Response-Time Analysis for Single Core Equivalence Framework

Renato Mancuso*, Rodolfo Pellizzoni†, Marco Caccamo*, Lui Sha*, Heechul Yun‡
*University of Illinois at Urbana-Champaign, USA, {rmancus2, mcaccamo, lrs}@illinois.edu
†University of Waterloo, Canada, rpellizz@uwaterloo.ca
‡University of Kansas, USA, heechul.yun@ku.edu

Abstract

Multi-core platforms represent the answer of the industry to the increasing demand for computational capabilities. From a real-time perspective, however, the inherent sharing of resources, such as memory subsystem and I/O channels, creates inter-core timing interference among critical tasks and applications deployed on different cores. As a result, modular per-core certification cannot be performed, meaning that: (1) current industrial engineering processes cannot be reused; (2) software developed and certified for single-core chips cannot be deployed on multi-core platforms as is.

In this work, we propose the Single Core Equivalence (SCE) technology: a framework of OS-level techniques designed for commercial (COTS) architectures that exports a set of equivalent single-core virtual machines from a multi-core platform. This allows per-core schedulability results to be calculated in isolation and to hold when all the cores of the system are composed together. Thus, SCE allows each core of a multi-core chip to be considered as a conventional single-core chip, ultimately enabling industries to reuse existing software, schedulability analysis methodologies and engineering processes.

I. Introduction

Multi-core chips are mainstream products. A multi-core chip allows the concurrent accesses of shared resources including DRAM, system bus, last level cache, and I/O channels. However very often one or more of these resources represents a bottleneck and creates severe inter-core interference. From a real-time perspective, if a given shared resource represents a bottleneck, the measured worst-case execution time of a task in a core can oscillate significantly when that resource is heavily used without deterministic regulation. This is because severe congestion can be experienced leading to unpredictable execution delays.

Avionic and automotive industry has a large base of certified software developed for single-core chips. However, due to inter-core interference, analysis results derived in single-core chips cannot be reused in the deployment on multi-core platforms. In this work, we propose the Single Core Equivalence technology (SCE technology). This represents a framework of OS-level techniques that: (A) can be implemented on commercial off-the-shelf (COTS) architectures and (B) exports a set of equivalent single-core virtual machines from a given multi-core platform, by means that schedulability results derived on the exported machines considered in isolation will hold when all the cores in the system are composed together.

We envision that SCE can allow industry to use each core of a multi-core chip as if it was a core in a conventional single-core chip. Such a property would allow to reuse not only software but also the development, schedulability analysis and certification process as is. As a result, SCE can provide both performance benefits, since multi-core platforms are fully exploited, and a reduction of production costs. While our technology has been originally developed in the context of multi-core avionics systems, it can be used in other industry domains such as automotive, medical devices and plant control where timing predictability is important.

In the past decades, single-core chips have become increasingly faster, so that several software applications in older and slower chips could be migrated to a single, faster and shared physical processor. In order to use faster chips while reusing consolidated components, Integrated Modular Avionics (IMA) was created. In fact, it provided the abstraction of multiple physical processors on a single chip, so that each IMA partition could be treated as a separate physical processor from an analytical standpoint. Thus, IMA facilitated the consolidation of applications running on multiple slow chips into a faster one, allowing engineers to port, develop, verify and certify applications at the granularity of the IMA partition. IMA as is cannot be reused in multi-core systems due to the discussed inter-core interference. Instead we propose SCE to enforce the same two main requirements as IMA on multi-core platforms, which are:

1) Software faults in one partition must not be able to corrupt the code or data in other partitions;
2) Software running in one partition is not allowed to slow down the execution of tasks in other partitions that are logically independent.

While Requirement 1 can be obtained using classic protection mechanisms for software components, in this work, we focus on how Requirement 2 can be met. Specifically, the goal of our SCE is to enforce new partitioning and protection mechanisms applied to the common resources shared by multiple cores. This way, each core can be viewed as a single-core chip inherently meeting the aforementioned Requirement 2. One of the consequences of the strict resource partitioning enforced by SCE is that classic IMA architecture can be reused on top of exported single-core machines for applications that share the same core.

In this paper, we describe how different partitioning mechanisms addressing the main sources of interference can be combined together to form the SCE framework. We specifically focus on the memory hierarchy and describe how system-wise schedulability analysis can be derived. In particular, we describe the following aspects:

Shared last-level cache allocation: we deterministically allocate portions of the last level cache to tasks, to simulate the cache architecture of single-core chips. The challenge is that the limited cache space represents a constraint. Thus, we rely on profiling for workload characterization in order to use the cache space in a predictable, yet efficient way.

Memory bandwidth partitioning: in a multi-core chip, when memory-intensive applications run on multiple cores, the generated memory traffic can saturate the DRAM controller. This scenario leads to severe inter-core interference, so that a predictable way to partition the available memory bandwidth must be employed.

Private DRAM bank assignment: in a multi-core architecture, different cores can concurrently access the DRAM memory. To avoid contention at the DRAM bank level, we enforce a memory allocation strategy that assigns disjoint sets of banks to tasks running on different cores. This is also known as assignment of private DRAM banks.

Note that in order to perform a complete end-to-end schedulability analysis, it is also fundamental to serialize access to shared I/O devices across cores. As a part of our SCE framework, we intend to delegate a specific core (I/O core) to serialize I/O transaction, using a technique similar to what described in [1]. However, since this work mainly focuses on memory hierarchy management, we do not describe how the I/O serialization is performed in our SCE and leave this as a future discussion point.

Overall, the coordinated partitioning and protection of shared resources in a multi-core chip is challenging. However the benefits of the SCE approach are significant since it allows engineers to port, develop, verify and certify applications core by core, and eventually partition by partition, as if running on a single-core chip. Without SCE, the combinatorial inter-core interference can easily prevent system integration and lead to undesired complexities at the level of liability management. In fact, from a business prospective, software applications are developed concurrently by different subcontractor. Thereby it is unacceptable for certified and delivered software to experience deadlines misses dependently on the characteristics of the workload deployed on other cores. It is important to underline, however, that even though SCE offers a solution to these problems, large improvement margins can be achieved by optimizing the coordinated shared resource allocation. Optimization considerations are currently left for future work.

The rest of the paper is organized as follows. Section II provides an overview of the related work. Next, a brief description of the main components of the SCE technology is provided in Section III, while the general methodology for system deployment is described in Section IV. The requirements and assumptions under which SCE can be applied are provided in Section V, while Section VII contains the derivation of the SCE WCET bound. Next, an experimental validation of our analysis is described in Section VIII. Finally, the paper concludes with Section IX.

II. Related Work

The problem of inter-core interference in multi-core architectures is well known and has been largely studied in literature. As a result, several different ways of approaching the new challenges introduced by multi-core platforms have been proposed.

