AFRL-RI-RS-TR-2011-266

PREDICTIVE CACHE MODELING AND ANALYSIS

LOCKHEED MARTIN AERONAUTICS CORP.

NOVEMBER 2011

FINAL TECHNICAL REPORT

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■**UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

REPORT DOCUMENTATION PAGE	*Form Approved* **OMB No. 0704-0188**

1. REPORT DATE *(DD-MM-YYYY)* NOV 2011	**2. REPORT TYPE** Final Technical Report	**3. DATES COVERED** *(From - To)* APR 2010 – SEP 2011

4. TITLE AND SUBTITLE PREDICTIVE CACHE MODELING AND ANALYSIS	**5a. CONTRACT NUMBER** FA8750-10-C-0041
	5b. GRANT NUMBER N/A
	5c. PROGRAM ELEMENT NUMBER 63788F
6. AUTHOR(S) Russell Kegley, Jonathan Preston, (Lockheed Martin Aeronautics) Brian Dougherty, Jules White (Virginia Polytechnic Institute) Anirudda Gokhale (Vanderbilt University)	**5d. PROJECT NUMBER** T35D
	5e. TASK NUMBER LM
	5f. WORK UNIT NUMBER CP
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Lockheed Martin Aeronautics Corporation 1 Lockheed Boulevard Fort Worth, TX 76108	**8. PERFORMING ORGANIZATION REPORT NUMBER** N/A
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory/RITA 525 Brooks Road Rome NY 13441-4505	**10. SPONSOR/MONITOR'S ACRONYM(S)** AFRL/RI
	11. SPONSORING/MONITORING AGENCY REPORT NUMBER AFRL-RI-RS-TR-2011-266

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. PA# 88 ABW-2011-6040
Date Cleared: 14 NOVEMBER 2011

13. SUPPLEMENTARY NOTES

14. ABSTRACT
This work applied particle swarm heuristic optimization techniques to the problem of finding a near-optimal order in which to schedule tasks in a real-time embedded system in order to minimize cache miss rates experienced by the software. Reducing the number of cache misses is an important component of runtime execution efficiency. We demonstrated runtime reductions of 3-5% in execution time, significant for embedded systems attempting to add new capability without upgrading hardware. The expectation is that these gains can be improved further by the use of hardware with pseudo-LRU cache behavior.

15. SUBJECT TERMS
Heuristic optimization; cache memory; rate monotonic analysis; performance tuning

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** WILLIAM E. MCKEEVER
a. REPORT U	**b. ABSTRACT** U	**c. THIS PAGE** U	UU	52	**19b. TELEPHONE NUMBER (*Include area code*)** N/A

Table of Contents

Table **Page**

List of Figures

List of Tables

1. Summary

The overall objective of this effort was to develop methods for achieving runtime efficiency gains in the design of integrated avionic systems and demonstrate those gains in a context that is representative of a major Department of Defense (DoD) weapon system.

Current-generation avionics designs generally employ networks of commercially-sourced single-core multiprocessors running multiple complex, multi-threaded applications. Such systems must be temporally predictable and guard against overrunning available computing and network capacities. Factors such as operating system threading models, stochastic interrupt arrival patterns, and effects of changing modes and system states make it extremely challenging to predict and control performance penalties associated with runtime cache behavior, forcing reliance on trial-and-error based methods. Figure 1 illustrates the importance of this problem. In both system utilization diagrams, the embedded applications software consumes the same amount of CPU bandwidth, 55%. However, the system shown on the left has only 8% reserve capacity, severely limiting the amount of capability growth that can be hosted on that processor without going to extraordinary, and expensive, measures to re-develop existing code. By reducing the amount of uncertainty in worst-case execution times, we can increase the usable reserve capacity to 24%, potentially eliminating or delaying the need to replace this piece of embedded hardware to expand capabilities of the weapon system.

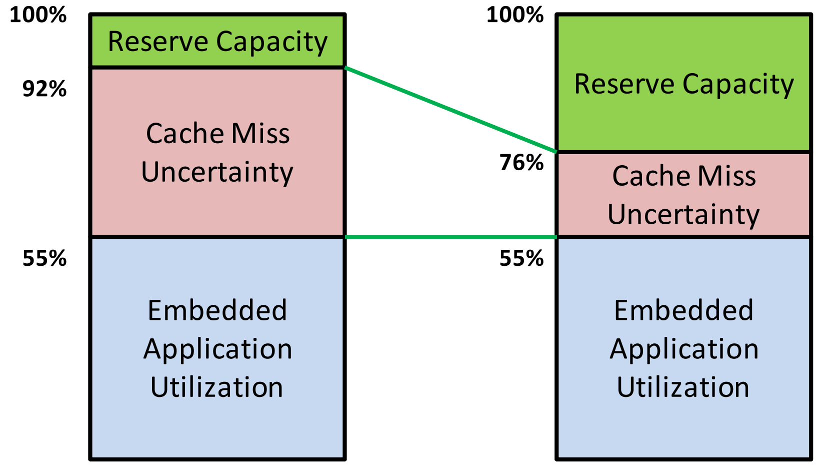


Figure 1. Effect of Cache Miss Uncertainty on Processor Reserve

The primary technical objective of this effort was to develop promising temporal analytic techniques in the context of realistic DoD avionics designs along with supporting analysis tools. Specific goals include: 1) increasing predictive precision associated with run-time phenomena such as cache miss penalties, modal effects, and aperiodic behaviors, 2) extending the single-core cache miss prediction to multicore processor architectures, and 3) providing an analytic basis for allocating application programs to processing cores that increases hardware utilization efficiency. Accurately predicting system performance is critical to making important system design and computing resource allocation decisions. Utilizing a processor cache can greatly benefit system performance and reduce execution time. Several factors that vary between system implementations, such as cache size, data sharing of software, and task execution schedule make it difficult to predict, quantify, and compare cache performance gains. A formal methodology for predicting performance gains due to the processor caching behavior of a system, makes it possible to compare multiple potential system implementations, resource allocations, and/or implement performance optimizations.

This effort provided five contributions to predictive performance evaluation of the benefits of processor caching in real-time avionic systems: 1) a case study of an industry avionics system that motivates the need for cache optimizations in which code-level software modifications are prohibitively expensive, 2) the System Metric for Application Cache Knowledge (SMACK), a formal methodology for quantifying the expected performance benefits of a system due to processor caching, 3) an empirical evaluation of execution time, L1 cache misses and L2 cache misses of 44 simulated software systems with different data sharing characteristics and execution schedules, 4) A demonstration of the relationship between SMACK score and system performance, 5) An examination of the impact of using SMACK as a heuristic to alter system execution schedules to reduce system execution time.

2. Introduction

2.1. Problem Statement

Modern avionics systems typically employ multiple interconnected commercial-based single-core CPU’s that communicate via a network messaging service. Multiple software applications are combined to execute on a single CPU as depicted in Figure 2. These applications exhibit large run-time performance variations when their execution order changes, often in the range of ±30%. Our experiments with application ordering have identified memory cache miss behaviors as a key factor in this variability. Currently, methods for predicting these “run together” effects are ad-hoc and imprecise. This imprecision forces system architects to be overly conservative in allocating applications to processors to accommodate presumed “worst case” behavior. As a result, processing margins are diminished. Our joint team believes that increasing the precision of cache miss behavior prediction will result in additional schedulable processor bandwidth that can benefit resource-constrained DoD systems by accommodating more work on current hardware, delaying or eliminating the need for expensive upgrades.

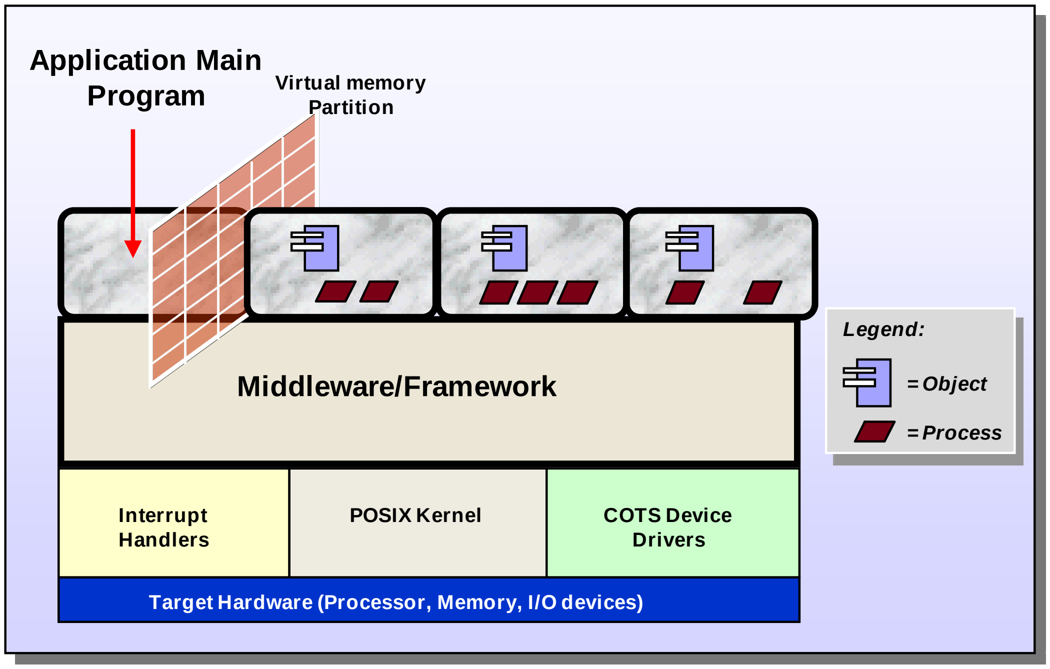


Figure 2. Single CPU Software Architecture

Referring again to Figure 2, each application is typically multi-threaded and occupies a single virtual memory space. The system executes different pieces of each application in response to triggering events, such as message arrivals or pilot commands. The rate and arrival times of these events can vary considerably between execution cycles, depending on system mode or the state of the battle situation. This variability complicates the interleaving of application execution on the computing platform, influencing how processor memory resources are accessed. Even minor changes in the arrival times of triggering events can significantly change memory access patterns, and the subsequent cache miss rate. Figure 3 shows the variance in execution times for a single application when changing only its order in the application set. Note that Ordering 1 yields a significantly higher average execution time than Ordering 2; none of the applications were changed in these observations. Simply rearranging the order in which applications execute seems to cause large changes in worst case execution times.

This variability in execution time leads to the problem illustrated in Figure 4, which shows two different embodiments of the same embedded applications on identical hardware. In both system

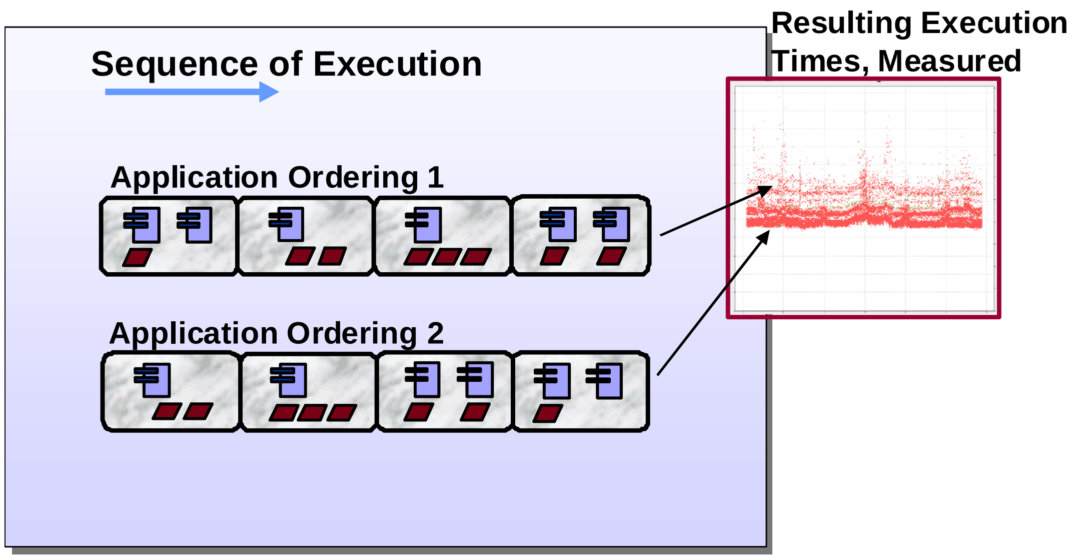


Figure 3. Execution Time Dependence on Application Ordering

utilization diagrams, the embedded applications software consumes the same amount of CPU bandwidth, 55%. However, the system shown on the left has only 8% reserve capacity, severely limiting the amount of capability growth that can be hosted on that processor without going to extraordinary, and expensive, measures to re-develop existing code. By reducing the amount of uncertainty in worst-case execution times, we can increase the usable reserve capacity to 24%, potentially eliminating or delaying the need to replace this piece of embedded hardware to expand the capabilities of the weapon system.

