On the Evaluation of the Impact of Shared Resources in Multithreaded COTS Processors in Time-Critical Environments

PETAR RADOJKOVIĆ, Barcelona Supercomputing Center
SYLVAIN GIRBAL and ARNAUD GRASSET, Thales Research and Technology
EDUARDO QUIÑONES, Barcelona Supercomputing Center
SAMI YEHIA, Thales Research and Technology
FRANCISCO J. CAZORLA, Barcelona Supercomputing Center and Spanish National Research Council (IIIA-CSIC)

Commercial Off-The-Shelf (COTS) processors are now commonly used in real-time embedded systems. The characteristics of these processors fulfill system requirements in terms of time-to-market, low cost, and high performance-per-watt ratio. However, multithreaded (MT) processors are still not widely used in real-time systems because the timing analysis is too complex. In MT processors, simultaneously-running tasks share and compete for processor resources, so the timing analysis has to estimate the possible impact that the inter-task interferences have on the execution time of the applications.

In this paper, we propose a method that quantifies the slowdown that simultaneously-running tasks may experience due to collision in shared processor resources. To that end, we designed benchmarks that stress specific processor resources and we used them to (1) estimate the upper limit of a slowdown that simultaneously-running tasks may experience because of collision in different shared processor resources, and (2) quantify the sensitivity of time-critical applications to collision in these resources. We used the presented method to determine if a given MT processor is a good candidate for systems with timing requirements. We also present a case study in which the method is used to analyze three multithreaded architectures exhibiting different configurations of resource sharing. Finally, we show that measuring the slowdown that real applications experience when simultaneously-running with resource-stressing benchmarks is an important step in measurement-based timing analysis. This information is a base for incremental verification of MT COTS architectures.

Categories and Subject Descriptors: J.7 [**Computer Applications**]: Computers in other systems—*Real time*

General Terms: Measurement

Additional Key Words and Phrases: Multithreaded COTS processors, Resource-stressing benchmarks, WCET, evaluation

This work was done as a part of the internship of Petar Radojković in Thales Research and Technology. The internship was funded by HiPEAC Network of Excellence. This work was also supported by the Ministry of Science and Innovation of Spain under contract TIN-2007-60625. The work of E. Quiñones was funded by the Spanish Ministry of Science and Innovation under the grant Juan de la Cierva JCI2009-05455. P. Radojković holds the FPU grant AP2008-02370 (Programa Nacional de Formación de Profesorado Universitario) of the Ministry of Education of Spain.
Authors' addresses: P. Radojković, E. Quiñones, and F. J. Cazorla, Barcelona Supercomputing Center, Nexus II Building, Jordi Girona, 29, 08034 Barcelona, Spain; S. Girbal, A. Grasset, and S. Yehia, Thales Research and Technology (France); Campus Polytechnique 1, Avenue Augustin Fresnel, 91767 Palaiseau Cedex France. Correspondence email: petar.radojkovic@bsc.es.

DOI 10.1145/2086696.2086713 http://doi.acm.org/10.1145/2086696.2086713

ACM Reference Format:
Radojković, P., Girbal, S., Grasset, A., Quiñones, E., Yehia, S., and Cazorla, F. J. 2012. On the evaluation of the impact of shared resources in multithreaded COTS processors in time-critical environments. ACM Trans. Architec. Code Optim. 8, 4, Article 34 (January 2012), 25 pages.
DOI = 10.1145/2086696.2086713 http://doi.acm.org/10.1145/2086696.2086713

1. INTRODUCTION

Commercial Off-The-Shelf (COTS) processors are increasingly being considered in the design of real-time and mission-critical embedded systems in order to reduce the non-recurring engineering (NRE) and time-to-market (TTM) costs [Baker 2002]. In such systems, ensuring timing predictability and meeting deadlines are of prime importance and therefore the analysis of the system and the target applications are essential before deployment [Wilhelm et al. 2008]. Time predictability is, in fact, a requirement not only in the real-time market, but also coming to be of primary importance in the mainstream market as recognized in the HiPEAC roadmap [Duranton et al.].

Currently, the COTS processor market is moving toward multithreaded (MT)[1] processor architectures. MT COTS processors are of special interest due to their good performance-per-watt ratio and high performance opportunities [Ungerer et al. 2010]. These architectures are particularly well suited for embedded *integrated architectures* in which several functions are integrated into the same processor, such as Integrated Modular Avionics (IMA) [Watkins and Walter 2007] in the avionics domain or Automotive Open System Architecture (AUTOSAR) [AUTOSAR; Natale and Sangiovanni-Vincentelli 2010]. In this context, MT processors can potentially schedule mixed criticality workloads, i.e. workloads composed of safety-critical, mission-critical, and non-critical applications inside the same processor, improving the hardware utilization and so reducing cost, size, weight, and energy consumption [MERASA].

