Technical Report

Single Core Equivalent Virtual Machines

for

Hard Real-Time Computing on Multicore Processors

Lui Sha, Marco Caccamo, Renato Mancuso, Jung-Eun Kim, and Man-Ki Yoon
University of Illinois at Urbana-Champaign

Rodolfo Pellizzoni
University of Waterloo

Heechul Yun
University of Kansas

Russell Kegley and Dennis Perlman
Lockheed Martin Corporation

Greg Arundale and Richard Bradford

Rockwell Collins Inc.

Department of Computer Science,
University of Illinois at Urbana-Champaign

http://hdl.handle.net/2142/55672
November 5, 2014

Single Core Equivalent Virtual Machines

for

Hard Real-Time Computing on Multicore Processors

Lui Sha, Marco Caccamo, Renato Mancuso, Jung-Eun Kim, and Man-Ki Yoon
University of Illinois at Urbana-Champaign

Rodolfo Pellizzoni
University of Waterloo

Heechul Yun
University of Kansas

Russell Kegley and Dennis Perlman
Lockheed Martin Corporation

Greg Arundale and Richard Bradford
Rockwell Collins Inc.

Acknowledgement: This paper reviews a technology package that is the results of a team effort. Lui Sha led the development of Single Core Equivalent architecture and system integration. Rodolfo Pellizzoni and Zheng Wu led the development of memory configuration and its schedulability impact analysis. Marco Caccamo and Renato Mancuso led the development of last level cache partitioning and hot code segment lockdown. Heechul Yun and Gang Yao led the development of memory bandwidth reservation mechanism. Jung-Eun Kim and Man-Ki Yoon led the development of Integrated Modular Avionics (IMA) partition scheduling with conflict-free I/O for multicore systems. Russell Kegley and Dennis Perlman guided the development and transition from the perspective of Lockheed Martin. Greg Arundale and Richard Bradford provided the guidance from the perspective of Rockwell Collins.

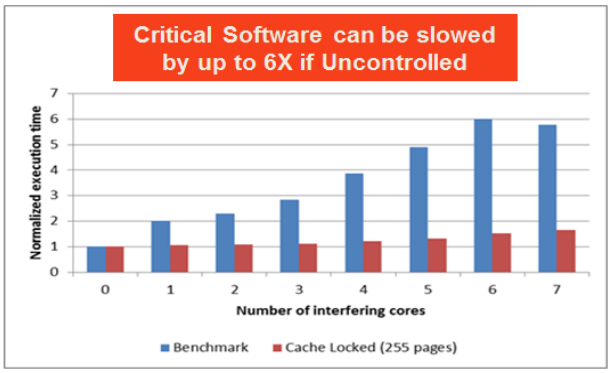
The material presented in this paper is based upon work supported by the National Science Foundation (NSF) under grant numbers CNS-1302563 and CNS-1219064, by ONR N00014-12-1-0046, Lockheed Martin 2009-00524, Rockwell Collins RPS#645038. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the NSF, ONR, LMC or RCI.

Abstract

The benefits of adopting emerging multicore processors include reductions in space, weight, power, and cooling, while increasing CPU bandwidth per processor. However, the existing real-time system engineering process is based on the ***constant worst case execution time (WCET) assumption***, which states that the measured worst case execution time of a software task when executed alone is the same as when that task is running together with other tasks. While this assumption is correct for single-core chips, it is NOT true for multicore chips. As it is now, the interference between cores can cause delay spikes as high as 600% in industry benchmarks. This paper reviews a technology package, namely Single Core Equivalence (SCE), that restores the constant WCET assumption so that engineers can treat each core in a multicore chip as if it were a single core chip. This is significant since FAA permits the use of only one core in a multicore chip due to inter-core interferences [11].

1.0 Introduction

The benefits of adopting emerging multicore processors include reductions in space, weight, power, and cooling, while increasing CPU bandwidth per processor. However, the existing real-time system engineering process is based on the ***constant worst case execution time (WCET) assumption***, which states that the measured worst case execution time of a software task when executed alone is the same as when that task is running together with other tasks. While this assumption is essentially correct for single-core chips, it is NOT true for multicore chips. As it is now, the interference between cores can cause delay spikes as high as 600% as shown in Figure 1. FAA permits the use of only one core in a multicore chip due to inter-core interferences [11].



Source: Lockheed Martin Space Systems Testbed

Figure 1: Delay spikes caused by inter-core interferences on Freescale P4080

The blue bars show the WCET of a single task running on Core 0 when an increasing number of tasks are run simultaneously without any change for a multicore architecture. As Figure 1 shows, the Core 0 task's WCET as a function of an increasing number of competing tasks running on Cores 1 through 7, reaches a peak of a 6X higher WCET, when 7, not 8 cores are used. This means that the worst case configuration of a multicore chip is non-deterministic, creating great challenges to system integration and certification of hard real time systems such as avionics. In contrast, with the Single Core Equivalence (SCE) technology described in this paper, the measured WCET (red bar) is much shorter and it increases monotonically as the number of cores in use increases. Notice that SCE technology preserves the constant worst case execution time assumption but WCET(m) is now function of the maximum number of active cores. This experimental result is in agreement with the model based WCET estimation discussed in Section 3.2.

Multicore real-time computing has many challenges, e.g., how to create near optimal assignments of tasks to cores and how to parallelize large applications such as real time synthetic vision. This paper focuses on how to create partitions of shared resources and optimize their use in such a way that each core can be viewed as if it were a single core computer, where the constant WCET assumption holds. In addition, this paper introduces a model based estimation of tasks' WCET for schedulability analysis that is useful for the sizing and planning of the migration and development of multicore real time systems.

2.0 Single Core Equivalence (SCE) Technology

There are multiple sources of core interference in multicore architectures. These include conflicts in accessing DRAM; bandwidth sharing when accessing DRAM; competition for shared cache resources and I/O resource contention between IMA partitions running in parallel on separate cores. The following sections discuss each of these issues along with mitigating solutions[1].

2.1 Minimizing DRAM Access Conflicts

In a multicore chip, DRAM is one of the major sources of contention. The DRAM structure is organized into ranks, banks, rows and columns. The degree of interference at the bank level is the greatest and we focus on it in this review. Figure 2 shows the average DRAM access latency of a synthetic memory benchmark while varying the number of interfering cores, each of which runs the same benchmark. In

[1] Proposed SCE techniques have been implemented at OS-level. It is recommended to encapsulate them at the level of operating system since some native OS data structures can be leveraged for implementation; however, a different software layering is also possible by encapsulating SCE within a hypervisor module.

SameBank, all cores access the same bank; while in DiffBank, each core accesses a different private bank. In this figure, when the same bank is shared by all cores, memory access latency increases as function of the number of concurrently accessing cores. On the other hand, when each core accesses its own DRAM bank, the memory access latency is not affected by other cores' activity. This shows the potential of improved performance isolation by partitioning DRAM banks among cores [1].

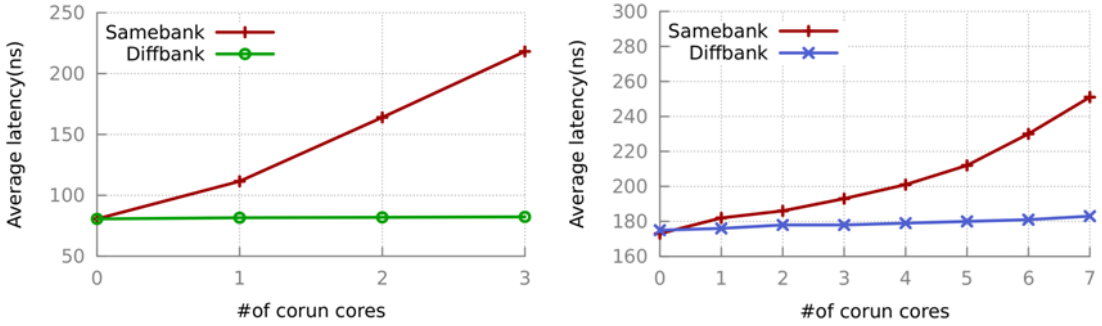


Figure 2: Effect of DRAM contention on Intel Xeon (left) and Freescale P4080 (right)

Unfortunately, current operating systems do not control how memory pages are mapped onto the DRAM banks, which results in unpredictable memory performance. Therefore, we developed a DRAM bank-aware OS level memory allocator, named PALLOC, which allows system designers to assign specific DRAM banks to applications. Using PALLOC, users can ensure that applications in different cores access disjoint sets of DRAM banks, as long as there are a sufficient number of banks to accommodate them.