Optimization of task placement is paramount for building cheaper, less energy consumptive,

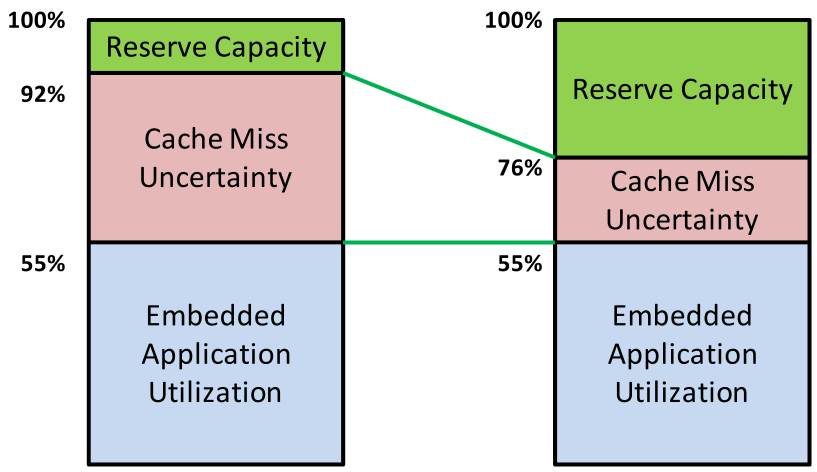


Figure 4. Effect of Cache Miss Uncertainty on Processor Reserve

reliable and more power-efficient distributed embedded systems. By packing software more

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

tightly onto hardware, fewer hardware resources can be used. For example, roughly four pounds of cooling, power supply, and other supporting hardware are needed for each pound of processing hardware in an aircraft. Reducing hardware not only decreases total system cost, but also can facilitate increased ranges for aircraft, decreased fuel and power consumption, and reduced long-term maintenance costs. Furthermore, reduced weight and power consumption enable longer operational ranges and increased payload capabilities.

The primary technical objective of this effort is to develop promising analytic techniques in the context of realistic DoD avionics designs and associated analysis tools. Specific goals include: 1) increasing predictive precision associated with run-time phenomena such as cache miss penalties, modal effects, and aperiodic behaviors, 2) extending the single-core cache miss prediction to multicore processor architectures, and 3) providing an analytic basis for allocating application programs to processing cores that increases hardware utilization efficiency.

2.2. Specific Execution Platform Characteristics

This section discusses the structure of the system under study from various aspects. While it is typical of avionics systems, it is not a precise representation of any particular aircraft produced by Lockheed Martin Aeronautics. It captures the essential aspects of the physical, software, and runtime architectures as they impact this research.

2.2.1. Physical Architecture

The system is physically expressed as a set of computing nodes connected by one or more networks as shown in Figure 5. Each node contains a single core computing element consisting of a COTS processor (typically PowerPC architecture, but this is not essential) with main memory and a two-level cache. The cache can be assumed to have the characteristics shown in Table 1; while the research should be parameterized with respect to these values, it may be helpful to have a concrete target for initial work.

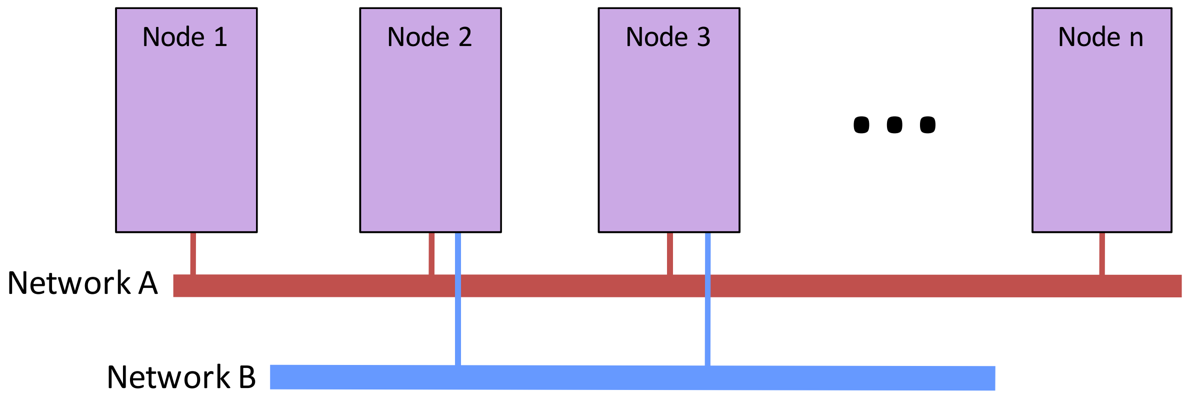


Figure 5. Notional System Physical Architecture

Table 1. Proposed Cache Memory Characteristics

Line size	32 bytes
Associatively	8-way
Total size	1 megabytes
Instruction/data allocation	Shared
Replacement policy	Pseudo-Least Recently Used

There may be multiple network connections available between the computing nodes in the system; however, this research can focus on a single, high-bandwidth connection between nodes, as the other connections typically carry much less traffic. For initial research purposes, the network can be assumed to be fiber optic with the characteristics shown in Table 2.

Table 2. Proposed Fiber Optic Network Characteristics

Bandwidth	400-1000 mbits per second
Topology	Switched fabric
Priority levels	1 (first-come/first served)

2.2.2. Software Architecture

The software is structured as a set of (primarily) singleton objects which 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 which is managed transparently by the system middleware. Objects are overwhelmingly structured as collections of callbacks 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 callback 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 callback 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. This document lays out a structure which prevents interaction between software components which are assumed to operate at different safety levels by partitioning space (memory) and execution time by safety criteria. Applications with lower safety levels are not allowed to be placed in the same partition as those with higher levels.

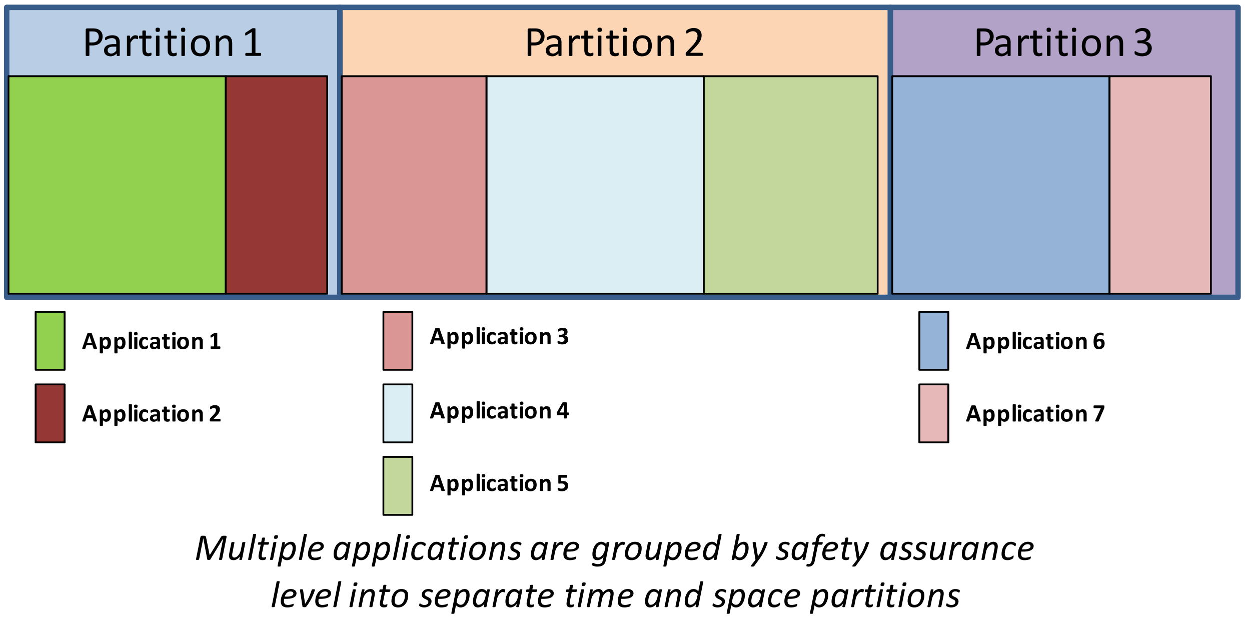


Figure 6. Time & Space Partitioned System Architecture

For the notional system under study, we assume that multiple time and space partitions execute on each node in the network. Figure 6 shows an example system structure with three partitions, in which seven different software applications are implemented.

Each partition is allocated a fixed time duration over 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 RTOS "activates" partitions in the specified sequence, allowing the applications inside each partition to execute in turn, then repeats the sequence.

2.2.3. Runtime Architecture

Each node executes its own system scheduler, which is part of either the operating system or system middleware. The system schedulers between nodes do not communicate explicitly with each other except for system-level status and other housekeeping operations, which are not of interest for this research. The system scheduler on each node implements a rate-based pattern for software execution, which breaks time into a series of numbered frames of equal duration as shown in Figure 7. Each frame is of duration $\frac{1}{N}$ seconds, where N is typically some integer between 40 and 100, and N is said to be the base rate of the system. For example, if the system is scheduled with $N = 75$, each frame will have duration $\frac{1}{75} = 13.33$ milliseconds. Since the base frame rate of the system is 75 times per second, software that is executed at that rate is also said to run at a frequency of 75 Hz.

Each base frame, the software that executes at that rate is scheduled to run, plus another rate of lower frequency. For example, with $N = 75$, at Frame 0 the scheduler will execute the software that runs at 75 Hz and the software that executes at $\frac{75}{2} = 37.5$ Hz, or half as frequently. This pattern continues as shown in Figure 7 until the lowest rate software in the system has completed; the pattern repeats indefinitely.

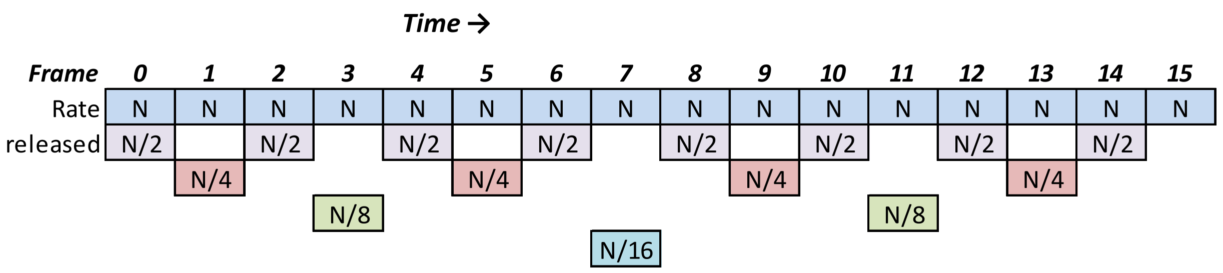


Figure 7. Time & Space Partitioned System Architecture

All of the scheduling of avionic application software 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 be 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.

Revisiting Figure 7 from above, Figure 8 illustrates the effect of priority-based interleaving of callbacks from multiple applications in a partition. As shown, multiple callbacks from Application 1 in Partition 1 may execute in a row, followed by one or more callbacks from Application 2, and so on. For the baseline system under study, we assume that there is no construct for applications to influence the interleaved order other than specifying priority, which is determined by execution frequency. This pattern is repeated in all of the partitions in the system.

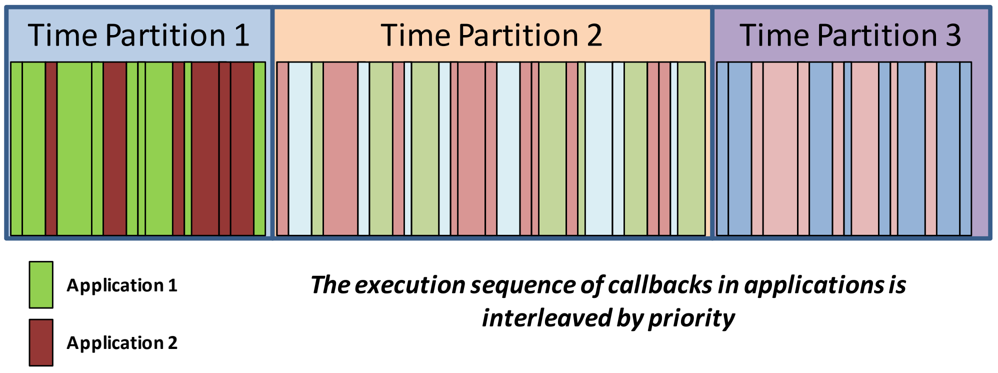

Time Partition 1
Time Partition 2
Time Partition 3
Application 1
Application 2
The execution sequence of callbacks in applications is interleaved by priority

Figure 8. Execution Interleaving inside Time Partition

2.3. Current Cache Optimization Technologies

This section discusses a variety of current techniques for optimizing cache references in general purpose software. The research literature discusses many techniques exist to increase the effectiveness of processor caches by altering software at the code level to change the order in which data is accessed.

