Optimizing Integrated Application Performance with Cache-aware Metascheduling

Brian Dougherty[1], Jules White[1], Russell Kegley[2], Jonathan Preston[2], Douglas C. Schmidt[3], and Aniruddha Gokhale[3]

[1] Virginia Tech
[2] Lockheed Martin Aeronautics
[3] Vanderbilt University
brianpd,julesw@vt.edu,
d.schmidt,a.gokhale@vanderbilt.edu,
russell.b.kegley,jonathan.d.preston@lmco.com†

Abstract. Integrated applications running in multi-tenant environments are often subject to quality-of-service (QoS) requirements, such as resource and performance constraints. It is hard to allocate resources between multiple users accessing these types of applications while meeting all QoS constraints, such as ensuring users complete execution prior to deadlines. Although a processor cache can reduce the time required for the tasks of a user to execute, multiple task execution schedules may exist that meet deadlines but differ in cache utilization efficiency. Determining which task execution schedules will utilize the processor cache most efficiently and provide the greatest reductions in execution time is hard without jeopardizing deadlines.

The work in this paper provides three key contributions to increasing the execution efficiency of integrated applications in multi-tenant environments while meeting QoS constraints. First, we present cache-aware metascheduling, which is a novel approach to modifying system execution schedules to increase cache-hit rate and reduce system execution time. Second, we apply cache-aware metascheduling to 11 simulated software systems to create 2 different execution schedules per system. Third, we empirically evaluate the impact of using cache-aware metascheduling to alter task schedules to reduce system execution time. Our results show that cache-aware metascheduling increases cache performance, reduces execution time, and satisfies scheduling constraints and safety requirements without requiring significant hardware or software changes.

1 Introduction

Current trends and challenges. Multi-tenant environments, such as Software-as-a-Service (SaaS) platforms and integrated avionics systems, are often subject to stringent quality-of-service (QoS) requirements, such as resource requirements and performance constraints [29]. To ensure that response time specified by service-level agreements (SLAs) are upheld, execution time must be minimized. One approach to reduce execution time is to reduce the time spent loading data from memory by efficiently utilizing processor caches.

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

Several research techniques utilize processor caches more efficiently to reduce execution time. For example, Bahar et al. [5] 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. [16] demonstrated a 25% reduction in execution time as a result of modifying the source-code of the executing software to use cache partitioning.

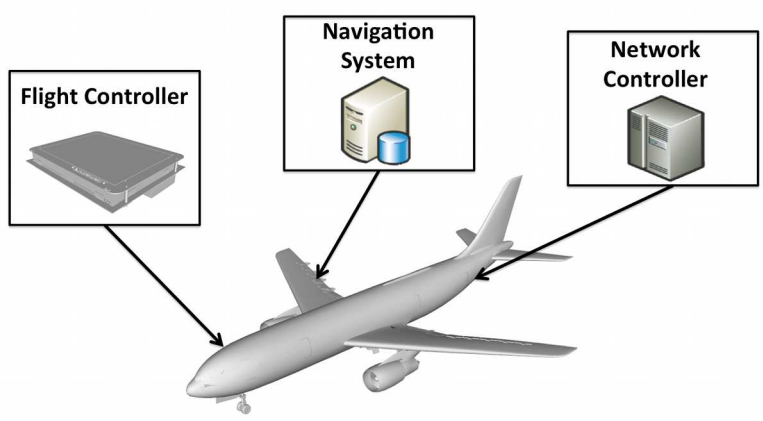

Fig. 1: Example of an Integrated Avionics Architecture

Many optimization techniques [21, 17, 27] 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 [13]. 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 [13, 16]. Increasing temporal locality increases the probability that data common to multiple tasks persists in the cache, thereby reducing cache-misses and software execution time [13, 16].

Open problem $\Rightarrow$ Increasing cache hit rate of integrated applications without source code modifications. *Integrated applications* are a class of systems consisting of a number of computing modules capable of supporting numerous applications of differing criticality levels [14]. Like other multi-tenant environments, integrated applications prohibit data sharing between multiple concurrent users, while requiring that execution completes within predefined deadlines.

Software architectures for integrated applications are built from separate components 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 multi-tenant environments built from multiple integrated applications. Systems based on the integration of multiple applications (such as the integrated avionics 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 [23] by introducing overflow or other faulty behavior.

