Optimizing DRE System Performance with the SMACK Cache Efficiency Metric

[1]Brian Dougherty, [2]Jules White, [3]Russell Kegley,
[3]Jonathan Preston, [1]Douglas C. Schmidt, and [1]Aniruddha Gokhale
[1]Vanderbilt University, {briand,schmidt,gokhale}@dre.vanderbilt.edu*
[2]Virginia Tech, julesw@vt.edu*
[3]Lockheed Martin Aeronautics, {russell.b.kegley,jonathan.d.preston}@lmco.com*

Abstract

Distributed real-time and embedded (DRE) systems are often subject to stringent timing constraints. Scheduling techniques, such as rate monotonic scheduling, can be used to ensure that real-time deadlines are met. Although a processor cache can reduce the time required for a task schedule to execute, multiple task execution schedules may exist that meet deadlines but differ in cache utilization efficiency. It is hard to determine which task execution schedules will utilize the processor cache most efficiently and provide the greatest reductions in execution time without jeopardizing real-time deadlines.

The work in this paper provides three key contributions to predictive performance evaluation of processor caching in DRE systems. First, we present the System Metric for Application Cache Knowledge (SMACK), which is a novel approach to quantify the expected cache utilization efficiency of different schedules. Second, we employ SMACK to predict the relative execution time and cache misses of 11 simulated software systems with 2 different execution schedules per system. Third, we empirically evaluate the impact of using SMACK as a heuristic to alter task schedules to reduce system execution time. Our results show that heuristic scheduling with SMACK increases cache performance, reduces execution time, and satisfies real-time scheduling constraints and safety requirements without requiring significant hardware or software changes.

*This work was sponsored in part by the Air Force Research Lab.

1 Introduction

Current trends and challenges. Distributed real-time and embedded (DRE) systems, such as integrated avionics systems, are subject to stringent timing constraints. These systems must minimize execution time to ensure that real-time deadlines are met. One approach to reduce execution time is to reduce the time spent loading data from memory by efficiently utilizing processor caches.

Several research techniques utilize processor caches more efficiently to reduce execution time. For example, Bahar et al. [4] examined several different cache techniques for reducing execution time by increasing cache hit rate. Their experiments showed that efficiently utilizing a processor cache can result in as much as a 24% reduction in execution time. Likewise, Manjikian et al. [13] demonstrated a 25% reduction in execution time as a result of modifying the source-code of the executing software to use cache partitioning.

Many optimization techniques [18, 14, 23] increase cache hit rate by enhancing source code to increase *temporal locality* of data accesses, which defines the proximity with which shared data is accessed in terms of time [11]. For example, loop interchange and loop fusion techniques can increase temporal locality of accessed data by modifying application source code to change the order in which application data is written to and read from a processor cache [11, 13]. Increasing temporal locality increases the probability that data common to multiple tasks persists in the cache, thereby reducing cache-misses and software execution time [11, 13].

Open problem $\Rightarrow$ Increasing cache hit rate of integrated applications without source code modifications.

Integrated applications are built from separate pieces of software that must be scheduled to execute in concert with one another. Prior work has generally focused on source-code level modifications for individual applications instead of integrated applications, which is problematic for DRE systems built from multiple integrated applications. DRE systems based on the integration of multiple applications (such as the architecture shown in Figure 1) often prohibit code-level modifications due to restricted access to proprietary source code and the potential to violate safety certifications [20] by introducing overflow or other faulty behavior.

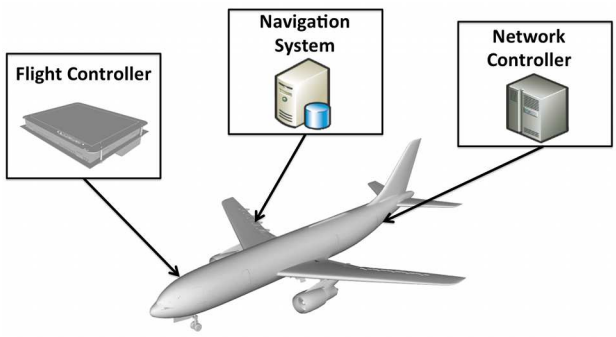

Figure 1: Example of an Integrated Avionics System

Solution approach → Heuristic-driven schedule alteration of same-rate tasks to increase cache hit rate. Priority-based scheduling techniques can help ensure DRE system software executes without missing real-time deadlines. For example, rate-monotonic scheduling [17] is a technique for creating task execution schedules that satisfy timing constraints by assigning priorities to tasks based on the task periodicity and ensuring utilization bounds are not exceeded. These tasks are then split into sets that contain tasks of the same priority/rate.

