WCET(m) Estimation in Multi-Core Systems using Single Core Equivalence

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 multiple cores of the system run in parallel. Thus, SCE allows each core of a multi-core chip to be considered as a conventional single-core chip, ultimately enabling industry 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. Unfortunately, often one or more of these resources creates a bottleneck causing severe inter-core interference. From a real-time perspective, if a given shared resource represents a bottleneck, the measured worst-case execution time (WCET) of a task in a core can vary 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 industries have a large base of certified software developed for single-core chips. However, due to inter-core interference, schedulability analysis results derived for single-core systems cannot be reused when migrating to multi-core platforms. In this work, we propose the Single Core Equivalence technology (SCE technology). Under SCE, access to shared memory resources is strictly regulated using a set of OS-level techniques. As such, each core in the system is assigned a partition of the memory hierarchy with predictable performance. If m is the number of active cores in a system, SCE dedicates $1/m$ of shared memory resources to each core. Consequently the WCET of a task directly depends on the number of active cores and can be denoted as WCET(m). In this work, we introduce the SCE technology and develop an analytical model of SCE that is used to estimate WCET(m) of a task under analysis given the knowledge of its behavior in isolation.

Currently, FAA and other international certification authorities are concerned about the use of multi-core platforms in safety critical avionics systems due to various inter-core interference channels [1]. We envision that SCE technology can be used by embedded industry to design certifiable multi-core systems with m cores. The use of SCE has two major benefits: a) first, if less than m cores are used in production, the remaining ones can be activated at a later upgrade without system-wide re-certification; b) second, any of the certified m cores can be treated as an independent processor in a conventional single-core chip, meaning that local workload changes require local re-validation. Such property allows reusing consolidated software components, development, schedulability analysis and certification process as is. As a result, SCE can provide both performance benefits, since multi-core platforms can be fully exploited, and a reduction of certification costs.

In this paper, we first describe how the SCE framework and its different partitioning mechanisms can be used to address the main sources of inter-core interference. Next, we detail the derivation of WCET(m) for a real-time task given a knowledge of its behavior in isolation. To the best of our knowledge, this is the first work to propose the idea of modeling the worst-case execution time as a parametric WCET(m) when tasks run on top of a performance isolation framework for COTS multi-core systems. Additionally, the proposed solution is implementable using existing hardware support. Thereby, this work makes the following contributions:

- Describe how techniques used to address contention at different levels of the memory hierarchy can be combined together to enforce strong inter-core performance isolation in modern COTS multi-core chips.
- Derive an analytical model to estimate the WCET(m) of a task running under SCE given the knowledge of its behavior in isolation.
- Show that by relying on the properties of WCET(m) it is possible to reuse, without substantial modifications, existing schedulability analysis techniques, such as response-time analysis.
- Validate proposed SCE technology as well as its analytical model through implementation on real COTS hardware.

Due to space constraints, this paper describes how SCE technology can be used as a means of mitigating the following multi-core interference channels: shared last-level cache, shared DRAM memory bandwidth, and shared DRAM banks. In order to perform a complete end-to-end schedulability analysis, it is also important to serialize access to I/O devices shared across cores. This can be done by delegating a specific core (I/O core) to serialize I/O transactions, as described in [2]. Since the focus of this paper is on memory hierarchy management, we do not discuss how I/O serialization can be performed. In

depth discussion and implementation details on how I/O can be integrated within the proposed SCE framework are provided in [3].

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 problem can easily jeopardize the system integration process and lead to undesired complexities at the level of liability management. Even though SCE offers a solution to these problems, it is worth to notice that large improvement margins can further be achieved by optimizing the coordinated shared resource allocation mechanisms. Optimization considerations are currently out of scope and left as 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 model and assumptions for SCE are reviewed in Section V. Next, in Section VI we establish the necessary theoretical results that are used for the derivation of WCET(m) in Section VII. An experimental validation of our analysis is described in Section VIII and the paper finally 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 classes of scheduling algorithms behave on multiprocessors [4]. Moreover, new scheduling strategies that are optimal on such platforms have been developed, such as RUN [5], Pfair [6, 7], LLREF [8], LRE-TL [9]. 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 complex multi-core scheduling strategies to be deployed on real systems.