In the current implementation, PALLOC modifies the Linux buddy allocator so that specific DRAM banks can be selected when allocating new memory pages. Desired banks are assigned to a set of processes via the CGROUP Linux interface. System designers can create multiple partitions (i.e., each partition is a single CGROUP) and specify desired DRAM banks for each partition. When a process in a CGROUP allocates a memory page (e.g., when a page fault occurs), PALLOC allocates only pages from the specified banks. Currently, all memory pages including text, data, and heap, must be allocated from the specified banks. A detailed discussion on how PALLOC works can be found in [1].

From an avionics application perspective, Integrated Modular Avionics (IMA) [2] is a widely used standard, where a low level time division multiple access (TDMA) schedule partitions CPU cycles for different applications. Within each IMA partition, fixed priority scheduling is used. Since only one IMA partition per core can be active at any time, giving each core a private set of banks completely eliminates inter-core bank conflicts.

However, there are two important considerations when applying the private banking strategy. First, each partition's memory space is restricted by the number of assigned DRAM banks. As an example, consider the Freescale P4080 platform used in our experiments: it includes a memory configuration with a total of 32 DRAM banks (2 DIMMs x 2 ranks/DIMM x 8 banks/rank = 32) and the size of each bank is 128MB. Therefore, if an application running in an IMA partition requires more than 128MB, for example, additional banks must be assigned to the corresponding core, in order to make the core fully private. If each core cannot be assigned its own private banks, then a mixed policy of assigning private and shared banks can be used. In this case, the most heavily used part of memory should be kept on private banks. Second, depending on applications, private banking can slightly reduce maximum achievable memory performance of each core, because the number of concurrently accessible banks per core is smaller. We found, however, the reduced peak performance per-core is not significant for most applications we tested; more detailed evaluation results can be found in [1].

As a final remark, it is important to notice that experiments of Figure 2 use a synthetic benchmark that does not saturate the shared memory data bus; hence when each core accesses a different bank, the benchmark latency is practically unaffected by the number of contending cores. In the general case, however, the memory bandwidth can also become a bottleneck, motivating the necessity of memory bandwidth control, which will be discussed in the next section.

2.2. DRAM Bandwidth Management

In a multicore chip, when applications running on multiple cores concurrently access memory, their memory bandwidth needs can interfere with each other since they all share the same main memory. At times, any application can demand high memory bandwidth that can cause significant delays to the memory accesses from critical real-time applications. To achieve better isolation in using memory bandwidth, we developed an OS level memory bandwidth management system, called MemGuard [3]. The goal of MemGuard is to provide bandwidth reservation to each core such that the stall time each application suffers due to the memory management system becomes predictable and analyzable (estimation of tasks' WCET and schedulability analysis will be discussed in Section 3.2). Intuitively, MemGuard enforces memory bandwidth reservations in the same way an IMA CPU partition enforces a CPU bandwidth reservation to protect the execution of its applications.

MemGuard has two main components: a per-core bandwidth regulator and a global bandwidth reclaiming manager. The per-core regulator is responsible for monitoring and enforcing its corresponding core memory bandwidth allocation. Each regulator is given a memory access budget Q_i for every regulation period P, which is global across the cores. In a chip with N cores, the sum of N memory bandwidth reservations is equal to the sustainable system memory bandwidth. When the given budget is exhausted, the regulator calls the OS scheduler to suspend all tasks assigned to that core. At the beginning of each regulation period the budget is replenished in full and the OS resumes suspended tasks. Regulation period P is a processor-wide parameter and should be much shorter than the minimal application task period. In our current implementation, P is one millisecond matching also the OS scheduler tick interval.

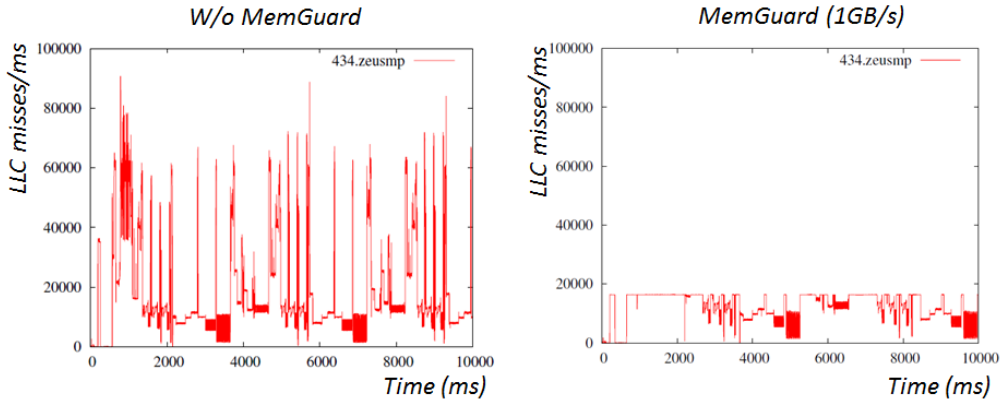


Figure 3: Last Level Cache Miss without and with MemGuard on Intel Xeon

Figure 3 shows the impact of bandwidth reservation provided by MemGuard. The experiment measures last level cache (LLC) misses with 1ms granularity over a 10 seconds duration. Without MemGuard (left), the LLC miss rate varies significantly over time. With MemGuard (right), however, the miss rate is regulated by the bandwidth reservation mechanism as depicted in the figure. In this example, the reservation parameter Q is 1GB/s (≡ 16384 LLC misses/ms).

By restricting maximal memory bandwidth usage of each core, MemGuard ensures that each core always has its reserved memory bandwidth reservation Q_i within the regulation period P. Since task worst-case execution times are used to set resource reservations in hard real time systems,

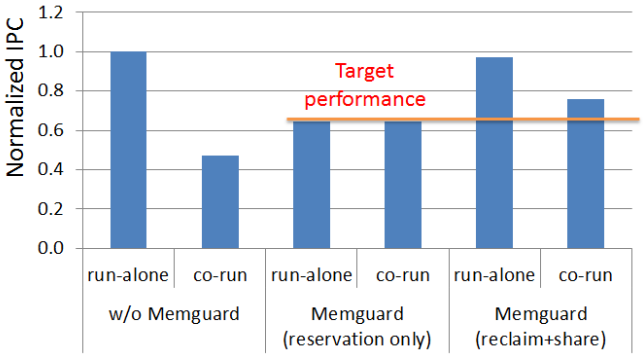


Figure 4: Performance impact of MemGuard on Intel Xeon

applications typically use less than their reservation. To achieve efficient memory bandwidth usage, MemGuard also offers different ways to share reserved but unused memory bandwidth across the cores. Details can be found in [3]. The sharing of memory bandwidth produces significant performance improvements at the cost of occasionally not receiving the full guaranteed reservation. Hence, it is important that cores running applications with high level of criticality do not use this additional feature.