Rate monotonic scheduling specifies that tasks of the same rate can be scheduled arbitrarily [7] as long as priority inversions between tasks are not introduced. Figure 2 shows two different valid task execution schedules generated with rate monotonic scheduling. Since task A2 and task B2 share the same priority, their execution order can be swapped without violating real-time constraints. This paper shows how to improve cache hit rates for DRE systems built from multiple integrated applications by intelligently ordering the execution of tasks within the same rate to increase temporal locality of task data accesses. We refer to this technique as *metascheduling*, which involves no source code modifications.

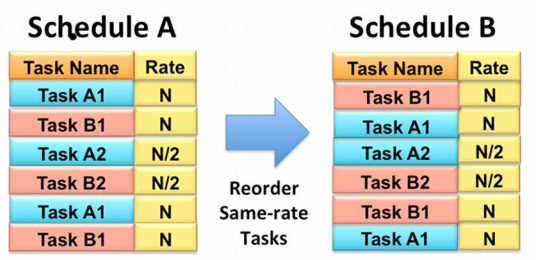

Figure 2: Valid Task Execution Schedules

This article presents a novel real-time scheduling optimization technique that uses metascheduling to improve cache effects in integrated DRE systems without violating real-time scheduling constraints or causing priority inversions. Since this technique requires no source code modifications, it can be applied to integrated DRE systems without requiring source software permissions or completely invalidating safety certifications. To quantify the temporal locality of the scheduling order of a set of same-rate tasks, known as the *metaschedule*, and guide the schedule modification process, we have created the *System Metric for Application Cache Knowledge* (SMACK) metric.

SMACK considers several factors, such as cache size and task execution schedule, to allow developers to determine which orderings of same-rate tasks result in a higher cache hit rate. Since DRE systems are subject to extremely tight resource constraints, reducing execution time by as little as 1% often frees enough resources to increase system capabilities or safety margins. Reaping these gains through increasing cache hit rate is particularly attractive to cost-conscious DRE system designers since no additional hardware or software is required.

This article provides the following contributions to R&D on real-time scheduling optimizations to increase cache hit rate in integrated DRE systems:

• We present a real-time metascheduling heuristic that satisfies real-time scheduling constraints and safety requirements, increases cache hits, and requires no new

hardware or software.

• We provide a formal methodology for calculating the "SMACK score," which provides a rough estimation of the temporal locality of different metaschedules in an integrated DRE system to help developers select better execution schedules.

• We present empirical results of 2 task execution schedules performance and demonstrate that the calculated SMACK score correlates with an increased cache hit rate.

Paper organization. The remainder of the paper is organized as follows:Section 2 summarizes the challenges of creating a metric that predicts integrated DRE system performance at design time and guides execution schedule modifications; Section 3 explains how the SMACK metric can be calculated to predict DRE system performance and applies it to create cache-effective execution schedules; Section 4 analyzes the results of experiments that evaluate how effectively SMACK creates schedules with fewer cache misses; Section 5 compares SMACK with related work; and Section 6 presents concluding remarks.

2 Challenges of Analyzing and Optimizing Integrated DRE System Architectures for Cache Effects

This section presents the challenges that DRE system integrators face when attempting to optimize application integration to improve cache hit rate. Safety-critical DRE systems are often subject to multiple design constraints, such as safety requirements and real-time deadlines, that may restrict which optimizations are applicable. This section describes three key challenges that must be overcome to optimize application integration by improving the cache hit rate of safety-critical DRE systems.

Challenge 1: Altering application source code may invalidate safety certification. Existing cache optimization techniques, such as loop fusion and data padding [10, 16], increase cache hit rate but requiring application source code modifications, which may invalidate previous safety certifications by introducing additional faults, such as overflow. Re-certification of system applications is a slow and expensive process, which increases cost and delays deployment. Moreover, the source code of proprietary applications may not be accessible to integrators. Even if application source code is available, integrators may not have the expertise required to make reliable modifications. What is needed, therefore, are techniques that alter the DRE system to improve cache hit rates without modifying system software.

Challenge 2: Optimization techniques must satisfy real-time scheduling constraints. Safety-critical DRE systems are often subject to stringent scheduling constraints and commonly use priority-based scheduling methods, such as rate monotonic scheduling, to ensure that software tasks execute predictably [24, 9]. These constraints prohibit many simple solutions that ignore task priority, such as executing all task sets of each application, that would greatly increase cache hit-rate. These techniques can cause the system to behave unpredictably, with potentially catastrophic results due to missed deadlines and priority inversions. What is needed, therefore, are techniques that can be applied and re-applied when necessary to increase the cache hit-rate and decrease system execution time without violating timing constraints.