Solution approach → Heuristic-driven schedule alteration of same-rate tasks to increase cache hit rate. Priority-based scheduling techniques can help ensure software executes without missing deadlines. For example, rate-monotonic scheduling [20] 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 [8] 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

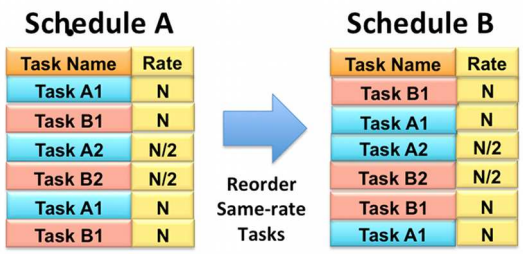

Fig. 2: Valid Task Execution Schedules

can be swapped without violating timing constraints. This paper shows how to improve cache hit rates for 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.

This paper presents *cache-aware metascheduling*, which is a novel scheduling optimization technique we developed to improve cache effects of integrated applications without violating scheduling constraints or causing priority inversions. Since this technique requires no source code modifications, it can be applied to integrated applications without requiring source software permissions or completely invalidating safety certifications.

This paper provides the following contributions to R&D on scheduling optimizations to increase the cache hit rate of integrated applications:

• We present a metascheduling technique that satisfies scheduling constraints and safety requirements, increases cache hits, and requires no new hardware or software.

• To motivate the need for scheduling enhancements to improve cache hit rate in integrated applications, we present an industry case study of an integrated avionics sys-

tem in which modifications to its constituent applications are prohibitively expensive due to safety (re)certification requirements.

• We present empirical results of 2 task execution schedules performance and demonstrate that applying cache-aware metascheduling can result in increased cache-hit rates and reduced system execution time.

Paper organization. The remainder of the paper is organized as follows: Section 2 examines an integrated avionics system designed to meet scheduling deadlines and assure required safety constraints; Section 3 summarizes the challenges of creating a metric that predicts integrated application performance at design time and guides execution schedule modifications; Section 4 describes a cache-aware metascheduling strategy for increasing cache hit-rate and reducing execution time of an integrated avionics system; Section 5 analyzes empirical results that demonstrate the effectiveness of cache-aware metascheduling for increasing cache hit-rate and reducing system execution time; Section 6 compares our cache-aware metascheduling approach with related work; and Section 7 presents concluding remarks.

2 Integrated Avionics System Case Study

This section presents a case study representative of an integration avionics system provided by Lockheed Martin that shows how integrated applications are configured in modern aircraft, such as the one shown in Figure 1. This case study underscores the similarity between multi-tenant environments and integrated applications in terms of response time requirements and data sharing restrictions. It also shows how scheduling methods for integrating applications can be applied to ensure safety constraints and scheduling requirements are met. Section 5.2 describes modifications to this method that increase the cache hit-rates of integrated application architectures.

In this architecture, task execution schedules are divided into frames in which a subset of tasks execute. Two tasks are schedule to execute sequentially at the start of each base frame. The task that executes at the base frame rate is scheduled to run, followed by another task at a rate of lower frequency. For example, at Frame 0 the scheduler will execute the software that runs at 75 Hz and the software that executes at 37.5 Hz, or half as frequently. This pattern continues repeatedly until the lowest rate software in the system has completed. All scheduling of integrated application tasks in the avionics system occurs in this manner.

One method for ensuring that tasks execute with a predetermined frequency is to set a base execution rate and then set all other tasks to execute proportionately often. The execution of the tasks can the be interleaved based on the relative execution rate, as shown in Figure 3. Applications 1 and 2 both have tasks that execute at rates N, N/2, and N/4. The rate N tasks from both applications always execute before any other tasks in a given frame. Although it is not necessarily the case that all rate N tasks from Application 1 will run before the rate N tasks from Application 2, our case study makes this order repeatable, *i.e.*, the interleaving A1/B2/A2 will not change from frame to frame after it is established when the system starts.

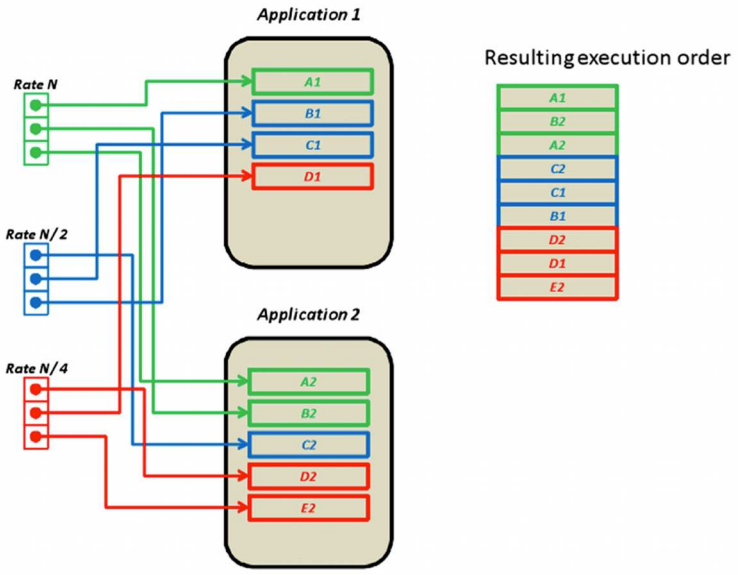

Fig. 3: Interleaved Execution Order is Repeatable

3 Challenges of Analyzing and Optimizing Integrated Applications for Cache Effects

This section presents the challenges faced by system integrators who attempt to optimize integrated applications to improve cache hit rate. Systems are often subject to multiple design constraints, such as safety requirements and scheduling 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 integrated applications.