One class of optimization, known as data access optimization [14], focuses on changing the manner in which loops are executed. An example of such a technique is loop interchange, which reorders multiple loops such that the data access of common elements in respect to time, referred to as temporal locality, is maximized [14, 2, 28, 29]. Another technique, referred to as loop fusion, is often then applied to further increase cache effectiveness. Loop fusion is the process of merging multiple loops into a single loop and altering data access order to maximize temporal locality [18, 22, 26, 28]. Temporal locality defines the closeness of two tasks in terms of execution start time [14].

Another technique for improving the cache effectiveness of software is to utilize prefetch instructions. A prefetch instruction is retrieves data from memory and writes to the cache before the data is requested by the application [8]. Prefetch instructions can be inserted manually into software at the code level and have been shown to reduce memory latency and/or cache miss rate [6, 8].

While these techniques have all been shown to increase the effectiveness of software utilizing processor caches, they all require source code-level modifications of the software. Many systems, such as the avionic system described in Section 2.2 are safety critical and undergo expensive certification and rigorous development techniques. Any alteration to these components can introduce unforeseen side effects and invalidate the safety certification. Further, developers may not have code-level proprietary components 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.

Another class of optimization techniques focus on increasing performance through intelligent system construction. Constructing valid distributed real-time embedded (DRE) system implementations by configuring prefabricated COTS components is non-trivial due to several constraints, such as real-time requirements, budgetary limitations, and strict resource constraints. However, substantial reductions in execution time, financial cost, and resource requirements can be realized by intelligently configuring DRE systems [21, 27].

Other techniques, such as Software Product Lines (SPLs), examine points of variability in the hardware and software of the system to determine if certain variants offer superior performance [7, 4]. These techniques are appropriate for constructing new system implementations or evolving existing system implementations so that all DRE constraints are met. However, these techniques do nothing to further optimize system performance after a valid configuration has been determined. For example, Bahar et al examined several different cache techniques and the tradeoff between increases in performance and power requirements [3] and saw enhancements as high as 24%. Manjikian et al. demonstrate a performance increase of 25% using cache partitioning and code-level modification techniques [16].

While the above are not the most desirable techniques for safety-certified software, we refer to them to demonstrate the effects of increasing temporal locality on cache effectiveness and performance.   These results suggest that it is possible to use 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 our proposed approach and the current methods is that no modifications are required to the underlying source code that is executing on the system, thereby allowing performance gains to be realized without requiring code-level access or component re-certification.

2.4. Heuristic Optimization as an Cache Miss Minimization Technology

To efficiently explore combinations and discover nearly-optimal task-assignment algorithms, we extended to our existing work on meta-heuristic driven bin-packing based task assignment for network optimization. Prior work by these researchers used a hybrid metaheuristic/bin-packing algorithm to optimize task placement based on task communication characterization. Our previous work on task allocation showed it was possible to use our algorithmic techniques to decrease network bandwidth consumption by ~25%. In this effort, we adapted these existing techniques to apply new cache characterization models to optimize task placement based on cache behavior. Specifically, we extended our prior work on task assignment with bin-packing algorithms to leverage probabilistic cache effect information when making task assignments. In addition, we created bin-packing algorithms that make placement decisions by better accounting for the benefits of co-locating tasks with positive cache effects, resulting in bin selection heuristics that promote co-location of tasks with compatible cache usage characterizations.

One end product of this research is a set of algorithms, heuristics, and analysis techniques which can be expressed in an easily applied form. In the future, these algorithms and heuristics could be incorporated into architectural design and analysis tools which leverage the experience-based knowledge of system architects through automation-assisted exploration of alternatives backed by a solid foundation of mathematical analysis techniques.

Altering the execution schedule of the tasks of an application is another technique for increasing processor effectiveness and reducing execution time. Since altering the execution order of tasks does not require code-level modifications, integrated system designers can apply this technique without needing code-level software permissions or violating safety certifications.

While modifying the execution schedule of the applications is allowed, ensuring that the resulting schedule will uphold the real-time scheduling constraints of the system is difficult. Priority-based scheduling techniques, such as rate-monotonic scheduling, can be used to ensure that the software of a system completes execution without exceeding predefined real-time deadlines. These techniques, however, must be modified to take into account the impact of changing the application execution order on temporal locality so that performance gains due to processor caching can be maximized.

Further, other system specific properties can influence the potential for performance benefits of altering the execution schedule. The effectiveness of processor caching of a system is dependent upon hardware properties, such as cache size and replacement policy, and software properties such as data sharing characteristics and task execution schedule [14, 15, 20, 26, 28, 29, 5, 17, 19, 23,2 4]. These properties must be taken into account when applying cache optimization techniques such as execution schedule alteration.

3. Research Results – The SMACK Methodology

Having discussed current approaches for cache optimization involving source code change by increasing temporal locality, we present a novel heuristic-driven scheduling integration for cache optimization approach to increasing the performance of integrated applications. Altering the execution schedule of integrated applications and executing tasks impacts the data that is stored in the cache at a given point in time, potentially increasing temporal locality. Further, modifying the execution schedule of tasks does not require any code-level alterations. Characteristics of the execution environment make this possible, in particular the repeatability of interleaved execution orders, as shown in Figure 9. The resulting execution order, once determined at system initialization, is repeated throughout the system lifetime; thus, changes in task execution order can be used to effect optimizations.

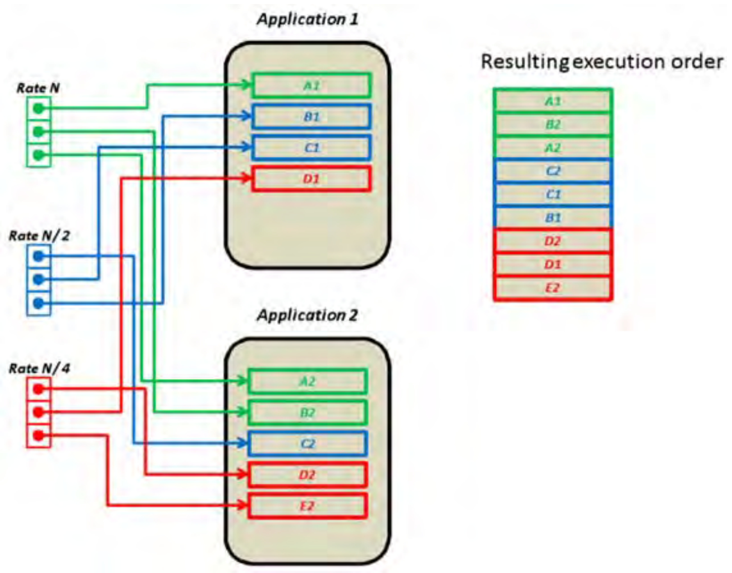


Figure 9. Interleaved Execution Order is Repeatable

To predict the system performance of integrated application execution schedules and guide the schedule modification process, we have created the System Metric for Application Cache Knowledge (SMACK). SMACK considers several factors, such as cache size, data sharing, and software execution schedule to predict the effectiveness of the processor cache. The use of SMACK enables system designers to manipulate models of system runtime behavior without having to repeatedly construct and measure specific implementations.

3.1. Challenges of Analyzing and Optimizing for Cache Effects

Accurately predicting and quantifying the performance of potential implementations for an integrated system such as the avionics systems described in Section 2.2 is critical for making intelligent design decisions. If a predictive metric can be devised that accurately reflects post-implementation performance, potential alterations to the system can be tested without the time and expense required for actual implementation. Determining which alterations should be performed to optimize this predictive metric is paramount for maximizing system performance. Mission-critical systems, however, are often subject to multiple design constraints, such as safety requirements and real-time deadlines that may restrict the optimizations that can be applied. In the case of the system described in Section 2.2, several factors, such as system recertification, unknown data coupling characteristics, and strict scheduling requirements make it difficult to construct optimization techniques for integrated systems. This section describes three challenges that must be overcome for a technique to be applicable for safety-critical DRE systems.

3.2. Challenge 1: Existing Optimization Techniques May Require Recertification

Integrated embedded systems are very often safety-critical. While software crashes may cause minor inconveniences for most system users, unpredictable system behavior in integrated avionics systems can lead to catastrophic system failure. 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 a passenger plane to crash. To ensure that these catastrophes will not occur, the software and hardware components of safety-critical integrated avionics must be certified. This certification guarantees 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 require modifications to the components to increase cache utilization and performance [12, 28]. Modifications of system components, however, may void any previous safety certifications. Re-certification of system components can be a prohibitively slow and expensive process, potentially resulting in dramatically increased system cost and considerable development delays. Therefore, techniques should be developed that alter the system to optimize a predictive performance metric while leaving the hardware and software of the system unmodified. These techniques could increase system performance through better cache utilization, resulting in decreased system runtime while avoiding the need for costly system recertification.

3.3. Challenge 2: Data Sharing Characteristics of Software May Be Unknown

As opposed to small, stand-alone software applications, integrated systems are comprised of several systems made up of many components working together in concert. System developers usually work on a small portion of the components of a single system and are unaware of the inner-workings of components developed by other manufacturers. For example, the software that controls the system flight controller may consist of components developed by multiple companies. It would be impractical to expect these manufacturers to interact with each other to make code-level design decisions. Therefore, system integrators usually remain intentionally unaware of the implementation details of the software components that may comprise the final system.

The data sharing characteristics of software can have a large impact on system performance due to processor caching. However, system size may make data coupling analysis excessively cumbersome and time consuming. Therefore, optimization techniques should be developed to increase performance without requiring that the amount of data shared between software be known a priori. These techniques could then be applied to systems where the data coupling profile is unknown to increase the predictive performance metric and ultimately reduce system execution time.

3.4. Challenge 3: Optimization Must Satisfy Real-time Scheduling Constraints

Another aspect of these safety-critical systems is that they are subject to strict scheduling constraints. These systems commonly use a priority based scheduling method, such as rate monotonic scheduling, to ensure that the software tasks execute in a predictable manner [9, 25]. For example, if a 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 ensures that tasks of higher priority will execute without causing tasks of lower priority to completely starve due to continuous preemption.

Any schedule optimization technique must result in a schedule that does not violate any of these scheduling restrictions. This constraint prohibits many simple solutions that would greatly increase the cache utilization but would also cause the system to behave in an unpredictable and potentially catastrophic manner. This difficulty is compounded by the fact that the priority of system tasks may fluctuate during system lifetime. Optimization techniques should be developed that can be applied and re-applied when necessary to increase the predictive performance metric and decrease system execution time for any set of system task priorities.

3.5. Using SMACK to Improve Cache Performance

Each node of the system described in Section 2.2 consists of multiple partitions of executing applications. The tasks that comprise these applications are scheduled for execution with a priority-based scheduler that is 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 of the partitions described in Section 2.2 is set to execute for a fixed-time duration determined by the expected execution time for all tasks of 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.

3.5.1. Goal: A Cache Hit Characterization Metric for Software Deployments

We propose SMACK, for predicting the relative performance that a specific task scheduling will yield. SMACK can be used to predict the relative performance for multiple execution schedules and to determine which schedule will result in the greatest reduction of required execution time.

We expect that by profiling the data sharing of software tasks and creating an execution schedule that increases the task interleavings that share data leverages temporal locality. We hypothesize that altering the execution schedule of the software tasks of integrated applications to increase temporal locality will result in increased cache hits and reduced system execution time without violating real-time constraints.

3.5.2. How Real-time Schedules can Potentially Impact Cache Hits

The physical structure of the system under study consists of multiple, separate nodes. Each node is divided into separate partitions in which applications execute. As described previously, each application executing in a partition is made up of 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 lead to more or less 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 a common 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 is allowed to begin executing. A major-frame is the set of minor-frames that must execute before all tasks of all rates are guaranteed to have executed. For example, Figure 10 shows an execution for a set of tasks. Tasks A1 through B1 execute in the same minor-frame. The major-frame is the execution of all tasks from minor-frame 0 to minor-frame 15.

3.5.3. Intra-frame and Extra-Frame Transitions

For this study, we examine the impact of transitioning between tasks on the effectiveness of the data cache. Transitioning between tasks executing in one or more minor frames can potentially result in a cache hit. For example, Figure 10 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. Since task A1 and B2 share the same minor-frame, we call the transition from A1 executing to B2 executing an intra-frame transition. If tasks A1 and B2 require common data, then there is potential for a cache hit.