Unfortunately, despite the benefits that MT COTS processors may offer in embedded real-time systems, particularly in integrated architectures, the time-critical market has not yet embraced such a shift. The main challenge that MT COTS architectures face is with predicting the impact that the collision among simultaneously-running tasks has on execution time of time-critical tasks. The loss of predictability is explained by the fact that co-running tasks have to share the hardware resources, even if they do not communicate with each other. When two or more tasks that share a hardware resource try to access it at the same time, the tasks experience *inter-task interference*. Inter-task interferences are handled by an arbitration mechanism, which may affect the execution time of running tasks. As a result, it is much more difficult to provide the worst-case execution time (WCET) estimations for applications running on MT processors than running on single-threaded processors. Several studies show that collision in processor resources between co-running tasks may cause significant impact to application execution time [Doucette and Fedorova 2007; Čakarević et al.] and therefore on WCET [Pellizzoni et al. 2010a].

Static WCET analysis computes WCET bound based on the extensive program analysis and detailed model of the hardware [Puschner and Burns 2000]. Static WCET analysis is currently the only approach that computes safe WCET bounds, i.e. that guarantees that the actual execution time of the program cannot be longer than the computed WCET bound. However, there are several reasons that make the use of static WCET analysis difficult on real industrial programs running on MT COTS

[1]In this paper, we will use the term "multithreaded processor" to refer to any processor that has support for more than one thread running at a time. Multicore, HyperThreading, Simultaneous Multithreading, Coarse-grain Multithreading, Fine-Grain Multithreading processors, or any combination of them are multithreaded processors.

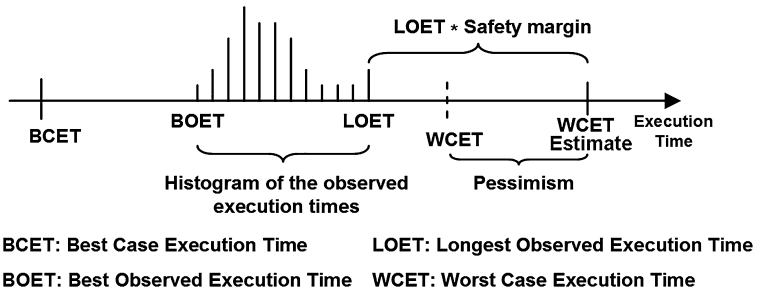


Fig. 1. Measurement-based timing analysis.

architectures [Kirner et al. 2005; Kirner and Puschner 2008; Mezzetti and Vardanega 2011]. Some of these reasons are: (1) Static WCET analysis of real industrial programs with a vast number of possible execution paths is a challenging task; (2) The implementation of accurate hardware models for new architectures requires a significant effort and a detailed description of the hardware, which is not always available; (3) Possible interference in shared hardware resources among tasks that simultaneously execute on MT architectures significantly increases the complexity of timing analysis.

This has motivated studies that analyze if changes in hardware can facilitate the effective timing analysis of real industrial programs running on MT architectures. There have been several hardware proposals [Across; Hansson et al. 2009; Genesys; Merasa; Pret; Tta; Predator] to ease the computation of composable WCET bounds or WCET estimates[2] of tasks running on multithreaded architectures. However, these proposals require changes in hardware and additional features that the current MT architectures do not have. Therefore, the industry that wants to use MT architectures in their current real-time system designs cannot benefit from them.

A measurement-based approach for single-threaded architectures computes the WCET estimate by multiplying the longest observed execution time (LOET) by a safety margin, usually provided by an expert with understanding of the target hardware architecture and the reference applications (see Figure 1) [Mezzetti and Vardanega 2011]. This method has been successfully used in the past to determine WCET estimate of applications running on single-threaded processors with a moderate difference between the Average Case Execution Time (ACET), the LOET, and the WCET estimate. However, the method provides no analytical guarantees that the estimated WCET is safe and it may depend significantly on the quality of the test-cases used as well as the experience of the expert(s) who compute the safety margin. A direct extension to the measurement-based analysis presented in Figure 1 to MT COTS processors would consist of running several reference applications simultaneously on the same processor, and monitoring the execution time for each application in the workload. However, in the case of MT architectures, the design of test-cases and the choice of the workload has more importance than for single-threaded processors. The effect of sharing processor hardware resources with simultaneously running (co-running) tasks may introduce significant variations in the execution time of applications. This can make a methodology based on safety margins fail. In addition to this, a change of any co-running task may affect the way that co-runners interfere, and necessitates repeating the WCET analysis

[2]The term, *WCET bound*, is used to refer to safe upper bound of program execution time that is provided by static WCET analysis. Static WCET analysis guarantees that the execution time of a given program will not exceed its WCET bound for all valid input configurations. *WCET estimate* refers to the upper bound of program execution time that is unlikely to be exceeded. The techniques that compute WCET estimate do not provide formal verification that program execution time does not exceed WCET estimate [Kirner et al. 2005; Kirner and Puschner 2008].

for all the tasks in the workload. Thus, the measurement-based timing analysis used for single-threaded processors cannot be directly extended for MT COTS architectures.