Challenge 1: Altering application source code may invalidate safety certification. Existing cache optimization techniques, such as loop fusion and data padding [12, 19], 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 integrated applications is a slow and expensive process, which increases cost and delays deployment. Proprietary source code also may not be accessible to integrators. Even if source code is available, moreover, integrators may not have the expertise required to make reliable modifications. What is needed, therefore, are techniques that improve cache hit rates without modifying integrated application software.

Challenge 2: Optimization techniques must satisfy scheduling constraints. Integrated applications are often subject to scheduling constraints and commonly use priority-based scheduling methods, such as rate monotonic scheduling, to ensure that software tasks execute predictably [28, 10]. 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 integrated applications 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 integrated application execution time without violating timing constraints.

Challenge 3: System complexity and limited access to source code. Current industry practice [3] for increasing cache hit rate require collecting detailed, instruction-level information that describe integrated application behavior with respect to the memory subsystem and data structure placement. Obtaining information of this granularity, however, can be an extremely laborious and time consuming for large-scale systems, such as integrated avionics systems containing millions of lines of codes and dozens of integrated applications. Moreover, these large-scale systems may be so complex that it is not 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 integrated applications without requiring intricate, low-level system knowledge.

4 Improving Cache Hit Rate Via Cache-aware Metascheduling

This section presents cache-aware metascheduling, which is a technique we developed to increase cache hit rate through re-ordering the execution schedule of same-rate tasks of integrated applications. Cache-ware metascheduling can potentially increase the cache hit-rate and reduce execution time of systems in which resources are not shared between concurrent executions, such as multi-tenant environments and integrated avionics systems.

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

Rate monotonic scheduling can be used to create task execution schedules for integrated applications that ensure scheduling deadlines are met. This technique, however, allows the definition of additional rules to determine the schedule of same-rate tasks [18, 4, 15]. As shown in Figure 4, reordering same-rate tasks, or metascheduling, can produce multiple valid execution schedules.

For example, Figure 4 shows how Task A1 can execute before or after Task B1. Either ordering of these same rate tasks meets scheduling constraints. Since the original schedule satisfies 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 scheduling 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 integrated applications often run concurrently on the same processor. 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, making it ideal for increasing the performance of existing integrated architectures and multi-tenant systems. Section 5 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.

4.2 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 4 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 execution schedules are the order in which tasks of the same-rate are executed.

It is not obvious which task execution schedule shown in Figure 4 will produce the best cache hit-rate. For example, Metaschedule 2 in Figure 4 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 4 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 integrated application developers with a mechanism for comparing multiple execution schedules and choosing which one would most yield the highest

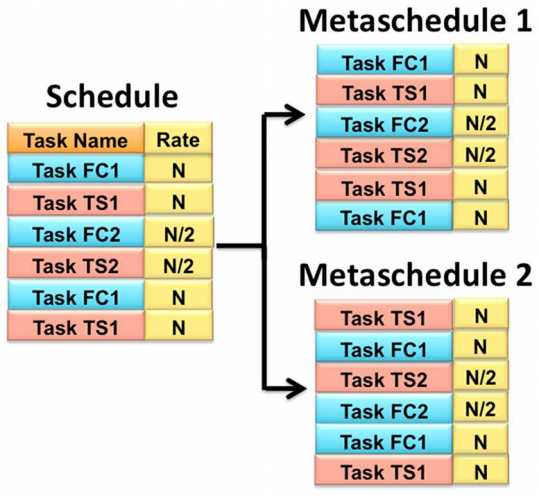

Fig. 4: Multiple Execution Schedules

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.

4.3 Using Cache-Half life to Drive Cache-aware Metascheduling

While metascheduling can be used to produce new execution schedules that continue to meet scheduling constraints, many of these schedules will not improve, and may even reduce the cache-hit rate and increase execution time of the system. We define a new heuristic, referred to as the cache half-life that can be used to drive the metascheduling process.

Cache Half-Life. We now explain the key factors that impact cache hit rate in integrated architectures and multi-tenant systems. 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 [22].

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.