Challenge 3: System complexity and limited access to source code. Current industry practice [2] for increasing cache hit rate require collecting detailed, instruction-level information that describe application behavior with respect to the memory subsystem and data structure placement. Obtaining system information of this granularity, however, can be an extremely laborious and time consuming for large-scale DRE systems, such as flight avionics that contain millions of lines of codes and dozens of applications. Moreover, large-scale DRE systems, such as flight avionics, may be so complex that it is not realistically feasible to collect this information. System integrators can more easily obtain higher level information, such as the percentage of total memory accesses made by a given task. What is needed, therefore, are techniques that allow system integrators to increase the cache hit rate of DRE systems without requiring intricate, low-level system knowledge.

3 Cache Aware Metascheduling to Improve Cache Hit Rate

This section presents cache-aware metascheduling, which is a technique for increasing cache hit rate through reordering the execution schedule of same-rate tasks.

3.1 Re-ordering Same-rate Tasks with Cache-aware Metascheduling

Rate mononotonic scheduling can be used to create task execution schedules that ensure real-time deadlines are met. This technique, however, allows the definition of additional rules to determine the schedule of same-rate tasks [15, 3, 12]. As shown in Figure 3, reordering same-rate tasks, or metascheduling, can produce multiple valid execution schedules.

For example, Figure 3 shows how Task A1 can execute before or after Task B1. Either ordering of these same rate tasks meets real-time scheduling constraints. Since the original schedule satisfies real-time constraints and reordering same rate tasks does not introduce priority inversions, schedules generated by metascheduling are valid. Moreover, metascheduling does not require alterations to application source code or low-level system knowledge.

The motivation behind metascheduling is that although different execution orders of same-rate tasks do not violate real-time constraints, they can impact the cache hit-rate. For example, if two same-rate tasks that share a large amount of data execute sequentially, then the first task may "warm up" the cache for the second task by preloading data needed by the second task. This type of cache warming behavior can improve the cache hit rate of the second task.

Same-rate task orderings can also negatively affect cache hit rate. For example, tasks from multiple applications often run concurrently on the same processor in an integrated DRE system. These tasks may be segregated into different processes, however, preventing tasks from different applications from sharing memory. If two tasks do not share memory there is no cache warmup benefit. Moreover, the first task may write a large amount of data to the cache and evict data needed by the second task from the cache, reducing the cache hit rate of the second task.

Cache-aware metascheduling is the process of reordering the execution of same-rate tasks to increase beneficial cache effects, such as cache warm up, and reduce negative effects, such as requiring reading data from main memory. Cache-aware metascheduling is relatively simple to implement, does not require in-depth knowledge of the instruction level execution details and memory layout of a large-scale system, and can be achieved without source code modifications to tasks. Section 4 shows that reordering same-rate tasks does improve cache hit rates and reduce execution time. A key question, however, is what formal metric can be used to choose between multiple potential same-rate task execution schedules.

3.2 Deciding Between Multiple Metaschedules

While cache-aware metascheduling can be used to produce multiple valid same-rate task execution schedules, it is not always apparent which schedule will produce the overall best hit-rate and application performance. For example, Figure 3 shows a schedule generated with rate monotonic scheduling and two additional valid schedules created by permuting the ordering of same-rate tasks for a flight controller (FC) application and a targeting system (TS) application. The only difference between the task

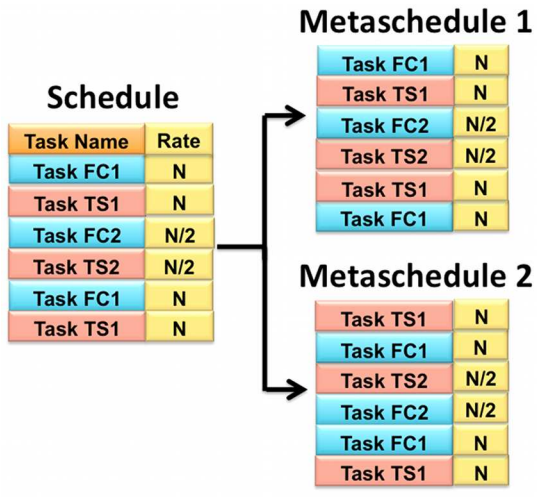

Figure 3: Multiple Execution Schedules

execution schedules are the order in which tasks of the

$$SMACK(S) = \sum_{i=0}^{|S|-1} \sum_{k=0}^{CHL-1} (CHit(S_i), FR(S_i), k))) * O(S_i), FR(S_i, k)) \quad (1)$$

same-rate are executed.

