The Journal of Systems and Software 98 (2014) 25–43

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

DRE system performance optimization with the SMACK cache efficiency metric

Hamilton Turner[a,*], Brian Dougherty[b], Jules White[b], Russell Kegley[c,1], Jonathan Preston[c,1], Douglas C. Schmidt[b], Aniruddha Gokhale[b]

[a] Virginia Polytechnic, United States
[b] Vanderbilt University, United States
[c] Lockheed Martin Aeronautics, United States

ARTICLE INFO

Article history:
Received 2 December 2013
Received in revised form 26 June 2014
Accepted 16 August 2014
Available online 27 August 2014

Keywords:
DRE
Deployment
Optimization
Heuristic
Cache

ABSTRACT

System performance improvements are critical for the resource-limited environment of multiple integrated applications executing inside a single distributed real-time and embedded (DRE) system, such as integrated avionics platform or vehtronics systems. While processor caches can effectively reduce execution time there are several factors, such as cache size, system data sharing, and task execution schedule, which make it hard to quantify, predict, and optimize the cache usage of a DRE system. This article presents SMACK, a novel heuristic for estimating the hardware cache usage of a DRE system, and describes a method of varying the runtime behavior of DRE system software without (1) requiring extensive safety recertification or (2) violating the real-time scheduling deadlines. By using SMACK as a maximization target, we were able to reduce integrated DRE system execution time by an average of 2.4% and a maximum of 4.34%.

1. Introduction

Current trends and challenges: Distributed real-time and embedded (DRE) systems, such as integrated avionics systems and vehtronics systems, are subject to stringent real-time constraints. To ensure these real-time requirements are met, these systems must minimize software execution time. One approach to reduce DRE execution time is to reduce the time spent loading data from memory by efficiently utilizing processor caching hardware.

Multiple design techniques have been researched for reducing system execution time by increasing processor cache utilization. For example, Bahar et al. (2005) examined several different cache techniques for reducing execution time by increasing cache utilization efficiency. Their experiments showed that efficiently utilizing a processor cache can result in as much as a 24% reduction in execution time. Likewise, Manjikian and Abdelrahman (1995) 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 (Reineke et al., 2007; Nayfeh and Olukotun, 1994; Sprangle et al., 2002) exist to increase how efficiently caches are utilized by modifying application source code to increase the *temporal locality* of data accesses, which defines the proximity with which shared data is accessed in terms of time (Kowarschik et al., 2003). For example, loop interchange and loop fusion techniques can be used to increase temporal locality of accessed data by modifying software application source code to change the order in which application data is written to and read from a processor cache (Kowarschik et al., 2003; Manjikian and Abdelrahman, 1995). Increasing temporal locality increases the probability that data common to multiple tasks will persist in the cache, resulting in reduced cache-misses and software execution time (Kowarschik et al., 2003; Manjikian and Abdelrahman, 1995).

In general, however, prior work has focused on source-code level modifications for single applications, which is problematic for DRE systems built from multiple integrated applications, such as the architecture shown in Fig. 1. Source code modifications in an integrated DRE system are infeasible due to 1) the proprietary nature of individual applications, and 2) safety requirements which necessitate extremely extensive certification after any source code modifications.

* Corresponding author. Tel.: +1 6158183577.
E-mail addresses: hamiltont@vt.edu, hamiltont@gmail.com, hturner0@vt.edu (H. Turner), briand@dre.vanderbilt.edu (B. Dougherty), julesw@dre.vanderbilt.edu (J. White), russell.b.kegley@lmco.com (R. Kegley), jonathan.d.preston@lmco.com (J. Preston), schmidt@dre.vanderbilt.edu (D.C. Schmidt), gokhale@dre.vanderbilt.edu (A. Gokhale).

[1] This work was sponsored in part by the Air Force Research Lab.

http://dx.doi.org/10.1016/j.jss.2014.08.031
0164-1212/

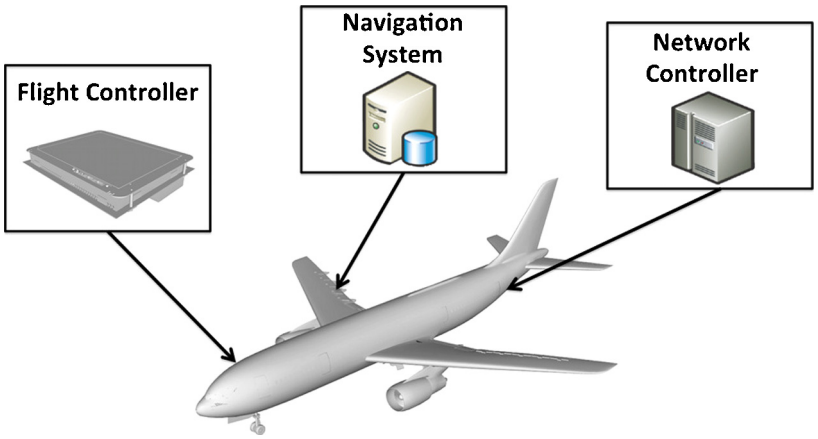

Fig. 1. Example of an integrated avionics system.

An integrated DRE system is composed of many individual applications, which are frequently provided by distinct subcontractors. License issues may prevent access to the source code, or prevent the right to modify or recompile the source code. Moreover, DRE system designers may not have the expertise to safely modify application-specific source code to improve the temporal locality of the source code.

Integrated DRE systems are often subject to stringent safety requirements, such as bounding the frequency at which hardware failures can occur and preventing code-level errors in application software, middleware, and operating systems. To help ensure predictable behavior, applications in integrated DRE systems must undergo a rigorous safety inspection process. After this process is completed and the result has been certified, any alteration to an application may invalidate the certification. Safety requirements thus pose a significant barrier to the use of cache optimization techniques that require source code alterations.

Solution approach → Heuristic-Driven Schedule Alteration of Same-rate Tasks to Increase Cache Utilization.

Priority-based scheduling techniques can help ensure DRE system software executes without missing real-time deadlines. For example, rate-monotonic scheduling (Pingali et al., 2007) is a technique for creating task execution schedules that satisfy real-time 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 (Dhall and Liu, 1978). Fig. 2 shows two different valid task execution schedules generated with rate-monotonic scheduling. As Task A1 and Task B1 share the same priority, their execution order can be swapped without violating real-time constraints. Each scheduling problem with at least one valid execution order therefore has a number of equally valid permutations, which can be created by rearranging the order of same-rate tasks. This research shows that it is possible to improve the cache hit-rate of integrated DRE systems by selecting a valid execution order that increases the temporal locality of data accesses.

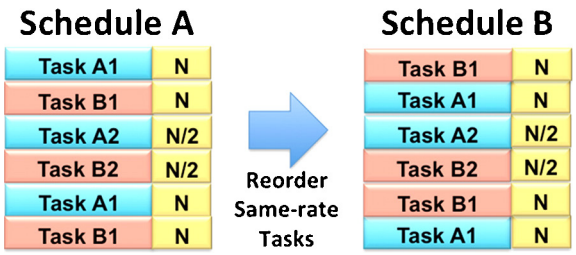

Fig. 2. Valid task execution schedules.

Critically, our approach for improving cache utilization of integrated DRE systems does not require any source code modifications. This allows DRE system integrators to improve integrated DRE system performance without invalidating application safety recertification, requiring legal agreements to share source code, or needing specialized expertise for each application. Our modifications only change the execution order of same-rate tasks, which results in a system that is still a valid rate-monotonically scheduled application as interchanging tasks of identical utilization does not change the value of the Liu–Layland bound (Dhall and Liu, 1978). To select the best valid schedule from the set of all possible valid schedules, we introduce the *System Metric for Application Cache Knowledge* (SMACK) heuristic, which measures the maximum possible cache utilization of a given execution schedule. SMACK considers several factors, such as cache size, data sharing, and task execution schedule, to provide developers with a way to determine which orderings of same-rate tasks have higher potential for cache utilization. SMACK enables DRE system designers to manipulate models of system runtime behavior without having to repeatedly construct and measure potential system implementations, thereby yielding performance increases without expending undue effort on multiple implementations.

This article provides the following contributions to integrated DRE system creation and deployment:

- We present a real-time scheduling heuristic for same-rate tasks that satisfies real-time scheduling constraints and safety requirements, increases cache hits, and requires no new hardware, software, or middleware. Proper use of this heuristic enables reduction in execution time of rate-monotonically scheduled DRE systems without requiring recertification.
- To motivate the need for scheduling enhancements to improve the efficiency of cache utilization in integrated DRE systems, we present an industry case study of an integrated avionics system in which modifications to the constituent applications are prohibitively expensive due to safety certification requirements.
- We provide a formal methodology for calculating the "SMACK score", which quantifies the temporal locality of different execution schedules for the same-rate tasks in an integrated DRE system.
- We present empirical results of the performance of 44 simulated integrated DRE systems, each with different data sharing characteristics and task execution schedules, and demonstrate that the calculated SMACK score correlates with an increased cache

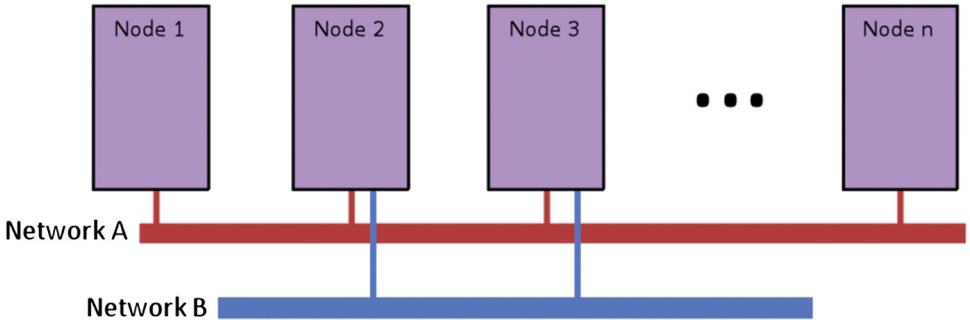

Fig. 3. Avionics system deployment architecture.

hit rate, thereby providing a means for comparing potential system implementations in terms of expected cache effectiveness without requiring actual implementation.
- We show that altering the task execution schedule to optimize the SMACK score can improve the cache hit-rate of integrated DRE systems while simultaneously satisfying safety constraints and real-time requirements.

This work extends our prior publication on cache efficiency (Dougherty et al., 2011). This work formalizes the prior research with a mathematical model of SMACK, and generalizes the heuristic to work with a more diverse range of architectures. Specifically, we addressed the need to optimize multiple compute nodes simultaneously, whereas our prior publication focused on the effect of using SMACK on a single multitenant compute node. We also provide additional empirical evaluations on the effect of using the SMACK heuristic in various production systems by showing the range of effects resulting from using our SMACK heuristic under various data sharing and cache contention factor values.

Paper organization: The remainder of the paper is organized as follows: Section 2 describes how a typical integrated avionics system is designed to meet real-time deadlines and provide required safety constraints; Section 3 summarizes the challenges of creating a metric that predicts integrated DRE system performance at design time and guides execution schedule modifications; Section 4 explains how the SMACK metric can be calculated to predict DRE system performance and applies it to create cache-effective execution schedules; Section 5 analyzes the results of experiments that demonstrate the effectiveness of SMACK for predicting cache hits and runtime reductions; Section 6 compares SMACK with related work; and Section 7 presents concluding remarks.

2. Integrated avionics system case study

This section presents a case study that shows current practices for integration of multiple applications in modern avionics systems, such as the system shown in Fig. 1. This case study also describes a generic method for integrating applications while simultaneously ensuring that avionic safety constraints and real-time scheduling requirements are met. Section 5.4 describes our contributions to this method of application integration, which increase cache hit-rates while maintaining the goals of real-time scheduling and safety constraints.

2.1. Avionics system integration architecture

The avionics system in our case study is realized as a set of computing nodes connected by one or more networks, as shown in Fig. 3. Each node contains a single core computing element consisting of a Consumer-Off-The-Shelf (COTS) processor with main memory and a two-level cache. The cache can be assumed to have the characteristics shown in Table 1.