Figure 4 shows the performance impact of MemGuard. The Y-axis shows the average Instruction Per Cycle (IPC) of a SPEC2006 benchmark 462.libquantum when it runs alone, and together with a memory intensive co-runner ("memory hog"). Without MemGuard, performance of the benchmark varies significantly: more than 50% reduction when it is co-scheduled with the memory hog. When we reserve memory bandwidth for the two tasks by using MemGuard (1000MB/s for libquamtum and 200MB/s for the memory hog), however, we observe a dramatic reduction in performance variation; in fact, run-alone and co-run IPCs are almost identical. However, this predictable behavior comes at the cost of reduced run-alone performance since the benchmark demands more than its reserved bandwidth. By enabling reclaiming and sharing features, MemGuard(reclaim+share), we achieve better performance in both run-alone and co-run cases since the unused bandwidth is efficiently redistributed at run-time. Details of the algorithms and evaluation results of MemGuard can be found in [3].

In summary, MemGuard improves performance isolation on multicore platforms by using efficient DRAM bandwidth reservation mechanisms, while PALLOC (Section 2.1) achieves similar results by partitioning DRAM banks. Together, they provide strong isolation when accessing a shared DRAM memory.

2.3. Cache Management

While DRAM is an important shared resource, cache memory is equally important and its management involves many aspects. From the perspective of SCE, we will focus on inter-core interference. In multicore chip architectures, the last level cache is commonly shared by all the cores. We propose an efficient yet flexible use of last level cache that involves two main stages [9]: an offline profiling stage, and an online cache allocation stage.

Profiling: The memory usage profiler is designed to identify the most frequently accessed virtual memory pages for a given executable. For each task, a profile is produced offline, in which memory pages are ranked by access frequency. However, such profile cannot be based on absolute virtual addresses, because virtual addresses change from execution to execution. A common abstraction in operating systems is the concept of a memory region: a range of virtual addresses assigned to a process to contain a given portion of its memory. Thus, processes own several memory regions to hold executable code, stack, heap memory and so on. A profiling tool was implemented that exploits such abstraction to create a profile where a frequently used page is expressed as an offset from the beginning of the memory region it belongs to. In this way, the profile needs to be created only once, and it is possible to determine the effective virtual address of frequently accessed pages at run-time. This information is used to perform cache allocation accordingly.

The online allocation stage relies on a combination of two mechanisms to provide deterministic guarantees and a fine-grained cache allocation granularity:

- **Page coloring**: multiple DRAM pages can be mapped to a given set of shared cache pages. The pages in the same set are said to have the same "color". Pages with the same color can be allocated across cache ways, so that as many pages as the number of ways can be allocated simultaneously in last level cache. Our OS techniques are able to reposition task memory pages within the available colors, in order to maximize allocation flexibility.
- **Lockdown**: Real time applications are dominated by periodic execution flows which often exhibit tight inner loops. This characteristic allows for an optimized use of last level cache by locking pages with the highest hit score first. Relying on profile data, we first color frequently accessed memory pages to remap them on available cache ways; next, we exploit hardware cache lockdown support to guarantee that such pages (once prefetched) will persist in the assigned location (locked), effectively overriding the default cache replacement policy. In avionics

applications using the IMA architecture, frequently accessed pages can be preloaded at the beginning of each IMA partition, achieving efficient and predictable cache usage.

The combination of the mentioned techniques takes the name of *Colored Lockdown* [9]. We recommend allocating last level cache evenly across cores as the default configuration.

Application transparency: In addition to the development of a cache usage profiling tool, Colored Lockdown has been implemented by modifying the Linux kernel's page management algorithm in such a way that color lockdown is invisible to application logic.

The online stage of last level cache allocation (Colored Lockdown) is depicted in Figure 5. We assume that profiling has been performed at an earlier stage. At startup, the task's profile is processed and frequently accessed memory pages are assigned to available ways through coloring. Next, lockdown is performed, during which the system prefetches and locks all (or a portion of) the hot memory areas of a given task in the last level cache. According to our methodology, this procedure can be deferred until the activation of the first job in the task that will address a certain set of hot memory areas.

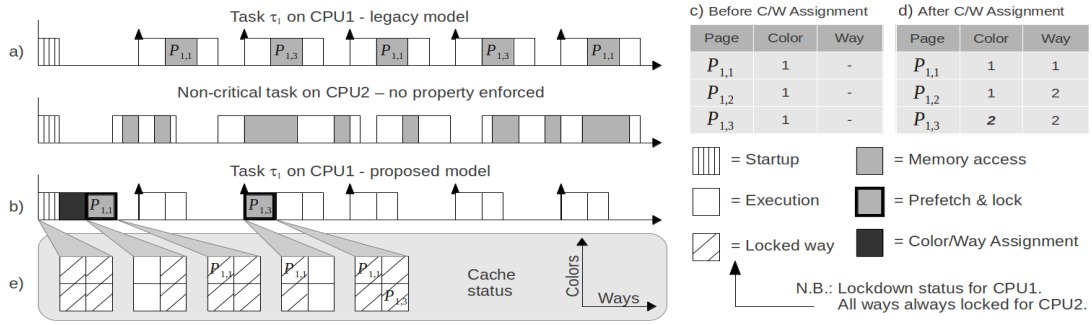


Figure 5: SCE vs Legacy System cache management

Note that colored lockdown allocates colors and locks data in cache at the granularity of virtual memory pages. Furthermore, a part of the cache area (locked for the critical tasks) is unusable by the non-critical ones, and this can have a negative impact on their average performance. However, the clock cycles needed to execute the critical tasks are significantly reduced, as shown in Figure 6, and temporal predictability is significantly improved.

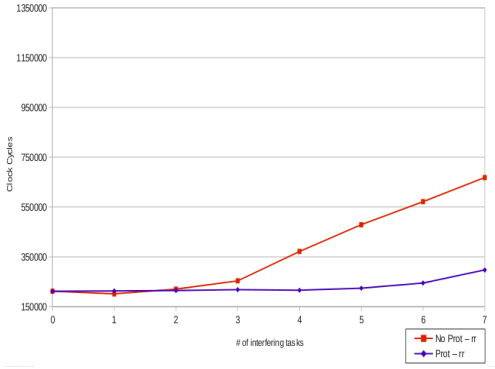


Figure 6: Performance of synthetic benchmark on Freescale P4080

It is worthwhile to mention that the experiments of Figure 6 were run without enabling either PALLOC or MemGuard. Strong performance isolation is achieved only when both of them are used, especially when the memory usage is high.

Using Colored Lockdown

To optimize cache allocation, we use the profiling information extracted for each task and use a fit curve that describes how the task WCET varies as a function of the number of memory pages locked in last-level cache.

Figure 7 shows the observed WCET of a task as a function of the number of frequently accessed memory pages allocated in cache. We call it as *Progressive Lockdown Curve*. Not surprisingly, it exhibits a convex shape since at profile time pages are ranked by number of observed accesses. The

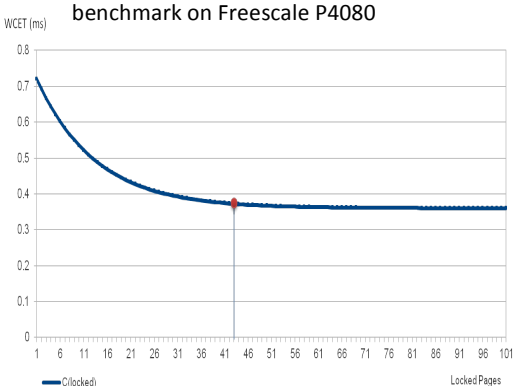


Figure 7: Progressive Lockdown Curve

progressive lockdown curve relates cache assignment with resulting WCET, so that it is possible to estimate the task WCET given the desired pages of cache assignment or vice-versa. In our example, the first 43 profile pages of the task under analysis are selected to be locked in last-level cache, resulting in a WCET of 0.37 ms. Since the amount of last-level cache is limited, the total number of locked pages for all the tasks in the same partition cannot exceed the per-core cache assignment size. The Progressive Lockdown Curve for each task can be experimentally derived by using a *"solo"* configuration where: a) even cache allocation is assigned to cores, b) all cores but one are kept idle, c) private DRAM banks are assigned to each core using PALLOC, and d) MemGuard is not activated, hence the task under analysis can retrieve data up to the peak memory bandwidth.