Our study provides to the industry a systematic methodology for measurement-based timing analysis of applications running on MT COTS architectures. The main contributions of our study are the following.

—We show that running workloads composed of real applications may not be sufficient to determine the slowdown that simultaneously running tasks may experience because of collision in shared processor resources. Thus, the measurement-based timing analysis used for ST processors cannot be directly extended to MT architectures.
—We present a method to determine if a given MT processor is a good candidate for systems with timing requirements.
—We also show that measuring the slowdown that real applications experience when co-running with designed resource-stressing benchmarks improves measurement-based timing analysis for MT architectures. This can be used as a base for incremental verification, a key feature of integrated implementations such as IMA or AUTOSAR.

In order to reach these objectives, we defined a set of specific *resource-stressing* benchmarks that introduce a high number of interferences on each potentially shared hardware resource. By using these resource-stressing benchmarks as co-runners, we obtain a good estimation of the worst-case slowdown that real applications may experience because of collision in shared processor resources. When a workload is composed only of resource-stressing benchmarks, the detected slowdown is unlikely to be exceeded for any workload composed of real applications. Therefore, the slowdown detected when using resource-stressing benchmarks may serve as an upper estimate of the effect of inter-task interference for a given processor.

We present several case studies in which we analyze three MT COTS architectures with different degrees of shared resources. We show that, for a given workload composed of several benchmarks, all three architecture types show low interference among co-running tasks and stable execution times. However, our method shows that the potential variation in the execution time of applications is different for each architecture under study.

As our study targets real COTS processors, we do not suggest any hardware change in the target architecture, but propose a way to improve the measurement-based approach for MT COTS processors. This is one of the main differences with previous works in the field that have suggested hardware modifications to improve architecture time predictability.

The rest of the paper is organized as follows: In Section 2, we propose a method for analysis of potential interference of co-running tasks in shared processor resources. Section 3 shows details of the experimental environment and methodology used in the study. Section 4 presents a case study in which we evaluate the suitability of three MT COTS architectures for time-critical environments. Related work is presented in Section 5, while Section 6 summarizes the conclusion of our study.

2. ANALYSIS OF INTER-TASK INTERFERENCES IN CURRENT MT PROCESSORS

In MT processors, the execution time of a task depends not only on the underlying hardware and the way the task is programmed, but also on the slowdown caused by the interference between simultaneously-running tasks. In order to provide a meaningful WCET estimation for tasks running on an MT processor, it is required to take into account the inter-task interference in shared processor resources. Several studies [Cullmann et al. 2010; Paolieri et al. 2009a, 2009b] and projects [Across; Hansson et al. 2009; Genesys; Merasa; Pret; Tta; Predator] address this problem for safety-critical

applications by introducing hardware mechanisms to define the upper bound of the delay a task can experience due to interferences in cache memory, bus, and access to the main memory. Unfortunately, these proposals have not yet been accepted by processor manufactures, and it is unclear if the COTS processor market will adopt them. Even if the hardware proposals were accepted, it would take at least five to ten years to implement them.

The analysis of the impact of inter-task interferences to application WCET is very complex. Without the hardware support proposed in the previously mentioned studies and projects, using static WCET analysis for MT COTS processor running real workloads is infeasible in practice. As we mentioned in Section 1, another solution could be to directly extend the measurement-based approach used for single-threaded architectures, and to estimate the application's WCET based on the longest observed execution time within all possible workloads.

The problem with this approach is that the number of different workloads in real systems may be large. For example, assume a task set composed of n tasks is to be executed on a target processor able to run up to k tasks at a time, in which $n > k$. Under this scenario, the number of workloads that have to be analyzed is $\frac{n!}{k!(n-k)!}$. Moreover, any change in the workload, e.g., a shift in time at which each application in the workload starts, would invalidate the previous analysis for all running tasks, especially when running mixed-criticality workloads with non real-time applications. Hence, measurement-based timing analysis considering all possible workloads is not feasible in practice for time-critical tasks running on MT COTS processors.

In this paper, we propose a software solution to this problem. In particular, we propose the design of resource-stressing benchmarks and a methodology for their use to quantify possible slowdown that co-running tasks may experience because of a collision in shared resources of MT COTS processors. The objective of the resource-stressing benchmarks is to introduce a high-load onto each of the processor hardware resources that the task which is under the analysis may use. Thus, the resource-stressing benchmarks can be used to provide good estimates of the potential slowdown that a set of simultaneously-running real applications may experience because of the collision in shared resources of a given processor. When the methodology is used on different MT COTS architectures, it can help to determine the suitability of each architecture to time-critical environments. The methodology can also be used for a given architecture to determine the potential variation in execution time that a particular task in a workload may experience if any of the co-runners change. This information is a key to providing incremental qualification [EMPRESS].