Table 1
Hypothetical cache memory characteristics.

Line size	32 bytes
Associativity	8-way
Total Size	1 MB
Instruction/data allocation	Shared
Replacement policy	Pseudo-least recently used

While it is possible for there to be multiple network connections available between the computing nodes in a system, we assume a single, high-bandwidth connection between nodes, as the other connections typically carry much less traffic. The network can be assumed to be fiber optic with the characteristics shown in Table 2.

2.1.1. Software integration architecture

The software integration is structured as a set of (primarily) singleton objects that interact through a publisher/subscriber protocol over the fiber optic network. While there are a few cases of a given class having more than one instance in the system, these are exceptional and address specific goals. A given application in the system typically corresponds to a single object, though some applications may comprise a small number of separate objects due to their size or other structural criteria.

Each object in the system executes on a single node, communicating with other objects through the use of a publisher/subscriber network protocol that is managed transparently by the system middleware. Objects are overwhelmingly structured as collections of tasks which operate on a set of private data structures. All communication of data between two objects is asynchronous and is organized through the publisher/subscriber message protocol, not through explicitly shared data structures.

Typically, an object will have a tasks identified for each message it can receive from some other object, from the system, or from some external source. When that message is received by the node on which the object resides, the system middleware arranges for the registered task for that message to be executed. The system uses a real-time operating system (RTOS) which implements the time and space partitioning specified in ARINC 653, Avionics Application Software Standard Interface (Committee, 1997).

To prevent interaction between integrated software applications the architecture in Fig. 4 uses the ARINC 653 standard to allow applications to operate at different safety levels by partitioning space (memory) and execution time along safety criteria.

Table 2
Proposed fiber optic network characteristics.

Bandwidth	400–1000 Mbps
Topology	Switched fabric
Priority	1 (e.g., first-come first-served)

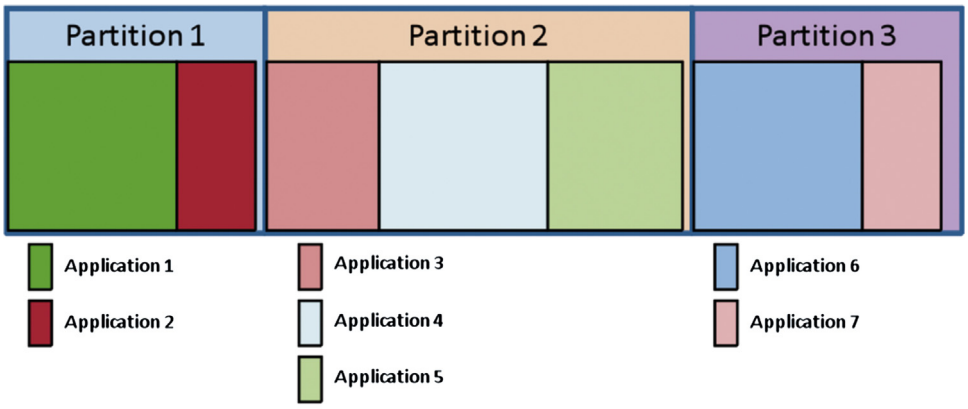

Fig. 4. Time and space partitioned system architecture.

Applications with lower safety levels cannot be integrated into the same partition as those with higher levels.

The case study also allows multiple time and space partitions to execute on each node in the network. Fig. 4 shows an example system structure with three partitions, in which seven different software applications are implemented.

Each partition is allocated a fixed time slot during which only its applications can be executed. The sum of the partition durations usually add up to the base frame duration discussed in the next section. The underlying real-time operating system (RTOS) "activates" partitions in the specified sequence, allowing the integrated applications inside each partition to execute in turn, then repeats the sequence.

2.2. Runtime integration architecture

To control application execution, each node executes its own system scheduler, which is part of either the RTOS or middleware. The system scheduler on each node in our case study implements a rate-based pattern for integrating software execution, which decomposes time into a series of numbered frames of equal duration as shown in Fig. 5. Each frame is of *duration* seconds, such as 60 s, and this duration span is said to be the base rate of the system. For example, if the system is scheduled with 75 Hz base rate, software that is executed at that rate is also said to run at a frequency of 75 Hz.

At the start of each base frame two tasks are scheduled to execute sequentially. 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 as shown in Fig. 5 until the lowest rate software in the system has completed. All scheduling of application avionic software tasks in the system occurs in this manner.

As previously noted, objects consist of collections of callbacks that operate on shared data. Each callback is scheduled to run for one of two reasons:

- The message registered for that callback has been received by the node.
- The callback is registered to be executed at a fixed frequency.

We define for simplicity that objects process messages at the same rate at which they are sent in the system, and further consider that messages are always transmitted at their defined rate, thus each callback in an object is executed at some fixed rate. The system sets scheduling priority according to the frequency at which software executes. For example, a callback which executes at rate 75 Hz is scheduled with a higher priority than callbacks at 37.5 Hz, which in turn run at a higher priority than callbacks at rate 18.75 Hz, and so on to the lowest defined rate in the system. Fig. 4 shows the effect of priority-based interleaving of task from multiple applications in a partition. In particular, it shows how multiple tasks from Application 1 in Partition 1 may execute in a row, followed by one or more tasks from Application 2, and so on. Our avionics case study assumes integrated applications cannot influence interleaved order other than specifying priority, which is determined by execution frequency. This pattern is repeated in all partitions in the system. (Fig. 6).

Fig. 7 shows the interleaved execution of software tasks. 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.

Safe coding practices require each object to allocate all the data structures it intends to use when the system starts. Henceforth,

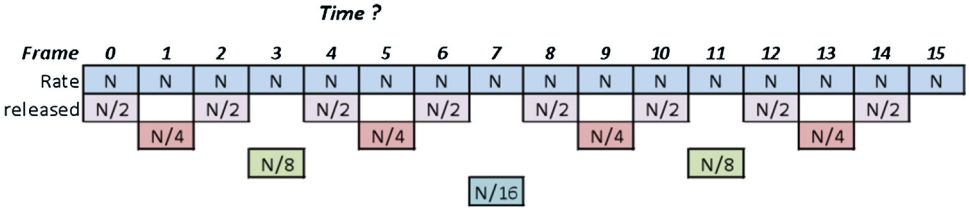

Fig. 5. Periodic scheduler interleaves task execution.

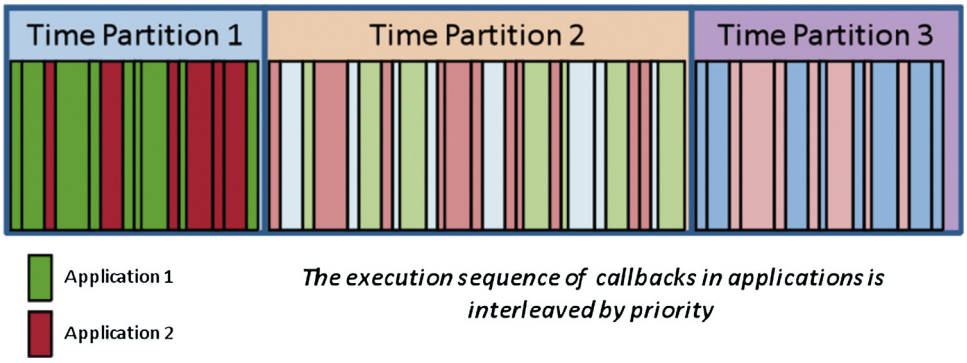

Fig. 6. Execution interleaving inside time partition.

data structures are neither released nor moved in the address space. Similarly, program text (instructions) are statically linked and do not move after they are loaded into main memory. Message buffers are also allocated when the system starts and do not move thereafter. In addition, if two or more objects on a given node subscribe to a received message, the two objects share a single read-only copy of the message.

3. 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. Mission-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. For example, the case system in Section 2 contains several factors, such as system recertification, unknown data coupling characteristics, and strict scheduling requirements, that make it hard to construct optimization techniques for integrated DRE systems. This section describes three challenges that must be overcome to ensure a processor cache optimization technique is applicable for safety-critical DRE systems.

3.1. Challenge 1: altering applications may invalidate safety certification

Integrated DRE systems, such as the avionics case study in Section 2, are often safety-critical. While software crashes may cause minor inconveniences for most system users, unpredictable system behavior in integrated avionics systems can yield catastrophic system failures. For example, an exception that forces a word processor to close unexpectedly may cause mild frustration or minor data loss, whereas a faulty system flight controller could cause an airplane to crash. To prevent these catastrophes, the software and hardware applications of safety-critical integrated avionics are heavily certified to ensure that as long as the software and hardware are not modified, the system will execute in a safe, predictable manner.

Existing cache optimization techniques, such as loop fusion and data padding (Kennedy and McKinley, 1994; Panda et al., 2002), require modifications to the applications to increase cache utilization and performance. Modifications of integrated applications, however, may void previous application safety certifications.

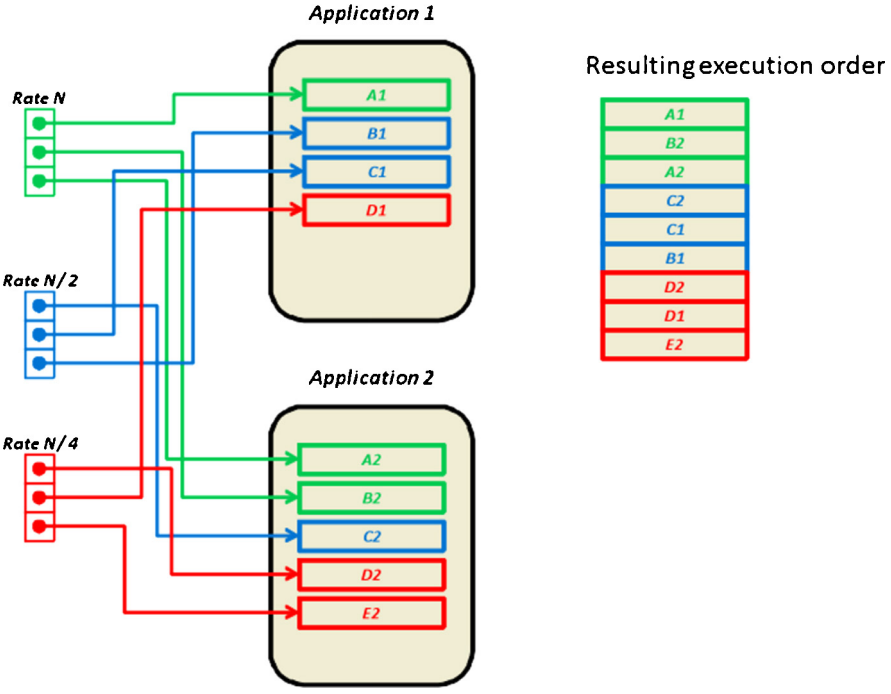

Fig. 7. Interleaved execution order is repeatable.

Re-certification of applications is a slow and expensive process, which increases cost and delays deployment. What is needed, therefore, are techniques that alter the DRE system to optimize a predictive performance metric while leaving the hardware and software of the integrated applications unmodified. These techniques would then not require recertification application for each application. However, there would need to be a system integration recertification performed to ensure that no unexpected behavior, such as network thrashing, emerge when all of the optimized applications are executed concurrently. However, recertification of the system is a minor cost compared to recertification of each application. Section 5.4 describes how altering task execuion schedule increases system performance through better cache utilization, resulting in decreased system runtime while avoiding the need for costly system recertification.

3.2. Challenge 2: data sharing characteristics of software applications vary

In contrast to small, stand-alone software applications, integrated DRE systems are comprised of several systems made up of many applications that work together. The data sharing characteristics of applications can potentially have a large impact on system performance due to processor caching. System scale and the opacity of the competive integration eco-system, however, makes data coupling analysis cumbersome and time consuming. What is needed, therefore, are optimization techniques that increase performance regardless of data sharing characteristics *a priori*. Sections 5.3 and 5.5 describe how altering task execution schedule effects the DRE system execution time and cache performance of systems, regardless of the amount of data sharing between tasks.