Once a Progressive Lockdown Curve is obtained for each task and the cache assignment has been performed, the value of WCET (C) and maximum number of residual cache misses (μ) are associated to each task. Since all the analysis so far has been performed in a single-core scenario, the value of WCET obtained at this step has to be inflated to account for the fraction of memory bandwidth allocated to each core. In Section 3.2, we describe how the inflated value of WCET (named C_{sce}) can be estimated off-line for each single task.

2.4. IMA Architecture and I/O Management

In IMA, real-time applications run within a *partition,* which is the basic execution environment of software applications. Since IMA supports partitions' temporal and spatial isolation from one another, real-time avionics functions with different safety-assurance levels can use different IMA partitions. IMA partitions also provide the basis for migrating single-core based IMA applications to multicore avionics systems. However, since IMA partitions share I/O devices and channels, interference among I/O transactions from application partitions residing on different cores raises the need to synchronize all the I/O transactions [5].

In IMA, zero partitions (I/O partitions) are used for I/O transactions in a consolidated fashion [5]. As a result, the multicore I/O synchronization problem reduces to the problem of I/O partition synchronization. In our approach, all I/O partitions are consolidated using a dedicated core, called *I/O core*, to simplify their management. The use of I/O core allows for IMA partitions in different cores to have different major cycle lengths and significantly improves the feasibility of I/O scheduling.

According to SCE framework, the I/O core is responsible for handling the peripherals so that no I/O transaction is initiated outside its assigned time slot (Device-Input/Output). This can be done by either adopting a polling scheme for incoming data or by unmasking interrupt lines for device-originated events during the assigned slot. Whenever the I/O core is ready to communicate with a given device, a DMA engine is used to directly perform the DRAM-to-device transfer or vice-versa. In order for the I/O core to respect the system-wide bandwidth allocation, DMA bandwidth control (BWC) features should be enabled upon servicing any memory transaction. These features are normally available in commercial DMA units and can be used to configure the number of consecutive bytes that a DMA engine is allowed to transfer between consecutive wait states. For instance, in the Freescale P4080 platform wait states can be inserted between DMA transfers every *b* bytes, where *b* is a power of two ranging between 1 and 1024.

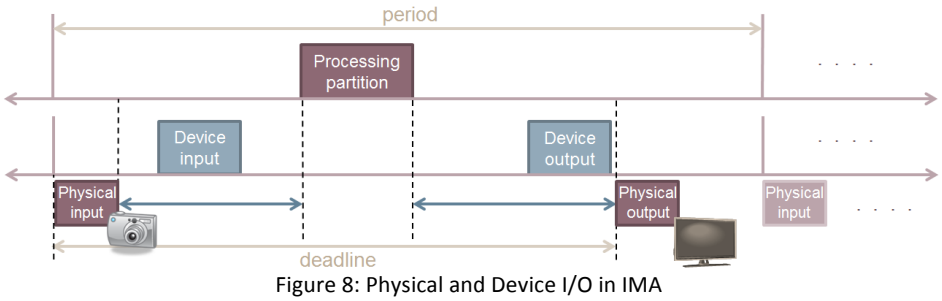


Figure 8: Physical and Device I/O in IMA

Each I/O transaction is divided into *Physical-I/O* and *Device-I/O*. Physical-I/O is for reading and writing raw data between the physical environment and an I/O device. For example, a transaction of a camera taking

pictures from the physical environment is a *Physical-Input*. A *Device-Input* then relays the buffered data for pictures to the main memory, where they are then processed by a *Processing* application running on a core. The output of the processing is buffered in the main memory and then transferred to an output device through a *Device-Output* transaction. The last transaction is a *Physical-Output* in which the resulting image is scattered on a final entity such as a synthetic vision system. Thus, these five transactions have a precedence relationship as shown in Figure 8.

From a scheduling perspective, we must observe the precedence relation from physical input, to device input, to processing, to device output and finally to physical output. In addition, the I/O partitions on the I/O core must not overlap, and the processing partitions on the same core do not overlap. Finally, physical I/O transactions at different devices are performed in parallel.

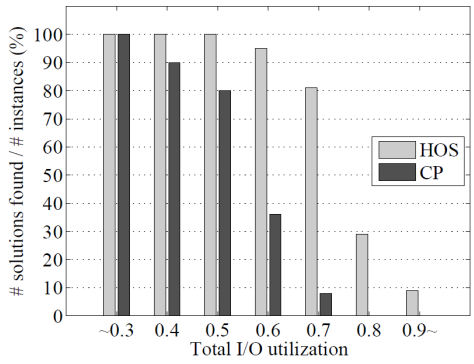


Figure 9: CP vs. HOS performance

This I/O scheduling problem can be formulated as a constraint programming (CP) problem. However, this is an NP-hard problem and CP does not scale. We developed a heuristic algorithm called hierarchically ordered scheduling (HOS). HOS starts with a random but partial assignment of the offsets of all physical-I/Os and processing partitions and then finds a complete solution by determining the offsets of all device-I/O partitions. This heuristic shares the idea of First Fit Decreasing for bin packing by meeting the hardest constraints first. Once the physical I/O and processing partition offsets are fixed, the search space for the rest, i.e., the Device-I/O partition offsets, can be represented as a set of periodic intervals, which reduces the problem size considerably. Through this hierarchical searching, the algorithm quickly finds a solution on average and scales well with the problem size as shown in Figure 9. For further details, the interested reader is referred to [4].

3.0 Single-Core Equivalence Methodology

Our methodology consists of two parts. Part 1 is to partition globally shared resources to create SCE cores. Part 2 is to estimate WCET of each task. We note that once WCET is obtained, the schedulability analysis for each task in a partition is the same as the one used in the single core chip [6].

3.1 Create Single Core Equivalent Cores

1. Hardware selection: We select a multicore chip that provides the primitives needed by SCE technologies, which include support for last level cache partition and locking together with specific performance counters required by MemGuard. An example is Freescale P4080 [10].
2. Resource partition: We minimize the DRAM conflicts as described before. For example, each core has its own memory banks and PALLOC is enabled. The default is to assign equal memory bandwidth to each core and enforce it by MemGuard, and to partition the last level cache evenly among the cores. Together with private DRAM banks for each core, this almost creates a set of virtual "single core chips". By "almost", we are mindful of the fact that cores still share the I/O channels at this stage.
3. Partition to core allocation: Before multicore chips, the move from slower single chips to a smaller number of faster single core chips has been done many times in avionics. We assume that the same allocation heuristic is used, where the partition size is also adjusted to the new chips' speeds plus, e.g., 30% slacks in CPU cycles, so that it will be easy to add new software features within a partition in the future. Still, there is a problem: these "almost virtual single core chips" do not have their private I/O channel partitions.
4. I/O channel partition: Using the HOS heuristic described in Section 2.4, we will check if the shared I/O channels can be partitioned temporally. That is, non-overlapping I/O (zero) partitions for all partitions running on the cores can be created and meet all the precedence and capacity

constraints. If not, go back to step 3. If this loop has no solution, more/faster chips are needed. If there is an I/O solution, we have now created a tentative set of virtual single core chips.

5. Last level cache partition optimization and schedulability analysis: We now optimize the usage of last level cache partitions using colored lockdown as discussed in Section 2.3. Colored Lockdown often offers significant reduction of tasks' WCET. This would likely create even more slack in each partition. Finally, WCET of each application is estimated, and the schedulability of partitions and I/O transactions is checked. If the schedulability check fails, go back to step 3. If this loop has no solution, more/faster chips are needed.