Important results have been achieved to understand how different class of scheduling algorithms behave on multi-processors [2]. Moreover, new scheduling strategies that are optimal on such platforms have been developed, such as RUN [3], Pfair [4, 5], LLREF [6], LRE-TL [7]. The commonly adopted model in multi-core scheduling theory assumes that WCET of tasks does not vary according to what is being scheduled on other cores at the time of their execution. In this sense, since the SCE layer creates an abstraction where the constant-WCET assumption is guaranteed, it can allow optimal multi-core scheduling strategies to be deployed on real systems.

Static analysis tools that leverage abstract interpretation [8, 9, 10] and symbolic execution [11, 12, 13, 14] have been developed. Using such techniques it is possible to analyze a program from binary reconstructing execution paths and extracting information about memory access patterns. These tools can achieve excellent results on single-core systems relying only on architectural modeling and knowledge about task under analysis. In order to scale to multi-cores, however, this approach requires extending the analysis to all those software components that can run in parallel to achieve tight bounds. We envision, instead, that these tools can rely on the deterministic properties exported by the proposed SCE without major changes in the adopted analysis approach. As we show in Section VII, we are able to provide a response-time analysis for a tasks under SCE, even though we expect that it can be largely improved by combining our framework with static analysis techniques.

On multi-core, static analysis for caches proposed by [15] has been extended to shared caches [16, 17, 18]. Similarly, shared memory buses have been analyzed in [19, 20]. A complete attempt of system-wise static analysis which includes shared cache and bus in a multi-core platform has been proposed in [21], on top of which a unified WCET framework has been formulated [22]. Overall, a interesting plethora of work has been developed in this direction. However, these work need to have strong assumptions about: (A) the behavior of some architectural components (e.g. TDMA-based buses or pure LRU-based caches); and (B) the presence of a fixed workload in the system, meaning that system-wise analysis must be reiterated if changes to individual software components are made. We believe these techniques could be reused in conjunction with the high-level guarantees exported by SCE, allowing a relaxation of their assumptions and ultimately becoming more suitable for practical use.

A different approach is to design multi-core and many-core architecture that provide better guarantees on the worst-case execution time of critical software components. The precision timed (PRET) architecture [23, 24] embed runtime control and deadline enforcement at the level of processor instruction set while proposing a set of hardware modifications to achieve good performance without sacrificing predictability. Similarly, predictability is enhanced in the MERASA project [25] by reducing inter-core interference at the hardware level. Finally, the P-SOCRATES [26] project aims at creating an hardware design for predictable many-core architectures. Although hardware that has been designed to be predictable can provide stronger guarantees, the goal of our SCE approach is to be implementable on existing COTS platforms by enforcing isolation in software with the aid of existing hardware support. This characteristic sets our SCE apart from hardware-based approaches, whose feasibility has been confirmed with an implementation on commodity hardware.

III. Background about SCE Technologies

A first version of SCE was produced as a result of our collaboration with two avionics industries, namely Rockwell Collins and Lockheed Martin. Even though single components were covered as a part of our previous work, the SCE framework was not in the public domain. Thus, in this section, we provide the required context and background about each of the integrated resource management techniques comprising SCE: Colored Lockdown for shared cache management; MemGuard for DRAM bandwidth reservation; PALLOC for DRAM bank partitioning.

A. Colored Lockdown - Cache Assignment

Multi-core architectures often feature several levels of CPU cache. Private caches are relatively easier to analyze, have smaller size and are less configurable via software than shared levels of cache. Thus, we primarily consider the problem of efficiently managing shared caches, even though some of the concepts can be easily reused to manage lower (private) cache levels. Specifically, we leverage a mechanism, Colored Lockdown, that is able to solve inter-core interference on allocated memory areas while providing a good tradeoff between efficiency and flexibility. Colored Lockdown involves two main stages: an offline profiling stage; and an online cache allocation stage.

1) Profiling: During the offline stage, each real-time task is analyzed using a memory profiler [27]. When the task runs in the profiling environment, memory accesses are traced and per-page access statistics are maintained. In this way, it is possible to: (A) rank pages of the task's addressing space by access frequency; and (B) produces a task profile which identifies frequently accessed (hot) memory pages by their relative position in the addressing space. Since the final profile can be used online to drive the cache allocation phase.

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

2) 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.

3) Lockdown: Real time applications are dominated by periodic execution flows. This characteristic allows for an optimized use of last level cache by locking hot pages 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.

The combination of the mentioned techniques takes the name of Colored Lockdown [27]. In order to export identical single-core machines, we consider *even* last level cache allocation across cores: i.e. each core is allowed to allocate a maximum number of pages that is equal to the total size of the cache (in pages) divided by the number of cores in the system.

For each task, by following its profile, it is possible to derive a curve that plots the observed WCET of a task as a function of the number of hot memory pages allocated in cache. We call this curve *progressive lockdown curve*. It relates cache assignment with resulting WCET, so that it is possible to estimate the task WCET given the desired cache assignment and vice-versa. Thereby, once the cache assignment has been performed, two parameters are derived: (1) a value of WCET C for the task running in isolation (single-core scenario); and (2) a residual maximum number of cache misses μ, corresponding to accesses to all those profile pages not allocated in cache. As we show in Section VII, the value of WCET obtained at this step has to be inflated to account for the fraction of memory bandwidth allocated to each core.

Since the pages are ranked by access frequency, the progressive lockdown curve will exhibit a convex shape and constrained convex optimization methods can be used to obtain the optimal cache allocation for a group of tasks. However, from an SCE perspective, the specific cache assignment strategy can be left to the system designer.

Such curve is experimentally derived by using a configuration where: (A) portions of cache of even size are assigned to cores; (B) all but one core are kept idle; (C) private DRAM banks are assigned to each core (see Section III-C).

B. MemGuard - Memory Bandwidth Partitioning

Similarly to shared caches, DRAM memory is one of main sources of inter-core interference. In fact, in multi-core chips, applications running on different cores can concurrently access main memory. This determines unregulated contention at the level of the memory controller that can introduce significant delay to the memory requests from critical real-time applications. To achieve better isolation, we developed an OS level memory bandwidth management system, called MemGuard [28]. 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 more predictable and analyzable, as we discuss it in Section VII.

MemGuard uses a series of per-core regulators that are responsible for monitoring and enforcing the memory bandwidth allocation. This can be done by relying on hardware-specific performance monitoring capabilities. Specifically, each regulator monitors the amount of DRAM transactions performed by each core (or alternatively, the number of last-level cache misses). By considering the worst-case latency L_{max} for a single memory request to be serviced, it is possible to derive a worst-case (guaranteed) bandwidth at which the memory subsystem can operate.

MemGuard operates as follows: it is configured to enforce the bandwidth assignment at a given period P. The amount of transactions K_q that a given core will be allowed to perform during P, under even bandwidth distribution, will simply be: $K_q = \frac{P}{mL_{max}}$, where m is the number of cores in the system. At the beginning of each period, MemGuard configures the hardware performance counters to trigger an event after any core exceeds the K_q threshold of completed DRAM memory requests. To

enforce the strict bandwidth assignment, upon reception of a budget-exhausted event, MemGuard idles the associated core. Any idled core resumes its activity at beginning of the next replenishment period. The length of the regulation period P is a system-wide parameter and should be much smaller than the minimal application task period. In our current implementation, P is one millisecond matching also the OS scheduler tick interval.