It is not obvious which task execution schedule shown in Figure 3 will produce the best cache hit-rate. For example, Metaschedule 2 in Figure 3 shows 2 tasks of Application FC executing sequentially, while no tasks of Application TS execute sequentially. If the tasks in Application FC share a large amount of data temporal locality should increase compared to the original schedule since the cache is "warmed up" for the execution of FC1 by FC2.

In Metaschedule 1, however, 2 tasks of Application TS execute sequentially while no tasks of Application FC execute sequentially. If Application TS shares more data than Application FC, Metaschedule 1 will yield greater temporal locality than both the original schedule and schedule FC since the cached will be warmed up with more data. It may also be the case that no data is shared between any tasks of any application, in which case all three schedules would yield similar temporal locality and cache hit rates.

Figure 3 shows it is hard to decide which schedule will yield the highest cache hit rate. Constructing a metric for estimating temporal locality of a task execution schedules could provide DRE system developers with a mechanism for comparing multiple execution schedules and choosing which one would most yield the highest cache hit rate. It is hard to estimate temporal locality, however, due to several factors, such as the presence and degree of data sharing between tasks. Below, we formally define a hard metric for comparing multiple task execution schedules and guiding the metascheduling process.

3.3 Quantifying Temporal Locality with SMACK

To estimate the temporal locality of task execution schedules we developed the *System Metric for Application Cache Knowledge* (SMACK). SMACK can estimate the temporal locality of multiple execution schedules, such as those shown in Figure 3. This metric can be used as a heuristic for metascheduling to determine task execution schedules that increase cache hit rate. The calculation of the SMACK metric is shown in Equation 1 where:

- S is the set of tasks scheduled to execute.
- CHL determines how many sequential task executions can occur before cached data written by executing the initial task may be overwritten as described in Section 3.3.1.
- $CHit$ yields the expected number cache hits due to a task executing after an initial task completes execution.
- FR provides the task that will execute k task executions after an initial task completes execution.
- O is a function that determines if two tasks are of the same application.

The remainder of this section explains other factors that impact cache hit-rate while. The components of SMACK are then formally defined in detail in Section 3.4.

3.3.1 Cache Half-Life

We now explain the key factors that impact cache hit rate. A beneficial effect occurs when task T1 executes before task T2 and loads data needed by T2 into the cache. The beneficial effect can occur if T1 and T2 execute sequentially or if any intermediate task executions do not clear out the data that T1 places into the cache that is used by T2. The *cache half-life* is this window of time between which T1 and T2 can execute before the shared data is evicted from the cache by data used for intermediate task executions. While this model is simpler than the actual complex cache data replacement behavior, it is effective enough to give a realistic representation of cache performance [19].

For example, assume there are 5 applications, each consisting of 2 tasks, with each task consuming 20 kilobytes of memory in a 64k cache. The hardware uses a *Least Recently Used* (LRU) replacement policy, which replaces the cache line that remained the longest without being read when new data is written to the cache. The cache half-life formulation will differ for other cache replacement policies.

Executing the tasks will require writing up to 200 kilobytes to cache. Since the cache can only store 64 kilobytes of data, all data from all applications cannot persist in the cache simultaneously. Assuming the cache is initially empty, it would take a minimum of 4 task executions writing 20 kilobytes each before any data written by the first task potentially becomes invalidated. This system would therefore have a cache half-life of 4.

3.3.2 Determining Total Cache Hits

Each sequential task execution occurring after a task executes yields a probability of a beneficial cache effect based on the DRE system's cache half-life. Each good cache effect increases the cache hit rate of a specific task and reduces the execution time of the system. The total probabilistic expected cache hits due to these beneficial cache effects yields the expected cache hits for this set of tasks.

3.4 Defining and Calculating SMACK Cache Metric

Section 3 provides a qualitative summary of our method for calculating the cache metric of a system deployment. Below, we define a formal method for determining the relative execution time savings due to caching of system deployments.