Static analysis tools that leverage abstract interpretation [10, 11, 12] and symbolic execution [13, 14, 15] 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 the 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 [16] has been extended to shared caches [17, 18, 19]. Similarly, shared memory buses have been analyzed in [20, 21]. A complete attempt of system-wise static analysis which includes shared cache and bus in a multi-core platform has been proposed in [22], on top of which a unified WCET framework has been formulated [23]. Overall, an interesting plethora of work has been developed in this direction. However, these works need to make 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 architectures that provide better guarantees on the worst-case execution time of critical software components. The precision timed (PRET) architecture [24, 25] embeds 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 [26] by reducing inter-core interference at the hardware level. Finally, the P-SOCRATES [27] is an ongoing project for the development of software techniques to provide soft real-time guarantees on top of commercial many-core platforms. 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 strong 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 avionic industries, namely Rockwell Collins and Lockheed Martin [3]. 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 [28], 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 [28]. When the task runs in the profiling environment, memory accesses are traced and per-page access statistics are maintained. Next, a) pages of the task's addressing space are ranked by access frequency; and b) a profile is produced identifying frequently accessed (hot) pages by their relative position in the addressing space. The final profile can be used online to drive the cache allocation phase. Given the produced profile, two mechanisms are used to provide deterministic guarantees and a fine-grained cache allocation granularity, described below.

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.

For each task, by following its profile, it is possible to derive a curve that plots the 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 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 WCET C obtained at this step is used to derive the value of WCET when m are the active cores in the system: WCET(m).

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. As discussed in Section VIII, the progressive lockdown curve can be either derived by using static analysis tools or experimentally in 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 isolation, we developed an OS level memory bandwidth regulation system, called MemGuard [29]. The goal of MemGuard is to provide bandwidth reservation for each core, so that the stall time each application suffers due to memory bandwidth regulation is predictable and can be analyzed, as discussed 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 active cores in the system. At the beginning of each period, MemGuard configures the hardware performance counters to trigger an event when any of the cores 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 the 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 the maximal memory bandwidth usage for each core, it is ensured that each core can always complete up to K_q memory requests within a 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 [29].

C. PALLOC - DRAM Bank Partitioning

The DRAM structure is organized into ranks, banks, rows and columns [30]. 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 [31, 32]. In a multi-core scenario, several cores can potentially access the same DRAM bank. In this case, the row-miss ratio of a task can increase as multiple cores access the same bank. In addition, reordering of requests at the bank level operated by the memory controller can significantly increase the worst-case per-bank queuing delay for the core under analysis [32].

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 [31], 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 [31].

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 briefly describe the main steps that are needed to deploy our SCE framework on top of multi-core platforms to support m concurrently active cores. The key property is that once SCE is in place, the value of WCET(m) calculated for each task on a given core (see Section VII) will not be affected by workload changes in other cores. This allows reusing consolidated schedulability techniques for single-core chips on each of the m active cores.

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;
2) Atomic instructions to prefetch and lock individual lines in last-level cache, or a per-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 studied 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, simplifying 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 m-times 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 [33] partitions are used on top of SCE, partition-to-core assignment is performed in this step.

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 WCET is fixed and its cache requirements are derived accordingly; or a given amount of cache is provided and the corresponding WCET 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 $\frac{\text{LLC cache size}}{m}$.

Finally, WCET(m) for tasks assigned on a given core can be derived and schedulability can be checked as 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 not to be 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 groups of tasks across cores.