The key insight is that by restricting maximal memory bandwidth usage of each core, it is ensured that each core always has its reserved memory bandwidth reservation within the regulation period P. Since non memory-intensive applications typically use less than their reservation, MemGuard also offers different ways to share reserved but unused memory bandwidth across cores to achieve significant average-case performance improvements. However, in this paper, we analyze the behavior of tasks with strict bandwidth assignment only, while additional details on bandwidth reclaiming and sharing mechanisms can be found in [28].

C. PALLOC - DRAM Bank Partitioning

The DRAM structure is organized into ranks, banks, rows and columns [29]. Whenever a given row is accessed in a bank, subsequent accesses on the same row (row-hits) can be serviced with a small latency. Conversely, if a subsequent access requires data in a different row (row-miss), a significant increase in the latency is introduced. Different banks of the same DRAM chip can satisfy requests in parallel. The mapping between physical addresses and banks is hardware-specific and can be determined as described in [30, 31].

In a multi-core scenario, several cores can potentially access the same DRAM bank, leading to a suboptimal utilization of the inherently parallel structure of the DRAM banks. Specifically, in the worst-case, requests from different cores to the same bank can result in continuous row-misses that would not be observed if different banks were assigned to each core. This also means that a different bank-level behavior can be experienced by a given task when it is analyzed in isolation compared to the case when the workload on different cores is composed together.

Unfortunately, current OSes do not control how physical memory pages are mapped to DRAM banks. In order to overcome the discussed shortcomings, SCE uses a DRAM bank-aware OS-level memory allocator, named PALLOC [30], which allows system designers to assign specific DRAM banks to cores (or applications). In the context of SCE, PALLOC is used to assign disjoint sets of DRAM banks (private banks) to applications running on different cores. This way, tasks running in parallel do not collide on DRAM banks and do not suffer inter-core conflicts at this level, as long as there is 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. System designers can create multiple partitions and specify desired DRAM banks for each partition through the CGROUP interface. When a process in a CGROUP requires physical memory, PALLOC allocates only pages from the specified banks. A detailed discussion on how PALLOC works can be found in [30].

In summary, MemGuard improves performance isolation on multi-core platforms by using efficient DRAM bandwidth reservation mechanisms, while PALLOC achieves similar results by partitioning DRAM banks. Together, they provide strong isolation when accessing shared DRAM memory.

IV. SCE Methodology for System Deployment

In this Section, we brielfy describe the main steps that are needed to deploy our SCE framework on top of multi-core platforms. Once SCE is in place and the key parameters of the underlying platform have been identified, schedulability techniques similar to what used in single-core chips can be employed on each exported equivalent machine, as we describe in Section VII.

The first step to obtain a SCE-compliant system is to select a commercial multi-core chip that features hardware primitives needed by SCE components. Specifically, the requirements are:

1) An MMU unit, in order to allow page-level coloring to be performed;
2) Atomic instructions to prefetch and lock individual lines in last-level cache, or a per-cache-way allocation mechanism;
3) Knowledge (either from technical reference manuals or from benchmarking) about the mapping of physical addresses to DRAM ranks, banks and columns, so that PALLOC can be deployed;
4) Extensive performance monitoring capabilities, allowing per-core DRAM transactions to be observed in software, ultimately allowing MemGuard to operate.

The second step comprises workload characterization. In this step, all but one core of the system are powered off and tasks are observed in isolation. For each task, we extract the complete memory profile and progressive lockdown curve, together with the maximum number of residual last-level cache misses for each data-point in the curve.

Next, we propose to evenly distribute shared resources to create the pool of exported equivalent single-core machines. The even distribution of resources is not a strict requirement as the techniques comprising SCE can also be used to perform uneven allocation. Nonetheless, enforcing even partitioning ensures that identical single-core machines are exported. This in turns simplifies migration of software components across cores without the need of reiterating a costly task schedulability analysis.

The even resource assignment is enforced on the DRAM subsystem by: (A) using PALLOC to allocate the same number of private banks to each core; and (B) by configuring MemGuard to allocate an equal fraction of the guaranteed bandwidth to each processor. At this point, tasks deployed on different cores will observe a slower but dedicated DRAM subsystem.

In the next step, the assignment of tasks to cores is performed. The assignment is application-specific, so that the exact task-to-core assignment strategy can be left to the system designer. If IMA partitions are used on top of SCE, partition-to-core assignment is performed in this step.

TABLE I
SUMMARY OF PARAMETERS FOR SCE RESPONSE-TIME ANALYSIS

Parameter	Meaning
m	Number of cores in the system
P	MemGuard budget replenishment period
C	WCET for the considered task in isolation
μ	Number of residual misses after last-level cache assignment using Colored Lockdown
L_{size}	The size in bytes of a single cache line
L_{min}	Minimum amount of time for a single request to the memory subsystem
L_{max}	Maximum amount of time for a single request to the memory subsystem
K_q	Maximum number of memory requests allowed by MemGuard in a single period P

After tasks/partitions have been allocated to cores, last-level cache assignment is performed. For this purpose, it is possible to rely on the previously derived progressive lockdown curve: either the desired task runtime is fixed and its cache requirements are derived accordingly; or a given amount of cache is provided and the corresponding runtime is determined. Either way, it must always hold that the cumulative amount of last-level cache (LLC) allocated to all the tasks on the same core (or partition) is less or equal than $\frac{\text{LLC cache size}}{\text{number of cores}}$.

Finally, the schedulability for tasks assigned on a given core can be checked using the analysis described in Section VII. If the set of tasks on each core is determined to be schedulable, then the whole system is schedulable. Conversely, if any of the tasksets is found to be not schedulable, the previous cache allocation step can be reiterated, rearranging cache resources inside the considered core. Alternatively, the initial task-to-core assignment can be reconsidered and schedulability issues can be solved by migrating group of tasks across cores.

V. SYSTEM MODEL AND ASSUMPTIONS

In this section, we provide details about the considered system model as well as the assumptions according to which the analysis is carried out.

Table I summarizes the list of parameters that will be used throughout the paper to derive the SCE bounds. Here, m represents the number of cores of the system. The parameter P, instead, represents the budget replenishment period of MemGuard such that the memory access budget for each core will be restored every P time units. In order to derive the response time analysis when MemGuard is operational and all the cores of the system are active, we assume that the worst-case execution time of the task running in isolation is C. This can be done by shutting down all the cores but the one under analysis and reusing established WCET estimation techniques for single-core systems. The value of C should also be derived once the last-level cache assignment for the task has been determined using Colored Lockdown. Under this scenario, the maximum value of residual cache misses μ can be obtained for the worst case scenario.