3.4.1 Calculating the Cache Half-Life

The cache half-life, *CHL*, estimates how many sequential task executions can potentially lead to a cache hit before all cached data from the original task is invalidated, as shown in Equation 2.

$$CHL = \frac{CS}{((DW(T)/|T|) * (1 - DS))} \tag{2}$$

In this equation, *CHL* is calculated by dividing the size of the cache, *CS*, by the average amount of data written per task. To determine the average amount of data written per task, the total amount of data written, *DW* is divided by the number of tasks $|T|$, and multiplied by the percent of task data shared between tasks, *DS*. *DS* is determined by dividing the total number of variables that are read by both tasks by the sum of the total number of variables read by both tasks.

$$FR(S_i,k) = \begin{cases} F_{i+k} & i+k < M(MF) \\ F_{((i+k)-M(S))\%M(S)} & i+k \geq M(MF) \end{cases} \tag{4}$$

3.4.2 Determining if Tasks Overlap

Our metric assumes that tasks of different applications share no data. Cache hits can therefore only occur if two tasks share the same application. Equation 3 returns 1 if two tasks are a part of the same application and 0 if they are not.

$$O(t_i,t_j) = \begin{cases} 1 & t_i == t_j \\ 0 & t_i != t_j \end{cases} \tag{3}$$

3.4.3 Quantifying Cache Hits for Variable Size Tasks

Software tasks of the same application may not read the same amount of memory. The number of cache hits that result from a task executing will therefore differ based on the amount of common data read. Equation 5 defines the maximum cache hits that can be expected if a task of an application executes after another task of the same application.

$$CHit(S_j,S_y) = DS * DR(S_y) \tag{5}$$

The maximum cache hits is equal to the percentage of data shared by the tasks multiplied by the amount of data read by the task executing later.

3.4.4 Cache Hits due to Sequential Task Executions

We calculate the cache hit probability "CHit" for all sequential executions of tasks on/off the processor in the schedule "S." After a task executes, the number of sequential executions that can occur before all data written by the task to the cache is invalidated is determined by the CHL. Each transition that occurs before the CHL is reached can therefore potentially yield a cache hit and must be investigated.

Determining which task executes *k* executions after an initial task executes is shown in Equation 4. We define

$M(S)$ as the number of tasks that execute in a given schedule. Two cases must be considered:

- A task may execute k steps ahead of a task, but in the same as shown in the first case of Equation 4.
- Since execution schedules are assumed to repeat, we must also take into account the impact of sequential task executions between sequential schedule executions. Incrementing by k transitions may exceed the boundary of the schedule, whereby the task is determined by wrapping back to the beginning of the schedule and incrementing any remaining schedule executions as shown in the second case of Equation 4.

Equation 1 accounts for all cache hits due to all sequential task executions in the execution schedule S. The first summation in Equation 1 accounts for all tasks in the schedule S'. The innermost summation in Equation 1 sums the expected cache hits CHit for tasks that share the same application, as given by O.

4 Empirical Results

This section analyzes the results of a performance analysis of multiple DRE systems with different SMACK scores. These systems differ in task execution schedules and the amount of memory shared between tasks. We investigate potential correlations between the SMACK score and L1 cache misses and runtime reductions for each system.

4.1 Overview of the Hardware and Software Testbed

To examine the relationship between SMACK score and DRE system performance, we collaborated with members of the Lockheed Martin Corporation to create multiple software systems that mimic the scale, execution schedule and data sharing of cutting-edge industry flight avionics systems. We specified the number of applications, number of tasks per application, the distribution of task priority, and the maximum amount of memory shared between each task for each system. We created a Java-based code generator to create C++ system code that possessed these characteristics. Rate monotonic scheduling was used to create a deterministic priority based schedule for the generated tasks that adheres to rate monotonic scheduling requirements.

The systems were compiled and executed on a Dell Latitude D820 with a 2.16Ghz Intel Core 2 processor with 2 x 32kb L1 instruction caches, 2 x 32 kb write-back data caches, a 4 MB L2 cache and 4GB of RAM running Windows Vista. For each experiment, each system was executed 50 times to obtain an average runtime. The cache performance of these executions were profiled using the Intel VTune Amplifier XE 2011.

4.2 Experiments: Correlating SMACK with Increased Cache Hit-rate and Runtime Reductions

Experiment design. The execution schedule of tasks can potentially impact both the runtime and number of cache misses of a system. We manipulated the execution order of a single software system with 20% shared data probability between 5 applications consisting of 10 tasks each to create 2 new execution schedules. First, rate monotonic scheduling was use to create the baseline schedule. This schedule was then permuted to change the total number of instances in which the execution of two tasks from a common application executing could potentially cause a cache hit to create the SMACK Optimized schedule.

The baseline execution schedule generated by rate monotonic scheduling resulted in a SMACK score of $3.82651x10^{10}$. Metascheduling was used to reorder same rate tasks to increase the SMACK score. This SMACK Optimized resulted in a a SMACK score of $4.679442x10^{10}$.