TABLE I
Summary of parameters for SCE response-time analysis

Param.	Interpretation
m	Number of active 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
L_{size}	The size in bytes of a single cache line
L_{min}	Minimum amount of time for a single memory request
L_{max}	Maximum amount of time for a single memory request
K_q	Maximum number of memory requests within each period P

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 estimate the WCET(m) of a task in a system where m represents the number of active cores. 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. The parameter C captures the WCET of the considered task running in isolation 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.

Note that the value of C and μ can be derived by using either static analysis or an experimental approach. Static analysis can be used whenever a micro-architectural model for the considered platform is available [10, 11]. However, for complex architectures, it is often common industrial practice to experimentally derive the value of WCET. The measurement-based approach may determine a WCET that does not correspond to the absolute WCET. However, for COTS platforms where many micro-architectural details are undisclosed this approach is the only viable option for practically dimensioning a complex system at design time. Existing tools that adopt this approach are part of the industry practice toward WCET determination on single-core platforms [34, 35].

The 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, and we assume that the DRAM controller, as well as the bus arbiter, implement a round-robin scheduling policy.

Many modern COTS multi-core chips (e.g. Freescale MPC56xx and MPC57xx chip families) designed for safety-critical operations can be configured to implement a round-robin policy on the main memory controller arbiter. Nonetheless our analysis can also be derived under the assumption of a FIFO policy, leading to more pessimistic results. Due to space constraints, however, we describe our analysis with a round-robin policy, leaving the discussion about the implications of other scheduling policies as a part of our future work.

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}. The latter parameter captures the maximum amount of delay

suffered on a single memory request due to the activity of other cores. In an experimental approach, L_{min} can be derived by ensuring that only the core under analysis is active, that all its 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 has a data dependency with the previous one; b) subsequent memory requests access different DRAM rows. These parameters can also be rewritten in terms of bandwidth: $L_{min} = \frac{L_{size}}{BW_{max}}$ and $L_{max} = \frac{L_{size}}{BW_{min}}$, where BW_{min} is the DRAM guaranteed bandwidth and BW_{max} is the maximum achievable DRAM bandwidth (as measured in isolation, without regulation and while all the other cores are off).

Finally, 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 $K_q = \frac{P}{mL_{max}}$.

In this work we do not address the problem of contention at the level of shared cache bus and controller. The cache bus is normally on-chip and features a much higher bandwidth compared to the main memory bus. Contention at this level has been found to affect some modern platforms [36]. However, due to its high bandwidth capabilities, cache buses are normally able to sustain, without hitting the saturation point, the simultaneous activity of several CPUs. According to our benchmarks, on the Freescale P4080 platform used for our evaluations, contention at this level produces no visible effects up to 4 active cores and yields negligible slowdowns with 8 active cores.

Additionally, COTS architectures are often not time-composable between CPU pipeline and cache hierarchy [37]. This means that under certain circumstances, a local reduction of execution time (e.g. a cache hit) can produce a global increment in the execution time and vice versa, determining what is known as timing anomaly. In SCE, however, cache blocks are allocated to tasks using lockdown. As a result, the memory locations that will produce a cache hit are known at profile time and do not change across different runs, while the rest of the cache is unavailable for allocation. Consequently, if timing anomalies exist, their effects are already embedded in the experimentally derived WCET. No additional timing anomalies will be experienced after production as there is no variation in the sets of accesses that hit/miss in cache.

The derived theoretical model also assumes *delay additivity*. If delays are not additive, a pending memory request (or other instruction) not only introduces a delay in the execution due to the current operation, but may also determine additional delays in subsequent instructions. As previously discussed, an exact micro-architectural model is typically not available for COTS systems. Thus, formally determining and modeling their delay additivity is often infeasible. Conversely, the goal of this paper is to derive WCET estimates that can be used at design time for complex systems through a measurement-based approach. In practice, however, the following consideration can be made: out-of-order processors like the one considered in our evaluation are likely to satisfy delay additivity. In fact, instructions that cause long delays, like those accessing memory, can execute in parallel with different operations or with instructions of the same type. This effect counterbalances memory-induced delays, so that the resulting increase in execution time can be less or equal than the single instruction delay.