Following parameters model the behavior of the key components of the underlying DRAM memory subsystem. First, L_{size} represents the size in bytes of a single cache line. As such, L_{size} bytes will be requested and transferred to last-level cache for every suffered cache miss. Prefetchers and speculative execution units are assumed to be disabled. Each core is allowed to have more than one outstanding memory request, but we assume that the bus arbiter implements a round-robin scheduling policy. Once a memory request is issued, we assume that the DRAM access time for a single transaction is bounded between the best-case access time L_{min} and the worst-case access-time L_{max}. Given that the timing behavior of the DRAM subsystem is captured by the described parameters, K_q represents the number of memory accesses that a core is allowed to perform within each MemGuard period P. Specifically, K_q can be calculated as $K_q = \frac{P}{mL_{max}}$.

Since private banks to each core are assigned using PALLOC. This implies that tasks do not suffer intra-bank interference and thus that the time for the execution of a single memory request does not change when the other cores of the system are activated.

VI. WORST-CASE REGULATION STALL

In this section, we identify the worst case memory access pattern and derive a bound for the maximum amount of stall that a task can suffer during any regulation period P.

A. Regulation-induced stall

Memory regulation (MemGuard) never allows one core to generate more than a given number of DRAM memory accesses (K_q) inside each regulation period (P). As long as the core performs K_q memory requests or less, it is not stalled inside a given regulation period P. Conversely, when a core tries to perform the $(K_q + 1)^{th}$ DRAM access, it is stalled until the beginning of the next period. We define *stall* as the sum of all the intervals of time during which the core under analysis has a pending memory transaction but is not being served by the memory subsystem. This can happen either because the memory subsystem is serving other cores (contention) or because it has been stalled by MemGuard (regulation).

The first lemma shows that under even bandwidth assignment across cores, the amount of stall induced on the task is upper-bounded by the regulation stall, assuming that the scheduling policy of the DRAM transactions on the bus follows a round-robin scheme.

Lemma 1 *Consider a system with even bandwidth regulation, where K_q is the number of DRAM accesses of each core during a regulation period of length P. If the policy for scheduling DRAM transactions at the bus arbiter and at the DRAM controller is round-robin, then each core is guaranteed to always perform up to K_q memory transactions within a regulation period P.*

Proof: This lemma can be easily proven by contradiction. Recall that $K_q = \frac{P}{mL_{max}}$. Suppose that the core under analysis takes $X > P$ to perform a burst of K_q memory accesses that was ready at the beginning of the period. Since the bus (memory-controller) arbitration policy is round-robin, X can be rewritten in terms of performed transactions:

$$X = K_q L_{max} + K_{oth} L_{max}$$

Where K_{oth} denotes the aggregate number of DRAM transactions performed by all the other cores during P. It follows that:

$$\begin{aligned} K_q L_{max} + K_{oth} L_{max} &> P \\ \Rightarrow (K_q + K_{oth}) L_{max} &> P \\ \Rightarrow K_q + K_{oth} &> \frac{P}{L_{max}} \\ \Rightarrow K_q + K_{oth} &> mK_q \end{aligned}$$

which contradicts the assumption that each core can perform a burst of K_q transactions per period P without being regulated. ■

Next, we find an upper bound on the amount of stall that can be observed in a single regulation period P for the core under analysis. In order to do this, we rely on the fact that at most K_q memory transactions can be performed by each core during a regulation period P (Lemma 1).

Lemma 2 *For any single regulation period P, $P - K_q L_{min}$ is an upper bound on the amount of stall suffered by the core under analysis.*

Proof: In order to prove this lemma, we consider two cases:

Case 1 (Regulation) The task under analysis needs to perform K_q DRAM memory accesses and no transactions are requested by other cores. In this case, the worst-case stall is suffered when the task under analysis accesses the DRAM at its peak bandwidth. Thus the K_q accesses will be satisfied in $K_q L_{min}$ time units, and the resulting regulation-induced stall will be: $P - K_q L_{min}$.

Note that this case also includes the case in which the memory budget is already exhausted when the task under analysis is scheduled.

Case 2 (Contention) The task under analysis needs to perform K_q DRAM memory accesses together with all the other cores. Since the bus (memory controller) scheduling policy is round-robin, in the worst-case the task will stall for $K_q(m-1)L_{max}$. Thus, it holds that:

$$\begin{aligned} (m-1)K_q L_{max} &= mK_q L_{max} - K_q L_{max} \\ &= P - K_q L_{max} \\ &\leq P - K_q L_{min} \end{aligned}$$

■

The following lemma identifies the worst case DRAM memory access pattern for a given task in the system, within an arbitrary time window. The timing analysis will be then derived according to this scenario. The key insight behind the lemma is that if all memory accesses concentrate on one region, it represents burst arrival case and could possibly take several regulation periods to complete. While the core is stalled due to regulation, no progress on the task can be made. Thus, a clustered memory access pattern represents the worst case in terms of how much delay is introduced on task's completion.

Lemma 3 *The memory access pattern that causes the worst-case regulation-induced stall on a given task is when all the accesses within the considered task are clustered, starting when the budget is exhausted.*

Proof: From Lemma 2 it follows that a task cannot suffer more stall than $P - K_p L_{min}$ (regulation-induced stall) per each regulation period P. Thus, the worst-case memory access pattern corresponds to the pattern that maximizes this stall. It follows that the worst-case memory access pattern corresponds to the case where all the μ memory accesses are clustered at the beginning of the task. If the number of memory accesses μ is not a multiple of K_q, we assume the worst-case contention

from all the other cores in the last period in which the leftover transactions occur. ■

B. Stall term

By using Lemma 3 and Lemma 2, we can derive the worst-case regulation-induced stall for a given task that performs μ DRAM memory accesses.

Theorem 1 *The worst-case regulation induced stall for a task that performs μ DRAM memory-accesses in a system with evenly distributed bandwidth regulation can be obtained as:*

$$stall = \left\lceil \frac{\mu}{K_q} \right\rceil (P - K_q L_{min}) + (m-1)(\mu - (\left\lceil \frac{\mu}{K_q} \right\rceil - 1)K_q) L_{max} \tag{1}$$

Proof: The theorem follows from Lemma 3 and Lemma 2. The formula can be divided into two terms. The first term captures the regulation-induced delay in all the intervals where regulation occurs. The considered task is regulated every K_q transactions, and it can be released right after the MemGuard budget has been exhausted. Thus, it will be regulated throughout its execution exactly $\lceil \frac{\mu}{K_q} \rceil$ times, every time suffering a worst-case stall of $P - K_q L_{min}$ (Lemma 3). The remaining $\mu - (\lceil \frac{\mu}{K_q} \rceil - 1)K_q$ transactions will not trigger the regulation mechanism and thus will be only subject to contention. Since the bus scheduling policy is round-robin, they will be stalled in the worst-case by $(m-1)L_{max}$ each. ■