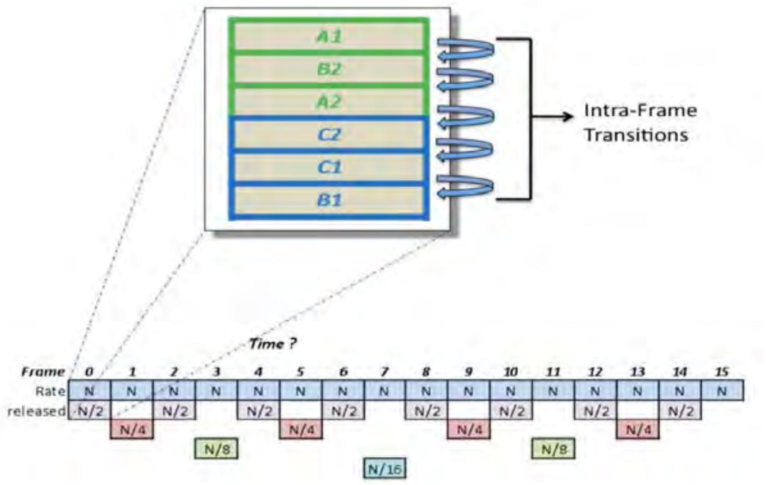


Figure 10. Scheduling with Intra-Frame Transitions

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 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, Figure 11 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.

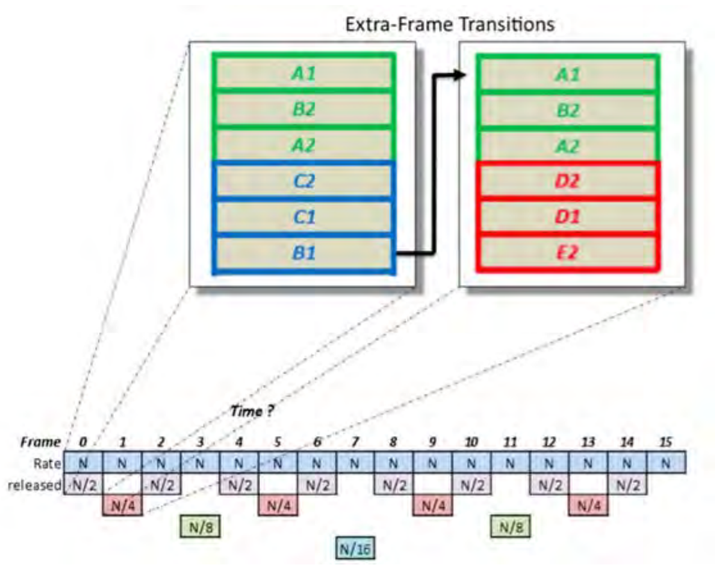


Figure 11. Scheduling with Extra-Frame Transition

3.5.4. Cache Contention Factor

Transitioning a new task onto the processor as described in Section 3.5.3 can result in a cache hit. However, the likelihood of a cache hit occurring as a result of transition is based on the cache contention factor. The cache contention factor is defined by the memory usage of the software, the size of the cache, and the cache replacement policy. The cache contention factor determines how many different transitions can occur before any of the data written to the cache by a task is invalidated, reducing the probability of a cache hit 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.

For example, assume there are 5 applications consisting of 2 tasks, each of which consumes 2 kilobytes of memory of a 64k cache. The hardware uses a Least Recently Used (LRU) replacement policy in which the cache line that has remained the longest without being read is replaced 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 kilobytes to memory. Since the cache can store 64 kilobytes of data, all data from all applications can remain in the cache. The cache contention factor in this case would be 32 since it would require 32 tasks to be executed before the next transition would violate all cached data. Now consider a system in which the total cache is only 2 kilobytes. Executing a task of Application A would write 2 kilobytes of cache to memory, thereby filling the cache. Next, a task of Application B executes writing 2 kilobytes of new data to the cache. Since the cache is only 2 kilobytes, the cached data from the first task will be invalidated. Executing a task from Application A will not result in cache hit since the cache data from the first task was invalidated by the data of Application B written by the second task. In this case, two tasks of the same application must execute consecutively to produce cache hits.

3.5.5. Determining Total Cache Hits

Each intra-frame and extra-frame transition yields a probability of a cache hit based on the contention factor of the system. Each of these cache hits tends to reduce the execution time of the system. The total probabilistic expected cache hits due to these transitions yields the expected cache hits for this set of tasks. Summing the expected cache hits for all set of tasks in all partitions of a given node will yield the total expected cache hits for the node.

The relative execution time reduction for a node due to caching can be determined by multiplying the total number of expected cache hits by the amount of time saved due to successful cache hit, which may differ between nodes. Finally, the relative execution time reduction for a system can be determined by taking the sum of execution time reductions of all nodes. In the following section, we formally define a methodology for determining the relative execution time savings due to caching of system deployments.

3.6. Defining and Calculating SMACK Cache Metric

Section 3.5 describes a high-level methodology for calculating the cache metric of a system deployment. In this section, we will formally define this calculation.

The cache contention factor, *CCF*, shown in the equation in Figure 12, estimates how many consecutive transitions can potentially lead to a cache hit before all cached data from the original task is invalidated. *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 read by both tasks.

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

Figure 12. Equation 1, Cache Contention Factor

In the integrated avionics systems of interest, it is assumed that tasks of different applications do not share any data. Therefore, cache hits can only occur if two tasks share the same application. Equation 2 shown in Figure 13 returns 1 if two tasks are a part of the same application and 0 if they are not.

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

Figure 13. Equation 2, Task overlap probability

Software tasks of the same application may not read the same amount of data in memory. As a result, the number of cache hits that result from a task executing will differ based on the amount of common data read. Equation 3 (Figure 14) defines the maximum cache hits that can be expected if a task of an application executes after another task of the same application. 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.

$$CHit(t(F_i)_j, t(F_x)_y) = DS * DR(t(F_x)_y)$$

Figure 14. Equation 3, Cache hits for variable size tasks

We now calculate the cache hit probability *CHit* for all intra-frame and extra-frame transitions in the major-frame MF. *CHit* takes two tasks specified by the transition in which they execute. For example, $t(F_i)_j$ specifies the task executing at transition *j* of frame *i*. The total set of transitions for a minor-frame *F* is given by *t(F). DS* is the average amount of data shared between tasks. $DR(t(F_x)_y)$ is the percentage of data read by task $t(F_x)_y$ that is shared with task $t(F_i)_j$. Once 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*. Therefore, each transition that occurs before the *CCF* is reached can potentially yield a cache hit and must be investigated

$FR(F_{ij},k)$ determines which task executes *k* transitions from a task as shown in Figure 15, Equation 4. We define *M(MF)* as the number of tasks that execute in a given major-frame. Two cases must be considered: First, a task may execute *k* steps ahead of a task, but in the same major-frame. This is shown in the first case of Equation 4 (Figure 15). In this case, add *k* steps to the index *j* (which specifies the frame in which a task executes) and divide this by the number of transitions per frame |*F*| to determine the frame in which the task *k* steps ahead executes. To determine the position in the frame that is k transitions ahead, add *k* to *i* and take the modulus of the number of tasks per frame |*F*|. Second, incrementing by k transitions may exceed the boundary of the major-frame. In this case, 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 Equation 4.

$$FR(F_{ij},k) = \begin{cases} F_{j+(\lfloor \frac{i+k}{|F|} \rfloor)((i+k)\%|F|)} & i+k < M(MF) \\ F_{j+(\lfloor \frac{(i+k)-M(MF)}{|F|} \rfloor)(((i+k)-M(MF))\%|F|)} & i+k \geq M(MF) \end{cases}$$

Figure 15. Equation 4. Intra-frame transitions

Similarly, Equation 5 (Figure 16) accounts for all cache hits due to all transitions in the super frame. The first summation in Equation 5 accounts for all frames in the major-frame. The second summation examines all frame transitions in the current frame. The innermost summation in Equation 5 sums the expected cache hits *CHit* for tasks that share the same application, as given by O.

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

Figure 16. Equation 5. Inter-frame transitions

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* in the set of all applications *A* executing on partition *p* must be summed as shown in Equation 6, Figure 17.

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

Figure 17. Equation 6. Total cache hits in a partition

However, all tasks for a given partition will be executing in the same major-frame *SF*. Therefore, the total number of caches hits due to all transitions in a major-frame will yield the total cache hits for the set of applications in a partition, as shown in Figure 18's Equation 7.

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

Figure 18. Equation 7. Total cache hits for a set of applications in a partition

Continuing to the next higher level in the system hierarchy, 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* from the set of all partitions *P* executing on node *n* as shown in Equation 8, given in Figure 19. This sum reflects the total probabilistic number of cache hits of the partitions executing on the node.

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

Figure 19. Equation 8. Total execution time reduction on a node

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 system designer or determined through profiling. Once we have calculated the total number of cache hits on the node as described in Figure 18, we multiply this value by the *Cc(n)* to determine the total average overhead reduction (ms) for the node as shown in Figure 19.

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 value is described in Equation 9 (Figure 20), which defines the SMACK metric of a total set of nodes *N*.

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

Figure 20. Equation 9, Total system time reduction

4. Results Analysis - Applying the SMACK Metric to Increase System Performance

This section describes how SMACK can be applied to potentially increase cache hit rate while resolving the challenges described in Section 3. The SMACK metric is used as a heuristic to determine the "score" of potential system configurations. For instance, if calculating the SMACK metric for system implementation A yields a higher score than doing the same for system implementation B, then we assume that executing system A will execute faster due to more efficient data caching.

To determine if SMACK can be applied to reduce the runtime of integrated systems by increasing the cache hit rate, we examine an instance of the avionics system described in Section 2. As discussed in Section 2.2, multiple tasks schedules exist that satisfy real-time scheduling requirements. As stated in Section 3.2 many existing cache optimization methods require altering the software, which then requires expensive and time consuming recertification. Changing the execution ordering of the tasks, however, does not actually alter the software and therefore does not require recertification. The SMACK score of one task ordering may be greater than that of others, thereby indicating a faster runtime. Maximizing the SMACK score can therefore be used as a heuristic for arranging schedules to maximize temporal locality and reduce execution time.

As specified in the system defined in Section 2, tasks of different applications do not share data. The Cache Contention Factor (CCF) determines how many task executions of other applications can occur after a task executes before the cache is potentially completely invalidated. As a result, executing tasks of the same application consecutively will increase the SMACK metric value and therefore decrease execution time, despite having limited or no knowledge of the data coupling between tasks, as stated in Section 3.3. If more is known about the data coupling characteristics of the tasks between common applications, the more accurate the SMACK value will be.

Finally, reordering the tasks to attempt to increase the SMACK value of the system cannot be done in a haphazard fashion. Any execution order must adhere to the real-time scheduling constraints defined in Section 3.4. This greatly restricts the total potential execution orders that satisfy all system constraints. Scheduling techniques, such as rate monotonic scheduling, can be applied to create schedules that enforce real-time constraints.

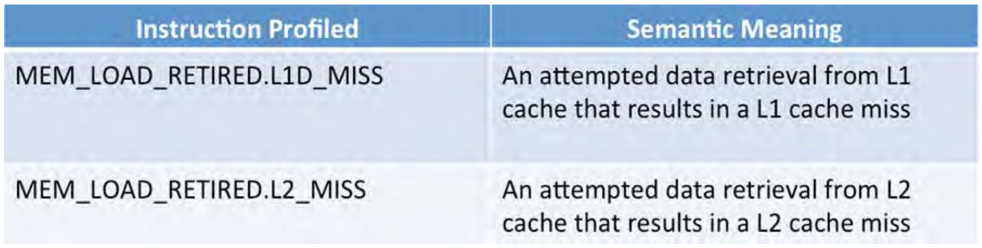
4.1. Empirical Results

This section presents an analysis of the performance of multiple systems with different SMACK values. These systems differ in task execution schedules and the amount of memory shared between tasks. For each system, we investigate potential correlations between the SMACK score and L1 cache misses, L2 cache misses, and runtime reductions.

To examine the relationship between SMACK score and system performance, we created multiple software systems to mimic the scale, execution schedule and data sharing of the system described in Section 2. For each system, 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. We created a Java based code generator to create C++ system code that possessed these characteristics. Rate monotonic scheduling was used to create a deterministic priority based schedule for the generated tasks that adheres to rate monotonic scheduling requirements.

4.1.1. Experimental Platform

The systems were compiled and executed on a Dell Latitude D820 with a 2.16 GHz Intel Core 2 processor with 2 x 32kb L1 instruction caches, 2 x 32 kb write-back data caches, a 4 MB L2 cache and 4GB of RAM running Windows Vista. For each experiment, each system was executed 50 times to obtain an average runtime. 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 L2 cache misses of System ’A’, System ’A’ is compiled and then executed with VTune configured to return the total times that the instruction MEM_LOAD_REQUIRED.L2_MISS is called. Figure 21 shows the instructions that were profiled in the following experiments as well as their semantic meanings.



Instruction 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

Figure 21. Processor Instructions Profiled with VTune

To test the SMACK based schedule modification technique, we created a software suite for generating C++ code representing mock integrated avionics systems that behave as specified in Section 2. As shown in Figure 22, these systems include a priority based scheduler and multiple sample avionics applications consisting of a variable number periodic avionic tasks. (More details of this application generator suite are given in Appendix B.)