Finally, although the effect of PALLOC does not appear directly in the analysis, it is important to underline that the enforcement of private DRAM banking across cores is fundamental for the soundness of the analysis. In fact, modern DRAM controllers perform reordering of requests at the bank level to maximize open-row hit ratio [32]. Due to reordering, if multiple cores are allowed to access the same bank, several requests from a different core can be processed before a request from the core under analysis, potentially leading to large delays.

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, given that the scheduling policy of the DRAM controller implements 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 and the memory-controller have a round-robin arbitration policy, 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:

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

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: $P - K_q L_{min}$. This also captures 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 bus and memory controller use a round-robin scheduling policy, 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 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:*

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

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. Thus, with a round-robin scheduling policy at the bus and memory controller, 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.

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

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. WCET(m) and Response Time Analysis

We consider a set of implicit-deadline tasks $\tau_1, \ldots, \tau_n$, where each task τ_i is characterized by a period T_i, a WCET in isolation $C_i = \text{WCET}(1)_i$, and a maximum number of residual last-level cache misses after lockdown $\hat{\mu}_i$. In this section we proceed as follows to derive $\text{WCET}(m)_i$ for a given task τ_i. First, we assume that the scheduling policy is based on tasks' fixed priority assignment (e.g. Rate Monotonic) upon a partitioned multi-core system; this is a common practice used in industrial applications. Next, we consider a level-i busy interval[1]. From Lemma 3, the worst-case memory access pattern can be obtained by clustering together all the memory accesses of task instances in the considered busy interval. Finally, we show that the value of $\text{WCET}(m)_i$ follows directly from this analysis. For ease of notation, we use the more compact term C_{sce} to indicate $\text{WCET}(m)$.

Equation 2 considers a level-i busy interval and computes the response time analysis for tasks with regulation-induced stall as calculated in Equation 1.

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

[1]Defined as a time interval during which only jobs of task i or belonging to higher priority tasks execute continuously on the processor.

Where $\mu^{(k)}$ captures the clustered memory accesses of all the task instances in the interval and is defined as follows:

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

Note that the *stall* term in this formulation only depends on the term $\mu_i^{(k)}$ since 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 WCET(m) with m active cores under SCE 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:

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

By expanding $\mu_i^{(k)}$ using Equation 3:

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

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

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

Which can be finally rewritten as:

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

■

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 over-approximated, 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 framework is deployed and the analysis is carried out as described in Section VII. Specifically, we show how the workload for a specific task can be characterized and how WCET(m) 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 from specifications and validated through benchmarking to be at 1.2 GB/s. Similarly, we have determined a peak bandwidth of 2.5 GB/s. Next, the MemGuard budget replenishment period has been configured as $P = 1\ ms$, which represents a good trade-off between regulation granularity and introduced overhead [29]. 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 locking and unlocking of cache lines, we use the `dcbtls` and `dcblc` atomic instructions. In this evaluation, we manage the shared L3 cache, and keep the lower cache levels unavailable for allocation. Thanks to its large size, the L3 allows more flexibility in the allocation strategy. Unfortunately, in the P4080 platform, this level of cache is not significantly faster than the DRAM subsystem, 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.

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

This and similar optimizations, however, are left for future work as we are mostly interested in enforcing performance isolation.

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 counters 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 the SCE theoretical model described in Section VII, we have used the San Diego Vision Benchmarks Suite (SD-VBS) [38]. 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 [28]. 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 across execution expressed in KB ("PeakVM"); ratio between runtime with no allocated pages and full L3 assignment ("Exec. Ratio").

We were able to observe the same performance trend across all the considered benchmarks. Thereby we show in the next section the detailed curves for one benchmark in particular, tracking, and summarize the rest in Table III. The tracking 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 [38], the considered benchmark implements the Kanade Lucas Tomasi (KLT) tracking algorithm.