For sake of simplicity, we show the derivation of the response time analysis assuming the case $\mu \% K_q = 0$. In case the assumption does not hold, an upper-bound on the number of requests can be considered: $\hat{\mu} = \lceil \frac{\mu}{K_q} \rceil K_q$. It is easy to see that $\hat{\mu} \% K_q = 0$ and that $\hat{\mu} \geq \mu$. Under this assumption, we can rewrite the stall term as follows.

$$\begin{aligned} stall &= \frac{\hat{\mu}}{K_q}(P - K_q L_{min}) + (m-1)K_q L_{max} \\ &= \frac{\hat{\mu}}{K_q}(P - \frac{P L_{min}}{m L_{max}}) + (m-1)K_q L_{max} \\ &= \frac{\hat{\mu} m L_{max}}{P}(P - \frac{P L_{min}}{m L_{max}}) + (m-1)K_q L_{max} \\ &= \hat{\mu} L_{max}(m - \frac{L_{min}}{L_{max}}) + (m-1)K_q L_{max} \end{aligned}$$

Since $\hat{\mu} L_{max}$ is the maximum time spent accessing memory, it is easy to see that under the assumption $L_{max} = L_{min}$, the stall term represents the case in which each task sees a memory channel that is m times slower than the isolation case. Moreover, the constant value $(m-1)K_q L_{max}$ represents a blocking term deriving from the fact that the bandwidth assignment is enforced by MemGuard at a finite granularity P. Ideally if the mechanism could be implemented such that $P \to 0 \Rightarrow K_q \to 0$, this effect would disappear.

VII. SCE Response Time Analysis

Assume that the scheduling policy is based on tasks' fixed priority assignment upon a partitioned multi-core system; this is a common practice used in industrial applications.

The response time analysis for a group of tasks assigned to a specific core in a multi-core system with bandwidth regulation and even bandwidth assignment can be carried out as shown below.

First, the following iterative formula can be used to compute the response time analysis for regulated tasks which include regulation-induced stall as calculated in Equation 1.

$$R_i^{(k+1)} = C_i + \sum_{\tau_j \in hp(i)} \left\lceil \frac{R_i^{(k)}}{T_j} \right\rceil C_j + stall(\mu_i^{(k)}) \tag{2}$$

Where $\mu^{(k)}$ is defined as follows:

$$\mu_i^{(k)} = \sum_{\tau_j \in hep(i)} \left\lceil \frac{R_i^{(k)}}{T_j} \right\rceil \hat{\mu}_j \tag{3}$$

It is important to note that the stall term in this formulation depends only the term $\mu_i^{(k)}$ defined above, because the amount of regulation-induced stall exclusively depends on the number of required DRAM memory accesses (last-level cache misses).

Theorem 2 *The response time of a periodic task in a system under bandwidth regulation and even bandwidth assignment can be calculated as:*

$$R_i^{(k+1)} = C_{sce_i} + \sum_{\tau_j \in hp(i)} \left\lceil \frac{R_i^{(k)}}{T_j} \right\rceil C_{sce_j} + K_q L_{max}(m-1) \tag{4}$$

Where C_{sce} represents the SCE execution time and is defined as:

$$C_{sce} = C + \hat{\mu} L_{max}(m - \frac{L_{min}}{L_{max}}) \tag{5}$$

Proof: Equation 4 follows from Equation 2 after expanding the stall term. Specifically:

$$\begin{aligned}
R_i^{(k+1)} &= C_i + \sum_{\tau_j \in hp(i)} \left\lceil \frac{R_i^{(k)}}{T_j} \right\rceil C_j + stall(\mu_i^{(k)}) \\
&= C_i + \sum_{\tau_j \in hp(i)} \left\lceil \frac{R_i^{(k)}}{T_j} \right\rceil C_j + K_q L_{max}(m-1) \\
&+ \mu_i^{(k)} L_{max}(m - \frac{L_{min}}{L_{max}}) \\
&= C_i + \sum_{\tau_j \in hp(i)} \left\lceil \frac{R_i^{(k)}}{T_j} \right\rceil C_j + K_q L_{max}(m-1) \\
&+ \big(\sum_{\tau_j \in hep(i)} \left\lceil \frac{R_i^{(k)}}{T_j} \right\rceil \hat{\mu}_j \big) L_{max}(m - \frac{L_{min}}{L_{max}}) \\
&= C_i + \sum_{\tau_j \in hp(i)} \left\lceil \frac{R_i^{(k)}}{T_j} \right\rceil C_j + K_q L_{max}(m-1) \\
&+ \big(\left\lceil \frac{R_i^{(k)}}{T_i} \right\rceil \hat{\mu}_i + \sum_{\tau_j \in hp(i)} \left\lceil \frac{R_i^{(k)}}{T_j} \right\rceil \hat{\mu}_j \big) L_{max}(m - \frac{L_{min}}{L_{max}})
\end{aligned}$$

Since $\left\lceil \frac{R_i^{(k)}}{T_i} \right\rceil = 1$ when checking schedulability for task τ_i, the following holds:

$$\begin{aligned}
&= C_i + \sum_{\tau_j \in hp(i)} \left\lceil \frac{R_i^{(k)}}{T_j} \right\rceil C_j + K_q L_{max}(m-1) \\
&+ \big(\hat{\mu}_i + \sum_{\tau_j \in hp(i)} \left\lceil \frac{R_i^{(k)}}{T_j} \right\rceil \hat{\mu}_j \big) L_{max}(m - \frac{L_{min}}{L_{max}}) \\
&= C_i + \hat{\mu}_i L_{max}(m - \frac{L_{min}}{L_{max}}) \\
&+ \sum_{\tau_j \in hp(i)} \left\lceil \frac{R_i^{(k)}}{T_j} \right\rceil \big(C_j + \hat{\mu}_j L_{max}(m - \frac{L_{min}}{L_{max}}) \big) \\
&+ K_q L_{max}(m-1)
\end{aligned}$$

Which can be finally rewritten as:

$$= C_{sce_i} + \sum_{\tau_j \in hp(i)} \left\lceil \frac{R_i^{(k)}}{T_j} \right\rceil C_{sce_j} + K_q L_{max}(m-1)$$

■

In order to perform SCE analysis, the parameters L_{min} and L_{max} need to be estimated. This can be done experimentally using synthetic benchmarks. Specifically, L_{min} can be derived by ensuring that only the core under analysis is active, that all his requests are hitting in the same DRAM row, and that there are no dependencies between subsequent requests. Conversely, L_{max} can be found by analyzing a benchmark in isolation where: (A) each memory transaction depends on the previous one (latency experiment); (B) subsequent memory requests access different DRAM rows. It can be noted that the parameters L_{min} and L_{max} can be rewritten in terms of bandwidth as follows:

1) $L_{min} = \frac{L_{size}}{BW_{max}}$
2) $L_{max} = \frac{L_{size}}{BW_{min}}$

Where BW_{min} is the DRAM guaranteed bandwidth; BW_{max} is the maximum requested DRAM bandwidth (as measured in isolation, without regulation and while all the other cores are off); and where L_{size} is the size of a cache line (i.e. the amount of bytes transferred from DRAM at each last-level cache miss). Under this formulation, C_{sce} can be rewritten as:

$$C_{sce} = C + \hat{\mu} L_{size} (\frac{m}{BW_{min}} - \frac{1}{BW_{max}}) \tag{6}$$

This formula highlights that the term BW_{max} can be overapproximated, or considered as $BW_{max} \to \infty$, in case a precise estimation cannot be performed. Thus, the formula can be rewritten without the dependency from BW_{max} as follows:

$$C_{sce} = C + \frac{\hat{\mu} L_{size} m}{BW_{min}} \tag{7}$$

VIII. Experimental Validation

In this section, we present some of the results obtained on a commercial multi-core platform when the SCE techniques are deployed and the analysis is carried out as described in Section VII. Specifically, we show the study of how the workload for a specific task can be characterized and how the SCE WCET can be derived.