Analysis of results.

4.2.1 Experiment 1: Using Cache-Aware Metascheduling to Reduce Cache Misses

Altering the task execution schedule can raise or lower the temporal locality of a sequence of data accesses. We hypothesized that using cache-aware metascheduling to increase temporal locality would reduce the number of cache misses. Figure 4 shows the L1 cache misses for both execution schedules. The baseline execution schedule resulted in $3.5076x10^9$ L1 cache misses

while the SMACK Optimized execution schedule generated $3.484x10^9$ cache misses. Therefore, this data validates our hypothesis that cache miss rates can be reduced by using cache-aware metascheduling to increase temporal locality.

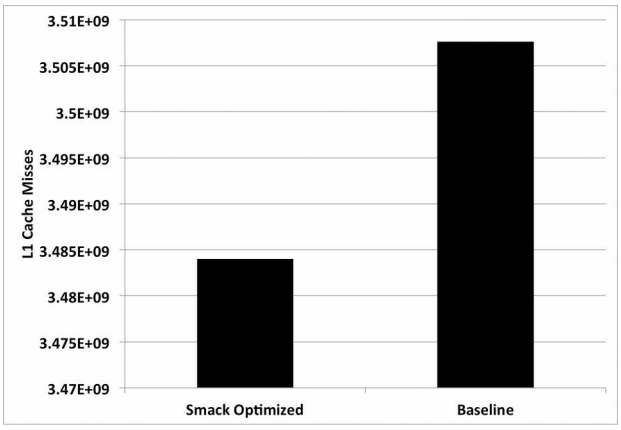

Figure 4: Execution Schedules vs L1 Cache Misses

4.2.2 Experiment 2: Reducing Execution Time with Cache-Aware Metascheduling

While Experiment 1 showed that applying cache aware metascheduling can reduce cache misses, we further hypothesized that execution time can be reduced as well. Figure 5 shows the average runtimes for the different execution schedules. As shown in this figure, the task execution order can have a large impact on runtime. The baseline execution schedule executed in 3,374 milliseconds. The SMACK Optimized execution schedule completed in 3,299 milliseconds, which was an 2.22% reduction in execution time from the baseline execution schedule.

4.2.3 Experiment 3: Impact of Data Sharing on Cache-Aware Metascheduling Effectiveness

Figure 5 shows the execution time of two execution schedules at only 20% data sharing. Data sharing of industry DRE systems, however, may vary to a large extent. Therefore, we created 10 other systems with different data sharing characteristics. We examine the impact of data

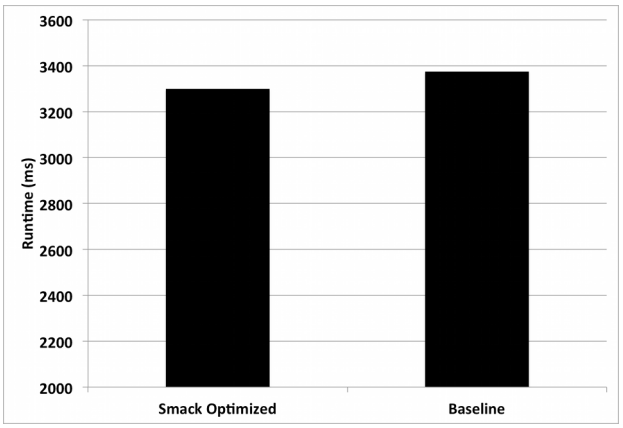

Figure 5: Runtimes of Various Execution Schedules

sharing on execution time reductions due to cache-aware metascheduling .

The execution time for the baseline and SMACK Optimized schedules is shown in Figure 6. The SMACK Optimized schedule consistently executed faster than the baseline schedule with an average execution time reduction of 2.54% without requiring alteration to application source-code and without violating real-time constraints. Moreover, this reduction required no purchasing nor implementing of any additional hardware or software or obtaining any low-level knowledge of the system. These results demonstrate that cache-aware metascheduling can be applied to reduce the execution time of an array of DRE systems, such as avionics systems, regardless of cost constraints, restricted access to software source code, real-time constraints, or instruction level-knowledge of the underlying architecture.

5 Related Work

This section compares our cache-aware metascheduling heuristic for increasing cache hit-rate with other techniques for optimizing cache hit rate.