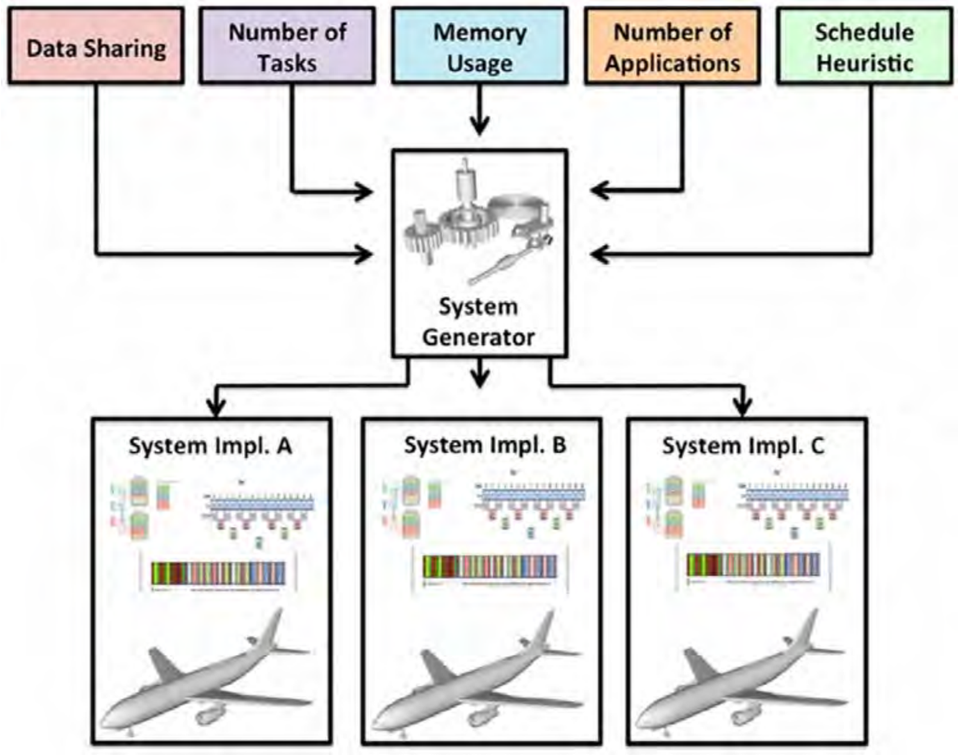


Figure 22. System Creation Process

Together, these components comprise a full test 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 the SMACK metric by showing that a lower SMACK value correlates with better performance in terms of execution time and cache misses.

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 that is shared between two applications, the more likely that the data in the cache can be utilized by tasks of the applications, resulting in reduced cache misses and system runtime. The system described in Section 2 prohibits data sharing between tasks of different applications. Therefore, all systems profiled in this section are also restricted to sharing data between tasks of the same application. However, applications that use a great deal of message data in common are likely to share memory. Due to this, it would not be a difficult extension to account for these architectures in the SMACK metric calculation and will be examined in future work.

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 a single task stores enough memory to completely overwrite any data in the cache, resulting in a Cache Contention Factor of 1. A task from App1 executes, completely filling 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. Therefore, multiple execution schedules effect performance differently and produce different SMACK scores.

4.1.2. Experiment 1: Variable Data Sharing

As stated in Section 2, 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. To examine the effect of data sharing between tasks of common applications, we constructed 10 software systems. Each of these systems contained 5 separate applications consisting of ten tasks each. The body of the tasks consisted of floating and integer addition operations. The total number of operations of the tasks was constant across all applications. The amount of data shared between the same tasks, however, was manipulated. 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 ten software systems, we executed each system 50 times and determined an average runtime of each system.

As can be seen in Figure 23, as the amount of data shared between tasks of a single application decreased, the faster the system can execute. In this case, sharing 100% of data resulted in an execution time of 2572.58 milliseconds, where as a sharing of no data between tasks, or 0%, resulted in an execution time of 3704.85 milliseconds, which is a difference of 44.01%. It is important to note, however, that the curve shown in Figure 23 is non-linear, with only an additional reduction of 9.40% occurring as a result of increasing the shared data amount from 50% to 100%.

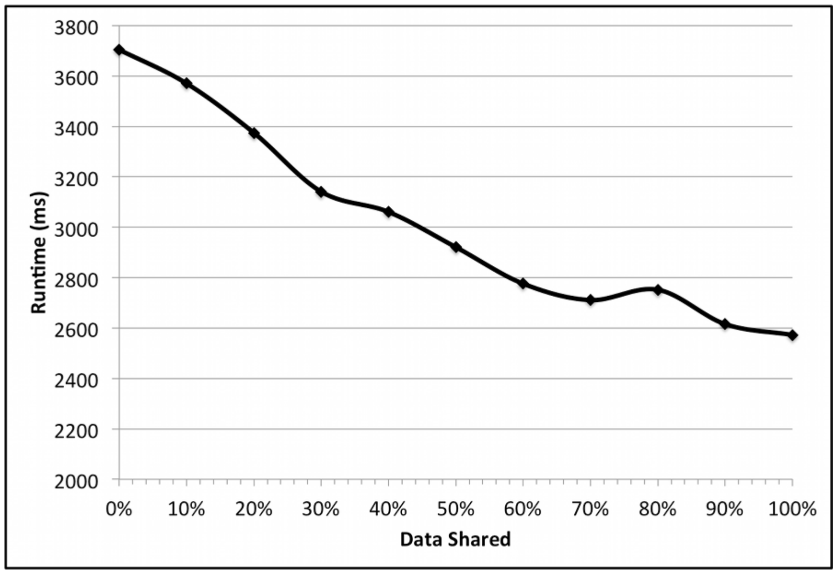


Figure 23. Amount of Data Shared vs Runtime

Increasing the amount of shared data between tasks also leads to a decrease in cache misses. As described in Section 4.1.1 VTune Amplifier XE 2011 was used 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 instructions. It is important to note that these instructions only take into account cache misses due to data write-back and do not include cache misses resulting from instruction fetching.

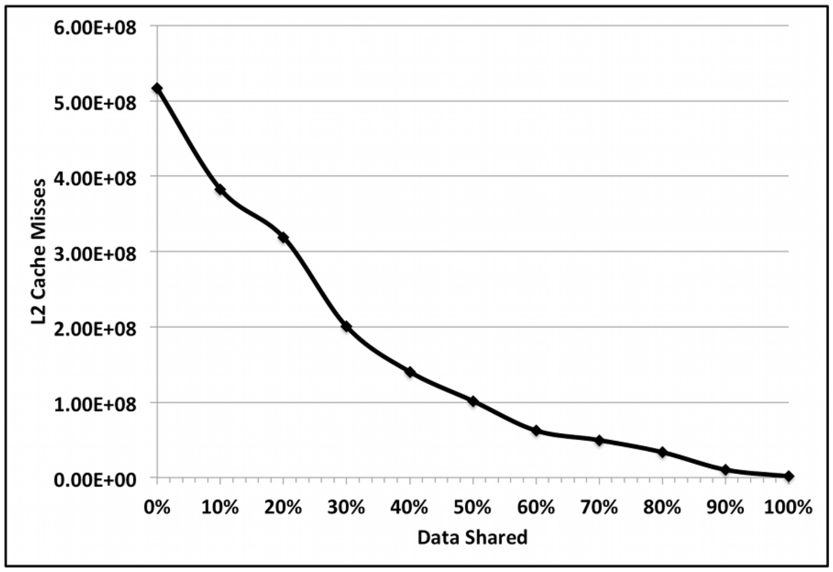


Figure 24. Amount of Data Shared vs L2 Cache Misses

Figure 24 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. In this case L2 cache misses decreased from $5.172x10^8$ to $1.6x10^5$, a reduction of 99.96%. 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. Figure 25 shows the number of L1 cache misses decrease as data sharing between tasks increases. In contrast to runtime and L2 cache misses, the decrease in L1 cache misses is considerably more linear.

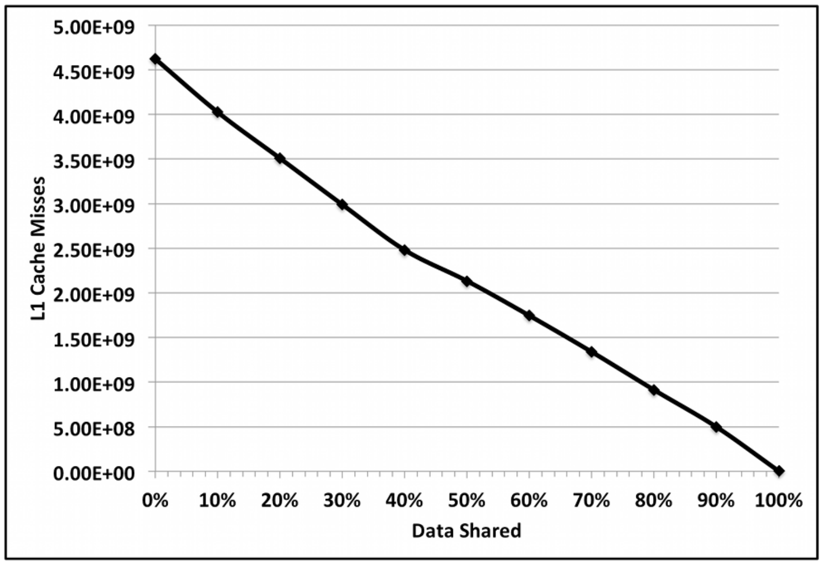


Figure 25. Amount of Data Shared vs L1 Cache Misses

4.1.3. Experiment 2: Execution Schedule Manipulation

As discussed in Section 3.6 the execution schedule of tasks can potentially impact both the runtime and number of cache misses of a system. In this experiment, 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 4 new execution schedules.

First, stride scheduling was used to create an execution ordering that meets all scheduling constraints. 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. Therefore, the Cache Contention Factor for this system would be two.

The original execution schedule generated by Stride Scheduling is referred to as "Unoptimized". The Cache Contention Factor for the experimental platform was 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 it contained 801 overlaps.

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.

The average runtimes for the different execution schedules can be seen in Figure 26. As can be seen, the task execution order can have a large impact on runtime. In this case, the “Best” execution schedule, consisting of 1743 overlaps, executed in 2790 milliseconds on average. The Optimized execution schedule completed in 3299 milliseconds, an 18.24% increase from the Best execution schedule. The Unoptimized and Worst execution schedules executed in 3337 and 3329 milliseconds respectively.

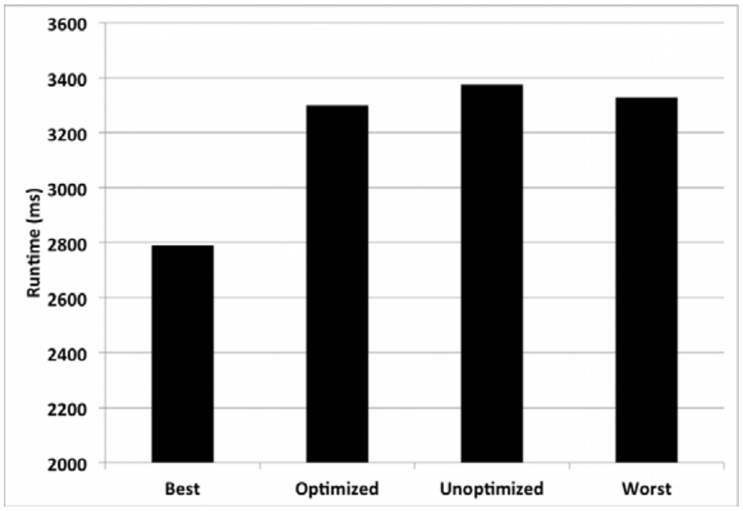


Figure 26. Runtimes of Various Execution Schedules

We refer to this execution order as the Worst execution order as it yields 0 overlaps when the Cache Contention Factor is one. As shown in Figure 27, 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 1743 overlaps. We refer to this execution ordering as the "Best" execution schedule. Note that the “Best” execution schedule is usually not feasible for actual hard real-time execution, since it in effect prioritizes tasks according to what would yield the best cache miss behavior. We use it as a comparison only to quantify the application set’s best-case cache miss statistics, so we can gauge how well the heuristic optimization comes to a maximum.