A. Methodology

We have performed an integrated implementation of Colored Lockdown, MemGuard and PALLOC on a Linux kernel. For our experiments we use a commercial multi-core 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: $L_{size} = 64$ bytes; $L_{min} = 2.38 \times 10^{-8} s$; $L_{max} = 4.96 \times 10^{-8} s$; $K_q = 2520$. In addition, the selected platform meets the hardware requirements for the main SCE components described in Section IV.

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. Thanks 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 the latter is used at full bandwidth. This characteristic is evident from the benchmarks, which only experience about a 2x performance improvement when full cache allocation is performed. This means that by managing all the cache levels, significant performance improvements can be obtained without violating SCE invariants. However, we are mostly interested in enforcing performance isolation while possible optimizations are left for future work.

In our integrated implementation, MemGuard directly monitors the DRAM activity in order to take access control decisions at the granularity of a single core. This can be 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 generated at the DRAM controller. Moreover, the unit can be configured to increment different hardware counter according to the ID of the core that has originated each DRAM transaction. Finally, each counter can be programmed to generate an interrupt when a threshold in the number of collected events is reached.

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 through the instructions that provide access to the time base register: `mftbu` and `mftb`.

B. Benchmark Selection and Profiling

In order to validate our implementation and to validate the SCE theoretical model described in Section VII, we have used the San Diego Vision Benchmarks Suite (SD-VBS) [32]. The suite includes a number of applications that implement image processing algorithms and are good examples 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 aircrafts for real-time synthetic vision.

For each of these benchmarks, we have performed profiling using the technique described in [27]. Table II 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 in across execution expressed in KB ("PeakVM"); ratio between runtime with 0 allocated pages and full L3 assignment ("Exec. Ratio").

Since we were able to observe the same performance trend across all the considered benchmarks, we show in the next section the detailed curves for one benchmark in particular: tracking. This benchmark extracts motion information from the input image-set, which involves feature extraction and feature movement detection. Thus, besides the avionic domain, this application is relevant for robotic vision, autonomous vehicles and surveillance. As mentioned in [32], the considered benchmark implements the Kanade Lucas Tomasi (KLT) tracking algorithm [33].

TABLE II
CHARACTERIZATION OF SD-VBS BENCHMARKS

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

C. Progressive Lockdown Curve

As reported in Table II, the complete profile for the tracking benchmark is comprised of 217 memory pages, for a total of 868 KB of memory. As mentioned in Section III-A3, the correspondent progressive lockdown curve can be obtained by measuring the exhibited runtime 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, we make the rest of the L3 cache unusable for the task by locking arbitrary portions of system memory.

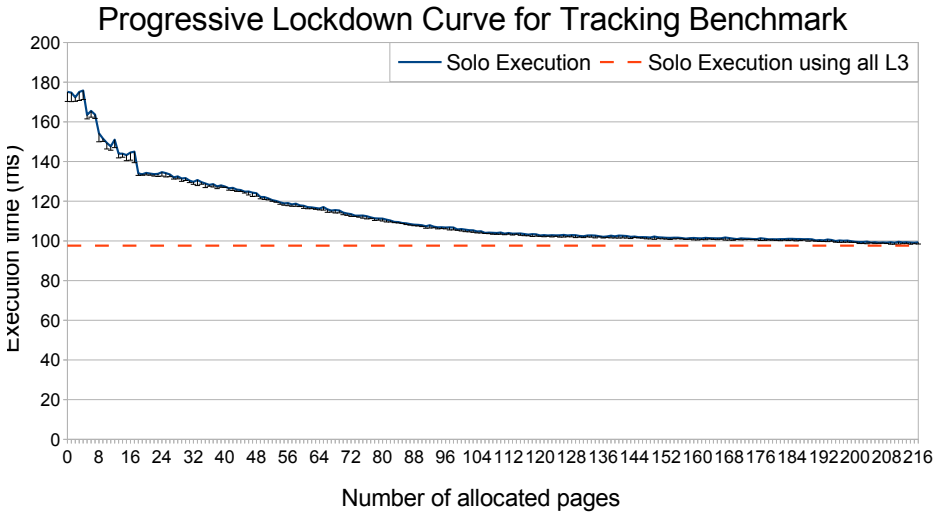

Fig. 1. Progressive lockdown curve for tracking benchmark.

Figure 1 depicts the resulting curve and compares the resulting execution time to the case in which all the L3 is left unmanaged and the benchmark is able to potentially allocate over L3's entire size. The continuous line represents the observed maximum among all the collected samples, while the negative error bars report the difference between maximum and minimum observed runtime. Three main aspects emerge from the plot: (A) by allocating about half of the most frequently accesses profile pages, it is possible to consistently reduce by a 75% the runtime of the considered benchmark; (B) increasing the amount of allocated pages quickly reduces the fluctuation of the observed execution time; and (C) by allocating only a subset of critical pages, the benchmark exhibits performance that are comparable to the case where the entire L3 is available for the application.

D. C_{sce} Estimation

Albeit not shown in the plots, to each time sample in the progressive lockdown curve, we associate the number of leftover generated DRAM transactions. These correspond to the parameter μ in the analysis and can be experimentally captured or derived from profile-time data. In this case we follow an experimental approach and simply rely on EPU counters to produce the correspondent μ for each execution. Once the task has been characterized with a progressive lockdown curve, the SCE execution time C_{sce} can be derived according to Equation 5.

In Figure 2, the continuous line at the top of the graph represents the obtained value of C_{sce} for each step in the progressive lockdown curve. As a reference, the original progressive lockdown curve (see Figure 1) is reported as a continuous line at the bottom of the plot. Finally, the dotted line represents the measured execution time when MemGuard is activated, even bandwidth assignment is enforced and memory-intensive benchmarks are deployed on other cores. For each data-point, the maximum and the observed fluctuation is reported. In this plot, three main features can be observed.