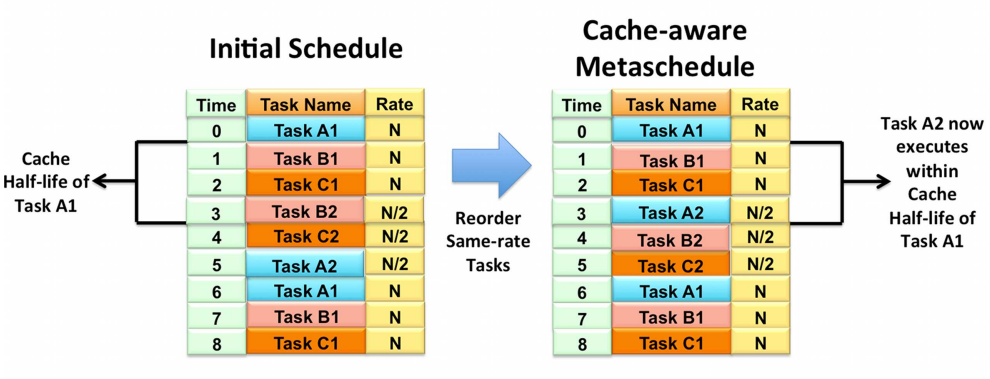

Fig. 5: Using Cache Half-life to Drive Cache-aware Metascheduling

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.

Our cache-aware metascheduling algorithm uses the cache half-life to increase the cache hit-rate and reduce system execution time. We attempt to maximize the number of tasks of the same application that execute before the cache half-life of the initial execution task expires.

For example, as shown in Figure 5 task A1 of Application 'A' executes at timestamp 0. The cache half-life of task A1 is 3 timestamps. As a result, for at least timestamps 2-4 data from Application 'A' will persist in the cache. Any tasks that share data with task A1 could use the data stored in the cache rather than accessing the data from main memory, resulting in a cache hit and reducing system execution time.

In this example, moving Task A2 from timestamp 5 to timestamp 3 will give Task A2 the opportunity to take advantage of the data cached by Task A1, resulting in a potentially higher cache hit-rate without violating scheduling constraints. Conversely, moving Task A2 to timestamp 4 will not increase the cache hit-rate as most or all of the data written by Task A1 to the cache will have been overwritten by this point due to tasks of other applications executing.

5 Empirical Results

This section analyzes the results of a performance analysis of integrated applications in multiple systems with different execution schedules generated through metascheduling. These systems also differ in the amount of memory shared between tasks. We investigate the impact of cache-aware metascheduling on L1 cache misses and runtime reductions for each system.

5.1 Overview of the Hardware and Software Testbed

To examine the impact of cache-aware metascheduling on integrated application performance, we collaborated with members of the Lockheed Martin Corporation to generated multiple systems that mimic the scale, execution schedule and data sharing of modern flight avionics systems. We specified the number of integrated applications, number of tasks per application, the distribution of task priority, and the maximum amount of memory shared between each task for each system. Together these integrated applications comprise a representative avionics system. We also developed a Java-based code generator to synthesize C++ system code that possessed these characteristics.

Figure 6 shows how the generated systems included a priority-based scheduler and multiple sample integrated applications that consisted of a variable number of periodic avionic tasks. Rate monotonic scheduling was used to create a deterministic priority

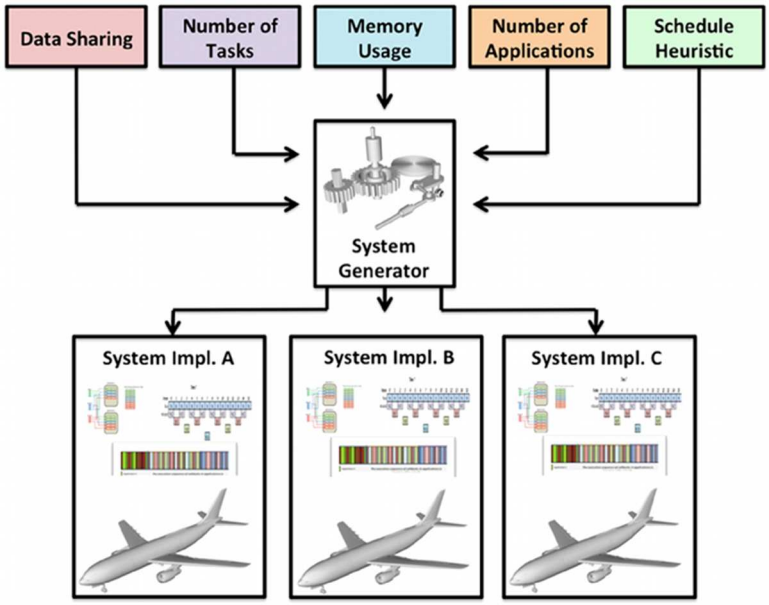

Fig. 6: System Creation Process

based schedule for the generated tasks that adheres to rate monotonic scheduling requirements. The systems then 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, every 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. VTune is a profiling tool that is capable of calculating the total number of times an instruction is executed by a processor.