2.1. Worst-Interference Benchmark

When designing a resource-stressing benchmark, we would like to have the *worst possible interference benchmark*. This benchmark would cause the highest possible interference in the shared resources that are used by the application under study. In this scenario, the slowdown of the application due to collision in processor resources could be used to compute a safe execution time bound. It is important to note that the worst-interference benchmark would be specific for the application under study and the target MT processor.

Unfortunately, the design of the worst-interference benchmark is extremely difficult or even impossible, even for a single hardware shared resource. In order to illustrate this, we analyze the design of a benchmark that should cause the worst interference to a given application in the shared instruction fetch unit (IFU) in an SMT processor. In this case, we assume that the IFU is designed to fetch up to one instruction in each cycle, and implements a least recently fetched policy, i.e. the thread that last fetched an instruction has the lowest priority. Under this fetch policy, whenever the

worst-interference benchmark fetches an instruction, the application under study becomes the least recently fetched task with higher priority in the next access to the IFU. In order to design the worst-interference IFU benchmark for a given application, it is required to (1) Determine precise cycles in which application under study will access IFU, (2) Determine the priority of the application in each IFU access, and (3) Consider how the interference between the application under study and the worst-interference benchmark and will delay any future IFU requests of both tasks.

In general, in real time systems, the design of worst-interference benchmark requires:

—Full knowledge of the processor implementation, including resource latency, arbitration policy, etc. However, many of these hardware features of COTS processors are not reported in the public documentation.
—Full knowledge of the application, including input data set, execution flow, resource usage pattern, etc. However, a common practice is that applications are provided by different suppliers (Tier 1) [Gereffi 1999], making it difficult to have full knowledge about all applications.
—The worst interference benchmark should stress all resources used by the application under study, needing to be perfectly aligned in the access to each of shared resources used by the application. Any misalignment between worst-interference benchmark and application under study can significantly reduce the impact of resource sharing.

All these requirements make the design of worst interference benchmarks infeasible in practice.

2.2. Resource-Stressing Benchmarks

A key design choice in our resource-stressing benchmarks is how to stress shared processor resources. (1) We can design benchmarks that specifically stress a single resource by putting a high load on it. For example, to stress the instruction fetch unit, we could design a benchmark that fetches an instruction in each cycle. The downside of this solution is that only one resource can be stressed at a time. (2) Alternatively, we can design benchmarks that stress several resources at a time. For example, a benchmark can comprise different instructions that access different resources. The downside of this solution is that the stress in each resource decreases as several resources are stressed simultaneously.

In architectures with two cores or hardware contexts, in which we can only run the application under study and one resource-stressing benchmark at a time, it is unclear which of the two approaches better show the sensitivity of applications to hardware resource sharing. However, as the number of cores or contexts increases, the methodology that uses benchmarks that stress a single hardware resource scales very well. Under that methodology, we can reserve 1 out of the N cores (or hardware contexts) to run the application under study on, and use the remaining N-1 to run different combinations of stressing benchmarks, each stressing a given resource. This improves the obtained slowdown as with several cores or hardware contexts we can observe the execution of the application under different resource-stressing conditions. In Section 4.4, we show a case study in which we use this feature of the methodology.

We start benchmark design by identifying common levels in the hardware resources shared in MT COTS processors. This allows the creation of a set of benchmarks that stress those particular resources. In particular, we identify three *resource sharing levels*.

(1) *Intra-core resources* include: (1.1) Front end of the pipeline; (1.2) Back end of the pipeline: Integer and Floating Point (FP) execution units; (1.3) The L1 data cache; (1.4) The L1 instruction cache.

(2) *Inter-core resources* such as the L2 cache memory (the last level of cache memory).
(3) *Interface to off-chip resources* such as the bandwidth to the main memory.

Although these levels of shared resources are common in MT COTS processors, the effect in time (delay) that the interference in each of these resources causes, depends on the particular processor. To measure that delay, for each level of shared resources, we designed at least one resource-stressing benchmark as described next.

(1.1) *Front end of the pipeline.* In order to stress front end of the pipeline, mainly the instruction fetch unit and the decode unit, we designed a benchmark that executes a series of *nop* instructions. The nop instruction is a low latency instruction that puts significant stress to the front end of the pipeline and negligibly stresses rest of the processor resources.