First, when MemGuard is activated and memory interference from other cores is generated, a component of noise appears in the measurements. This noise results from different components, such as: interleaving of memory transactions from different cores on the bus; and overhead, in terms of DRAM transactions and timing, introduced by the OS routines (which becomes particularly visible since L1 to L3 caches are not available for allocation). In our current setup, we were able to eliminate a portion of this noise by allocating MemGuard periodic routines into the L1 cache. In addition, as a part of our future work, we

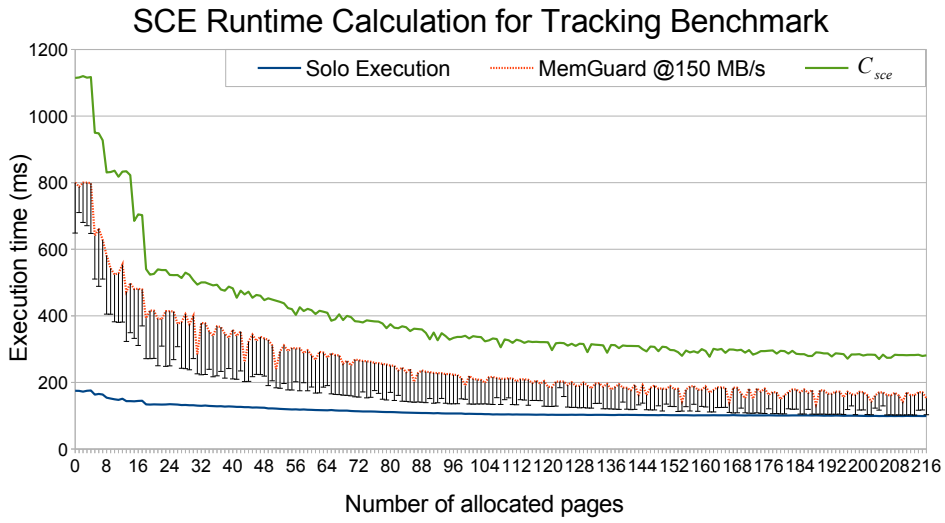

Fig. 2. C_{sce} calculation and experimental MemGuard runtime for tracking.

TABLE III
RUNTIME AND C_{sce} ESTIMATION WITH FIXED CACHE ALLOCATION

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

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 the DRAM noise.

Second, despite the noise, it can be observed that by combining MemGuard, Colored Lockdown and PALLOC it becomes possible to enforce strict resource allocation to prevent inter-core interference with a reasonable loss in performance. In fact, note that after about 140 pages have been allocated, in some of the collected samples, we observe a runtime that is very close to what observed in isolation. On the other hand, when not enough critical pages are allocated in cache, a significant worst-case slowdown is experienced. This effect is expected since an $m = 8$ times slower DRAM subsystem is exported by SCE.

Third, it is easy to note that the estimated C_{sce}, calculated according to the equations in Section VII, always upper-bounds the experimentally produced samples. This property represents a confirmation of the validity of the proposed response-time analysis. As previously mentioned, the same result has been obtained on all the benchmarks of the suite and the results are summarized in Table III. In this table, we have fixed the allocation at half the number of profile pages and report the time for execution in isolation ("Solo"); execution with MemGuard while memory-intensive tasks are active on different cores ("MemGuard"); and calculated value of C_{sce}. All the times are in milliseconds.

In the plot a certain degree of pessimism is visible. This is not surprising since at least two main sources of pessimism can be identified: (A) in order to be conservative, the analysis assumes the memory access pattern that realizes the worst-case regulation-induced stall. However, this does not adhere to real benchmarks where non-memory instructions and memory accesses are not clustered, but rather interleaved; (B) the additional noise in the system 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. Moreover, additional knowledge of task-specific memory access patterns could be derived using static analysis tools and leveraged to lower the current pessimism.

E. Sensitivity to BW_{max} Parameter

Finally, we have mentioned how the simpler Equation 7 can be considered for practical purposes instead of Equation 6, if an exact estimation of the parameter L_{min} (i.e. BW_{max}) cannot be performed. Thereby, in our last plot we show how sensitive is the derived C_{sce} to variations of the discussed parameter. For this purpose, Figure 3 compares the C_{sce} curve reported in Figure 2 with the same calculation performed under the unrealistic assumption that $BW_{max} = 1000$ GB/s (dotted curve). Despite the large gap in the value of this parameter, however, only a 7% increase is visible in the resulting curve.

IX. CONCLUSION

A significant increase in the demand for computational capabilities in real-time and safety-critical platforms has been observed over the past decades. The resulting fast growth in the production of multi-core chips is pushing the industries toward

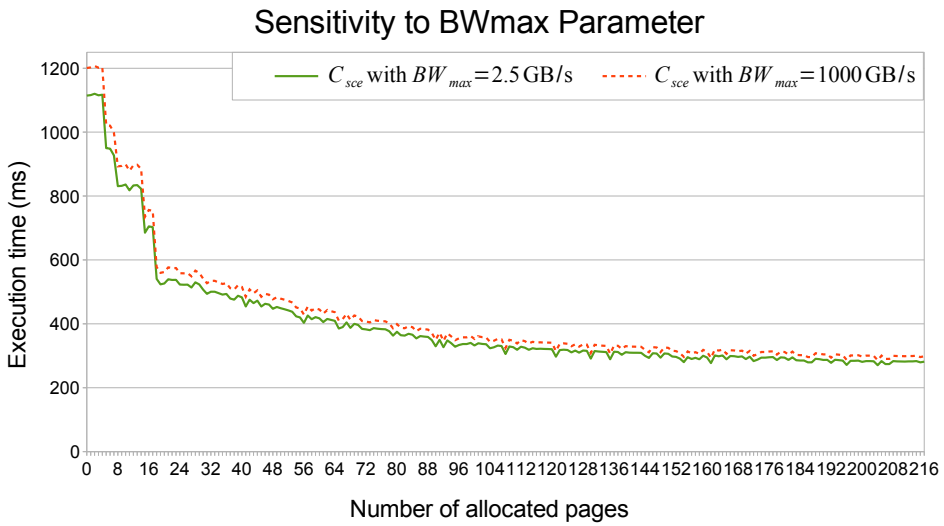

Fig. 3. Sensitivity of C_{sce} to variations in L_{min} (i.e BW_{max}) parameter.

an inevitable migration from single-core systems. Unfortunately, however, real-time schedulability has become significantly harder to analyze because the fundamental assumption that WCET of critical tasks can be derived in isolation and treated as a constant for analysis purposes does not hold anymore in COTS multi-core systems.

In this work, we have described our Single Core Equivalence (SCE): a framework of OS-level techniques that can be deployed on COTS platforms and that restores the constant-WCET assumption. SCE achieves isolation by performing strict allocation of shared resources to CPUs, ultimately allowing modular verification and certification. In this work we have provided a methodology to perform response-time analysis of an SCE-compliant system and validated our theoretical results by implementing SCE on commercial hardware.

ACKNOWLEDGMENT

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.

REFERENCES

[1] 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 *Computer Software and Applications Conference (COMPSAC), 2014 IEEE 38th Annual*, pages 321–331, July 2014.