For example, to determine the L1 cache misses of System A, we compiled and then executed it with VTune configured to return the total times that the instruction MEM_LOAD_REQUIRED.L1D_MISS is called. The data sharing and memory usage of these integrated applications, as well as the metacheduling strategy, are all parameterized and varied to generate a range of test systems. We use these simulated systems to validate cache-aware metascheduling by showing that taking into account data sharing when selecting a metaschedule performance in terms of execution time and cache misses.

System Size vs. Cache Size. The amount of memory required for the system has a major impact on the caching efficiency of the system. For example, consider a system that requires 6 kilobytes of memory executing on a processor with an L1 cache of 60 kilobytes. Assuming this is the sole executing system, all of data can be stored in the cache, leaving 90% of the cache free. The cache effects would therefore be the same for any system that does not require memory that exceeds the available memory in the cache.

Cache-aware metascheduling currently does not take into account available cache size since this may vary drastically from platform to platform. For our experiments, we require that memory requirements of all generated software systems exceed the memory available in the cache. Otherwise, the cache could store all of the data used by all applications simultaneously, removing any contention for cache storage space, which is unrealistic in industry systems.

Data Sharing Characteristics. The data shared between applications and shared between tasks of the same integrated application can greatly impact the cache effectiveness of a system. For example, the more data shared between two applications, the more likely the data in the cache can be utilized by tasks of the applications, resulting in reduced cache misses and faster system runtime. The system described in Section 2 prohibits data sharing between tasks of different integrated applications.

All systems profiled in this section are also restricted to sharing data between tasks of the same application. Integrated applications that exchange a great deal of common message data, however, are likely to share memory. To account for this sharing, our future work is examining cache-aware metascehduling strategies that account for these architectures.

Task Execution Schedule. The execution schedule of the software tasks of the system can potentially affect system performance. For example, assume there are two integrated applications named App1 and App2 that do not share data. Each application contains 1,000 task methods, with tasks of the same application sharing a large amount of data. The execution of a single task stores enough memory to completely overwrite any data in the cache, resulting in a cache half-life of 1.

When a task from App1 executes it completely fills the cache with data that is only used by App1. If the same or another task from App1 executes next, data could reside in the cache that could potentially result in a cache hit. Since no data is shared with App2,

however, executing a task from App2 could not result in a cache hit and would overwrite all data used by App1 in the class. We predict that multiple execution schedules therefore effect performance differently in terms of cache hit-rate and execution time.

5.2 Experiments: Determining the Impact of Cache-aware Metascheduling on 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

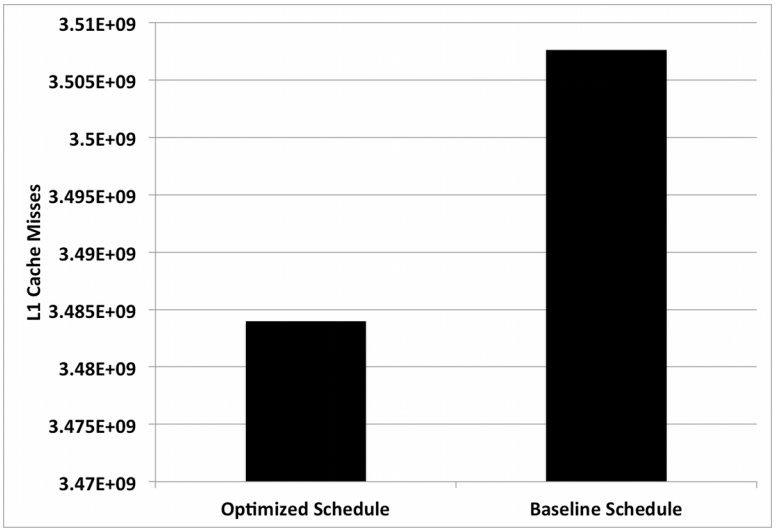

Fig. 7: Execution Schedules vs L1 Cache Misses

scheduling was use to create the baseline schedule. This cache-aware metascheduling was then applied to reorder same rate tasks to increase the temporal proximity between executions of tasks that share data to the Optimized schedule.

Experiment 1: Using Cache-Aware Metascheduling to Reduce Cache Misses. This experiment measures the impact of applying cache-aware metascheduling on the total L1D cache misses generated by an execution schedule.