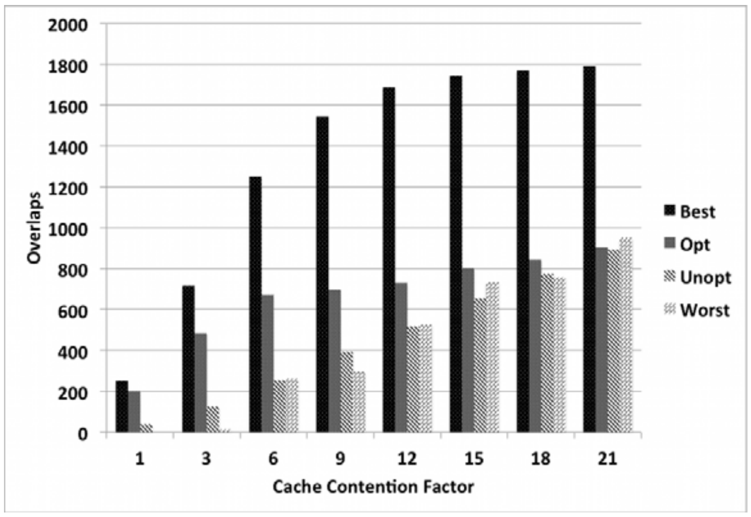


Figure 27. Cache Contention Factor vs Overlaps

Execution order was also shown to impact the number of cache misses. Figure 28 shows the L1 cache misses for all execution schedules. Once again, the execution schedule with the most overlaps, Best, performed the best of all execution orders, resulting in only 3.2584x10^9 cache misses. The Optimized execution schedule, consisting of 801 overlaps, generated 3.484x10^9 cache misses, an increase of 6.92% from the L1 cache misses of the Best execution order. Next, the Unoptimized execution schedule, consisting of 655 overlaps, resulted in 3.5076x10^9 L1 cache misses. Finally, the Worst execution order resulted in 3.5336x10^9 L1 cache misses, the most of all execution orders.

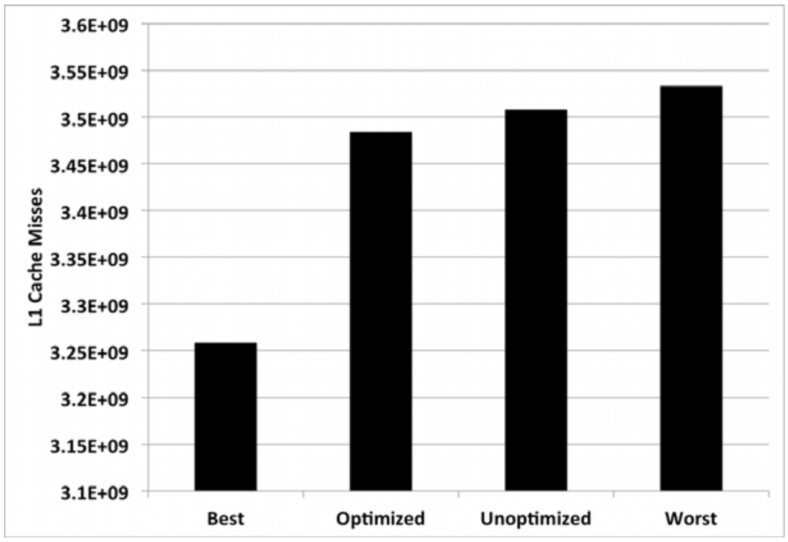


Figure 28. Execution Schedules vs L1 Cache Misses

The impact of execution order on L2 cache misses can be seen in Figure 29. Similarly to L1 cache misses and runtime, the execution schedule with the most overlaps, Best, produced the lowest results with 1.588x10^8 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.

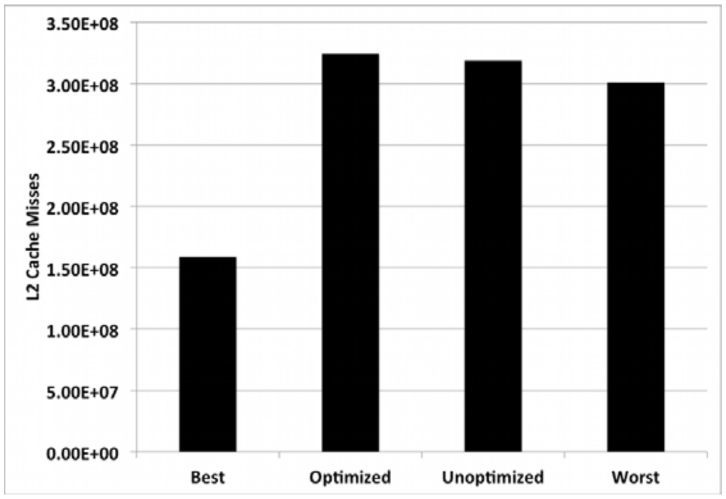


Figure 29. Execution Schedules vs L2 Cache Misses

4.1.4. Experiment 3: Dynamic Execution Order and Data Sharing

Sections 4.1.2 and 4.1.3 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 concurrently. In this section we examine 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.

As can be seen in Figure 30, system execution time decreases as the amount of shared data increases. However, the decrease in runtime is not constant across all execution orders. For example, the Best execution order decreases from 2884 milliseconds when 0% of data is shared to 2398 milliseconds when 100% of data is shared, a total decrease of 486 milliseconds or 16.85%. The Optimized execution order decreases from 3592 milliseconds to 2582 milliseconds as the shared data increase from 0% to 100%, for a total runtime decrement of 1010 milliseconds 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. Therefore, it can be seen that altering the amount of data shared has a larger impact on runtime for systems with less efficient execution orders.

The number of L1 cache misses also decreases as the number of overlaps in the execution order and/or the amount of shared data increases as shown in Figure 31. 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. Further, L1 cache misses for all execution orders decreased at near-linear rate.

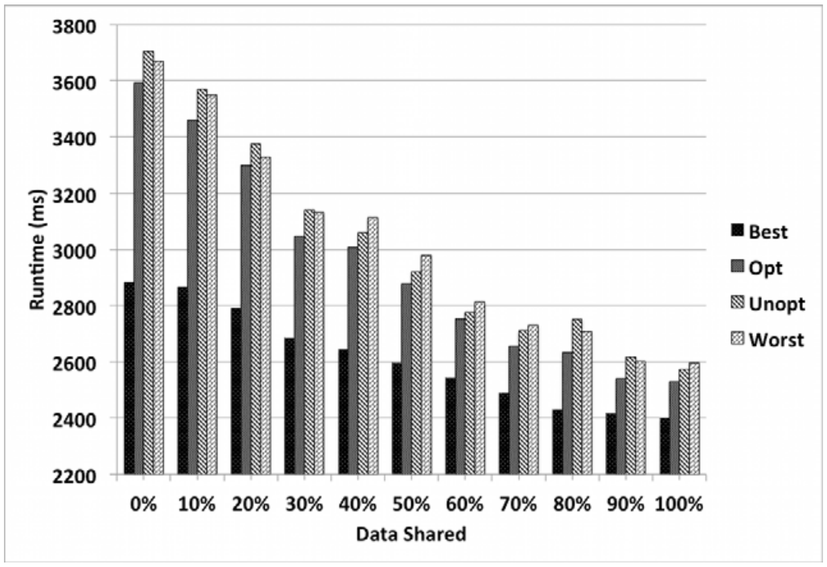


Figure 30. Runtime vs Data Shared and Execution Order

As can be seen in Figure 32, however, 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. 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%.

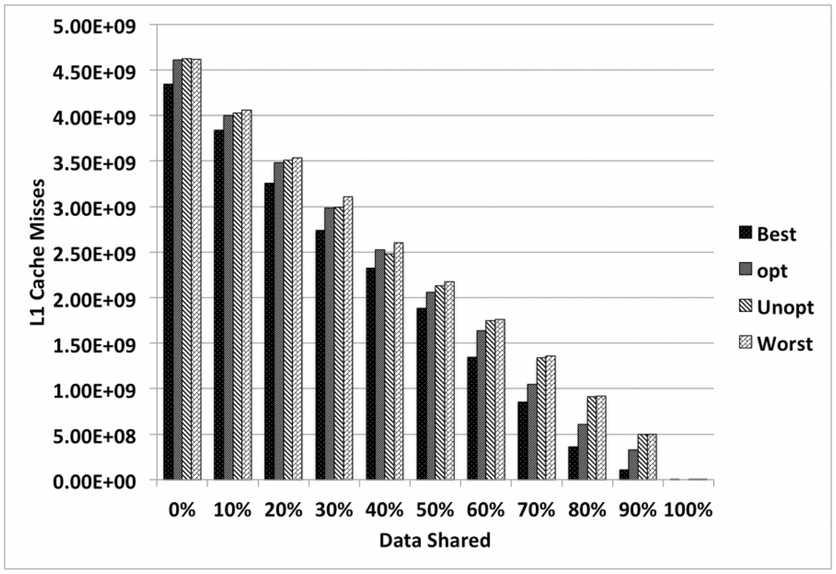


Figure 31. L1 Cache Misses vs Data Shared and Execution Order

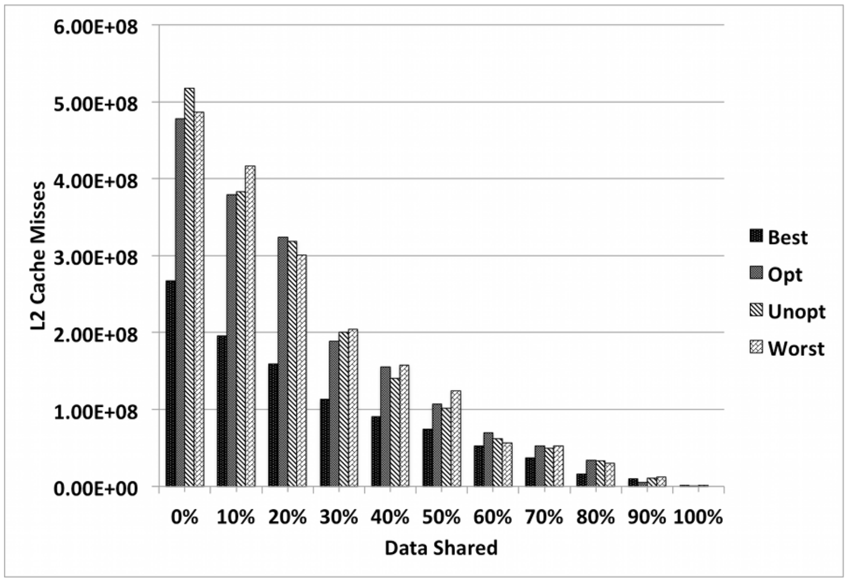


Figure 32. L2 Cache Misses vs Data Shared and Execution Order

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.

As can be seen, altering the execution order leads to a greater reduction in system runtime for systems that share less data between tasks. As data sharing increases, this reduction is not as great. It should also be noted that for the execution orders that satisfy scheduling constraints, the Optimized execution order, resulted in faster runtimes than the Unoptimized execution order. Therefore, runtime reductions can be realized by manipulating execution order without violating priority scheduling constraints.

4.1.5. Experiment 4: Predicting Performance with SMACK

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

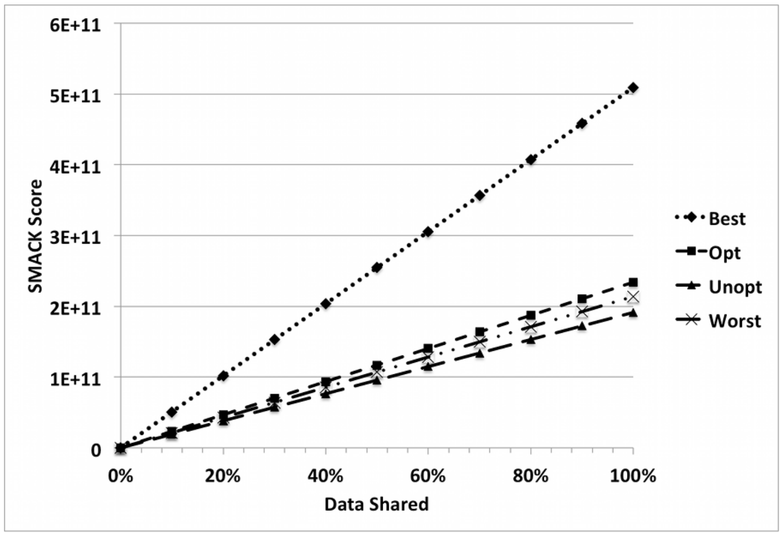


Figure 33. SMACK Score vs Data Shared and Execution Order

Section 4.1.3 presents 44 different systems with data sharing ranging from 0%-100% and four different execution schedules for each. Each system was executed on the same hardware, thereby producing the same value for the contention factor. The SMACK value is calculated for each system taking into account the contention factor, the execution schedule, and data sharing characteristics. Figure 33 presents the SMACK values 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 Figure 33 to the actual system execution times shown in Figure 30 shows that the SMACK score does correlate with an increase in performance. Similarly to runtime, optimizing the execution schedule of a system is also shown to increase the SMACK score. Therefore, the SMACK metric is effective for predicting and comparing the performance of multiple software systems.

Section 4.1.2 presents four 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 such that overlaps are increased without violating priority scheduling constraints.