DRE system developers often focus on a small portion of the applications in a single subsystem, and are unaware of the inner workings of applications developed by other suppliers. For example, the software that controls the system flight controller may consist of applications developed by multiple companies that compete for sponsor funds. It is uncommon for these competing suppliers to interact with each other when making their source code design decisions. System integrators thus often lack knowledge of key implementation details within the software applications that comprise the final DRE system. Therefore, direct manipulation of applications is infeasible and dangerous.

3.3. Challenge 3: 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 (Stewart and Barr, 2002; Ghosh et al., 1998; Naghibzadeh, 2002). This constraint prohibits many simple solutions that would greatly increase cache hit-rate but would cause the system to behave unpredictably, with potentially catastrophic results. What is needed, therefore, are optimization techniques that can be applied and re-applied when necessary to increase the cache hit-rate and decrease system execution time for any set of system task priorities. Section 5.4 describes how task execution schedules can be altered to increase the temporal locality of task execution schedules without violating real-time constraints.

For example, if Task A is assigned a priority that is twice the rate of Task B, then Task B must execute twice before Task A executes a second time. This method ensures that tasks of higher priority will execute completely, and that tasks of lower priority will not be starved of resources due to continuous preemption. Any schedule optimization technique must result in a schedule that does not violate the restrictions outlined above.

4. Cache aware metascheduling with SMACK to improve cache utilization efficiency

Altering the task execution schedule of applications can increase cache hits without requiring code-level modifications, but may violate real-time constraints. Rate-monotonic scheduling, however, allows for same-rate tasks to be scheduled arbitrarily while ensuring real-time constraints are satisfied (Dhall and Liu, 1978). In this section, we examine the re-ordering of same rate tasks, or *metascheduling*, which permutes existing execution schedules to rapidly generate new valid schedules.

Metascheduling, however, often generates multiple valid task execution schedules such as those shown in Fig. 8. As can be seen, determining which task execution schedule that will yield the largest increase in temporal locality and best utilize the cache is not always immediately apparent. Metrics can be developed, however, to quantify temporal locality of multiple execution schedules.

This section presents a metric for quantifying the temporal locality of task execution schedules of integrated DRE architectures that can guide the selection of a metaschedule to increase the efficiency of processor cache utilization. We first describe how scheduling decisions affect temporal locality and subsequently cache utilization efficiency. We then provide a formal definition of a metric for quantifying the temporal locality of task execution schedules. Finally, we show how metascheduling can use this metric as a heuristic to generate task execution schedules that increase cache utilization efficiency without violating real-time constraints or requiring code-level modifications.

Each node of the avionics system case study described in Section 2 consists of multiple partitions of executing applications, as shown in Fig. 4. The tasks that comprise these applications described in Section 2.2 are scheduled for execution with a priority-based scheduler local to each node. As tasks execute, cache hits may occur between tasks that share a common partition. These cache hits can yield substantial reductions in the total execution time of the partition.

Each partition in Section 2.2 executes for a fixed-time duration determined by the expected execution time for all tasks in the partition. This fixed-time duration, however, does not take into account cache hits. The size of the partitions can be more finely configured by factoring in the expected execution time savings due to cache hits.

4.1. Goal: a cache hit characterization metric for software deployments

To predict the relative performance that a specific task scheduling will yield we developed the *System Metric for Application Cache Knowledge* (SMACK). SMACK can predict relative performance for multiple execution schedules, such as those shown in Section 2.2. Fig. 8. This metric can the be used as a heuristic for metascheduling to determine task execution schedules that increase cache utilization efficiency.

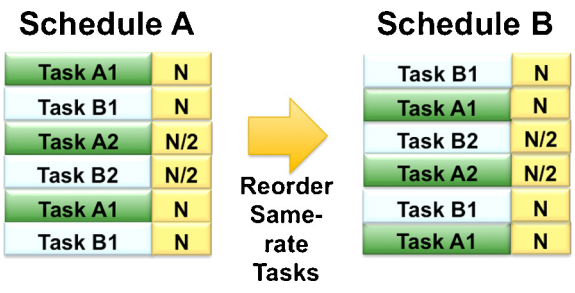

Fig. 8. Multiple execution schedules.

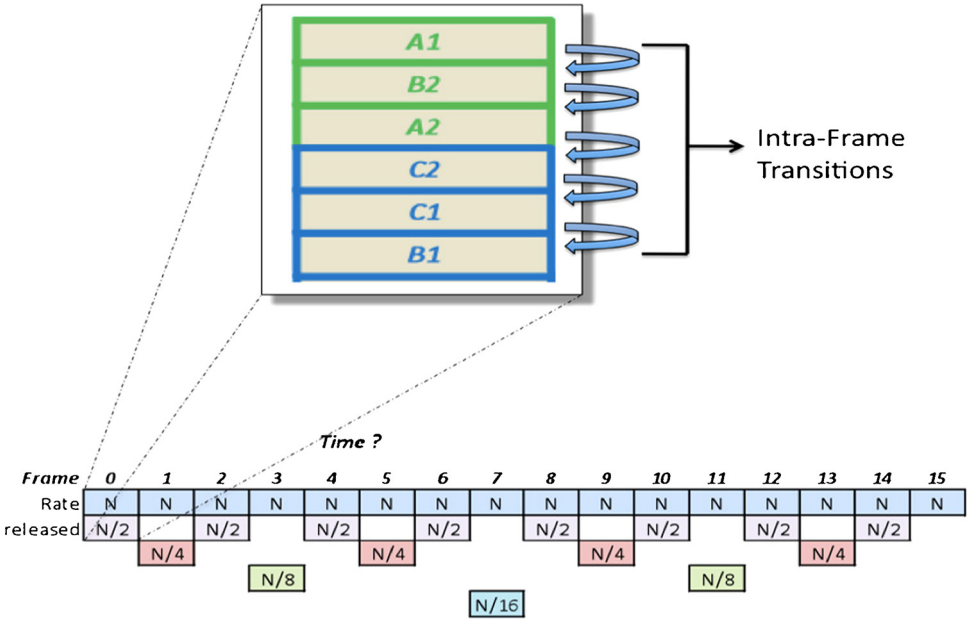

Fig. 9. Scheduling with intra frame transitions.

4.2. How real-time schedules may impact cache hits

Section 2 shows how the structure of our avionics system case study consists of multiple, separate nodes, where each node is divided into separate partitions in which applications execute. Section 2.2 describes how each application executing in a partition contains tasks executing at various rates and priorities. Each node is equipped with a priority-based scheduler that determines the execution order of these tasks. Different execution schedules can yield more or fewer cache hits. While the reduction in system execution time resulting from a successful cache hit may differ from node to node, we assume it is the same for applications executing on the same node.

A task execution schedule is divided into minor frames and major frames. A minor frame is a subset of tasks that execute before the next set of tasks can begin executing, and is analogous to the base frames of the system described in Section 2. A major frame is the set of minor frames that must execute before all tasks of all rates are guaranteed to execute, similar to the partitions of an integrated DRE system. For example, Fig. 7 shows an execution for a set of tasks. Tasks A1 through B1 execute in the same minor frame, whereas the major frame is the execution of all tasks from minor frame 0 to minor frame 15.

4.2.1. Intra frame transitions

Our case study examines the impact of transitioning between tasks on the effectiveness of the data cache. Transitioning between tasks executing in one or more minor frames potentially results in a cache hit. For example, Fig. 9 shows a scheduling of multiple tasks with six tasks scheduled to execute in the same minor frame.

Task A1 executes and then Task B2 is scheduled to execute next. We call the transition from A1 executing to B2 executing an *intra frame transition* since task A1 and B2 share the same minor frame. If Tasks A1 and B2 require common data, however, there is potential for a cache hit.

4.2.2. Extra frame transitions

Tasks may also be scheduled to execute in separate minor frames. A cache hit may result from a transition from the final task to execute in one minor frame and the first task to execute in the next minor frame. We call this type of transition between separate minor frames an *extra frame transition*. For example, Fig. 10 shows two sets of tasks executing in separate minor frames. An extra frame transition exists between Task B1 and A1. The probability of a cache hit occurring due to extra frame and intra frame transitions, however, differs based on the cache contention factor.

4.2.3. Cache contention factor

Transitioning a new task onto the processor can result in a cache hit, as described in Sections 4.2.1 and 4.2.2. We define the cache contention factor, *CCF*, to estimate how many consecutive transitions can potentially lead to a cache hit before all cached data from the original task is invalidated, as shown in Eq. (1). The cache contention factor is defined by the memory usage of the software, the size of the cache, and the cache replacement policy.

$$CCF = \frac{CS}{(DW/|T|) * (1 - DS)} \quad (1)$$

In this equation, *CCF* 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 in the two tasks.

After invalidation, the probability of a cache hit is reduced to 0. 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 (Robinson and Devarakonda, 1990).

For example, assume there are 5 applications consisting of 2 tasks, each consuming 2 kilobytes of memory of a 64 kB cache. The hardware uses a *Least Recently Used* (LRU) replacement policy. This policy replaces the cache line that remained the longest without being read when new data is written to the cache. The cache contention factor formulation will differ for other cache replacement policies. Executing the tasks will require writing 20 kB to memory. Since the cache can store 64 kB of data, all data from all applications can remain in the cache. The cache contention factor in this case would be 32, as the data from any one task might remain in the cache for 32 consecutive tasks. After 32 the data from that

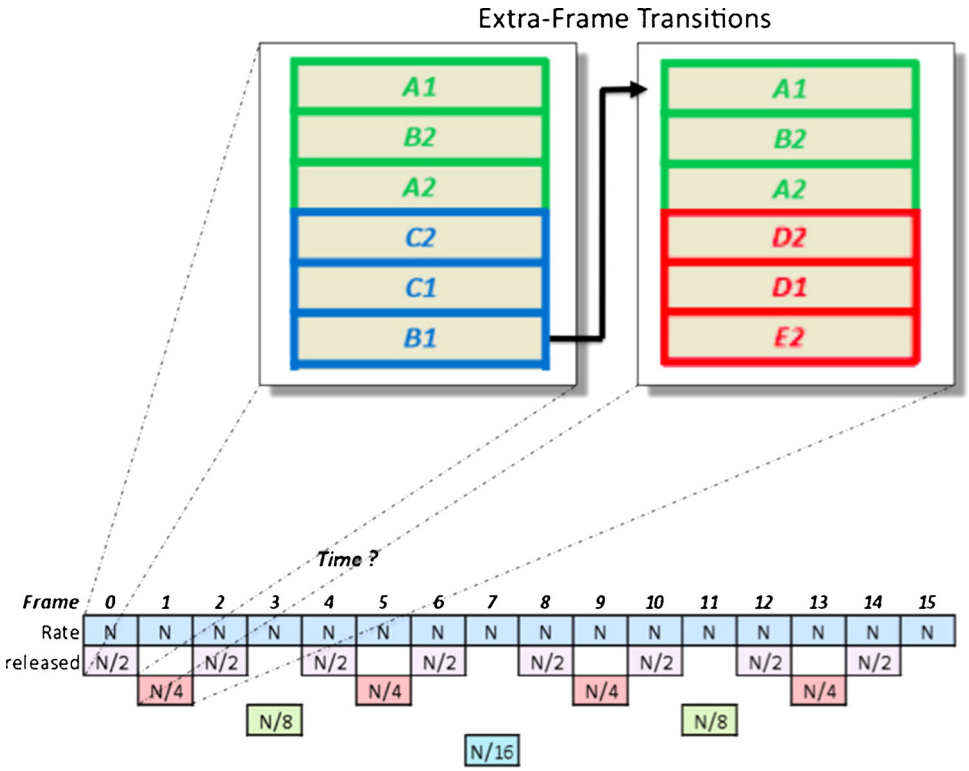

Fig. 10. Scheduling with extra frame transition.

execution of that task will no longer remain in the cache, although other tasks which accessed the same data may cause it to remain in the cache.