Before presenting the comprehensive example of Section 4.0 about SCE methodology, Section 3.2 gives some insights on how to perform WCET estimation for applications running on the virtual single core chips.

3.2 WCET Estimation

We have mentioned earlier that the constant WCET assumption represents a key feature for schedulability analysis as well as system integration, testing, and certification. That is, the WCET estimation for each task must hold regardless of future software changes in any of the cores; otherwise, WCET estimation for every task needs to be repeated whenever any software component in any core is modified.

Fully Experimental Approach: Because of the isolation provided by resource partitioning described in Section 3.1, the schedulability analysis will be exactly the same as in a single core system. The challenge is how to find bounds on each task's WCET that will remain valid when software is changed in different cores. Under Single-Core Equivalence (SCE), it can be done experimentally in two steps.

1. Create an environment that maximizes inter-core interference, now bounded since SCE enforces resource partitioning. As such, before we estimate a task's WCET in a given core, we replace all other application cores' applications with memory intensive synthetic tasks.
2. In each core i, we then run each task by itself alone to estimate its WCET as in single core chips.

The challenge here is to ensure the creation of the maximally inter-core interfering environment as described in Step 1. Although this is conceptually simple, the implementation can be difficult and time consuming. It is best done as a part of the validation process near the end of timing analysis and verification.

Model Based WCET Estimation: Under Single-Core Equivalence, cores can be studied in isolation (i.e., when all the other cores are powered off), analytically accounting for the bounded inter-core interference.

DRAM transactions are generated as a result of last-level cache misses. Let S_{line} be the number of bytes of a single cache line that is transferred during each transaction. This parameter is architecture-specific and is provided in the specifications.

Let L_{max} be the maximum delay on a DRAM transaction suffered by the core under analysis. For worst-case delay analysis purposes, L_{max} is a key parameter to be derived. This can be done either experimentally by using specifically engineered memory benchmarks or by using DRAM analysis techniques as in [7].

If an experimental approach is used, as detailed in [8], it is possible to measure the DRAM memory transfer bandwidth BW_{min} when: a) each memory transaction has a data dependency with the previous one (latency experiment), and b) subsequent memory requests access different DRAM rows. From BW_{min}, L_{max} can be derived since $BW_{min} = \frac{S_{line}}{L_{max}}$ [8]. Finally, we use the term *Stall* to indicate the MemGuard regulation-induced task delay. This stall term directly depends on BW_{min} (as well as L_{max}), and it can be conservatively computed as:

$$Stall = \frac{m\mu S_{line}}{BW_{min}} \qquad (1)$$

Note that in Equation 1, m is the number of active cores and μ is the maximum number of residual cache misses that is obtained as part of colored lockdown process (see Section 2.3). The inflated WCET time of each task, named C_{sce}, includes now the effect of DRAM bandwidth partitioning and is computed as: C_{sce} = *Stall* + C, where C is a task's *"solo"* WCET obtained from the progressive lockdown curve given the desired amount of locked last level cache.

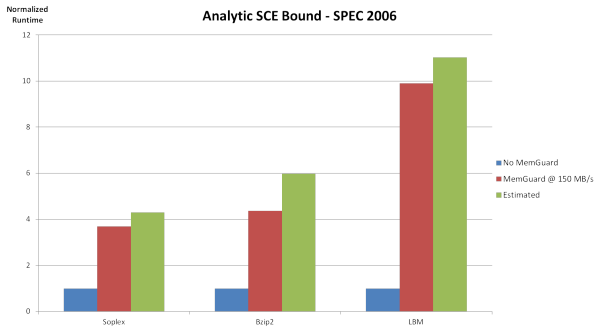


Figure 10: Model based WCET estimation on Freescale P4080

Figure 10 depicts a set of experiments. In each of the three bar plots, the blue bar is a task's WCET when it runs in *"solo"* configuration as described in Section 2.3. The red bar is the measured inflated WCET using the experimental method described at the beginning of this section; finally, the green bar is the analytically derived worst-case execution time C_{sce} by using the described stall formula. As confirmed by run experiments, the stall formula provides a conservative computation of a task's inflated WCET.

The stall formula also gives us a useful insight on how DRAM bandwidth allocation impacts a task's memory latency (its stall time). First, it is proportional to a) the number of active cores m; b) the maximum number of residual cache misses μ, and c) the size of the cache line S_{line}. Additionally, the stall time is inversely proportional to the aggregated DRAM memory bandwidth BW_{min}.

4.0 Experimental Validation

In this section, we present some of the results obtained on a commercial multicore platform when the SCE techniques are deployed and the schedulability analysis is carried out as described in [8].

We have performed an integrated implementation of Colored Lockdown, MemGuard and PALLOC on a Linux kernel. For our experiments we use a commercial multicore platform that provides the necessary hardware support to deploy the discussed techniques. Specifically, we have used a Freescale P4080 platform. The P4080 features m = 8 PowerPC cores with 2 levels of private cache, 2 MB of shared L3 (last-level cache), and 4 GB of DRAM. Each core operates at 1.5 GHz, while the minimum (guaranteed) DRAM bandwidth has been calculated and validated through benchmarking to be at 1.2 GB/s. We consider a peak bandwidth of 2.5 GB/s and a MemGuard budget replenishment period P = 1 ms. Thereby, under the current configuration, the value of the remaining parameters necessary for the WCET analysis are the following: S_{line} = 64 bytes; $L_{max} = 4.96 \times 10^{-8}$ s.

In order to perform lockdown of cache lines, we use the `dcbtls` instruction, while the `dcblc` instruction allows to selectively unlock memory lines from the selected level of cache. In this evaluation, we perform cache management of the shared L3 cache, and keep the lower cache levels disabled. Due to its large size, the L3 allows more flexibility in the allocation strategy. Unfortunately, in the P4080 platform this level of cache is particularly slow with respect to DRAM transactions when main memory is used at full bandwidth. This characteristic is evident from the benchmarks, which only experience approximately 2x performance improvement when full cache allocation is performed. This means that significant performance improvements can be further obtained by managing all the cache levels; however, SCE technology focuses on enforcing performance isolation while additional optimizations are left for future work.

In our integrated implementation, MemGuard directly monitors the DRAM activity in order to make access control decisions at the level of single cores. This is done on the selected platform by relying on the on-chip Event Processing Unit (EPU). This unit is able to internally collect and process Nexus debug messages [12] generated by the DRAM controller. Moreover, the unit can be configured to increment different hardware counters based on the ID of the core that has originated each DRAM transaction. Finally, each counter is programmed to generate an interrupt when the number of collected events

reaches a set threshold. In our evaluation, time samples have been obtained using the core's internal timestamp counters in order to have cycle-accurate measurements. On the selected platform, this can be done using the instructions that provide access to the time base register: `mftbu` and `mftb`.

Benchmark Selection and Profiling: in order to validate the SCE technology and the analytic derivation of MemGuard regulation-induced task delay as summarized[2] in Section 3.2, we have used the San Diego Vision Benchmarks Suite (SD-VBS). The suite includes a number of applications that implement image processing algorithms and it is a good example of memory-intensive workload. In addition, since the suite includes motion tracking, object localization and image feature detection algorithms, it represents a realistic example of data-centric applications deployed on modern avionic and automotive systems for real-time synthetic vision.