Software cache optimization techniques. Many techniques change the order in which data is accessed to increase cache hit rate by altering software at the source code level. These optimizations, known as data access optimizations [11], focus on changing the manner in which

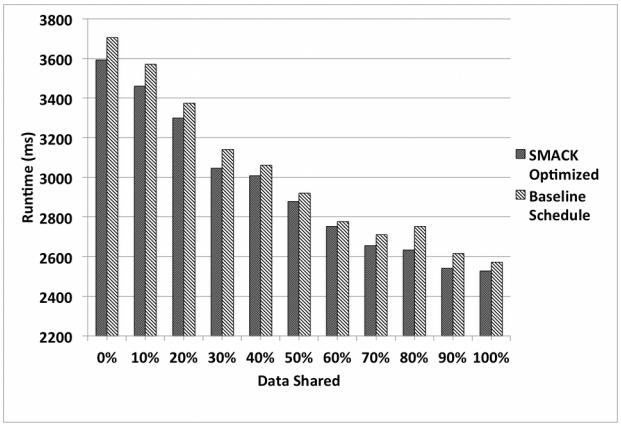

Figure 6: Runtimes of Multiple Levels of Data Sharing

loops are executed. One technique, known as *loop interchange* [26], can be used to reorder multiple loops to maximize the data access of common elements in respect to time, referred to as *temporal locality* [1, 27, 26, 21].

Another technique, known as *loop fusion* [22], is often applied to further increase cache hit rate. Loop fusion maximizes temporal locality by merging multiple loops into a single loop and altering data access order [22, 10, 25, 5]. Yet another technique for improving cache hit rate is to utilize *prefetch* instructions, which retrieves data from memory into the cache before the data is requested by the application [11]. Prefetch instructions inserted manually into software at the source code level can significantly reduce memory latency and/or cache miss rate [6, 8].

While these techniques can increase cache hit rate of applications, they all require source code optimizations. Many systems, such as avionic systems are safety critical and must undergo expensive certification and rigorous development techniques. The fundamental difference, however, between cache-aware metascheduling and these methods is that no modifications are required to the underlying software that is executing on the system, thereby achieving performance gains without requiring source code access or additional re-certification of hardware or software.

6 Concluding Remarks

Processor data caching can substantially increase DRE system performance. It is hard, however, to create valid task execution schedules that increase cache effects and satisfy timing constraints. Metascheduling can be used to generate multiple valid execution schedules with various levels of temporal locality and different cache hit rates. Without a formal methodology for quantifying the temporal locality of an task execution schedule, moreover, it is hard to determine which task execution schedule will yield the highest cache hit rate.

This paper presents a cache-aware metascheduling heuristic that uses the *System Metric for Application Cache Knowledge* (SMACK) to quantify the performance gains of processor caching of a system. The temporal locality of multiple cache task execution schedules can be quantified and compared based on SMACK score. We empirically evaluated four task execution schedules with different SMACK scores in terms of L1 cache misses and execution time. We learned the following lessons from increasing cache hit rate with our cache-aware metascheduling heuristic:

• **Cache-aware metascheduling increases cache hit rate.** Using cache-aware metascheduling led to runtime reductions of as much as 5% without requiring code-level modifications, violating real-time scheduling constraints or implementing any additional hardware, middleware, or software, and thus can be applied to broad range of DRE systems.

• **Relatively minor system knowledge yields effective cache performance assessments.** Calculating the SMACK value of a system does not require an expert understanding of the underlying software. Reasonable estimates of data sharing and knowledge of the executing software tasks are all that is required to determine schedules that yield effective reductions in computation time.

• **Algorithmic techniques to maximize SMACK are needed to optimize cache-hit rate.** The task execution schedule was shown to have a large impact on system performance and SMACK score. Moreover, the performance of task execution schedules differed based on the cache half-life. Our future work will examine algorithmic techniques that use cache-aware metascheduling and SMACK as a heuristic for determining the optimal execution order for tasks in specific systems to maximize cache hit rate.

The source code simulating the avionics system discussed in Section 4 can be downloaded at `ascent-design-studio.googlecode.com`.

References

[1] J. Allen and K. Kennedy. Automatic loop interchange. In *Proceedings of the 1984 SIGPLAN symposium on Compiler construction*, page 246. ACM, 1984.

[2] A. Asaduzzaman and I. Mahgoub. Cache Optimization for Embedded Systems Running H. 264/AVC Video Decoder. In *Computer Systems and Applications, 2006. IEEE International Conference on.*, pages 665–672. IEEE, 2006.

[3] A. Atlas and A. Bestavros. Statistical rate monotonic scheduling. In *Real-Time Systems Symposium, 1998. Proceedings., The 19th IEEE*, pages 123–132. IEEE, 1998.