Figure 34 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% though was shown to be as high as 4.34%. This reduction can be realized without altering the underlying hardware or software executing on the system. Therefore, optimizing system execution schedules to maximize SMACK scores can lead to substantial reductions in system execution time without requiring extensive knowledge of the software, access to the code, recertification, or alterations to the hardware.

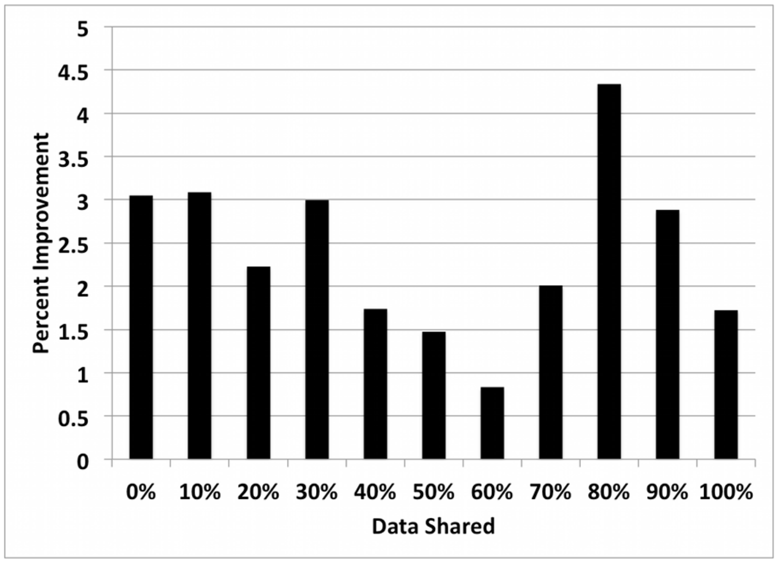


Figure 34. Percent Runtime Reduction vs. Data Shared

It should be noted that the Optimized execution order presented here is not the optimal execution order that would lead to the minimal SMACK score. Even for 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 4.1.3 demonstrates that the Cache Contention Factor of a system changes the effectiveness of an execution schedule. In future work, we will investigate creating an algorithmic technique that takes into account the Cache Contention Factor of a system to maximize the SMACK score and performance gains.

The previous section demonstrated the effectiveness of using SMACK to assess the cache effectiveness of different task execution schedules. However, other techniques exist for optimizing cache hits and system performance. This section compares the SMACK metric and its use as a heuristic for cache optimization with (1) software cache optimization techniques, (2) hardware cache optimization techniques, and (3) other DRE system performance optimization techniques.

While these techniques have all been shown to increase the effectiveness of software utilizing processor caches, they all require code-level optimizations of the software. Many systems, such as the avionic system described in Section 2 are safety critical and must undergo expensive certification and rigorous development techniques. Any alteration to these components can introduce unforeseen side effects and invalidate the safety certification. Further, developers may not have code-level proprietary data for components 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 pre-fetch 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 allowing performance gains to be realized without requiring code-level access or component re-certification.

Several techniques also exist for altering systems at the hardware level to increase the effectiveness of processor caches. One technique is to alter the cache replacement policy that is used by the processor 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 (LFU), First In First Out (FIFO), and random replacement [1, 10, 11].

Which policy is used can substantially influence the performance of a system. For example, LRU is effective for systems in which the same data is likely to be accessed again before enough data has been written to the cache to completely overwrite the cache. However, if enough new data is written to the cache that previously cached data is always overwritten before it can be accessed then performance gains will be minimal. In these cases, a random replacement policy will probably yield increased cache effectiveness.

SMACK does not alter the cache replacement policy of hardware in any way. Similarly to altering software at the code-level, altering the hardware could invalidate previous safety certifications. Further, SMACK uses a heuristic to alter execution order to reduce L1 cache misses. It has been shown that LRU replacement policies are the most effective for reducing L1 cache hits for systems with high temporal locality. Therefore, we believe that SMACK will show the largest performance gains in systems that utilize an LRU cache replacement policy. In future work, we will examine the impact of cache replacement policy on the performance gains of schedules altered with SMACK.

Processor data caching can substantially increase DRE system performance and reduce system runtime. Several factors, such as task execution schedule, data sharing characteristics, and system hardware can influence the caching effects of a system, making it difficult to predict performance gains. Without a formal methodology for predicting performance gains due to the processor caching behavior of a system, it is extremely difficult to compare multiple potential system implementations or apply performance optimizations.

4.2. Experimental Dead-Ends

This section briefly discusses some experimental techniques which the researchers tried and found to be not useful or not optimal.

4.2.1. Using Only Execution Time as a Metric

The primary metric of interest is execution time. Since we target embedded real-time systems, that measurement is the biggest factor in computing CPU utilization, which in turn decides whether new capabilities can be integrated into an existing system without expensive hardware platform upgrades. Once the experimental test bed was implemented, we began running several different execution schedules and recording the run times. While our initial results showed that our scheduling heuristic more often than not yielded superior execution times, this was not always the case. Without an accurate picture of what was occurring at the instruction level of the processor, there was no way to ensure that differences in execution time were in fact due to cache effects. Other system factors, such as jitter due to other background processes preempting experiments, could have potentially been causing the difference in run times.

This caused the experiments to begin recording another metric of interest, the cache miss rate. To obtain a profile of the instruction level hardware events of the system during schedule attention, we used VTune, a processor profiling program of the Intel Amplifier suite. Intel VTune allows the user to create manual instruction level analysis profiles that specify one or more of 100's of Intel based hardware events to sample. Despite the considerable effort required to configure VTune, we were able to show that altering execution schedules did impact the hardware events profiled by Intel VTune, that are representative of cache misses.

4.2.2. Configuring Intel VTune

As stated above, the Intel Amplifier suite (and more specifically, Intel VTune) provided a means for monitoring many user specified hardware events of the system. The process of setting up manual hardware event based samplings, however, was neither straightforward nor well-documented. Determining which hardware events correspond with cache-hits/cache-misses took extensive research through the somewhat poorly organized documentation of Intel VTune.

Also, the default method for creating and executing VTune hardware event-based analysis is through interaction with the VTune graphical user interface. However, this requires the user to manually provide several inputs every time an individual experiment runs. In our analysis, we ran over 44 experiments, which would have required extensive monitoring of the GUI to begin experiment execution and aggregate results.

VTune provides a command-line interface to run analysis. However, the command-line interface is even more poorly documented than the GUI, especially when conducting hardware event based sampling. It wasn't until long after our initial experiments that we finally determined (through extensive hacking of existing configuration files) how to create our own hardware event based sample analyses to be run from the command line. We also eventually figured out how to take the results of VTune and aggregate them in a CSV format that could then be exported into Microsoft Excel for further analysis. Once we were able to run all of our experiments from the command-line as shown in Figure 35, the experimentation process became considerably more automated.

```
sudo amplxe-cl -report hw-events
        -r VTuneResult-TrialRun1 -csv-delimiter "|"
        > "VTuneResult-TrialRun1-output.txt"
```

Figure 35. Command for Collecting Hardware Events for Analysis

4.2.3. Need for Better Understanding of Compiler Optimizations

Since our work relied on the binaries for the tasks, our metascheduling techniques and the SMACK metrics fail to identify the impact of specific compiler optimizations on cache usage and cache optimizations. Consequently, we did not include different metascheduling techniques that are tuned to work most effectively with specific compiler optimization techniques.

5. Conclusion

5.1. Outcomes from PCMA Research

This effort has produced a formal methodology for predicting performance gains related to processor caching behavior. The methodology consists of metrics, analysis methods, and techniques that can be used during systems engineering when computer applications are allocated to processing hardware.

System performance of multiple system implementations can be assessed and compared based on SMACK score. The system with the lowest SMACK score will better utilize the processor cache than other system implementations, resulting in decreased system execution time. Further, certain aspects of the systems could potentially be altered, such as execution schedule, to optimize the SMACK score and decrease system execution time. We empirically evaluated applying the SMACK metric to 44 different simulated industry avionics systems to determine if a correlation exists between the SMACK metric and runtime reductions due to processor caching.

As a result of these efforts, we discovered the following relationships and principles related to 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 execution order of system task affects the SMACK score differently based on the cache contention factor.
- 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 leads to a decreased SMACK score. Reducing the SMACK score correlated with an average runtime reduction of 2.4%. Therefore, multiple system implementations can be compared based on their SMACK scores.
- 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 and First In First Out.
- Relatively minor system knowledge yields accurate 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.
- Algorithmic techniques to maximize SMACK should be developed. The execution order of tasks was shown to have a large impact on system performance and SMACK score. Further, the performance of execution schedules differed base on the Cache Contention Factor. In future work, we will examine the development of algorithmic techniques that use SMACK and the Cache Contention Factor as a heuristics for determining the optimal execution order for the tasks of specific systems.

The above relationships and principles represent new insights that are now available to practicing DoD avionics systems engineers and development teams. They provide additional quantitatively assessable factors and perspectives that have the potential to improve the overall computing resource optimality of a given avionics design. The SMACK methodology is a well formed framework and set of analysis tools that aptly targets a DoD need and accomplishes the original objectives of this research effort.

In addition, SMACK applies techniques at an effective level of abstraction within a given system. This enables ease of use, assessment of key optimization relationships, and effective decision making without requiring extensive detailed knowledge of application implementations.

SMACK optimization potential results are significant to real systems. Typically, avionic systems are highly resource constrained and processors are often required to run at nearly full capacity. Even small percentage gains in run-time efficiency are significant in that they can: 1) allow a high value application to be added to a system that is near capacity limits, 2) reduce the need to hand optimize source code to match resource capacity, and 3) stave off the need to perform a near term or expensive avionics upgrade.

Because SMACK was developed and evaluated using representative and appropriately scaled data, it is applicable to a wide variety of complex legacy and emerging avionic system designs. Ongoing efforts within Lockheed Martin will seek to directly apply these research results to real DoD systems and related design challenges such as legacy system upgrades.

5.2. Future Directions Spurred by this Research

5.2.1. Extension of the Synthetic Application Generation Framework

The synthetic benchmark generation capability referred to in Section 3.6 and discussed in more detail in Appendix B proved to be very versatile, able to create quite large applications which exhibit known properties very quickly and with minimum effort. One of the big problems with using standard benchmark suites is that they are generally written to perform some specific calculation, such as matrix multiplication or linear equation solutions. While perhaps useful in comparing expected throughput of multiple systems, where one executes the same problem on different computers, it is usually difficult to know the amount of cache sharing between tasks in the system. Furthermore, these standard benchmarks are usually represented as a single monolithic program, not as separate tasks which can be scheduled in varying orders.

This approach has been adopted by ongoing research inside Lockheed Martin which is investigating the cache effects of a variety of methods for assigning threads to cores in a large scale multicore processor (8 to 12 cores, currently.) One of the hypotheses being evaluated is whether the time partitioned scheduling paradigm discussed above reduces the inter-frame cache advantage to the point where thread migration costs approach zero, removing one of the reasons to prevent thread migration. The allocation of threads to cores becomes much simpler if this constraint is removed.

5.2.2. Investigating Other Cache Replacement Policies

The cache replacement policy of a processor architecture determines when existing cached data is overwritten. Our heuristic for schedule alteration was built assuming a Least-Recently-Used (LRU) processor cache replacement policy. The LRU policy specifies that the cache line that has not been accessed in the longest amount of time is selected for replacement. This policy is common to many processor manufacturers, such as AMD and Freescale. (In practice, most manufacturers employ a mechanism known as pseudo-LRU which is cheaper and faster to implement, but which asymptotically yields very similar results to true LRU.)

The Intel architectures available for experimentation, however, used a random replacement in which cache lines were selected in a pseudorandom fashion for eviction. The SMACK methodology still performed well with this architecture, because serializing the execution of tasks that tend to reuse the same cache lines reduces the number of evictions required. In the future, we intend to execute experiments on a processor architecture which uses pseudo-LRU so that we can compare the impact of that policy against random replacement. We anticipate that the SMACK methodology will perform even better in this environment.

5.2.3. Extension of SMACK to Mobile Architectures

One of the major benefits of our approach is that performance gains can be obtained in architectures that have extremely limited resource, power, and physical weight requirements. Obviously, the relative benefits that can be gained from utilizing a processor cache more efficiently are not as vital when the cache rarely becomes filled to the point that overwrites occur. Desktop PCs, for example, contain large caches that require extensive computation to fill. As a result, it takes extremely large experiments that take a considerable amount of time to execute to push these caches to their capacity.

Figure 36 depicts the extension of our methodology to mobile devices. These devices tend to be subject to much more stringent resource constraints. Smartphones contain memory and caches that are a fraction of the size of those found on desktop PC's. In future work, we plan to also port our experiments to execute on Android™ devices to examine the same factors we explored in embedded devices. First, the smaller cache of the Android device will allow us to fill the processor cache much faster, allowing us to examine the impact of execution schedules on smaller caches filled to capacity. Second, we would also like to examine reductions of power consumption on mobile devices that could be yielded by executing more cache-efficient task execution schedules. Finally, we would like to examine methods for dynamically altering the Android activity manager to determine faster, more power efficient execution orders of Android applications and services.