Hypothesis: Increasing temporal locality through cache-aware metascheduling will result in less cache misses. Altering the task execution schedule can raise or lower the temporal locality of a sequence of data accesses. This change in temporal locality could potential affect the cache hit-rate resulting from executing a specific schedule. One way to potentially raise temporal locality is to increase the instances in

which a task executes before the cache half-life of a previous task with which it shares memory expires. We hypothesize that increasing temporal locality through cache-aware metascheduling will result in less cache misses.

Experiment 1 Results. We hypothesized that using cache-aware metascheduling to increase temporal locality would reduce the number of cache misses. Figure 7 shows the L1 cache misses for both execution schedules. The baseline execution schedule resulted in $3.5076x10^9$ L1 cache misses while the 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.

Experiment 2: Reducing Execution Time with Cache-Aware Metascheduling. This experiment measures and compares the total execution time of a system execution schedule generated with rate monotonic scheduling and the schedule resulting from applying cache-aware metascheduling.

Hypothesis: Using cache-aware metascheduling to increase temporal locality of schedule will reduce execution time. While Experiment 1 showed that applying cache-aware metascheduling can reduce cache misses, the impact of cache-aware metascheduling on system execution time remains unclear. We hypothesize that the schedule generated with cache-aware metascheduling will execute faster than the schedule generated with normal rate monotonic scheduling.

Experiment 2 Results. Figure 8 shows the average runtimes for the different execution schedules. As shown in this figure, the task execution order can have a large im-

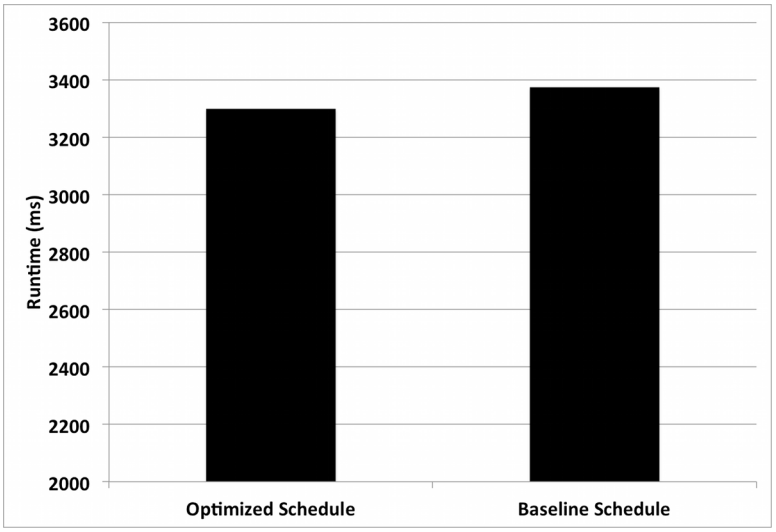

Fig. 8: Runtimes of Various Execution Schedules

pact on runtime. The baseline execution schedule executed in 3,374 milliseconds. The Optimized execution schedule completed in 3,299 milliseconds, which was an 2.22% reduction in execution time from the baseline execution schedule. These results demonstrate that applying cache-aware metascheduling can reduce the total execution time of a schedule.

Experiment 3: Impact of Data Sharing on Cache-Aware Metascheduling Effectiveness. This experiment measures the impact of data sharing on execution time reductions due to cache-aware metascheduling.

Hypothesis: Applying cache-aware metascheduling will reduce execution time for all levels of data sharing.

Figure 8 shows the execution time of two execution schedules at only 20% data sharing. Data sharing of industry systems, however, may vary to a large extent. Therefore, we created 10 other systems with different data sharing characteristics. We hypothesize that cache-aware metascheduling will lead execution time reductions regardless of the amount of data shared between tasks.

Experiment 3 Results. The execution time for the baseline and Optimized schedules is shown in Figure 9. The 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 systems, such as integrated avionics systems, regardless of cost constraints, restricted access to software source code, real-time constraints, or instruction level-knowledge of the underlying architecture.

6 Related Work

This section compares the cache-aware metascheduling and its use for cache optimization with other techniques for optimizing cache hits and system performance, including (1) software cache optimization techniques and (2) hardware cache optimization techniques.

Software cache optimization techniques. Many techniques change the order in which data is accessed to increase the effectiveness of processor caches by altering software at the source code level. These optimizations, known as data access optimizations [13], focus on changing the manner in which loops are executed. One technique, known as loop interchange [30], can be used to reorder multiple loops to maximize the data access of common elements in respect to time, referred to as *temporal locality* [2, 31, 30, 24].