Benchmark	Profile Pages	Input Res.	PeakVM	Exec. Ratio
disparity	173	128x96	7736	2.29
localization	80	128x96	2988	1.61
mser	115	128x96	3304	2.11
tracking	217	128x96	3468	1.88
multi-ncut	87	33x44	2996	1.19
sift	930	128x96	6528	2.04
texture	404	352x288	4616	2.25

Table 1: Characterization of SD-VBS benchmarks

For each of these benchmarks, we have performed profiling using the technique described in [9]. Table 1 reports a summary of their characteristics, such as: number of memory pages in the produced profile ("Profile Pages"); resolution of input images in pixels ("Input Res."); peak virtual memory footprint during execution expressed in KB ("PeakVM"); ratio between runtime with 0 allocated pages and full L3 assignment ("Exec. Ratio"). Since we were able to observe similar performance trends across all the considered benchmarks, next section shows the detailed curves for one specific benchmark: the tracking one. This benchmark extracts motion information from the input image-set, which involves feature extraction and feature movement detection. Thus, this application is significant not just for avionic systems, but also for robotic vision, autonomous vehicles and surveillance. The considered benchmark implements the Kanade Lucas Tomasi (KLT) tracking algorithm.

Progressive Lockdown Curve: as reported in Table 1, the complete profile for the tracking benchmark is comprised of 217 memory pages, for a total of 868 KB of memory. Its progressive lockdown curve can be obtained by experimentally estimating the benchmark WCET when an increasing number of profile pages are allocated in cache. Since in the final system the leftover cache space may be assigned to different tasks, once the desired pages are prefetched and locked, the rest of the L3 cache is made unusable for allocation by locking data that do not belong to the task under analysis.

[2] See [8] for more details.

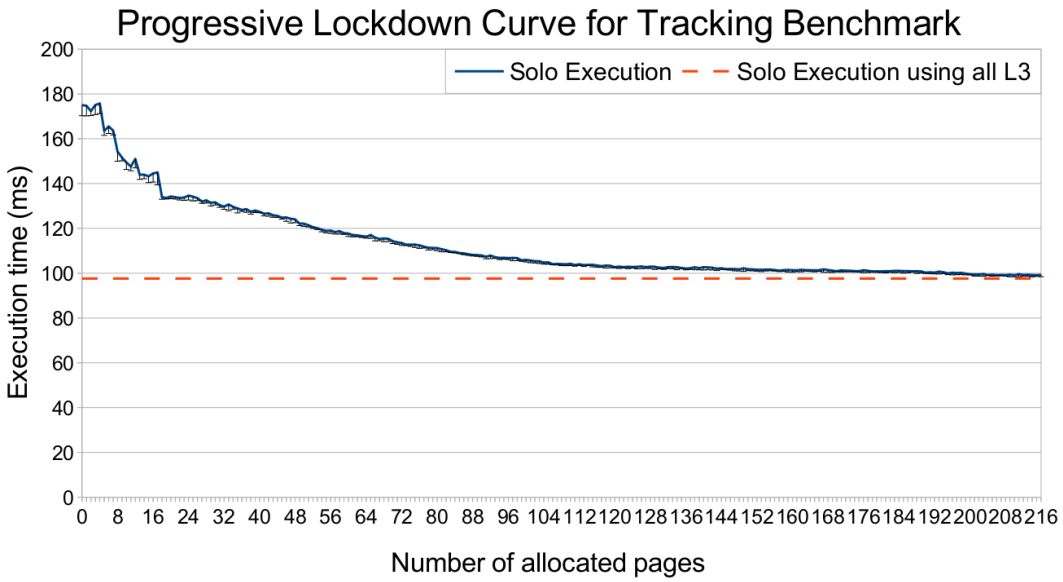


Figure 11: Progressive lockdown curve for tracking benchmark.

Figure 11 depicts the resulting progressive lockdown curve and compares it to the case when all L3 cache is left unmanaged and the benchmark is able to potentially allocate over L3's entire size. The continuous line represents the experimentally estimated WCET among all the collected samples, while the negative error bars report the difference between WCET and best-case execution time. Three main aspects emerge from the plot: a) by allocating about half of the most frequently accessed profile pages, it is possible to consistently reduce by 75% the WCET of the considered benchmark; b) increasing the amount of allocated pages quickly reduces the fluctuation of the measured execution time; and c) by allocating only a subset of critical pages, the benchmark exhibits performance that are comparable to the case when the entire L3 cache is available to the application.

C_{sce} Estimation: at each time sample of the progressive lockdown curve, the maximum number of residual DRAM transactions is also estimated. It corresponds to the parameter μ of Section 3.2 and it can be experimentally estimated or derived from profile-time data. In this experiment, EPU counters are used to estimate μ for each benchmark execution. Once the task is characterized by its progressive lockdown curve, the SCE execution time C_{sce} can be analytically derived.

In Figure 12, the continuous line at the top of the graph represents the obtained value of C_{sce} for each sample in the progressive lockdown curve. The original progressive lockdown curve (see Figure 11) is also included at the bottom of the plot as a continuous line. Finally, the dotted line represents the measured WCET when MemGuard is activated, even memory bandwidth assignment is enforced, and memory-intensive benchmarks are deployed on other cores. For each data-point, the estimated WCET and best-case execution time are reported. In this plot, three main features can be observed.

First, when MemGuard is activated and memory interference from other cores is generated, some noise appears in the measurements. This noise results from different components such as: interleaving of memory transactions from different cores on the bus, and OS overhead (in terms of DRAM transactions and timing) introduced by its routines[3]. In the experimental setup of Figure 12, only a portion of OS noise was reduced by allocating MemGuard periodic routines into L1 cache. As part of future work, we plan to perform an extensive analysis to identify the set of critical routines/data structures of the OS that need to be retained in cache to reduce OS overhead.

[3] In this experiment, OS overhead is particularly visible since L1 to L3 caches are not available for allocation.

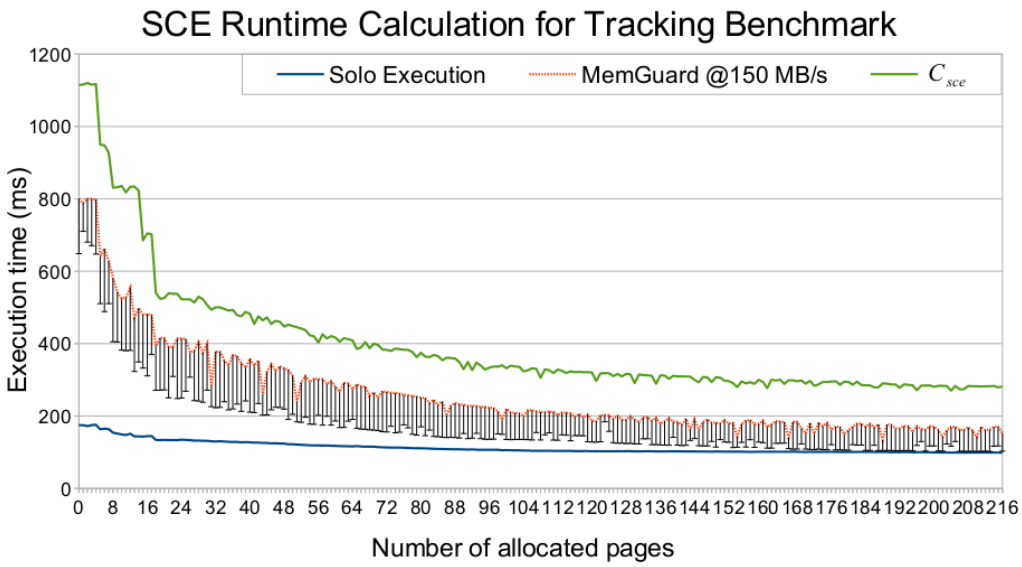


Figure 12: C_{sce} calculation and experimental MemGuard runtime for tracking

Second, despite the noise, it can be observed that by combining MemGuard, Colored Lockdown and PALLOC it becomes possible to enforce strict resource allocation and prevent inter-core interference with a reasonable loss in performance. In fact, note that after about 140 pages have been allocated, we experimentally observe a WCET that is very close to what observed in isolation. On the other hand, when not enough critical pages are allocated in cache, a significant slowdown is experienced. This effect is expected since P4080 features eight cores and an eight times slower DRAM subsystem is exported by SCE.