(1.2) *Back end of the pipeline (Integer and FP execution units)*: In state-of-the-art processors, different integer and FP instruction may execute in different execution units and may have different behavior (e.g. may be pipelined or non-pipelined). To cover the different cases of possible interference, we design six benchmarks for stressing integer and FP execution units. *intAdd*, *intMul*, and *intDiv* benchmarks consist of a sequence of integer addition, multiplication, and division instruction, respectively. *fpAdd*, *fpMul*, and *fpDiv* benchmarks execute a serial of floating point addition, multiplication, and division instructions, respectively.

(1.3) *The L1_dcache* benchmark consists of a sequence of load instructions that access different cache lines of the L1 data cache. The size of the array that the benchmark traverses is the same as the size of the L1 data cache. Therefore, when the L1_dcache benchmark executes in isolation, most of the loads hit in the L1 data cache. However, when L1_dcache cache is scheduled with a co-runner that uses L1 cache, the data sets of both benchmarks do not fit in the cache, which causes L1 misses and longer execution time.

(1.4) *The L1_icache* benchmark consists of a sequence of jump instructions that access different cache lines in the instruction cache. The size of the code is equal to the size of the instruction cache.

(2) *The L2 benchmark* is designed using the same principle as L1_dcache benchmark. The only difference is that the size of the array that the benchmark traverses is equal to the L2 cache size.

(3) *Interface to off-chip resources (Memory bandwidth).* One of the factors which has a significant impact on the applications performance is access to the off-chip resources. The *mem_bw* benchmark is a resource-stressing benchmark that has the same structure as L1_dcache or L2 benchmark (a sequence of load instructions that traverse the array). Since the purpose of the benchmark is to stress the bandwidth to memory, but not to cause any collision in the main memory, the size of the array that the benchmark traverses has to be chosen to cause misses in the last level of cache, but not to cause page faults in the main memory. For the configurations we test in this study, the size of the *mem_bw* array is four times larger than the size of the last level of cache.

The set of resource-stressing benchmarks we present in this paper can be easily extended to stress execution units and I/O interfaces of different processors. For each architecture under study, the user should determine the set of shared resources in which co-running tasks may collide and design resource-stressing benchmarks for each of them.

2.3. Implementation

The framework for automatic execution of the experiments and for processing of the results is implemented in *C* programming language using POSIX threads [Butenhof 1997]. The real-time and resource stressing benchmark that are used in each experiment are read from the input file defined by the user.

Table I. Structure of *intAdd* Resource-Stressing Benchmark

Line	Source code	Explanation
001	movl %1, %ecx	initialize loop counter *ecx* (%1 is an input parameter)
002	label_intAdd:	beginning of the loop
003	add %eax, %ebx	target instruction
004	add %ebx, %eax	target instruction
...	...	...
...	...	...
252	add %ebx, %eax	target instruction
253	decl %ecx	decrement loop counter
254	cmp %ecx, $0	compare loop counter with 0
255	jne label_intAdd	if (counter != 0) jump to the beginning of the loop

The core of our framework consists of benchmarks that stress specific processor resources. Each resource-stressing benchmark is defined in the function that is included in the framework. The benchmark functions are implemented directly in assembly of the target processor in order to: (1) Provide the programmer with the maximum control over the instructions that are to be executed, and (2) Prevent a compiler from applying any optimization that changes the core of the benchmarks. The assembly functions are inlined in a *C* code in order to avoid the overhead of the function call.

All the resource-stressing benchmarks are designed using the same principle that is presented in Table I. Each benchmark is comprised of three parts: (1) The register used as a loop iteration counter (*ecx*) is initialized to the value of the input parameter (line 001). The initial value of *ecx* determines the number of loop iterations and the duration of the benchmark. (2) The main part of the benchmark is a sequence of instructions of target type (lines from 003 to 252). The *add* instruction is changed by the corresponding target instruction in each of the resource stressing benchmarks: *nop* instruction[3] in the case of the *nop* benchmark, integer multiplication in the case of the *intMul* benchmark, etc. (3) Finally, the sequence of target instructions is followed with the decrement of the loop counter register (line 253), comparison of the counter value with zero (line 254), and a conditional branch to the beginning of the loop (line 255). The overhead of the loop and the calling code is very low - more than 99% of the instructions targets the specific resource we want to stress.

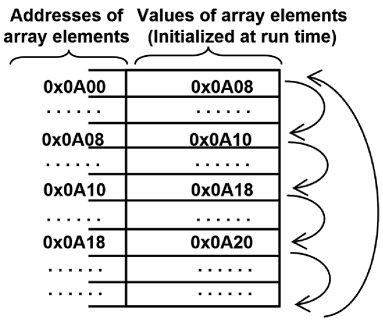
An overview of the benchmarks that stress the cache memory and the memory bandwidth is shown in Figure 2. The benchmarks are implemented using the concept of *pointer chasing*. In the benchmark initialization, which is done in the *C* code, we allocate a contiguous section of memory and initialize it in such a way that a given array element contains the address of the next array element (memory location) that we want to access, see Figure 2(a). The benchmarks are initialized to (1) Traverse the whole array, and (2) Access different cache lines in each memory access. An example of a benchmark memory access pattern is shown in Figure 2(b). Finally, Figure 2(c) shows the assembly code that stresses the memory subsystem. First, the register used as a loop iteration counter (*ecx*) is initialized to the value of the input parameter (line 001). The initial address of the array is passed to the assembly code as an input parameter (line 002). The main part of the benchmark (lines from 004 to 253) is a sequence of