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 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, we make the rest of the L3 cache unusable for the task by locking arbitrary portions of system memory.

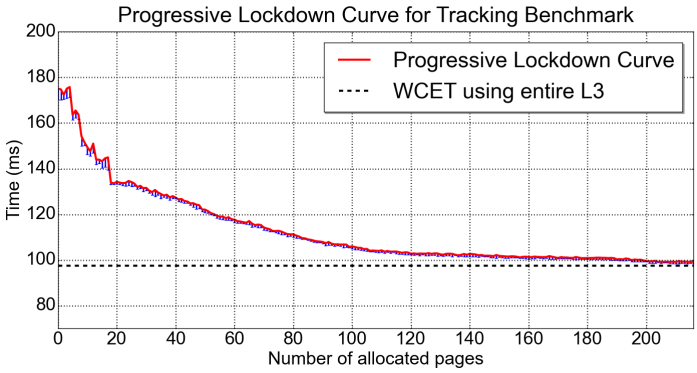

Fig. 1. Progressive lockdown curve for tracking benchmark.

Figure 1 depicts the resulting curve and compares the resulting WCET 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 measured WCET, while the negative error bars report the difference between maximum and minimum observed runtime. Each data point summarizes 50 different observations. 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 where the entire L3 is available for the application.

D. C_{sce} Estimation

Albeit not shown in the plots, the number of residual generated DRAM transactions is associated to each data-point in the progressive lockdown curve. 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 value of WCET(m) with $m = 8$ active cores (i.e. C_{sce}) can be derived according to Equation 5.

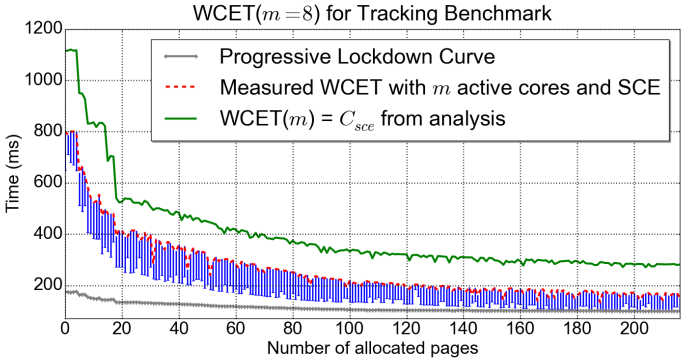

Fig. 2. C_{sce} calculation and experimental MemGuard runtime for tracking.

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

Benchmark	Solo	Exp. w/ SCE	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

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 and visible at the bottom of the plot. Finally, the dotted line represents the measured WCET when MemGuard is activated, even bandwidth assignment is enforced and memory-intensive benchmarks are deployed on all the remaining 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. The latter are 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 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 is possible to enforce strict resource allocation to prevent inter-core interference with a reasonable loss in performance. In fact, after about 140 allocated pages, we observe that some of the collected samples present 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 WCET. This property confirms the validity of the proposed model and analysis. As previously mentioned, the same result has been obtained on all the benchmarks of the suite. Due to space constraints, 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 measured WCET in isolation ("Solo"); measured WCET under SCE while memory-intensive tasks are active on different cores ("Exp. WCET w/ SCE"); 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 (Lemma 3). 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

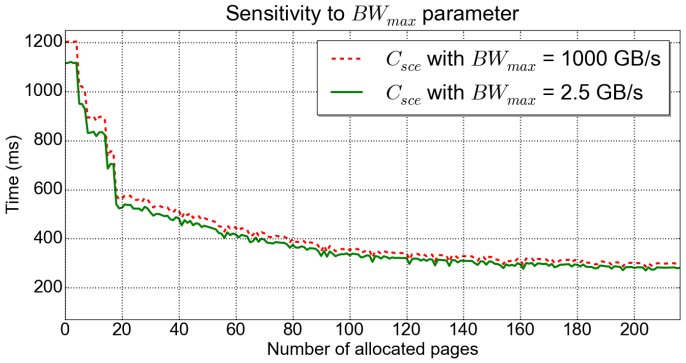

Fig. 3. Sensitivity of C_{sce} to variations in L_{min} (i.e BW_{max}) parameter.

