information. ACM Trans. Program. Lang. Syst. 21, 5 (Sep. 1999), pp. 977-1027.

[4] Alan Dain Samples. Profile Driven Compilation. PhD thesis, U.C. Berkeley, April 1991. U.C. Berkeley CSD-91-627.

[5] Ramos, J., Rego, V., Sang. An Improved Computational Algorithm for Round-Robin Service. in Proceedings of Winter Simulation Conference, December, 2003.

[6] Knuth. D. The Art of Computer Programming -- Vol. 1, Fundamental Algorithms. Addison Wesley, 1973.

[7] Torrellas, J., Xia, C., Daigle, R. Optimizing Instruction Cache Performance for Operating System Intensive Workloads. Symposium on High-Perf. Computer Arch., pp. 360-369, Jan 1995.

[8] Stone, H. High Performance Computer Architecture, 3rd Edition. Addison Wesley. 1992. pp. 76-84.

[9] Gloy, N. Code Placement using Temporal Profile Information. PhD thesis, Harvard University, September 1998. Cambridge, MA.

[10] Lennard, C., Mista, D. Taking Design to the System Level. http://www.arm.com/pdfs/ARM_ESL_20_3_JC.pdf. ARM LTD, April 2005.

[11] Guthaus, M., et al. MiBench: A free, commercially representative embedded benchmark suite. IEEE 4th Annual Workshop on Workload Characterization, pp. 1-12. Austin, TX, December 2001.

[12] Ghosh, S., Martonosi, M., Malik, S. Automated Cache Optimizations using CME Driven Diagnosis. Proc. of the 2000 Int. Conference on Supercomputing. pp. 316-326.

[13] Kalamatianos, J., Kaeli, D. Temporal Based Procedure Reording for Imporved Instruction Cache Performance. Proceedings of HPCA, Febuary 1998.