[3]In the current version of the nop resource-stressing benchmark, we use one-byte nop instruction which is an alias mnemonic for the *XCHG (E)AX, (E)AX* instruction in Intel architectures. This instruction performs no operation and does not impact machine context, except for the instruction pointer register [Intel Corporation 2011]. The same effect could be achieved by using multi-byte nop in processors that have a support for this instruction [Intel Corporation 2011]. As a part of future work, we plan to analyze whether using more complex opcodes would put higher stress to the decode unit and cause higher collision among simultaneously-running tasks that share this processor resource.

(a) Initialization code

```
for(cnt=0; cnt < array_size; cnt+=stride)
{
    if(cnt<array_size-stride)
    {
        // Each array element contains the address of ...
        // ... the following array element we want to access.
        array[cnt] = (int)&array[cnt+stride];
    }
    else
    {
        // The last accessed element in the array points ...
        // ... to the first element of the array that we access.
        array[cnt] = (int)array;
    }
}
```

(b) An example of a memory access pattern



(c) Assembly stressing code

Line	Source code
001	movl %1 %ecx
002	movl %2, %eax
003	label_mem:
004	mov (%eax), %eax
005	mov (%eax), %eax
...	...
...	...
253	mov (%eax), %eax
254	decl %ecx
255	cmp %ecx, $0
256	jne label_mem

Fig. 2. Overview of the memory stressing benchmarks.

indirect load instructions ($mov(\%eax), \%eax$) that follow the memory access pattern specified in the initialization.

In order to design the L1_icache benchmark, we used a sequence of unconditional jump instructions that point to different labels (jump destinations). The code is designed in such a way that, at run-time, the benchmark (1) Traverses the whole instruction cache, and (2) Accesses different cache lines in each jump instruction.

We analyzed not only the functionality of the resource-stressing benchmarks, but also their portability to different processors with different cache organization or different ISA. By setting the values of parameters *array_size* and *stride* to proper values (see Figure 2(a)), a user can easily adjust the memory stressing benchmarks to stress different parts of the memory subsystem of different architectures. The stressing assembly code of the L1_icache benchmark is automatically generated. In order to adjust the L1_icache benchmark to stress instruction caches with different size and organization, the user only has to specify the desired size of the stressing code and the stride between consecutive jump instructions (i.e. the stride between consecutive accesses to instruction cache). The resource stressing benchmarks are implemented in x86 ISA. In order to port benchmarks to architectures with different ISA, e.g. POWER or SPARC, the programmer only needs to change the x86 instructions in the assembly stressing code with the corresponding instructions of the target ISA. The assembly stressing

code of the benchmarks is very simple, see Table I and Figure 2(c), thus a little effort is required to port this code to architectures with different ISAs. As a part of future work, we plan to develop a tool that automatically generates the resource-stressing benchmarks for different architectures based on the architecture's ISA and a brief description of the hardware resources (e.g. the size and the organization of shared cache memory).

2.4. Using the Resource-Stressing Benchmarks

We used the resource-stressing benchmarks to: (1) Estimate the upper limit of a slowdown that co-running tasks may experience because of collision in different shared resources of a processor. This slowdown can be used to determine if a given MT processor is suitable for a time-critical environment; (2) Quantify the possible impact of inter-task interference to the execution time of an application. This improves the WCET estimation for applications that execute in MT COTS processors.

2.4.1. Worst-Case Slowdown in Shared Processor Resources. In order to quantify the slowdown that an application may experience due to collision in the different shared resources of a given MT COTS processor, we deploy our resource-stressing benchmark in the following way. First, we measure the execution time of each resource-stressing benchmark when it runs in isolation ($ET_{isolation}$). Second, we measure the execution time when several resource-stressing benchmarks run concurrently (ET_{MT}). In order to quantify the interference in processor resources, we compute the slowdown or normalized execution time as the relative difference between the benchmark execution time when it shares processor resources with other co-running benchmarks and when it runs in isolation: $slowdown = ET_{MT}/ET_{isolation}$. As resource-stressing benchmarks put high load on different hardware resources of a processor, the computed slowdown presents a good estimation of the worst-case slowdown that real applications may experience because of collision in these resources. Understanding the slowdown that co-runners may experience because of collision in shared resources can be used to define the suitability of a processor for a time-critical environment.