Benchmark	**Solo**	**MemGuard**	**C_{sce}**
disparity	171.1	339.4	598.7
localization	63.3	95.2	158.5
mser	15.1	17.3	22.5
tracking	108.1	212.9	335.9
multi-ncut	670.1	703.6	761.7
sift	471.3	640.2	967.5
texture	440.1	893.5	1465.6

Table 2: Experimental WCET and C_{sce} values with fixed cache allocation.

Third, it is important to note that the analytically derived C_{sce} always upper-bounds the experimentally measured WCET. Within the limits of an experimental setting, these measurements validate the analytic derivation of MemGuard regulation-induced task delay and SCE response-time analysis discussed in [8]. As previously mentioned, similar results have been obtained for all the benchmarks of the SD-VBS suite and the results are summarized in Table 2. In these experiments, cache allocation is fixed at half the number of profile pages and the table reports WCET in isolation ("Solo"); WCET with MemGuard while memory-intensive tasks are active on different cores ("MemGuard"); and calculated value of C_{sce}. All the times are expressed in milliseconds. The experiments show some degree of pessimism for the analytically derived C_{sce}. This is not surprising since at least two main sources of pessimism exist: (A) for the bound to be conservative, a worst-case memory access pattern is considered when generating the worst-case regulation-induced stall. However, rarely a real benchmark experiences such a worst-case pattern since non-memory instructions and memory accesses are not clustered, but rather interleaved; (B) OS noise increases the number of DRAM transactions that the task is being accounted for. The latter effect can be alleviated by performing aforementioned OS-level optimizations.

5.0 SCE Example

In this section, we present an example that uses a simplified workload to show how the SCE methodology described in Section 3.1 can be applied. We assume that steps 1 and 2 have been performed, in order to skip the details about hardware selection, task-to-partition and partition-to-core assignments. After the profiling for Task 1 is completed, Figure 13 shows graphically how the lockdown curve can be used to select the desired runtime in isolation for a given task. In this example, Task 1 has a period of 36 time units and its execution time is 13 time units when no pages are locked. This same curve needs to be derived for all the tasks in the system.

Next, a cache assignment for the three most frequently accessed memory pages is performed for this task, as depicted in Figure 14. This determines a reduction in the execution time of the task from 13 to 7 time units when it is executed in isolation. However, as shown in Figure 15a, once the interfering cores are turned on, unregulated contention on main memory is generated. As a result, memory accesses performed by Task 1 to all those locations that were not allocated in cache can cause a fluctuation of the execution time for the task under analysis. This execution variance can be severe and represents a major source of pessimism when estimating the worst-case execution time of tasks running on multicore.

Then, MemGuard is used to enforce a predictable regulation of core accesses to main memory. Specifically, we enforce an even memory bandwidth partitioning across the cores of the system (see Section 3.1) to obtain a safe bound for task worst-case execution times. Since MemGuard performs an even distribution of the available bandwidth, each core will see a slower (but predictable) memory subsystem. The inflated worst-case execution time (C_{sce}) for each task can be computed as described in Section 3.2. In our example, Task 1 will have an inflated worst-case execution time of 8 time units, as depicted in Figure 15b.

This step completes the procedure to derive the final parameters for a given task. In our example, Task 1 will have a period of 36 time units, with a WCET of 8 time units and 3 memory pages allocated in last level cache.

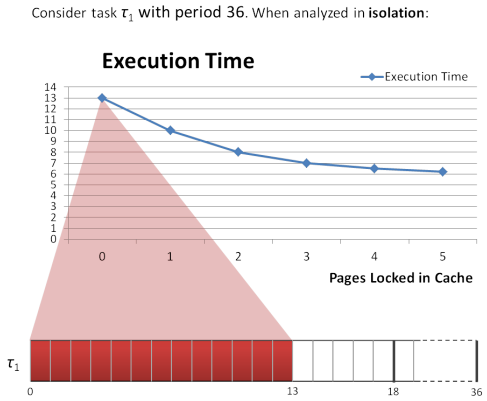


Figure 13: Lockdown curve for Task 1

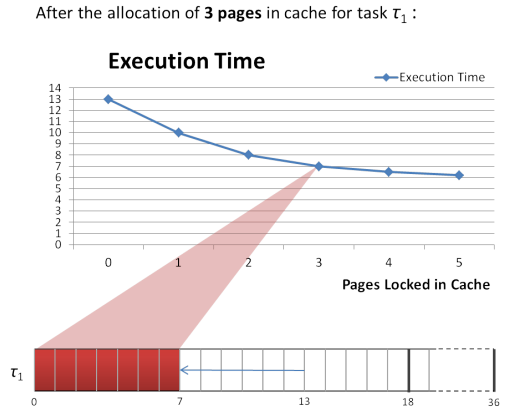


Figure 14: Task execution time after cache assignment

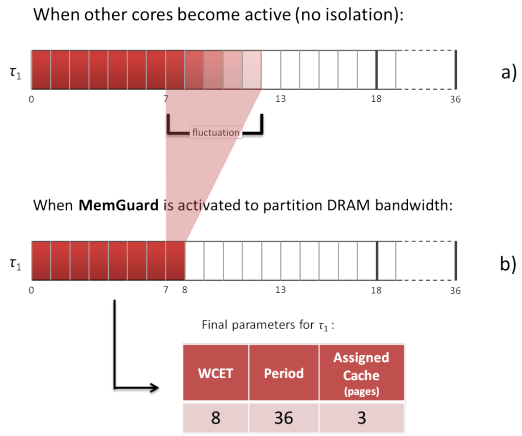


Figure 15: Using MemGuard to reduce execution time fluctuation

A complete view of the workload deployed on one of the CPUs of the system under analysis is presented in Figure 16a. Specifically, two IMA partitions (Partition 1 and Partition 2) are active on CPU 1, with a period of 18 and 36 time units respectively, and a reservation of 6 and 12 time units respectively. Inside Partition 1, two tasks are running: Task 3 with period 18 and worst-case execution time 3; and Task 4 with period 36 and worst-case execution time 3. Similarly, Task 1 (analyzed in Figures 13-15) and Task 2 are running inside Partition 2. Such tasks have periods 36 and 72, respectively, and experience a worst-case execution time of 8 and 3, respectively. Inside each IMA partition, tasks are scheduled rate-monotonically and at activation time of each partition, prefetch and lock (Colored Lockdown) of the allocated cache pages are performed for all tasks running inside the partition. The prefetch and lock operations are highlighted in the figure using a striped pattern.

Figure 16b shows the top-level scheduling of IMA partitions on CPU 1 and CPU 2 in the considered system. Once tasks are assigned to partitions and partitions are assigned to CPUs (see Section 3.1), the schedule of IMA partitions can be done offline (cyclic executive). Since each partition is synchronized with one or more I/O devices, each instance of a partition will be activated at a constant offset from the beginning of its period. Moreover, since prefetch and lock of allocated memory pages is performed at the beginning of each partition instance, IMA partitions execute non-preemptively on the assigned CPU.

The final schedule of IMA partitions is determined using the methodology described in Section 2.4. In this step, I/O requirements for system partitions are considered and a table of I/O transactions performed on the I/O-Core is derived. Figure 17 shows a possible schedule for these transactions, assuming that both input and output of data from/to peripherals can be completed at most within one unit of time. As depicted by the figure, precedence constraints need to be honored by each partition and its related I/O peripherals: data transfers can be performed on the I/O-core only between the end (beginning) of the physical input (output) of data at the device and the activation (completion) of the next (previous) partition instance.