[4] R. Bahar, G. Albera, and S. Manne. Power and performance tradeoffs using various caching strategies. In *Low Power Electronics and Design, 1998. Proceedings. 1998 International Symposium on*, pages 64–69. IEEE, 2005.

[5] K. Beyls and E. DâĂŹHollander. Reuse distance as a metric for cache behavior. In *Proceedings of the IASTED Conference on Parallel and Distributed Computing and systems*, volume 14, pages 350–360. Citeseer, 2001.

[6] T. Chen and J. Baer. Reducing memory latency via non-blocking and prefetching caches. *ACM SIGPLAN Notices*, 27(9):51–61, 1992.

[7] S. Dhall and C. Liu. On a real-time scheduling problem. *Operations Research*, 26(1):127–140, 1978.

[8] J. Fu, J. Patel, and B. Janssens. Stride directed prefetching in scalar processors. In *Proceedings of the 25th annual international symposium on Microarchitecture*, pages 102–110. IEEE Computer Society Press, 1992.

[9] S. Ghosh, R. Melhem, D. Mossé, and J. Sarma. Fault-tolerant rate-monotonic scheduling. *Real-Time Systems*, 15(2):149–181, 1998.

[10] K. Kennedy and K. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. *Languages and Compilers for Parallel Computing*, pages 301–320, 1994.

[11] M. Kowarschik and C. Weiß. An overview of cache optimization techniques and cache-aware numerical algorithms. *Algorithms for Memory Hierarchies*, pages 213–232, 2003.

[12] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Real Time Systems Symposium, 1989., Proceedings.*, pages 166–171. IEEE, 1987.

[13] N. Manjikian and T. Abdelrahman. Array data layout for the reduction of cache conflicts. In *Proceedings of the 8th International Conference on Parallel and Distributed Computing Systems*, pages 1–8. Citeseer, 1995.

[14] B. Nayfeh and K. Olukotun. Exploring the design space for a shared-cache multiprocessor. In *Proceedings of the 21ST annual international symposium on Computer architecture*, page 175. IEEE Computer Society Press, 1994.

[15] J. Orozco, R. Cayssials, J. Santos, and E. Ferro. 802.4 rate monotonic scheduling in hard real-time environments: Setting the medium access control parameters. *Information Processing Letters*, 62(1):47 – 55, 1997.

[16] P. Panda, H. Nakamura, N. Dutt, and A. Nicolau. Augmenting loop tiling with data alignment for improved cache performance. *Computers, IEEE Transactions on*, 48(2):142–149, 2002.

[17] S. Pingali, J. Kurose, and D. Towsley. On Comparing the Number of Preemptions under Earliest Deadline and Rate Monotonic Scheduling Algorithms. 2007.

[18] J. Reineke, D. Grund, C. Berg, and R. Wilhelm. Timing predictability of cache replacement policies. *Real-Time Systems*, 37(2):99–122, 2007.

[19] J. Robinson and M. Devarakonda. Data cache management using frequency-based replacement. *ACM SIGMETRICS Performance Evaluation Review*, 18(1):134–142, 1990.

[20] P. Rodríguez-Dapena. Software safety certification: a multidomain problem. *Software, IEEE*, 16(4):31–38, 1999.

[21] W. Shiue and C. Chakrabarti. Memory design and exploration for low power, embedded systems. *The Journal of VLSI Signal Processing*, 29(3):167–178, 2001.

[22] S. Singhai and K. McKinley. A parametrized loop fusion algorithm for improving parallelism and cache locality. *The Computer Journal*, 40(6):340, 1997.

[23] E. Sprangle and D. Carmean. Increasing processor performance by implementing deeper pipelines. In *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*, pages 25–34. IEEE, 2002.

[24] D. Stewart and M. Barr. Rate monotonic scheduling. *Embedded Systems Programming*, pages 79–80, 2002.

[25] S. Verdoolaege, M. Bruynooghe, G. Janssens, and P. Catthoor. Multi-dimensional incremental loop fusion for data locality. In *Application-Specific Systems, Architectures, and Processors, 2003. Proceedings. IEEE International Conference on*, pages 17–27. IEEE, 2003.

[26] M. Wolf, D. Maydan, and D. Chen. Combining loop transformations considering caches and scheduling. In *micro*, page 274. Published by the IEEE Computer Society, 1996.

[27] Q. Yi and K. Kennedy. Improving memory hierarchy performance through combined loop interchange and multi-level fusion. *International Journal of High Performance Computing Applications*, 18(2):237, 2004.