4.2.4. Determining if tasks overlap

The integrated avionics system described in Section 2 assumes that tasks of different applications do not share any data. Cache hits can therefore only occur if two tasks share the same application. While Eq. (1) identifies a set of new task transitions that may result in a cache hit, Eq. (2) further reduces this set to only contain new task transitions that belong to the same application as the current task being examined. These two equations constitute the necessary conditions for a cache hit to be possible. Eq. (2) returns 1 if two tasks are a part of the same application and 0 if they are not.

$$O(t_i, t_j) = \begin{cases} 1 & \text{if } t_i \equiv t_j \\ 0 & \text{if } t_i \neq t_j \end{cases} \tag{2}$$

4.2.5. 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. Eq. (3) defines the maximum cache hits that can be expected if a task of an application executes after another task of the same application. The total set of transitions for a minor frame F is given by $t(F)$.

$$CHit(t(F_i)_j, t(F_x)_y) = DS * DR(t(F_x)_y) \tag{3}$$

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.

4.2.6. Determining total cache hits

Each intra frame and extra frame transition yields a probability of a cache hit, based on the DRE system's cache contention factor and the task executing in the frame. We consider each task execution, and sum the total probabilistic cache hits for all sets of tasks in all partitions of a given node. Naturally, not all of the probabilistic cache hits are ensured to occur, so this is an upper bound. Each cache hit reduces the execution time of the system.

4.2.7. Cache hits due to intra frame and extra frame transitions

We calculate the maximum cache hits *CHit* for all intra frame and extra frame transitions in the major frame *MF*. After a task executes, the number of transitions that can occur before all data written by the task to the cache is invalidated is determined by the *CCF*. Each transition that occurs before the *CCF* is reached can therefore potentially yield a cache hit and must be investigated.

Determining which task executes k transitions from a task is shown in Eq. (4).

$$FR(F_{ij}, k) = \begin{cases} F\left[j + \left\lfloor \frac{i+k}{|F|} \right\rfloor\right]_{[(i+k)\%|F|]} & \text{if } i+k < M(MF) \\ F\left[j + \left\lfloor \frac{(i+k)-M(MF)}{|F|} \right\rfloor\right]_{[((i+k)-M(MF))\%|F|]} & \text{if } i+k \geq M(MF) \end{cases} \tag{4}$$

$$TTot(SF) = \sum_{i=0}^{|SF|-1} \sum_{j=0}^{|F|-1} \sum_{k=0}^{CCF-1} [CHit(t(F_i)_j, FR(t(F_i)_j, k))] * O(t(F_i)_j, FR(t(F_i)_j, k)) \tag{5}$$

We define $M(MF)$ as the number of tasks that execute in a given major frame. Two cases must be considered:

- A task may execute k steps ahead of a task, but in the same major frame, as shown in the first case of Eq. (4).
- Incrementing by k transitions may exceed the boundary of the major frame, whereby the task is determined by wrapping back

to the beginning of the major frame and incrementing any remaining transitions as shown in the second case of Eq. (4). This equation only handles transitions up to one MF ahead, which is reasonable given the case study in Section 2, but can easily be extended.

Eq. (5) accounts for all cache hits due to all transitions in the major frame. The first summation in Eq. (5) accounts for all frames in the major frame. The second summation examines all frame transitions in the current frame. The innermost summation in Eq. (5) sums the expected cache hits *CHit* for tasks that share the same application, as given by *O*.

4.2.8. Total cache hits of a partition

Each partition consists of one or more executing applications. To determine the total expected cache hits for a given partition p, the total expected cache hits of each application a for each application $a \in A$ executing on partition p must be summed, as shown in Eq. (6).

$$\theta(p) = \sum_{k=0}^{|A|-1} \beta(a_k) \tag{6}$$

All tasks for a given partition, however, will execute in the same major frame *SF*. The total number of caches hits due to all transitions in a major frame will therefore yield the total cache hits for the set of applications in a partition, as shown in Eq. (7).

$$\theta(p) = \sum_{k=0}^{|A|-1} \beta(a_k) = TTot(SF) \tag{7}$$

4.2.9. Execution time reduction on heterogeneous nodes

The overhead execution time reduction resulting from a successful cache hit may differ from node to node. We define this reduction as the Cache Constant or $Cc(n)$. This value must be supplied by the DRE system designer or determined through profiling. $CC(n)$ is occasionally referred to as 'cache miss impact,' and tools such as Intel VTune can be used to generate a precise value by monitoring differences in execution time and the event *MEM_LOAD_RETIRED.L2_LINE_MISS*. After calculating the total number of cache hits on the node as described in Eq. (7) we multiply this value by the $Cc(n)$ to determine the total average overhead reduction (ms) for the node, as shown in Eq. (8).

4.2.10. Total cache hits of a node

Each node consists of one or more executing partitions. To calculate the cache benefits $Cm(n)$ of a single node n, we must first determine the sum of the cache hits p for each partition $p \in P$ executing on node n, as shown in Eq. (8).

$$Cm(n) = Cc(n) * \sum_{j=0}^{|P|-1} \theta(p_j) \tag{8}$$

This sum reflects an upper bound on the total probabilistic number of cache hits of the partitions executing on the node.

4.2.11. Total execution time reduction of a system

Finally, the physical structure of the system consists of multiple, separate nodes. To quantify the total cache benefits (the total reduction of system overhead due to successful cache hits) of the system, the cache benefits of each node must be calculated and summed. This process is described in Eq. (9), which defines the SMACK score *SMACK* of a total set of nodes *N*.

$$SMACK(N) = \sum_{i=0}^{|N|-1} Cm(n_i) \tag{9}$$

The SMACK score is an estimation of the improvement of the entire system. This includes the improvement of each node contained within the system. While each node originally used a schedule derived via rate monotonic scheduling, we transform these original schedules via the rules laid out in this publication. The SMACK metric is an estimation of the total system time reduced by this transformation process.

4.3. Applying SMACK to increase system performance

We now describe how SMACK can be combined with metascheduling to increase cache hit-rate while resolving the challenges described in Section 3. Scheduling techniques, such as rate monotonic scheduling, can be used to generate new task execution schedules that ensure real-time constraints are satisfied. Metascheduling, or reordering same-rate tasks, can then be used to generate new task execution schedules that satisfy real-time requirements. It is not always clear, however, which of these schedules will most efficiently utilize the processor cache.

SMACK can be used to determine the "score" of task execution schedules. For instance, if the SMACK score for schedule A is higher than the score for schedule B, we can conclude that schedule A has potential to utilize the cache more efficiently. The SMACK score of one task re-ordering may be greater than that of others indicating more efficient cache utlization. Maximizing the SMACK score can therefore be used as a heuristic to drive metascheduling to maximize temporal locality and increase cache utilization efficiency.

As discussed in Section 2, tasks of different applications do not share data. The cache contention factor determines how many task executions of other applications can occur after a task executes before the cache is potentially completely invalidated. Arranging tasks of the same application to execute consecutively will therefore increase the SMACK score and therefore may decrease execution time, despite having limited or no knowledge of the data coupling between tasks, as stated in Section 3.2. The more that is known about the data coupling characteristics of the tasks between common applications, the more accurate the SMACK score will be.

Finally, reordering the tasks to attempt to increase the SMACK score of the system cannot be done in a haphazard fashion. Any execution order must adhere to the real-time scheduling constraints defined in Section 3.3. This constraint greatly restricts the total potential execution orders that satisfy all system deadlines. Scheduling techniques, such as rate monotonic scheduling and metascheduling, must be applied to create task execution schedules that meet real-time requirements.

5. 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, L2 cache misses, and runtime reductions for each system. All data shown in this section is the average value resulting from 30+ trial runs.

To examine the relationship between SMACK score and DRE system performance, we created multiple software systems to mimic the scale, execution schedule and data sharing of the system described in Section 2. 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

Event Profiled	Semantic Meaning
MEM_LOAD_RETIRED.L1D_MISS	An attempted data retrieval from L1 cache that results in a L1 cache miss
MEM_LOAD_RETIRED.L2_MISS	An attempted data retrieval from L2 cache that results in a L2 cache miss

Fig. 11. Processor signal events profiled with VTune.

priority-based schedule for the generated tasks that adheres to rate-monotonic scheduling requirements.

5.1. Overview of the hardware and software testbed

The systems were compiled and executed on a Dell Latitude D820 with a 2.16 GHz Intel Core 2 processor with 2×32 kB L1 instruction caches, 2×32 kB write-back data caches, a 4 MB L2 cache and 4 GB of RAM running Windows Vista. For each experiment, each system was executed 50 times to obtain an average runtime. These executions were profiled using the Intel VTune Amplifier XE 2011. VTune is a profiling tool that is capable of calculating Intel processor metrics by monitoring processor signal events. For example, to determine the number of L2 cache misses of System A, we compiled and then executed it with VTune configured to return the total times that the event MEM_LOAD_REQUIRED.L2_MISS is triggered. Fig. 11 shows the processor events that were profiled in the following experiments, as well as their semantic meanings.

5.2. Method for creating simulated DRE systems

To test the SMACK-based schedule modification technique, we created a software suite for generating the C++ code of mock integrated avionics systems that behave as specified in Section 2. As shown in Fig. 12 these systems include a priority-based scheduler and multiple sample avionics applications consisting of a variable number of periodic avionic tasks.

Together these applications comprise a representative avionics system. The data sharing and memory usage of these applications, as well as the scheduling technique, are all parameterized and varied to generate a range of test systems. We use these simulated systems to validate SMACK by showing that a lower SMACK score correlates with better performance in terms of execution time and cache misses.

5.2.1. Data sharing characteristics

The data shared between applications and shared between tasks of the same 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 applications. All systems profiled in this section are thus also restricted to sharing data between tasks of the same application. Applications that use a great deal of message data in common, however, are likely to share memory. To account for this sharing, our future work will extend the SMACK calculation to account for these architectures.

5.2.2. Task execution schedule

The execution schedule of the software tasks of the system can potentially affect system performance. For example, assume there are two applications named App1 and App2 that do not share data. Each application contains 1000 task methods, with tasks of the same application sharing a large amount of data. The execution of

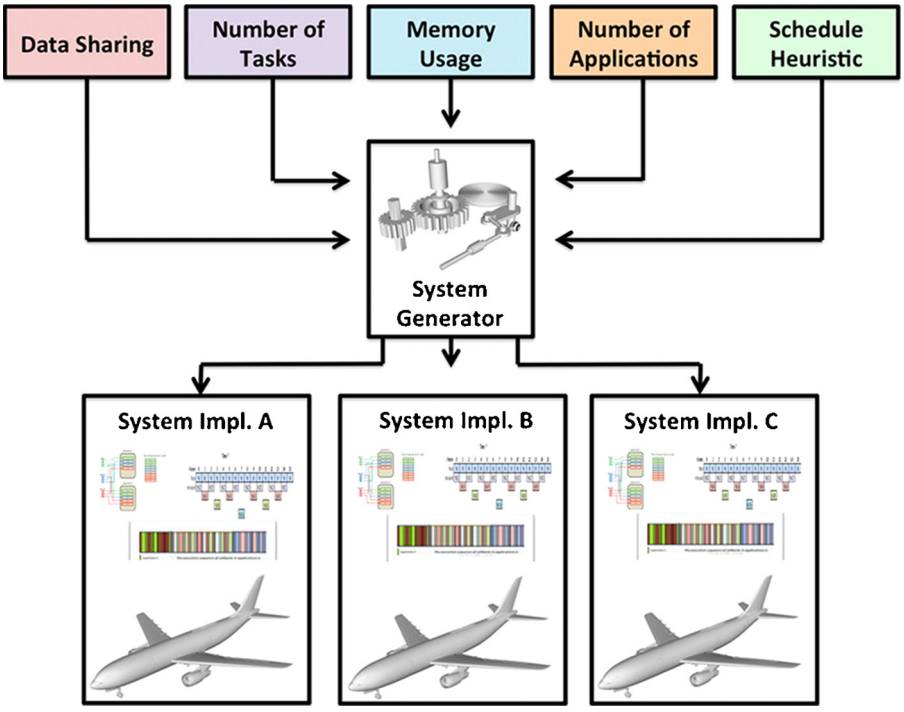

Fig. 12. System creation process.