—When co-runners experience a significant slowdown due to resource sharing, this shows a potentially high variation in execution time of applications running on the processor. This means that the processor is not a good candidate for systems that have timing requirements.
—When co-runners experience a low slowdown due to resource sharing, it means that the processor is suitable for running time-critical applications. Low variation in execution time would allow accurate timing analysis for applications running on these architectures even if the set of co-runners change.

2.4.2. WCET Estimation for Real Applications. In order to provide a meaningful WCET estimate of real applications running on a given MT COTS processor, it is important to quantify the slowdown that applications may experience due to collision in shared processor resources. Resource-stressing benchmarks can be used to measure this slowdown. In order to quantify a slowdown due to inter-task interferences in shared processor resources, we deploy our resource-stressing benchmark in the following way. We measure the execution time of a real application when it runs in isolation ($ET_{isolation}$) and when it simultaneously executes with different resource-stressing benchmarks ($ET_{RS[i]}$). The *sensitivity* of the application to sharing *a given resource X* of the processor is computed as $sensitivity[X] = ET_{RS[X]}/ET_{isolation}$, where $ET_{RS[X]}$ is the execution time of the application when co-running with a resource-stressing benchmark that targets resource X.

The *sensitivity* of an application running on a given processor is computed as the maximum value of sensitivity for all analyzed processor resources. If this sensitivity low, the application is not sensitive to the resource sharing on a given processor. This may be because the application does not stress shared processor resources. If the application experiences a significant slowdown when it executes with a benchmark stressing a given processor resource, the application will show a high sensitivity to that resource. Collision in that hardware resource is a potential source of high execution time variation that the application may experience.

In processors with more than two virtual CPUs (i.e. cores or hardware contexts), we can run the real application under study and several resource-stressing benchmarks at a time. These experiments combine the effect of the interferences in different resources, which may increase a measured slowdown and improve the WCET estimation. We expect that this methodology will show even better results in future MT COTS processors in which the number of cores and hardware contexts will increase.

3. EXPERIMENTAL ENVIRONMENT

In this section, we present the experimental environment used in the case study. We describe the different MT COTS processors, real benchmarks, and the methodology used in the experiments.

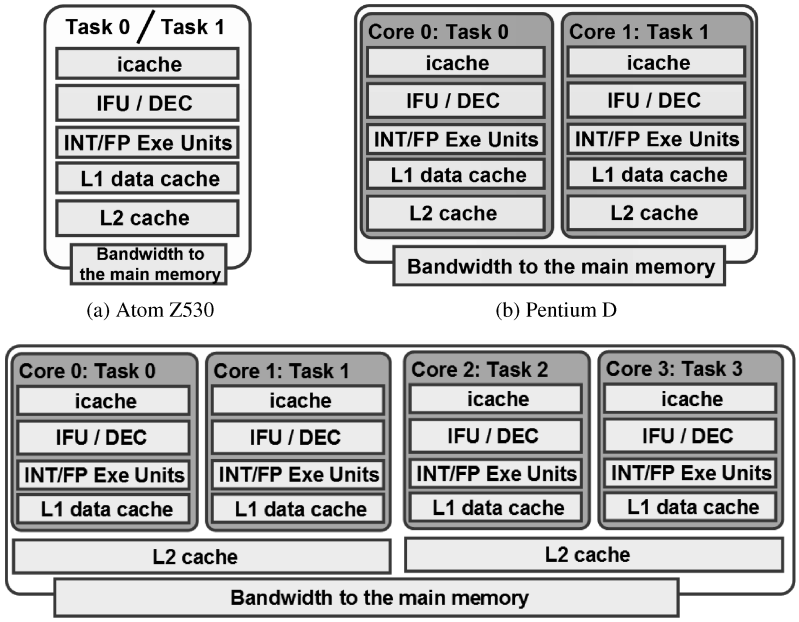
3.1. Hardware Environment

In this paper, we present a methodology that can be used to analyze how collision in shared processor resources can impact on the execution time of an application running on MT architectures. Still, in order to determine if a given processor is suitable for embedded real-time systems, it is important to consider several characteristics such as performance, energy consumption, dissipation, etc. The processors we used in the case studies are not necessarily the type of MT COTS processors used in embedded real-time systems because they do not necessarily meet those requirements. However, the processors used exhibit several configurations of resource sharing that allow us to show the application of our methodology for different types of target architectures. The three MT COTS processors considered in this study are the following.

(1) *Atom Z530* [Atom Z530 2009] is a HyperThreading processor. The schematic view of the processor is shown on in Figure 3(a). Atom Z530 has one core that supports the simultaneous execution of two tasks (two software threads). Most of the processor resources are shared among co-running tasks: from the front-end of the pipeline to the memory bandwidth. The platform based on the Atom processor contains 500MB of main memory.