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 AND FUTURE WORK

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 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 restore the property of a constant WCET in relation to the m active cores configured in the system. This property can be achieved by using the presented Single Core Equivalence (SCE): a framework of OS-level techniques that achieves isolation by performing strict partitioning of shared memory resources among cores. SCE can be deployed on COTS platforms and ultimately allows modular verification and certification on a per-core basis. In this work we have also provided a methodology to perform response-time analysis of an SCE-compliant system and validated our theoretical results by implementing SCE on commercial hardware.

As a part of our future work, we plan to extend SCE in several directions: first, the properties exported by our framework can be integrated in static analysis tools to automatically derive task WCET in multi-core systems. Next, we plan to refine WCET(m) calculation to consider more complex memory controller policies such as FR-FCFS. Additionally, we intend to integrate I/O serialization in the framework to derive end-to-end schedulability analysis. Finally, several optimizations can be performed to lower both the pessimism of the analysis and the experimental noise.

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] FAA position paper on multi–core processors, CAST32 (rev 0). http://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_papers/media/cast32.pdf. Accessed: 2015-01-26.

[2] 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.

[3] L Sha, M. Caccamo, R. Mancuso, J.E. Kim, M.K. Yoon, H. Pellizzoni, R. Yun, R. Kegley, D. Perlman, G. Arundale, and R. Bradford. Single core equivalent virtual machines for hard real–time computing on multicore processors. Technical Report http://hdl.handle.net/2142/55672, University of Illinois at Urbana–Champaign, November 2014.

[4] 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.

[5] 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.

[6] 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 25th Annual ACM Symposium on Theory of Computing*, STOC '93, pages 345–354, New York, NY, USA, 1993. ACM.

[7] 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.

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

[9] 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.

[10] 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.

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

[12] 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.

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

[14] M. Papadakis and N. Malevris. A symbolic execution tool based on the elimination of infeasible paths. In *Software Engineering Advances (ICSEA), 5th International Conference on*, pages 435–440, Aug 2010.

[15] 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.

[16] 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.

[17] 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 (RTAS), IEEE*, pages 80–89, April 2008.

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

[19] D. Hardy, T. Piquet, and I. Puaut. Using bypass to tighten WCET estimates for multi-core processors with shared instruction caches. In *30th IEEE Real-Time Systems Symposium (RTSS'09)*, pages 68–77, 2009.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

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

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

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

[31] 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.

[32] 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.

[33] Airlines Electronic Engineering Committee (AEEC) and Aeronautical Radio Inc. ARINC Specification 651: Design guidance for integrated modular avionics, ser. ARINC report., November 1991.

[34] R. Kirner, P. Puschner, and I. Wenzel. Measurement-based worst-case execution time analysis using automatic test-data generation. In *Proceedings of the IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS)*, pages 7–10, 2004.

[35] L. Kosmidis, E. Quiñones, J. Abella, G. Farrall, F. Wartel, and F. J. Cazorla. Containing timing-related certification cost in automotive systems deploying complex hardware. In *Proceedings of the 51st Annual Design Automation Conference*, DAC '14, pages 22:1–22:6, New York, NY, USA, 2014. ACM.

[36] G. Gracioli and A. Fröhlich. On the influence of shared memory contention in real-time multicore applications. In *Proceedings of the IV Brazilian Symposium on Computing Systems Engineering (SBESC)*. IEEE, 2014.

[37] T. Lundqvist and P. Stenström. Timing anomalies in dynamically scheduled microprocessors. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, pages 12–, Washington, DC, USA, 1999.

[38] 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.