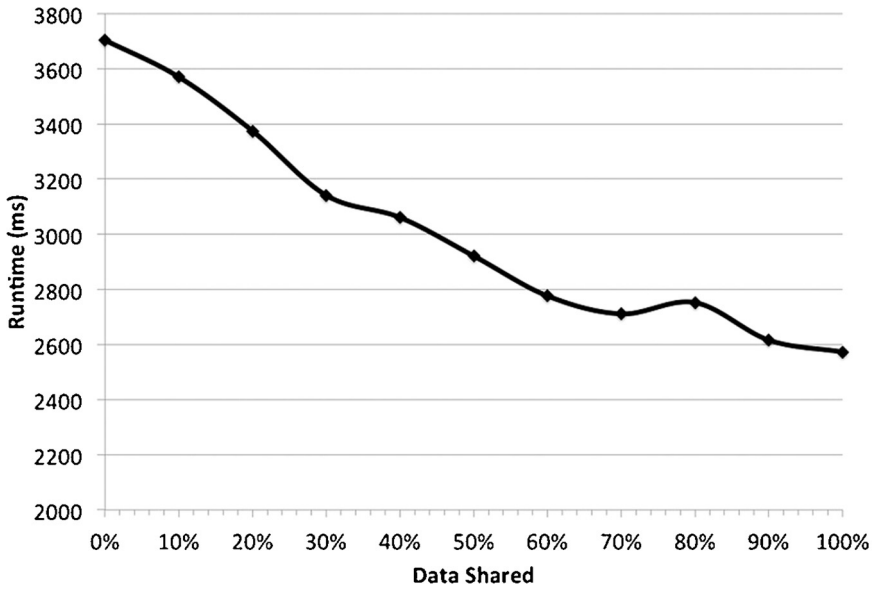

Fig. 13. Amount of data shared vs runtime.

a single task stores enough memory to completely overwrite any data in the cache, resulting in a Cache Contention Factor 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. Multiple execution schedules therefore affect performance differently and produce different SMACK scores.

5.3. Experiment 1: variable data sharing

Experiment design: The amount of data shared between multiple tasks can potentially have a large impact on the performance of a system in terms of cache misses and system runtime, as discussed in Section 5.2.1. Experiment 1 constructed 10 software systems to examine the effect of data sharing between tasks of common applications. Each system contained 5 separate applications consisting of 10 tasks each. The body of the tasks consisted of floating and integer addition operations.

Although the total number of operations of the tasks was constant across all applications, the amount of data shared between the same tasks was varied. For example, if the data sharing between tasks was set to 20%, then each tasks shared approximately 20% of the variables used in operations with all other tasks. After generating these 10 software systems, we executed each system 50 times and determined an average runtime of each system.

Analysis of results: Fig. 13 shows that as the amount of data shared between tasks of a single application increases, system execution time decreases. In this case, sharing 100% of data resulted in an execution time of 2572.58 ms, where as a sharing of no data between tasks, or 0%, resulted in an execution time of 3704.85 ms, which 30.56% slower. The curve shown in Fig. 13 is non-linear, however, with only an additional reduction of 9.40% occurring as a result of increasing the shared data amount from 50% to 100%.

Increasing the amount of shared data between tasks also leads to a decrease in cache misses. We used the VTune Amplifier XE 2011 to determine the total number of L2 and L1 cache misses by monitoring for MEM_LOAD_RETIRED.L2_MISS and MEM_LOAD_RETIRED.L1D_MISS events. These events only take into account cache misses due to data write-back, however, and do not include cache misses resulting from instruction fetching.

Fig. 14 shows the number of L2 cache misses as data sharing between tasks increases. As the data sharing increases the number of L2 cache misses decrease at an exponential rate, from 5.172×10^8 to 1.6×10^5, which is a reduction of 99.69%. As with runtime, the vast majority of L2 cache miss reductions occurred by increasing the amount of shared data from 0% to 50% or greater, resulting in an 80.36% L2 cache miss reduction. Fig. 15 shows the number of L1 cache misses decrease as data between tasks increases. In contrast to runtime and L2 cache misses, the decrease in L1 cache misses is considerably more linear.

5.4. Experiment 2: execution schedule manipulation

Experiment design: The execution schedule of tasks can potentially impact both the runtime and number of cache misses of a system, as discussed in Section 5.2.2. Experiment 2 manipulated the execution order of a single software system with 20% shared data probability between 5 applications consisting of 10 tasks each to create 4 new execution schedules. First, stride scheduling was used to create an execution ordering that met all scheduling constraints (Waldspurger and Weihl, 1995). 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, referred to as "overlaps."

The number of overlaps that exist in an execution schedule is affected by the number of task executions that must occur before the amount of data written to the cache exceeds the size of the cache, defined by the Cache Contention Factor. For example, if each task writes 30k to memory and the cache size is 50k, then most data written to the cache by the first task executing would persist through the execution of two more tasks. The Cache Contention Factor for this system would therefore be two.

The original execution schedule generated by Stride Scheduling is referred to as "Unoptimized." The experimental platform models real-world DRE systems such as the one described in Section 2, and resulted in a Cache Contention Factor of 15, thereby yielding 655 overlaps for the Unoptimized schedule. This schedule was then permuted to increase the number of overlaps while satisfying priority scheduling constraints. This schedule is referred to as the "Optimized" ordering and contained 801 overlaps.

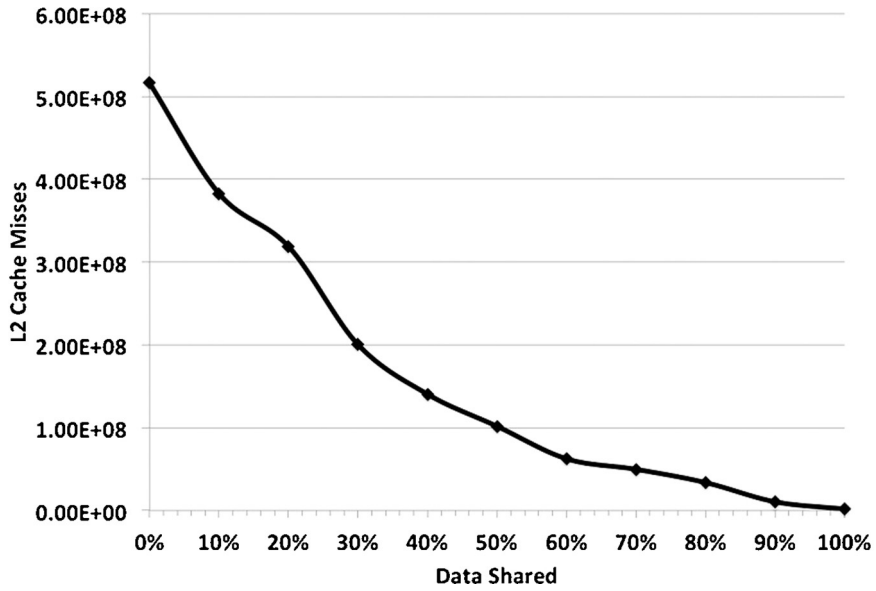

Fig. 14. Amount of data shared vs L2 cache misses.

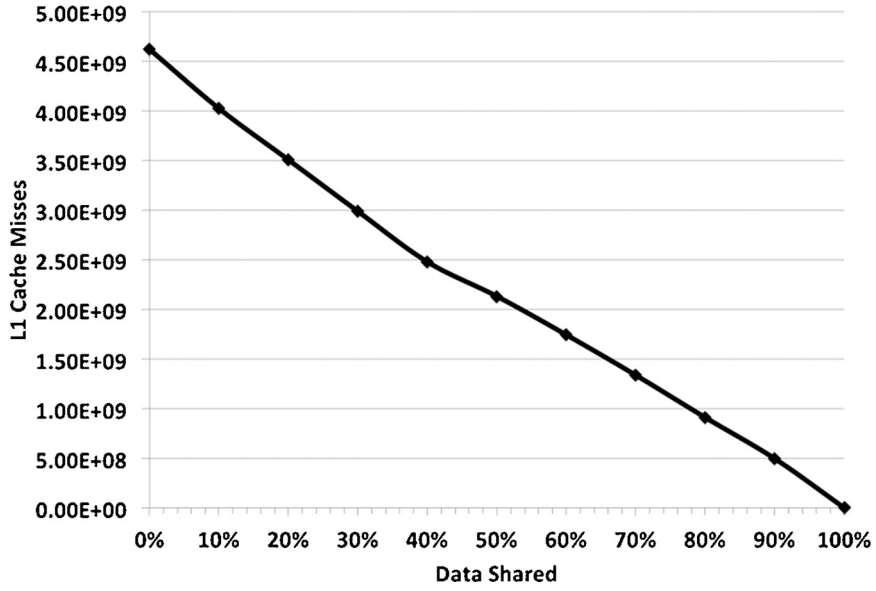

Fig. 15. Amount of data shared vs L1 cache misses.

We also created two execution schedules that do not satisfy the priority scheduling requirement to maximize and minimize the number of overlaps. To minimize the number of overlaps, we permuted the execution order such that no two tasks of the same application executed consecutively, resulting in the "Worst" case execution order with 732 overlaps.[2]

Fig. 16 shows that as cache size increases, the Worst execution order may result in more overlaps than other execution orders. Finally, we maximized the number of overlaps by executing all tasks of each application consecutively, resulting in 1,743 overlaps. We refer to this execution ordering as the "Best" execution schedule. The Best schedule represents an upper bound on the performance improvement that can be achieved using task reordering techniques.

Analysis of results. Fig. 17 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 Best execution schedule, consisting of 1,743 overlaps, executed in 2790 ms on average. The Optimized execution schedule completed in 3299 ms, which was an 18.24% increase from the Best execution schedule. The Unoptimized and Worst execution schedules executed in 3337 and 3329 ms, respectively.

Execution order was also shown to impact the number of cache misses. Fig. 18 shows the L1 cache misses for all execution schedules. Once again, the execution schedule with the most overlaps—the Best execution schedule—performed the best of all execution orders, resulting in only 3.26×10^9 cache misses. The Optimized execution schedule, consisting of 801 overlaps,

[2] This execution order is the "Worst" since it yields 0 overlaps when the Cache Contention Factor is one.

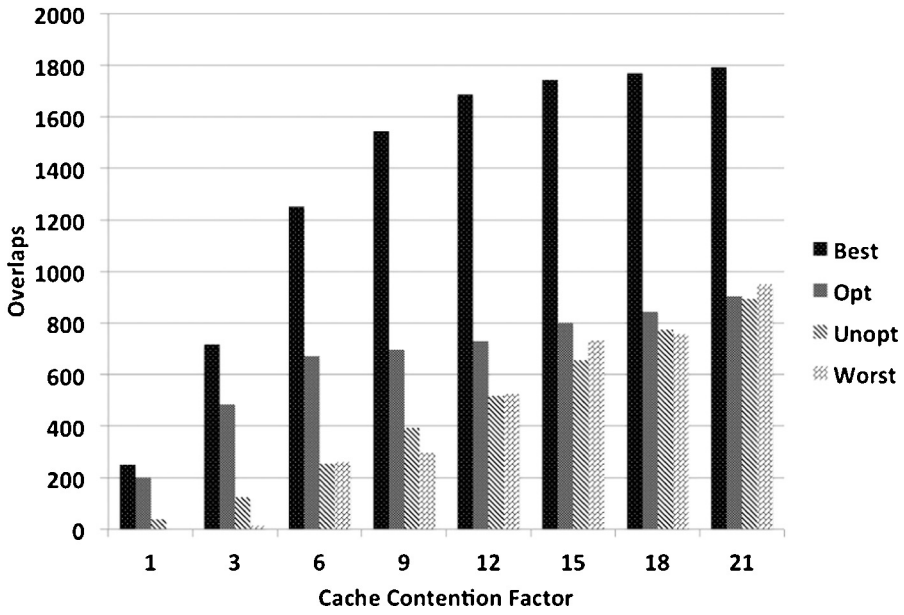

Fig. 16. Cache contention factor vs overlaps.

generated 3.48×10^9 cache misses, an increase of 6.47% from the L1 cache misses of the Best execution order. Next, the Unoptimized execution schedule, consisting of 655 overlaps, resulted in 3.51×10^9 L1 cache misses. Finally, the Worst execution order resulted in 3.53×10^9 L1 cache misses, the most of all execution orders.