Another technique, known as loop fusion [25], is often applied to further increase cache effectiveness. Loop fusion maximizes temporal locality by merging multiple loops into a single loop and altering data access order [25, 12, 6]. Yet another technique for improving software cache effectiveness is to utilize *prefetch* instructions, which

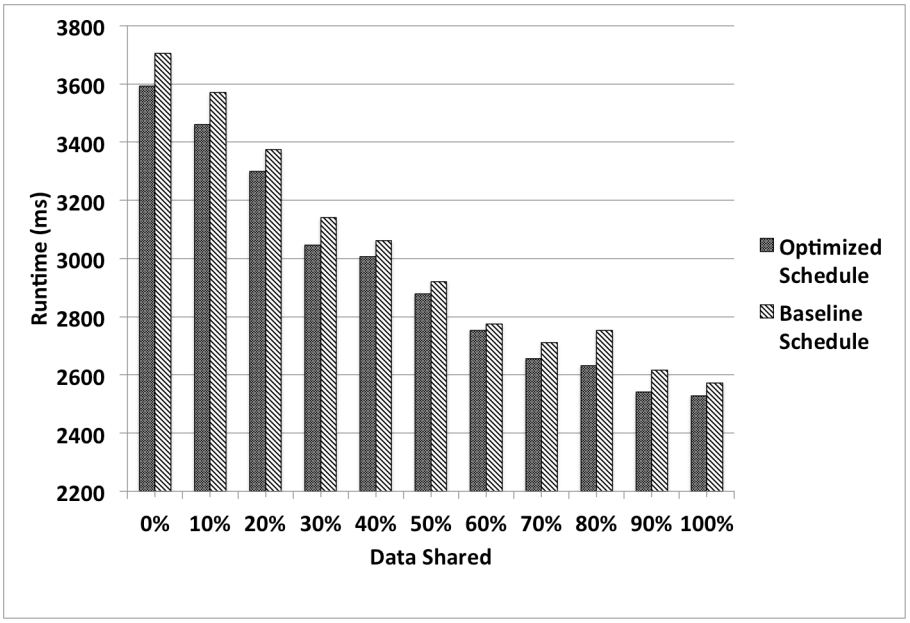

Fig. 9: Runtimes of Multiple Levels of Data Sharing

retrieves data from memory into the cache before the data is requested by the application [13]. Prefetch instructions inserted manually into software at the source code level can significantly reduce memory latency and/or cache miss rate [7, 9].

While these techniques can increase the effectiveness of software utilizing processor caches, they all require source code optimizations. Many systems, such as the avionic system case study described in Section 2, are safety critical and must undergo expensive certification and rigorous development techniques. Any alteration to these applications can introduce unforeseen side effects and invalidate the safety certification. Moreover, developers may not have source code proprietary applications that are purchased. These restrictions prohibit the use of any code-level modifications, such as those used in loop fusion and loop interchange, as well as manually adding prefetch instructions.

These techniques, however, demonstrate the effects of increasing temporal locality on cache effectiveness and performance. cache-aware metascheduling can be used as a heuristic to change the execution order of the software tasks to increase cache effectiveness and performance by ordering the tasks in such a way that temporal locality is increased. The fundamental difference, however, between using cache-aware metascheduling for cache optimization 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 application re-certification.

Hardware cache optimization techniques. Several techniques alter systems at the hardware level to increase the effectiveness of processor caches. One technique is to

alter the *cache replacement policy* processors use to determine which line of cache is replaced when new data is written to the cache. Several policies exist, such as Least Recently Used (LRU), Least Frequently Used (LRU), First In First Out (FIFO), and random replacement [1, 11].

The cache replacement policy can substantially influence DRE system performance. For example, LRU is effective for systems in which the same data will likely be accessed again before enough data has been written to the cache to completely overwrite the cache. Performance gains will be minimal, however, if enough new data is written to the cache such that previously cached data is always overwritten before it can be accessed [26]. In these cases, a random replacement policy may yield the most cache effectiveness [26].

Moreover, certain policies are shown to work better for different cache levels [1], with LRU performing well for L1 cache levels, but not as well for large data sets that may completely exhaust the cache. Unfortunately, it is hard—and often impossible for users—to alter the cache policy of existing hardware. Cache replacement policies should therefore be considered when choosing hardware to maximize the effects of cache optimizations made at the software or execution schedule level.

Cache-aware metascheduling does not alter the cache replacement policy of hardware since altering the hardware could invalidate previous safety certifications, similar to altering software at the source code level. Moreover, cache-aware metascheduling can be used a heuristic to increase temporal locality by altering the task execution order schedule. While many replacement policies exist, the metascheduling strategies we apply assumes an LRU replacement policy. Our future work is examining the impact of cache replacement policy on the performance gains of schedules altered via cache-aware metascheduling.