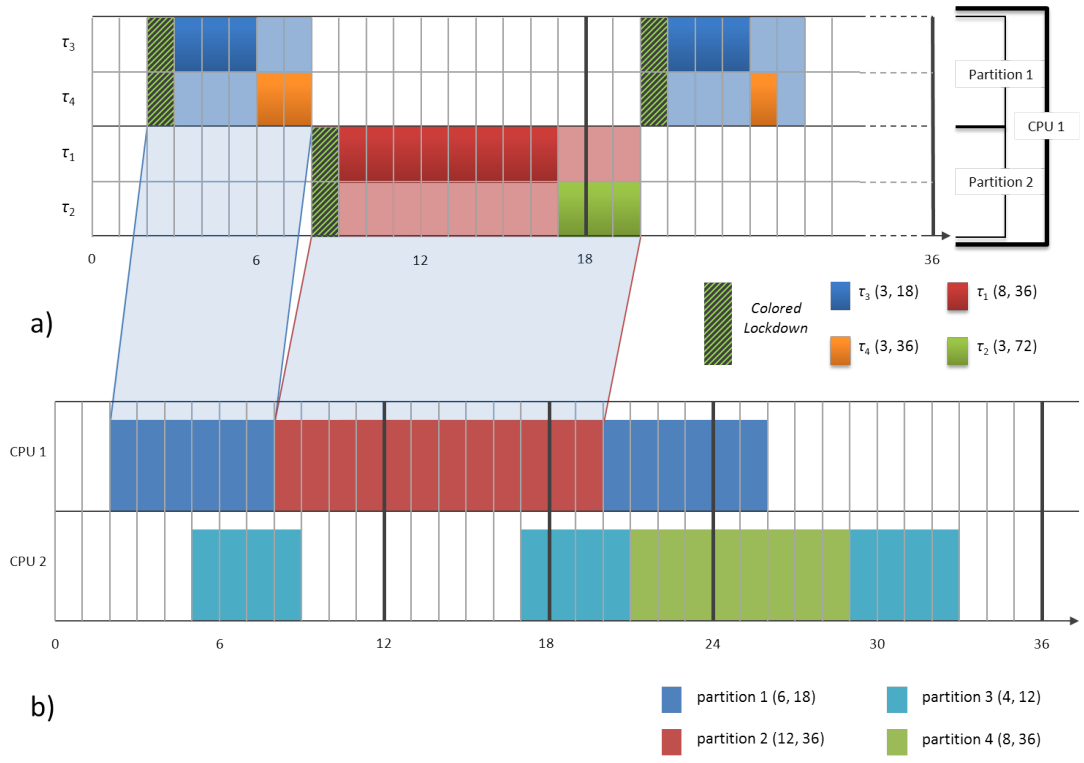


Figure 16: Overview of the workload in applicative cores (Core 1 and Core 2) with IMA partition view

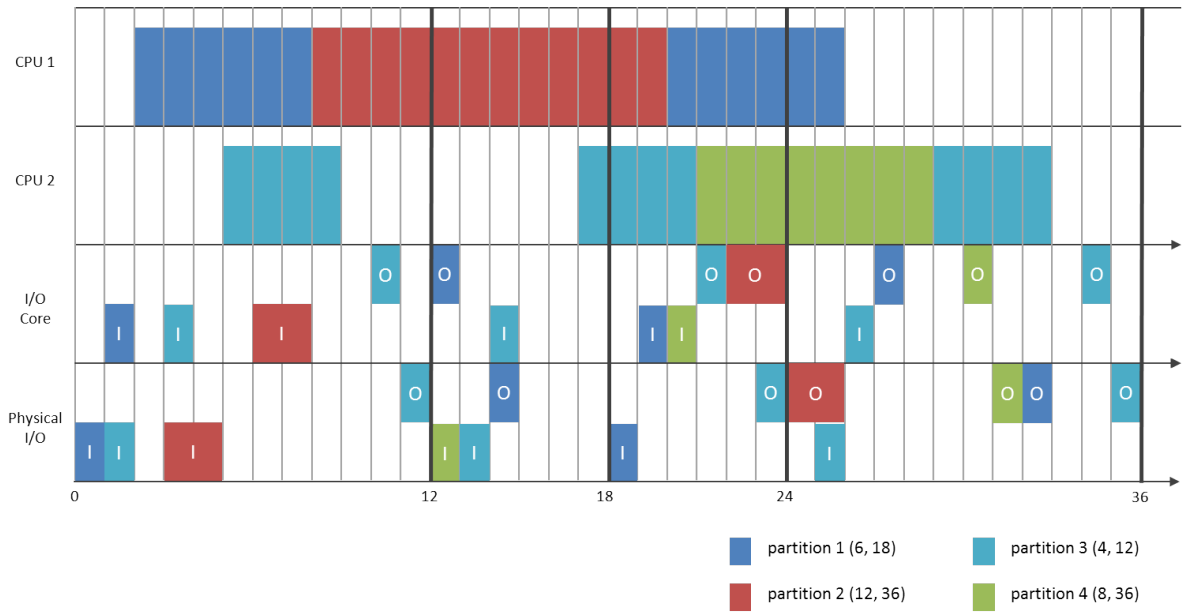


Figure 17: System view with IMA partitions and serialized I/O transactions

5.1 Summary and Conclusion

The increasing demand for computational power in safety-critical systems is pushing embedded industry to an inevitable migration to multicore systems. Unfortunately, the fundamental assumption that the WCET of tasks can be analyzed in isolation and treated as a constant for the purpose of schedulability analysis does not hold in COTS multicore systems. Proposed Single Core Equivalence (SCE) framework performs a strict allocation of shared resources to CPUs, it can be implemented at OS-level on COTS platforms, and it preserves the constant worst case execution time assumption; however, WCET(m) is now function of the maximum number of active cores. Finally, SCE technology allows for modular verification and certification.

References

[1] H. Yun, R. Mancuso, Z. P. Wu, R. Pellizzoni. "PALLOC: DRAM Bank-Aware Memory Allocator for Performance Isolation on Multicore Platform ", in IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014.

[2] *ARINC Specification 651: Design Guidance for Integrated Modular Avionics*. ARINC report. Airlines Electronic Engineering Committee (AEEC) and Aeronautical Radio Inc, Nov. 1991.

[3] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. "MemGuard: Memory Bandwidth Reservation System for Efficient Performance Isolation in Multi-core Platforms", in IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013.

[4] J.-E. Kim, M.-K. Yoon, R. Bradford, and L. Sha, “Integrated Modular Avionics (IMA) Partition Scheduling with Conflict-Free I/O for Multicore Avionics Systems”, in IEEE Computer Software and Applications Conference, July 2014.

[5] J. Rushby. Partitioning in avionics architectures: Requirements, mechanisms, and assurance. NASA Langley Technical Report, Mar. 1999.

[6] Lui Sha, Chang-Gun Lee: Real-time virtual machines for avionics software migration. IJES 2(3/4): 156-165 (2006).

[7] H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, R. Rajkumar. “Bounding Memory Interference Delay in COTS-based Multi-Core Systems”, in IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014.

[8] R. Mancuso, R. Pellizzoni, M. Caccamo, L. Sha, H. Yun. “Response-Time Analysis for Single Core Equivalence Framework”, Dept. of Computer Science, University of Illinois at Urbana-Champaign, Urbana, USA, Technical Report. http://hdl.handle.net/2142/55570, Oct. 28, 2014.

[9] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, R. Pellizzoni. “Real-Time Cache Management Framework for Multi-core Architectures”, in IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013.

[10] Freescale P4080 Platform – Technical Reference Manual. Online resource available at: http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=P4080

[11] FAA Position Paper on Multi-Core Processors, CAST-32 (Rev 0) , Online resource available at: http://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_papers/media/cast-32.pdf. Accessed Oct. 27, 2014.

[12] IEEE-ISTO Nexus 5001 Port Standard Specification. Online resource available at: http://www.nexus5001.org/standard. Accessed Oct. 30, 2014.