The impact of execution order on L2 cache misses can be seen in Fig. 19. Similar to L1 cache misses and runtime, the execution schedule with the most overlaps—the Best execution schedule—produced the lowest results with 1.59×10^8 L2 cache misses. The Worst case execution schedule generated fewer L2 cache misses than the Unoptimized schedule, which in turn generated fewer L2 cache misses than the Optimized schedule.

5.5. Experiment 3: dynamic execution order and data sharing

Experiment design: Sections 5.2.1 and 5.4 demonstrate the effects of the data sharing characteristics of applications and execution order of tasks on runtime and cache misses. These sections, however, do not examine the impact of altering both of these aspects simultaneously. Experiment 3 therefore examined multiple execution orders for multiple systems with different data sharing characteristics. For example, the reduction in system cache misses could be substantially different by altering the execution order of a system with 80% shared data than a system with only 10% shared data.

Analysis of results: Fig. 20 shows that the number of L1 cache misses decreases as the number of overlaps in the execution order

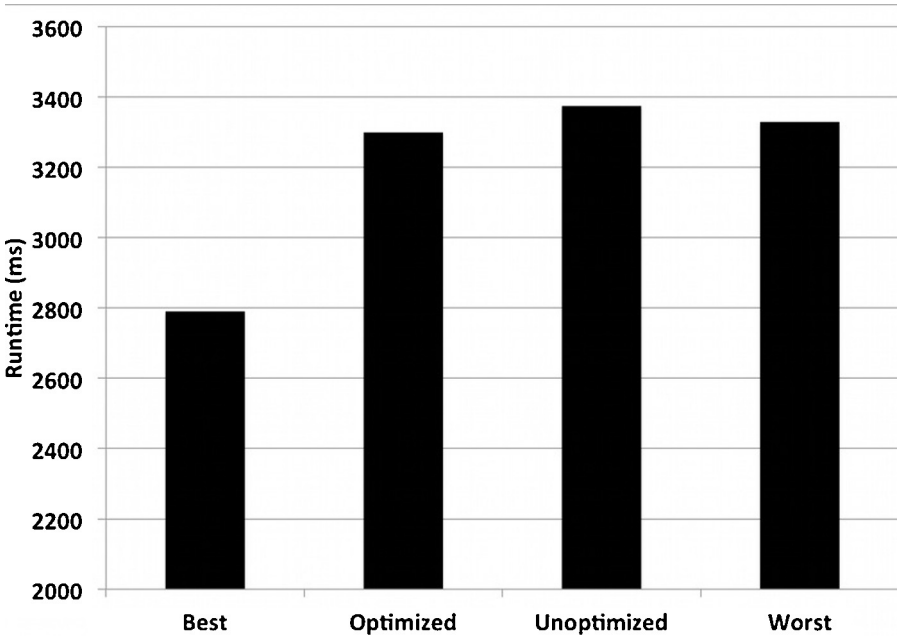

Fig. 17. Runtimes of various execution schedules.

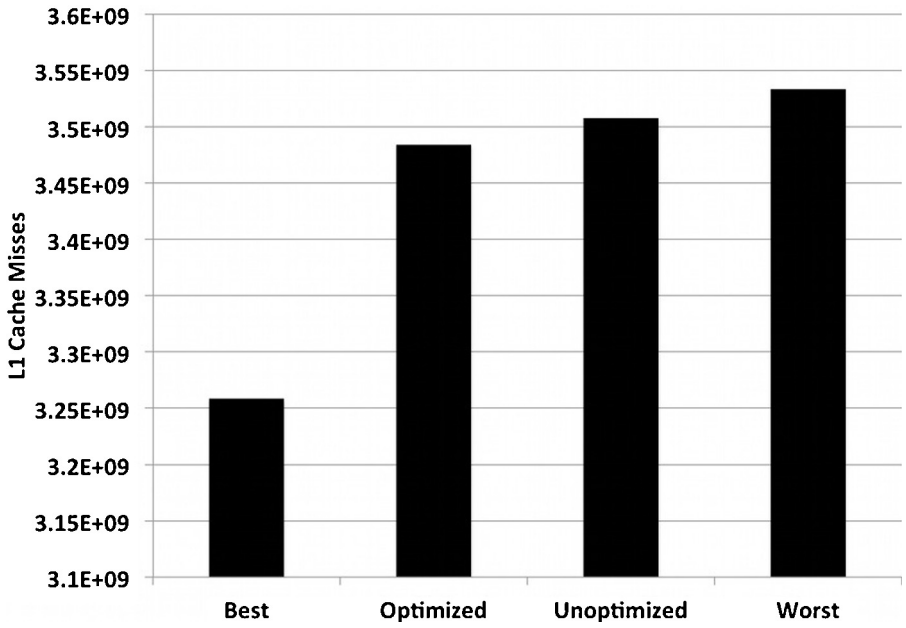

Fig. 18. Execution schedules vs L1 cache misses.

and/or the amount of shared data increases. Again, the Best execution order consisting of the most overlaps resulted in the fewest L1 cache misses for all software systems. Unlike runtime, however, the number of L1 cache misses are only slightly less than those of the other execution orders. Moreover, L1 cache misses for all execution orders decreased at near-linear rate.

Fig. 21 shows that the L2 cache misses decreased at an exponential rate. Once again, the Best execution order resulted in the lowest number of cache misses for almost all trials, with the exception of the system with 90% data sharing in which the number of L2 cache misses were comparably negligible. The exponential nature of the decrease in cache misses show that the greatest reduction in L2 cache misses can be made by increasing the total amount of shared data, if less than 50% of data is shared. Therefore, a reasonable source code design decision may be to use a small (relative to the cache) central state container for frequently accessed system state.

For example, increasing the amount of data shared from 0% to 50% for the Optimized execution order resulted in an L2 cache miss reduction of 77.64%. Increasing data sharing from 50% to 90%, however, does not yield as extreme benefits. Increasing the amount of data shared from 50% to 90% for the Optimized execution order resulted in an additional reduction of only 21.18%.

Fig. 22 shows the system execution time decreases as the amount of shared data increases. The decrease in runtime, however, is not constant across all execution orders. For example, the Best execution order decreases from 2884 ms when 0% of data is shared to 2398 ms when 100% of data is shared, a total decrease of 486 ms or 16.85%. The Optimized execution order decreases from 3592 ms to 2,582 ms as the shared data increase from 0% to 100%, for a total

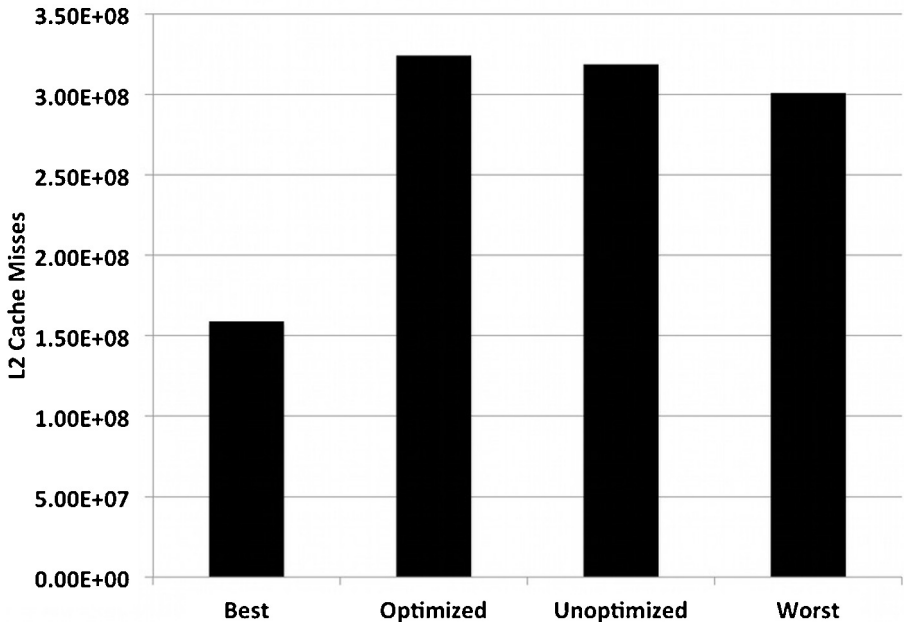

Fig. 19. Execution schedules vs L2 cache misses.

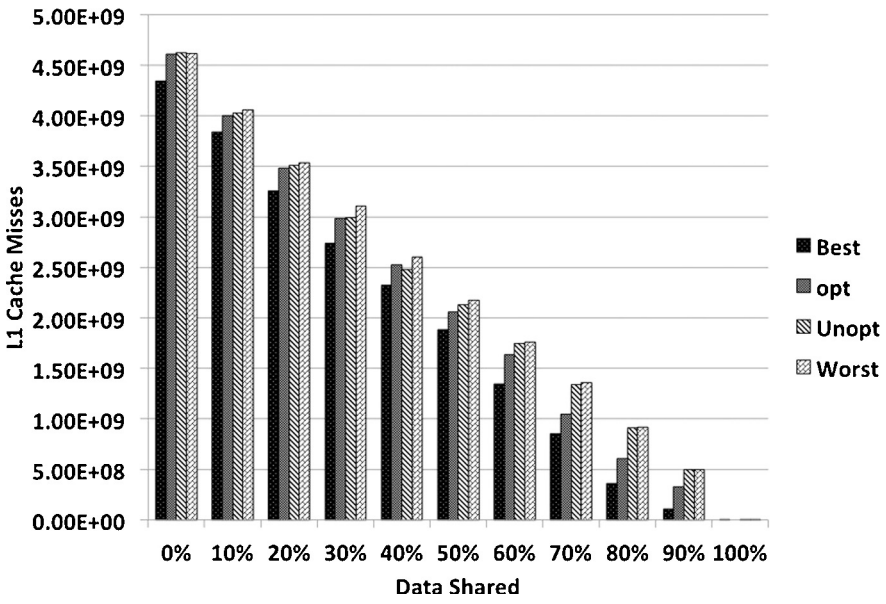

Fig. 20. L1 cache misses vs data shared and execution order.

runtime decrement of 1010 ms or 28.12%. Altering the amount of data shared reduced the system runtime of the Optimized execution order by 107.82% more than the same data alteration with the Best optimized execution order. These results therefore show that altering the amount of data shared has a larger impact on runtime for systems with less efficient execution orders.

While adjusting the data sharing characteristics of a system may be acceptable at design time, safety certification and other factors may prohibit altering the data sharing characteristics of a system. Manipulating the execution order of the software tasks, however, is permitted. Fig. 22 shows the potential benefits of altering system order for systems with different data sharing characteristics, which yields greater reduction in system runtime for systems that share less data between tasks. As data sharing increases, this reduction is not as great. For execution orders that satisfy scheduling constraints, the Optimized execution order consistently resulted in faster runtimes than the Unoptimized execution order. Runtime reductions can therefore be realized by manipulating the task execution schedule without violating priority scheduling constraints.

5.6. Experiment 4: predicting performance with SMACK

Experiment design: The previous experiments demonstrated the impact of the data sharing and execution schedule of several different systems. Experiment 4 examines the correlation between the SMACK score and actual runtime for a system. SMACK uses the execution order and data sharing characteristics in conjunction with a Cache Contention Factor to score systems in terms of expected performance, as described in Section 4. SMACK provides a basis for comparing multiple DRE systems in terms of expected performance. For example, if System A produces a higher SMACK score than System B, then System A has the potential to have a lower runtime than System B.