7 Concluding Remarks

Processor data caching can substantially increase performance of systems (such as integrated applications and other multi-tenant environments) in which SLAs provide response time assurance and QoS policies that restrict resource sharing. 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.

This paper presents a cache-aware metascheduling to increase the performance gains due to processor caching of integrated applications. We empirically evaluated four task execution schedules generated with cache-aware metascheduling in terms of L1 cache misses and execution time. We learned the following lessons from increasing cache hit-rate with cache-aware metascheduling:

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

• **Relatively minor system knowledge yields effective metascheduling strategies for increasing cache performance.** Developing cache-aware metascheduling strate-

gies 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 cache-hit rate improvements due to cache-aware metascheduling should be developed.** The task execution schedule was shown to have a large impact on system performance. Our future work is examining algorithmic techniques for optimizing cache-aware metascheduling to determine the optimal execution order for tasks in specific systems (such as multi-tenant environments) to maximize cache hit rate.

• **Cache-aware metascheduling should be applied to cloud-based multi-tenant environments.** Given the similarities (such as response time requirements and data sharing restrictions) between integration applications and multi-tenant environments, we expect cache-aware metascheduling to also increase the efficiency of multi-tenant cloud environments. In future work, we will apply cache-aware metascheduling to multi-tenant clouds to determine what degree of performance enhancements can be achieved.

The source code simulating the integrated avionics system discussed in Section 5 can be downloaded at `ascent-design-studio.googlecode.com`.

References

1. G. Abandah and A. Abdelkarim. A Study on Cache Replacement Policies. 2009.
2. J. Allen and K. Kennedy. Automatic loop interchange. In *Proceedings of the 1984 SIGPLAN symposium on Compiler construction*, page 246. ACM, 1984.
3. 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.
4. A. Atlas and A. Bestavros. Statistical rate monotonic scheduling. In *Real-Time Systems Symposium, 1998. Proceedings., The 19th IEEE*, pages 123–132. IEEE, 1998.
5. 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.
6. 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.
7. T. Chen and J. Baer. Reducing memory latency via non-blocking and prefetching caches. *ACM SIGPLAN Notices*, 27(9):51–61, 1992.
8. S. Dhall and C. Liu. On a real-time scheduling problem. *Operations Research*, 26(1):127–140, 1978.
9. 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.
10. S. Ghosh, R. Melhem, D. Mossé, and J. Sarma. Fault-tolerant rate-monotonic scheduling. *Real-Time Systems*, 15(2):149–181, 1998.
11. F. Guo and Y. Solihin. An analytical model for cache replacement policy performance. In *Proceedings of the joint international conference on Measurement and modeling of computer systems*, pages 228–239. ACM, 2006.

12. 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.
13. M. Kowarschik and C. Weiß. An overview of cache optimization techniques and cache-aware numerical algorithms. *Algorithms for Memory Hierarchies*, pages 213–232, 2003.
14. Y. Lee, D. Kim, M. Younis, J. Zhou, and J. McElroy. Resource scheduling in dependable integrated modular avionics. In *Dependable Systems and Networks, 2000. DSN 2000. Proceedings International Conference on*, pages 14–23. IEEE, 2000.
15. 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.
16. 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.
17. 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.
18. 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.
19. 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.
20. S. Pingali, J. Kurose, and D. Towsley. On Comparing the Number of Preemptions under Earliest Deadline and Rate Monotonic Scheduling Algorithms. 2007.
21. J. Reineke, D. Grund, C. Berg, and R. Wilhelm. Timing predictability of cache replacement policies. *Real-Time Systems*, 37(2):99–122, 2007.
22. J. Robinson and M. Devarakonda. Data cache management using frequency-based replacement. *ACM SIGMETRICS Performance Evaluation Review*, 18(1):134–142, 1990.
23. P. Rodríguez-Dapena. Software safety certification: a multidomain problem. *Software, IEEE*, 16(4):31–38, 1999.
24. 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.
25. S. Singhai and K. McKinley. A parametrized loop fusion algorithm for improving parallelism and cache locality. *The Computer Journal*, 40(6):340, 1997.
26. J. Smith and J. Goodman. Instruction cache replacement policies and organizations. *IEEE Transactions on Computers*, pages 234–241, 1985.
27. 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.
28. D. Stewart and M. Barr. Rate monotonic scheduling. *Embedded Systems Programming*, pages 79–80, 2002.
29. Z. Wang, C. Guo, B. Gao, W. Sun, Z. Zhang, and W. An. A study and performance evaluation of the multi-tenant data tier design patterns for service oriented computing. In *IEEE International Conference on e-Business Engineering*, pages 94–101. IEEE, 2008.
30. 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.
31. 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.