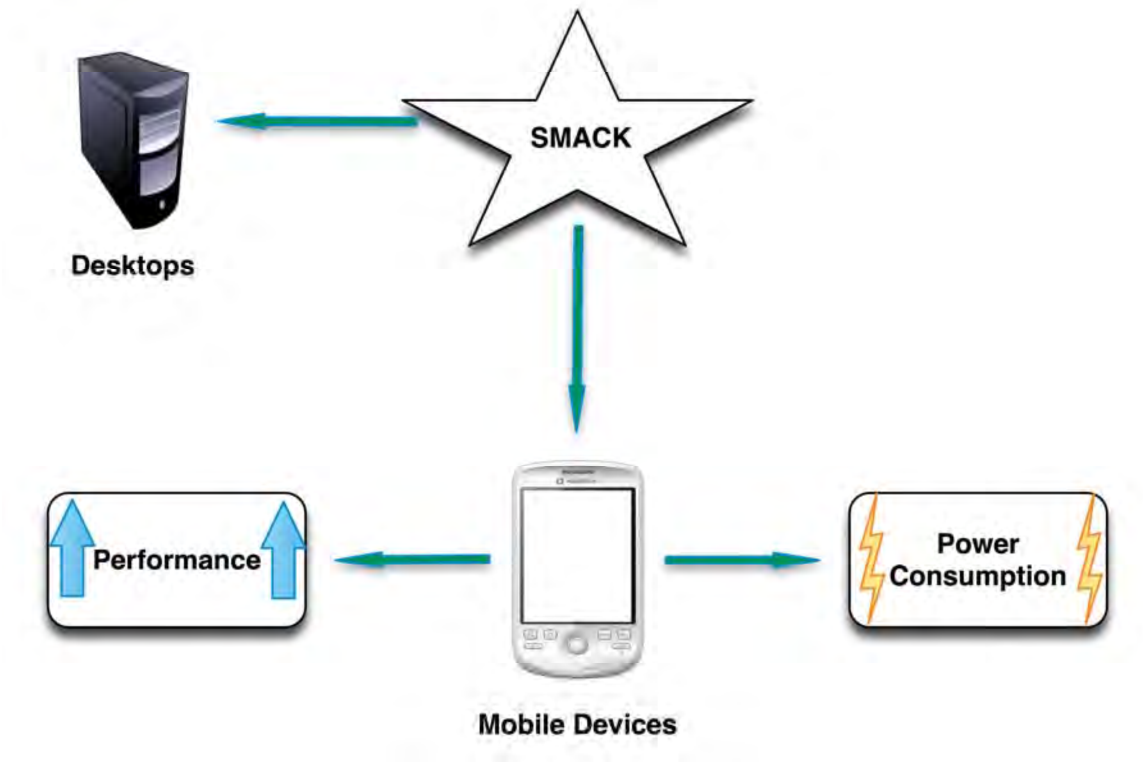


Figure 36. Extending SMACK to Mobile Devices

5.2.4. Rigorous Analysis of Task Order

The problems addressed by this research have spawned more investigation into the analytical methods required to accurately model the effects of execution order on cache misses. Rate Monotonic Analysis, a successful "gold standard" technique for certifying the temporal stability of embedded real-time systems for more than 30 years, depends on the fact that execution times of two different tasks are additive without respect to order. That is, given two tasks A and B, classical Rate Monotonic Analysis assumes that the total time for the tasks executed in the order A→B is the same as when the tasks are executed in the order B→A. Architectures using cache memory invalidate this assumption.

Real-time practitioners have recognized this shortcoming for years, and have either ignored the effect or assumed that the worst case observed measurement contains the worst case order. As long as the effects are relatively small, this approach is satisfactory; however, the system is still subject to chance occurrences where cache misses cascade, causing execution times to increase by large factors, up to 30-40%.

This research seeded a problem that is being investigated by a group of researchers from the Software Engineering Institute, Virginia Tech, and Lockheed Martin Aeronautics.

6. 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] 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.

[4] D. Benavides, P. Trinidad, and A. Ruiz-Cortes. Automated Reasoning on Feature Models. 17th Conference on Advanced Information Systems Engineering (CAiSEÂŠ05, Proceedings), LNCS, 3520:491–503, 2005.

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

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

[7] B. Dougherty, J. White, C. Thompson, and D. C. Schmidt. 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, Apr. 2009.

[8] J. Fu, J. Patel, and B. Janssens. Stride directed prefetching in scalar processors. In Proceedings of the 25th annual international symposium on Microarchitecture, pages

102–110. IEEE Computer Society Press, 1992.

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

[10] 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.

[11] A. Hassidim. Cache replacement policies for multicore processors. In Proceedings of The First Symposium on Innovations in Computer Science. Tsinghua University Press, 2010.

[12] R. Karedla, J. Love, and B. Wherry. Caching strategies to improve disk system performance. Computer, 27(3):38–46, 2002.

[13] 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.

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

[15] A. Lebeck and D. Wood. Cache profiling and the SPEC benchmarks: A case study. Computer, 27(10):15–26, 2002.

[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] 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.

[19] P. Panda, N. Nicolau, and A. Nicolau. Data cache sizing for embedded processor applications. In Design, Automation and Test in Europe, 1998., Proceedings, pages 925–926. IEEE, 2002.

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

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

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

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

[24] M. Squillante and E. Lazowska. Using processor- cache affinity information in shared-memory multiprocessor scheduling. Parallel and Distributed Systems, IEEE Transactions on, 4(2):131–143, 2002.

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

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

[27] J. White, K. Czarnecki, D. C. Schmidt, G. Lenz, C. Wienands, E. Wuchner, and L. Fiege. Automated Model-based Configuration of Enterprise Java Applications. In EDOC 2007, October 2007.

[28] 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.

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

Appendix A. Published Papers and Presentations

Brian Dougherty, Jules White, Russell Kegley, Jonathan Preston, Douglas C. Schmidt, and Aniruddha Gokhale, Optimizing Integrated Application Performance with Cache-aware Metascheduling, 1st International Symposium on Secure Virtual Infrastructures (DOA-SVI'11), October 17-19, 2011, Crete, Greece.

Brian Dougherty. *Deployment and Configuration Strategies for Distributed Real-time and Embedded Systems*. Dissertation, Vanderbilt University: 2011.

Appendix B. Synthetic Application Generator

To investigate the optimization of execution performance through schedule reordering, we need to have applications with known characteristics. Since the primary effect we are attempting to optimize depends on task executions "priming" cache memory for subsequent tasks, application tasks must share data. If the tasks are completely independent, task ordering can have no effect due to cache memory usage.

Typically available benchmarks fall short of the characteristics we need for several reasons. First, the structure of the benchmarks is usually a single thread, with no independent sets of tasks that can be reordered. Second, the memory usage of the benchmarks is not explicit, since the purpose of most available benchmarks is to measure CPU usage executing a given algorithm, not measuring the amount of optimization possible through schedule reordering. Third, it is very difficult to estimate the amount of data used in common between tasks in an application.

To address these shortcomings of typical available benchmarks, we created a simple system for generating synthetic applications with easily reordered tasks with known characteristics. Figure B-1 shows an excerpt from one of the generated tasks.

```
void TaskA07()
{
    int i1, i2, i3, i4, i5, i6, i7, i8, i9, i10;
    float f1, f2, f3, f4, f5, f6, f7, f8, f9, f10;

    ...


    i1 = P1I.i007749 + P1I.i007944 + P1I.i007917 + P1I.i007911 +
         P1I.i007715 + P1I.i007888 + P1I.i007848 + P1I.i007692;
    f1 = P1F.f007653 + P1F.f007654 + P1F.f007655 + P1F.f007656 +
         P1F.f007657 + P1F.f007658 + P1F.f007659 + P1F.f007660;
    i2 = P1I.i007904 + P1I.i007745 + P1I.i007728 + P1I.i007743 +
         P1I.i007706 + P1I.i007850 + P1I.i007883 + P1I.i007730;
    f2 = P1F.f007661 + P1F.f007662 + P1F.f007663 + P1F.f007664 +
         P1F.f007665 + P1F.f007666 + P1F.f007667 + P1F.f007668;

    ...

}
```

Figure B-1. Structure of the Generated Application

The structure of the task is very simple. After declaring some local stack variables, the text of the task consists of simple add statements which reference variables from a shared namespace. The first statement assigns the local variable i1 the value of adding integer namespace variables i007749, i007944, i007917, i007911, etc.

Structuring the application tasks very simply as straight-through execution tends to maximize pressure on the memory, eliminating pipeline stalls due to looping control structures. By keeping the memory fetch architecture as busy as possible, this application highlights the effect of cache misses on overall application performance.

Creating the applications is likewise relatively simple. The application generator starts with a pair of symbol tables, one for integer variables and one for floating point variables. Figure y shows a sample code fragment which generates four tasks, named A01 through A04. Each task is generated using a number between 0 and 1 called the share ratio probability, and a number of integer and floating variables to reference. For a given task in the generated application, a sequence of C++ addition expressions is generated, assigning the value to one of the stack variables created as part of the task definition. Each variable referenced on the right hand side of the assignment statements is chosen by drawing a random number. If the random number is less than or equal to the share ratio probability, the generator chooses a variable name that has already been generated at random from the appropriate symbol table. If the random number is greater than the share ratio, the generator creates a new variable name in the symbol table and uses that reference. In the example in Figure B-2, task A01 will reference variables from symbol table P1, and generate 4000 variable references total with a share probability of 5%.

```
     DefineTask("A", "01", "P1", 4000, 0.05f);
     DefineTask("A", "02", "P1", 2500, 0.1f);
     DefineTask("A", "03", "P1", 1000, 0.1f);
     DefineTask("A", "04", "P1", 3000, 0.01f);
     ...
```

Figure B-2. Sample code fragment showing task generation

The effect of this simple algorithm is a set of tasks which share data in relatively random patterns, but with a very accurately known percentage of sharing. For example, it is easy to generate three different applications with share ratios at 0%, 50%, and 100%, meaning an application where tasks share no data references, 50% of the references in each task are shared with some other task, and 100% of the references in a task are also used by another task, respectively.

Though very simple, the synthetic application generator proved to be quite powerful in use, quickly developing applications with tasks that can easily be reordered at execution time. Since the data sharing between application tasks is known, generating and measuring a set of applications with a variety of sharing ratios yields information about the sensitivity of a given machine architecture to cache reuse optimization.

Appendix C. Notation Quick Reference Guide

This section serves as a quick reference to the notation introduced and used throughout the development of the SMACK metrics in Section 3.

- *Cm(N)* is the Cache Metric, or the expected probabilistic amount of time(ms) saved through cache effects for all *N.*
- *N* is the set of hardware processing nodes.
- *SMACK(N)* predicts the performance of the set of processing nodes *N*.
- *O(ti , t j)* returns 1 if the tasks are of the same application and 0 if not.
- *M(MF)* as the number of tasks that execute in a given major-frame.
- *Cm(n)* is the expected probabilistic amount of time saved through cache effects for a single node *n*.
- *Cc(n)* is the amount of time(ms) that each cache hit saves on a given node n. *P* is the set of partitions for a given node n.
- *θ(p)* total number of expected cache hits for all applications *A* in a given partition *p*.
- *β(a)* is the total number of expected cache hits for all tasks *T* in a given application *a*.
- $Fi \in F$ is the set of tasks that execute in the *i*th minorframe.
- *CHit(Ti,Tj)* returns the probabilistic number of cache hits that will occur when executing *Ti* before *Tj.*
- *MF,* called a major-frame, is the set of minor-frames that execute before the all tasks of all priorities are guaranteed to have executed.
- *CS,* is the size of the processor cache.
- *T* is the set of all software tasks to execute.
- *DS* is the average percentage of application member variables read that are shared between tasks, i.e., if all tasks read the same variables then *DS* is 1.
- *DW(T)* is the total amount of data written to the cache by all tasks.
- *CCC,* called the *Cache Contention Metric*, uses the cache size, *CS,* number of tasks,
- *TTot(SF)* is the total number of cache hits due to extra-frame and intra-frame transitions for superframe *SF*.

List of Acronyms

ARINC	Avionics Application Standard Software Interface
CCF	Cache Contention Factor
Chit	Cache Hit
CS	Cache Size
CM	Cache Metric
COTS	Commercial Off The Shelf
CPU	Central Processing Unit
CSV	Comma-Separated Values
DoD	Department of Defense
DRE	Distributed Real-time Embedded
FIFO	First In First Out
GUI	Graphical User Interface
Hz	Hertz
LFU	Least Frequently Used
LRU	Least Recently Used
MBits	Megabits
RTOS	Real-time Operating System
SMACK	System Metric for Application Cache Knowledge
SPL	Software Product Line