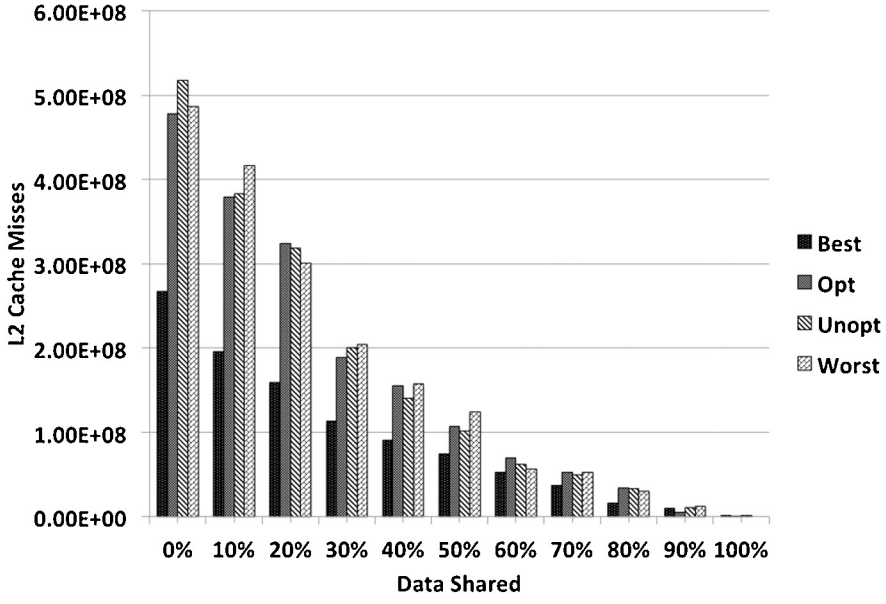

Fig. 21. L2 cache misses vs data shared and execution order.

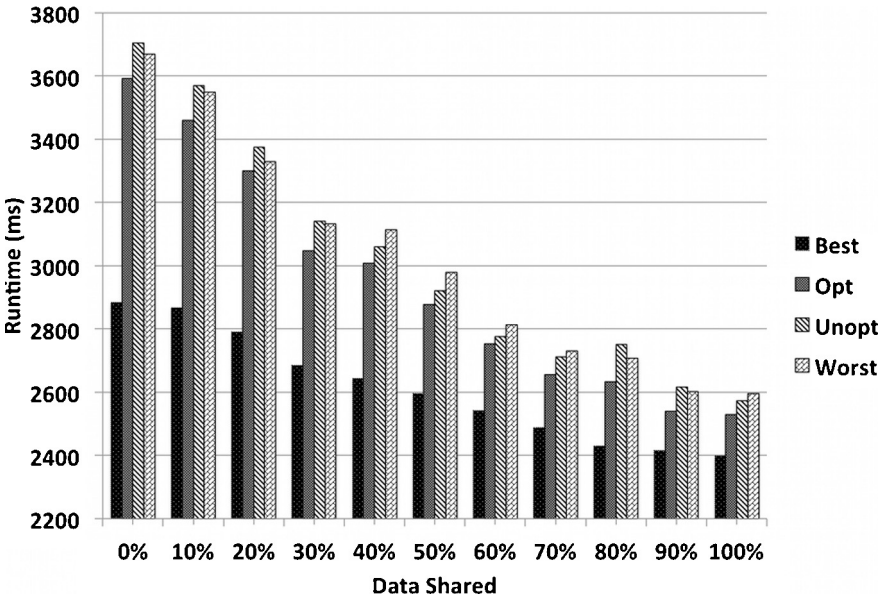

Fig. 22. Runtime vs data shared and execution order.

Section 5.5 presents 44 different systems with data sharing ranging from 0%-100% and 4 different execution schedules for each. Each system was executed on the same hardware, thereby producing the same value for the contention factor. The SMACK score is calculated for each system taking into account the contention factor, the execution schedule, and data sharing characteristics.

Analysis of results: Fig. 23 presents the SMACK scores for each system. As the amount of data sharing increases, the SMACK score increases, indicating a reduction in runtime. Comparing the SMACK scores shown in Fig. 23 to the actual system execution times shown in Fig. 22 shows that the SMACK score correlates with increased performance. Similarly to runtime, optimizing the execution schedule of a system is also increases the SMACK score. SMACK can therefore effectively predict and compare the performance of multiple DRE systems.

Section 5.4 presents 4 different execution schedules used to execute the software systems tested. Of these execution schedules, only the Unoptimized and Optimized execution schedules satisfy priority based scheduling constraints. The Unoptimized schedule was built without taking into account the effect of overlaps on system performance. The Optimized execution order is created by reordering the tasks executions so that overlaps are increased without violating priority scheduling constraints.

Fig. 24 shows the percentage reduction in runtime by changing the unoptimized execution order to increase overlaps and create the Optimized execution order. Altering the execution order resulted in an average runtime reduction of 2.4%, with a maximum reduction of 4.34%. This reduction can be achieved without altering the underlying hardware or software executing on the system. Optimizing system execution schedules to maximize SMACK scores can therefore yield substantial reductions in system execution time without requiring extensive knowledge of the software, access to the code, recertification, or alterations to the hardware.

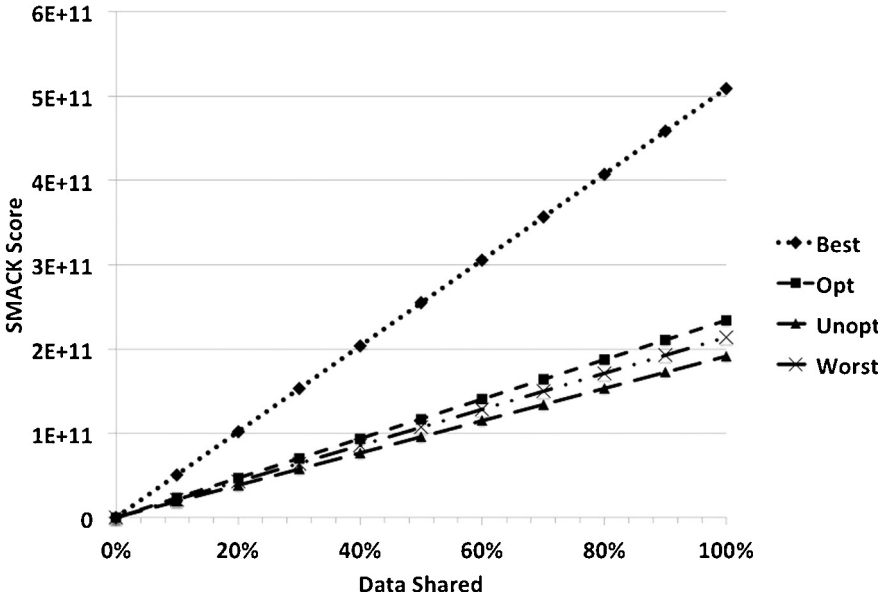

Fig. 23. Smack score vs data shared and execution order.

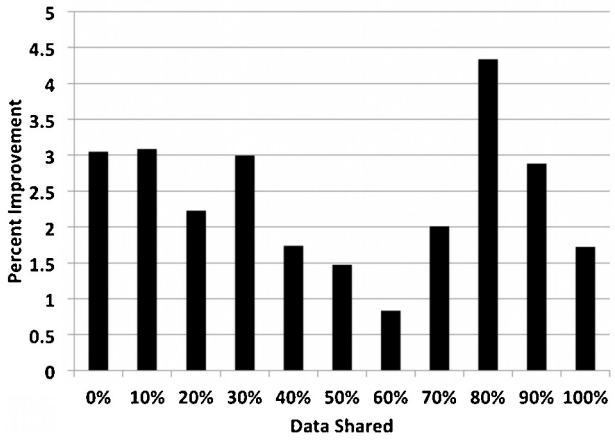

Fig. 24. Percent runtime reduction vs data shared.

The optimized execution order presented above is not the *optimal* execution order that would minimize the SMACK score. Even for DRE systems with the same software, the hardware can have a large impact on the Cache Contention Factor, which is an integral part of the SMACK score calculation. Section 5.4 demonstrates that the Cache Contention Factor of a DRE system changes the effectiveness of an execution schedule. Our future work is investigating an algorithmic technique that takes into account the Cache Contention Factor of a system to maximze the SMACK score and corresponding performance gains.

6. Related work

This section compares the SMACK metric and its use as a heuristic for cache optimization with other techniques for optimizing cache hits and system performance, including (1) software cache optimization techniques, (2) hardware cache optimization techniques, and (3) other DRE system configuration 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 (Kowarschik et al., 2003), focus on changing the manner in which loops are executed. One technique, known as loop interchange (Wolf et al., 1996), can be used to reorder multiple loops to maximize the data access of common elements in respect to time, referred to as *temporal locality* (Allen and Kennedy, 1984; Yi and Kennedy, 2004; Wolf et al., 1996; Shiue and Chakrabarti, 2001). Another technique, known as loop fusion (Singhai and McKinley, 1997), 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 (Singhai and McKinley, 1997; Kennedy and McKinley, 1994; Verdoolaege et al., 2003; Beyls and D'Hollander, 2001). Yet another technique for improving software cache effectiveness is to utilize *prefetch* instructions, which retrieves data from memory into the cache before the data is requested by the application (Kowarschik et al., 2003). Prefetch instructions inserted manually into software at the source code level can significantly reduce memory latency and/or cache miss rate (Chen and Baer, 1992; Fu et al., 1992).

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. SMACK 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 SMACK as a heuristic 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 (Abandah and Abdelkarim, 2009; Guo and Solihin, 2006; Hassidim, 2010).

The cache replacement policy can substantially influence DRE system performance. For example, LRU is effective for DRE 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 (Smith and Goodman, 1985). In these cases, a random replacement policy may yield the most cache effectiveness (Smith and Goodman, 1985).

Moreover, certain policies are shown to work better for different cache levels (Abandah and Abdelkarim, 2009), 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.

SMACK does not alter the cache replacement policy of hardware since altering the hardware could invalidate previous safety certifications, similar to to altering software at the source code level. SMACK is used a heuristic to increase temporal locality by altering the task execution order schedule. While many replacement policies exist, the SMACK calculation assumes an LRU replacement policy. Our future work will examine the impact of cache replacement policy on the performance gains of schedules altered via SMACK.

DRE system configuration optimizations: While heuristic techniques, such as heuristic-based scheduling with SMACK, can be applied to increase the processor cache effects of existing DRE systems, other techniques focus on increasing performance through intelligent system construction. Constructing valid DRE system implementations by configuring prefabricated COTS applications is non-trivial due to several constraints, such as real-time requirements, budgetary limitations, and strict resource constraints. Substantial reductions in execution time, financial cost, and resource requirements can be realized, however, by configuring DRE systems intelligently (Dougherty et al., 2009, 2009).

Other techniques, such as software product lines (Borba and Silva, 2009), examine points of variability in the hardware and software of the system to determine if certain variants offer superior performance (Benavides et al., 2005; White et al., 2007). These techniques are appropriate for constructing new DRE system

implementations or evolving existing system implementations so that all DRE system constraints are met. These techniques, however, do nothing to further optimize system performance after a valid configuration has been determined.

SMACK can be used as a heuristic to optimize system performance benefits due to processor caching. SMACK should not be used, however, to construct or evolve the configurations of existing DRE systems or alter system applications since it does not take into account several constraints that must be honored for a system to be valid. Instead, SMACK should be used to improve the performance of existing *valid* configurations, as only the execution schedule of the system is affected. Fortunately, optimizing execution schedules with SMACK takes into account real-time scheduling requirements and therefore ensures real-time constraints are satisfied.

7. Concluding remarks

Processor data caching can substantially increase DRE system performance. Several factors, such as task execution schedule, data sharing characteristics, and system hardware, can influence the caching effects of a system, which makes it hard to predict cache behavior. Moreover, satisfying real-time constraints makes it hard to create valid task execution schedules that increase cache effects. Without a formal methodology for predicting the processor cache behavior of a system and ensuring schedulability, moreover, it is hard to compare multiple potential system implementations or apply performance optimizations.