[2] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms. In *Handbook on Scheduling Algorithms, Methods, and Models*. Chapman Hall/CRC, Boca, 2004.

[3] P. Regnier, G. Lima, E. Massa, G. Levin, and S. Brandt. RUN: Optimal Multiprocessor Real-Time Scheduling via Reduction to Uniprocessor. In *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pages 104–115, Nov 2011.

[4] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress: A notion of fairness in resource allocation. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing*, STOC '93, pages 345–354, New York, NY, USA, 1993. ACM. ISBN 0-89791-591-7.

[5] M. Moir and S. Ramamurthy. Pfair scheduling of fixed and migrating periodic tasks on multiple resources. In *Real-Time Systems Symposium, 1999. Proceedings. The 20th IEEE*, pages 294–303, 1999.

[6] C Hyeonjoong, B. Ravindran, and E.D. Jensen. An optimal real-time scheduling algorithm for multiprocessors. In *Real-Time Systems Symposium, 2006. RTSS '06. 27th IEEE International*, pages 101–110, Dec 2006.

[7] S. Funk. LRE-TL: An optimal multiprocessor algorithm for sporadic task sets with unconstrained deadlines. *Real-Time Syst.*, 46(3):332–359, December 2010. ISSN 0922-6443.

[8] P. Cousot. Abstract interpretation based formal methods and future challenges. In *Informatics - 10 Years Back. 10 Years Ahead.*, pages 138–156, London, UK, UK, 2001. Springer-Verlag. ISBN 3-540-41635-8.

[9] AbsInt. aiT worst-case execution time analyzers, 2014. URL `http://www.absint.com/ait/`.

[10] X. Li, Y. Liang, T. Mitra, and A. Roychoudhury. Chronos: A timing analyzer for embedded software. *Sci. Comput. Program.*, 69(1-3):56–67, December 2007. ISSN 0167-6423.

[11] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976. ISSN 0001-0782.

[12] M. Papadakis and N. Malevris. A symbolic execution tool based on the elimination of infeasible paths. In *Software Engineering Advances (ICSEA), 2010 Fifth International Conference on*, pages 435–440, Aug 2010.

[13] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. *SIGSOFT Softw. Eng. Notes*, 30(5):263–272, September 2005. ISSN 0163-5948.

[14] M. N. Islam. *Introduction to WCET An Analysis Approach with SWEET: SWEET-THE WCET Tool*. LAP Lambert Academic Publishing, Germany, 2012. ISBN 3848437538, 9783848437535.

[15] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. *Real-Time Syst.*, 18(2/3):157–179, May 2000. ISSN 0922-6443.

[16] J. Yan and W. Zhang. WCET analysis for multi-core processors with shared L2 instruction caches. In *Real-Time and Embedded Technology and Applications Symposium, 2008. RTAS '08. IEEE*, pages 80–89, April 2008.

[17] Y. Li, V. Suhendra, Y. Liang, T. Mitra, and A. Roychoudhury. Timing analysis of concurrent programs running on shared cache multi-cores. In *Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE*, pages 57–67, Dec 2009.

[18] D. Hardy, T. Piquet, and I. Puaut. Using bypass to tighten WCET estimates for multi-core processors with shared instruction caches. In *Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE*, pages 68–77, Dec 2009.

[19] J. Rosen, A. Andrei, P. Eles, and Zebo Peng. Bus access optimization for predictable implementation of real-time applications on multiprocessor Systems-on-Chip. In *Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International*, pages 49–60, Dec 2007.

[20] T. Kelter, H. Falk, P. Marwedel, S. Chattopadhyay, and A. Roychoudhury. Bus-aware multicore wcet analysis through tdma offset bounds. In *Real-Time Systems (ECRTS), 2011 23rd Euromicro Conference on*, pages 3–12, July 2011.

[21] S. Chattopadhyay, A. Roychoudhury, and T. Mitra. Modeling shared cache and bus in multi-cores for timing analysis. In *Proceedings of the 13th International Workshop on Software & Compilers for Embedded Systems*, SCOPES '10, pages 6:1–6:10, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0084-1.

[22] S. Chattopadhyay, C.L. Kee, A. Roychoudhury, T. Kelter, P. Marwedel, and H. Falk. A unified wcet analysis framework for multi-core platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2012 IEEE 18th*, pages 99–108, April 2012.

[23] Stephen A. Edwards and Edward A. Lee. The case for the precision timed (PRET) machine. In *Proceedings of the 44th annual Design Automation Conference*, DAC '07, pages 264–265, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-627-1.

[24] D. Bui, E.A. Lee, I. Liu, H. Patel, and J. Reineke. Temporal isolation on multiprocessing architectures. In *Design Automation Conference (DAC)*, pages 274 – 279, June 2011.

[25] T. Ungerer, F. Cazorla, P. Sainrat, G. Bernat, Z. Petrov, C. Rochange, E. Quinones, M. Gerdes, M. Paolieri, J. Wolf, H. Casse, S. Uhrig, I. Guliashvili, M. Houston, F. Kluge, S. Metzlaff, and J. Mische. MERASA: Multicore execution of hard real-time applications supporting analyzability. *IEEE Micro*, 30(5):66–75, 2010. ISSN 0272-1732.

[26] V. Nélis, P. M. Yomsi, L. M. Pinho, J. C. Fonseca, M. Bertogna, E. Quiñones, R. Vargas, and A. Marongiu. The Challenge of Time-Predictability in Modern Many-Core Architectures. In Heiko Falk, editor, *14th International Workshop on Worst-Case Execution Time Analysis*, volume 39 of *OpenAccess Series in Informatics (OASIcs)*, pages 63–72, Dagstuhl, Germany, 2014. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-939897-69-9.

[27] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-time cache management framework for multi-core architectures. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, RTAS'13, pages 45–54, Philadelphia, PA, USA, April 2013. IEEE Computer Society.

[28] 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 *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 55–64, 2013.

[29] B. Jacob, S. Ng, and D. Wang. *Memory systems: cache, DRAM, disk*. Morgan Kaufmann, 2007. ISBN 9780123797513.

[30] H. Yun, R. Mancuso, Z.-P. Wu, and R. Pellizzoni. PALLOC: DRAM Bank-Aware Memory Allocator for Performance Isolation on Multicore Platforms. In *Proceedings of the IEEE Intl. Conference on Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Berlin, Germany, April 2014.

[31] H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. R. Rajkumar. Bounding memory interference delay in cots-based multi-core systems. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, RTAS'14, Berlin, Germany, April 2014. IEEE Computer Society.

[32] S.K. Venkata, I. Ahn, Donghwan Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, and M.B. Taylor. SD-VBS: The San Diego Vision Benchmark Suite. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 55–64, Oct 2009.

[33] J. Shi and C. Tomasi. Good features to track. In *Computer Vision and Pattern Recognition, 1994. Proceedings CVPR '94., 1994 IEEE Computer Society Conference on*, pages 593–600, Jun 1994.