This paper presents the *System Metric for Application Cache Knowledge* (SMACK) for quantifying the performance gains of processor caching of a system. System performance of multiple system implementations can be assessed and compared based on SMACK score. The system with the highest SMACK score will better utilize the processor cache than other system implementations, resulting in decreased system execution time. Moreover, certain aspects of the systems (such as the task execution schedule) could potentially be altered to optimize the SMACK score and decrease system execution time while ensuring real-time deadlines are met. We empirically evaluated applying the SMACK metric to 44 different simulated industry avionics systems.

As a result of these efforts, we learned the following lessons from predicting the impact of processor caching on system performance:

- **Both hardware and software design decisions affect the SMACK score of a system**: The processor cache size, data sharing characteristics and task execution have a large impact on the SMACK score. The SMACK score tends to improve with increases in cache size and data sharing. The task execution schedule of the system effects the SMACK score more dramatically when data sharing between tasks is low.
- **Decreases in the SMACK score correlate with increased system performance and decreased system execution time**: Increasing the data sharing and/or altering the execution order of a system yields a decreased SMACK score. Reducing the SMACK score correlated with an average runtime reduction of 2.4% without requiring any additional software, hardware, or middleware. Multiple system implementations can thus be compared based on their SMACK scores. Future work may include extending SMACK to evaluate aperiodic and/or mixed-critical real-time schedules.
- **Effects of other cache replacement policies should be investigated**: The SMACK metric does not take into account the cache replacement policy of a system and was only tested with random replacement policy. The effectiveness of SMACK should be investigated for other cache policies, such as Least Recently Used (LRU) and First In First Out (FIFO).
- **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 determining schedules that yield effective reductions in computation time.
- **Real-world DRE architectures are required for continued progress**: A predominant challenge in this work is the use of randomly-generated DRE architectures. While minor efforts have been made to publish industry challenge problems, there is a distinct lack of easily accessible real-world challenges. There are likely many practical challenges for using a SMACK-esq metric in a real system, with edge cases such as specialized data structures, unique programming paradigms, or large dependencies. Future work on large-scale DRE system improvements would be greatly advanced by a set of industrial-scale challenge problems.
- **Algorithmic techniques to maximize SMACK should be developed**: 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 Contention Factor. Our future work will examine algorithmic techniques that use SMACK and the Cache Contention Factor as a heuristic for determining the optimal execution order for tasks in specific systems.

Acknowledgements

We thank Steve Drager and Bill McKeever of the Air Force Research Labs for providing us with the opportunity to execute this research through contract FA8750-10-C-0041.

References

Abandah, G., Abdelkarim, A., 2009. A Study on Cache Replacement Policies.
Allen, J., Kennedy, K.,1984. Automatic loop interchange. In: Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction. ACM, p. 246.
Bahar, R., Albera, G., Manne, S.,2005. Power and performance tradeoffs using various caching strategies. In: Proceedings: 1998 International Symposium on Low Power Electronics and Design. IEEE, pp. 64–69.
Benavides, D., Trinidad, P., Ruiz-Cortes, A., 2005. Automated Reasoning on Feature Models. 17th Conference on Advanced Information Systems Engineering. LNCS 3520, 491–503.
Beyls, K., D'Hollander, E.,2001. Reuse distance as a metric for cache behavior. In: Proceedings of the IASTED Conference on Parallel and Distributed Computing and Systems, vol. 14. Citeseer, pp. 350–360.
Borba, C., Silva, C., 2009. A comparison of goal-oriented approaches to model software product lines variability. In: Advances in Conceptual Modeling-Challenging Perspectives, pp. 244–253.
Chen, T., Baer, J., 1992. Reducing memory latency via non-blocking and prefetching caches. ACM SIGPLAN Notices 27 (9), 51–61.
Committee, A.E.E.et al., 1997. Avionics Application Software Standard Interface. Aeronautical Radio.
Dhall, S., Liu, C., 1978. On a real-time scheduling problem. Oper. Res. 26 (1), 127–140.
Dougherty, B., White, J., Kegley, R., Preston, J., Schmidt, D., Gokhale, A., 2011. Optimizing integrated application performance with cache-aware metascheduling. On the Move to Meaningful Internet Systems: OTM 2011. LNCS 7045, 432–450, Springer.
Dougherty, B., White, J., Thompson, C., Schmidt, D.C., 2009. Automating hardware and software evolution analysis. In: 16th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS), San Francisco, CA, April.
Fu, J., Patel, J., Janssens, B.,1992. Stride directed prefetching in scalar processors. In: Proceedings of the 25th Annual International Symposium on Microarchitecture. IEEE Computer Society Press, pp. 102–110.
Ghosh, S., Melhem, R., Mossé, D., Sarma, J., 1998. Fault-tolerant rate-monotonic scheduling. Real-Time Syst. 15 (2), 149–181.
Guo, F., Solihin, Y.,2006. An analytical model for cache replacement policy performance. In: Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems. ACM, pp. 228–239.
Hassidim, A.,2010. Cache replacement policies for multicore processors. In: Proceedings of The First Symposium on Innovations in Computer Science. Tsinghua University Press.
Kennedy, K., McKinley, K., 1994. Maximizing loop parallelism and improving data locality via loop fusion and distribution. Lang. Compil. Parall. Comput., 301–320.
Kowarschik, M., Weiß, C., 2003. An overview of cache optimization techniques and cache-aware numerical algorithms. Algorithms Mem. Hierarchies, 213–232.

Manjikian, N., Abdelrahman, T.,1995. Array data layout for the reduction of cache conflicts. In: Proceedings of the 8th International Conference on Parallel and Distributed Computing Systems. Citeseer, pp. 1–8.
Naghibzadeh, M.,2002. A modified version of rate-monotonic scheduling algorithm and its' efficiency assessment. In: Proceedings of the Seventh International Workshop on Object-Oriented Real-Time Dependable Systems, 2002 (WORDS 2002). IEEE, pp. 289–294.
Nayfeh, B., Olukotun, K.,1994. Exploring the design space for a shared-cache multiprocessor. In: Proceedings of the 21st Annual International Symposium on Computer Architecture. IEEE Computer Society Press, p. 175.
Panda, P., Nakamura, H., Dutt, N., Nicolau, A., 2002. Augmenting loop tiling with data alignment for improved cache performance. IEEE Trans. Comp. 48 (2), 142–149.
Pingali, S., Kurose, J., Towsley, D., 2007. On Comparing the Number of Preemptions under Earliest Deadline and Rate Monotonic Scheduling Algorithms.
Reineke, J., Grund, D., Berg, C., Wilhelm, R., 2007. Timing predictability of cache replacement policies. Real-Time Syst. 37 (2), 99–122.
Robinson, J., Devarakonda, M., 1990. Data cache management using frequency-based replacement. ACM SIGMETRICS Perform. Eval. Rev. 18 (1), 134–142.
Shiue, W., Chakrabarti, C., 2001. Memory design and exploration for low power embedded systems. J. VLSI Signal Process. 29 (3), 167–178.
Singhai, S., McKinley, K., 1997. A parametrized loop fusion algorithm for improving parallelism and cache locality. Comp. J. 40 (6), 340.
Smith, J., Goodman, J., 1985. Instruction cache replacement policies and organizations. IEEE Trans. Comp., 234–241.
Sprangle, E., Carmean, D.,2002. Increasing processor performance by implementing deeper pipelines. In: Proceedings: 29th Annual International Symposium on Computer Architecture, 2002. IEEE, pp. 25–34.
Stewart, D., Barr, M., 2002. Rate monotonic scheduling. Embed. Syst. Program., 79–80.
Verdoolaege, S., Bruynooghe, M., Janssens, G., Catthoor, P.,2003. Multi-dimensional incremental loop fusion for data locality. In: Proceedings: IEEE International Conference on Application-Specific Systems, Architectures, and Processors, 2003. IEEE, pp. 17–27.
Waldspurger, C., Weihl, W., 1995. Stride scheduling: Deterministic proportional-share resource management. Massachusetts Institute of Technology, Cambridge, MA, Tech. Rep. MIT/LCS/TM-528.
White, J., Czarnecki, K., Schmidt, D.C., Lenz, G., Wienands, C., Wuchner, E., Fiege, L., 2007. Automated model-based configuration of enterprise Java applications. In: EDOC 2007, October.
Wolf, M.E., Maydan, D.E., Chen, D.-K.,1996. Combining loop transformations considering caches and scheduling. In: Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture. IEEE Computer Society, pp. 274–286.
Yi, Q., Kennedy, K., 2004. Improving memory hierarchy performance through combined loop interchange and multi-level fusion. Int. J. High Perform. Comp. Appl. 18 (2), 237.

Hamilton Turner is a PhD student at Virginia Tech University in Blacksburg, Virginia. His research interests are mobile and ubiquitous computing, cloud computing, and security. He is the leader of the ATAACK project, which enables large-scale simulation of mobile phone software on a distributed cloud ecosystem. He plans to obtain his PhD by Fall 2014 and continue towards becoming a faculty member.

Brian Dougherty holds a PhD in computer science from Vanderbilt University. Brian's research focuses on hardware/software co-design, heuristic constraint-based deployment algorithms, and design space exploration. He is the co-leader of the ASCENT project, a tool for analyzing hardware/software co- design solution spaces. Brian is also a developer for the Generic Eclipse Modeling System (GEMS). He received his BS in computer science from Centre College, Danville, KY in 2007.

Jules White is an assistant professor of computer science at Vanderbilt University. He received his BA in computer science from Brown University, his MS and PhD from Vanderbilt University. His research focuses on cyber-physical systems as well as deployment and configuration optimization, mobile computing, model-driven engineering, distributed systems and cloud computing, and autonomic computing. In conjunction with Siemens AG, Lockheed Martin, IBM and others, he has developed scalable constraint and heuristic techniques for software deployment and configuration. He is the Project-Lead of the Eclipse Foundation's Generic Eclipse Modelling System (GEMS; http://www.eclipse.org/gmt/gems).

Russell B. Kegley holds the position of Lockheed Martin Fellow in Lockheed Martin Aeronautics Company. He has worked in operating systems development and real-time computing since 1978, beginning with kernel development for mainframe computers and continuing into more diverse platforms as they have evolved. Since joining Lockheed Martin in 1986, he has worked on a long series of avionics architecture thrusts, including leading the first application development on the Carnegie-Mellon real time distributed Alpha operating system. His other experiences at Lockheed Martin include development of schedulability analysis frameworks for the F-22 and F-35 fighter programs, internal consulting in advanced design techniques, and leading multiple IRAD efforts investigating schedulability tools, middleware architectures, and cache contention reduction. He provides leadership and technical support to multiple Lockheed Martin-funded university and government agency research efforts.

Jonathan D. Preston is a technology program manager within the Advanced Development Programs branch of Lockheed Martin Aeronautics Company. Throughout his career, he has managed several government-funded technology programs that have transferred technologies to major weapon system programs. Preston earned a bachelor's degree in electrical engineering from the Pennsylvania State University and a master's degree in computer science engineering from the University of Texas at Arlington.

Douglas C. Schmidt is a full professor in the Electrical Engineering and Computer Science Department and associate chair of the Computer Science and Engineering program at Vanderbilt University, Nashville, TN. During the past two decades he has led pioneering research on patterns, optimization techniques and empirical analyses of object-oriented and component-based frameworks and model-driven development tools that facilitate the development of middleware and applications for distributed real-time and embedded (DRE) systems. He is an expert on DRE computing patterns and middleware frameworks and has published over 450 technical papers and nine books that cover a range of topics including high-performance communication software systems, parallel processing for high-speed networking protocols, quality-of-service (QoS)-enabled distributed object computing, object-oriented patterns for concurrent and distributed systems model-driven development tools. He received his PhD in Computer Science from the University of California, Irvine in 1994 (URL: www.dre.vanderbilt.edu/ schmidt).

Aniruddha Gokhale is an associate professor of computer science and engineering in the Department of Electrical Engineering and Computer Science at Vanderbilt University. His research expertise includes model-driven engineering and QoS-enablers for middleware solutions for distributed real-time and embedded systems.