(2) The *Pentium D* processor [Pentium 2007] contains two cores, and each of them can execute only one task at a time, see Figure 3(b). The front end of the pipeline, integer and FP execution units, L1 data, instruction, and L2 caches are private to each core. Simultaneously-running tasks on this processor can only collide in the bandwidth to the main memory. Platform based on Pentium D processor contains 2GB of main memory.

(3) The *Core2Quad* processor [Core2Quad] that we use in the study (Q9550) contains four cores, and each of them can execute only one task at a time, see Figure 3(c). The front end of the pipeline, integer and FP execution units, L1 data and instruction caches, are private to each core. The L2 cache memory has two partitions. Each partition is shared by tasks running on two cores: Partition 1 is shared by tasks running on Core 0 and Core 1, while partition 2 is shared among Core 2 and Core 3. This means that the distribution of simultaneously-running tasks on the processor determines if tasks share the L2 cache.



(a) Atom Z530

(b) Pentium D

(c) Core2Quad

Fig. 3. Schematic view of the processors used in the study.

In order to evaluate all the possible scenarios, we run two sets of experiments for the Core2Quad processor: (1) Reference and stressing benchmarks are bound to Core 0 and Core 1, while no benchmarks are running on cores 2 and 3 (RS-nn). In this distribution, reference and stressing benchmarks share the L2 cache. (2) A reference benchmark is bound to Core 0, while no benchmark is running on Core 1, and two stressing benchmarks are running on Core 2 and Core 3 (Rn-SS distribution). In this distribution, reference and stressing benchmarks do not share L2 cache. Finally, on the Core2Quad processor, the bandwidth of the main memory is shared among all simultaneously running tasks. The platform based on the Core2Quad processor contains 4GB of main memory.

As mentioned in Section 2, when a processor contains more than two cores of hardware contexts, we can run several combinations of resource-stressing benchmarks in order to improve the worst-case slowdown obtained by our methodology. In the case of the Core2Quad processor, we will bind the benchmark under study to Core 0, and run all possible combinations of L2 and mem_bw benchmarks in Core 1, Core 2, and Core 3.

3.2. Benchmarks

In addition to the set of resource-stressing benchmarks presented in Section 2, we used the following benchmarks to evaluate the proposed methodology.

STAP Radar. Space Time Adaptive Processing (STAP) Radar is a mission-critical application developed by Thales Research & Technology [Chevalier and Maria 2006]. The application is a simplified view of a moving target indication application, whose goal is to receive the echo of a periodic sequence of radar pulses and to detect the objects that are moving on the ground. The main characteristics of the applications are the following. First, a large part of the application is data-flow, manipulating multidimensional arrays of data. Second, data reordering (switching dimensions of arrays) is often needed. Third, the processing chain uses different operators, with

different specific needs in terms of precision and dynamic range. And, fourth, real-time performance is one of the key requirements, both in terms of computation throughput and latency.

CoreMark [CoreMark] is a benchmark developed by the Embedded Microprocessor Benchmark Consortium (EEMBC) [Eembc] and is designed specifically to test the functionality of a processor core. The EEMBC CoreMark suite contains three types of functionality that are representative of real-time environments: (1) Matrix-related functionality: Matrix multiplications, matrix addition, addition of a constant to a matrix; (2) Linked list management operations: Operations like insert, remove, or find the element in the linked list; (3) Finite State Machine (automata) operations. In each run, CoreMark benchmark executes all three sets of functions.

H264_Encode is a benchmark included in MediaBench II suite [Fritts et al.]. The benchmark encodes videos using H.264 compression standard. This compression standard is widely used for recording, compression, and distribution of high definition video. H264_encode is a good representative of soft real-time applications. In addition to this, video encoding using H.264 standard is used for video streaming in high-performance mission-critical networks [Andritsopoulos et al. 2007; Detti et al. 2010; Higgins 2004].

3.3. Experimental Methodology

The experiments were designed in such a way as to provide reliable results and minimize the impact of the operating system processes. Each experiment was repeated 50 times and each time we measured the execution time of the application under study. As our study addresses timing predictability and timing bounds of applications, we reported and analyzed the Longest Observed Execution Time (LOET) in 50 repetitions. We measured the execution time of an experiment by reading the time stamp counter (*rdtsc*). The time stamp counter is a 64-bit register that counts the number of ticks since reset on x86 processors. We accessed the counter register using the macro written in x86 assembly that is included in the experimental framework. This way, we avoided any system call for measuring time.

As the experiments were executed on a full-fledged Linux operating system (OS), we paid special attention to minimizing the impact of the OS to our measurements [Gioiosa et al. 2003; Petrini et al. 2003; Radojković et al. 2008]. To avoid task migration among different cores (virtual CPUs, hardware contexts, strands) of a processor, we bound each benchmark to the corresponding core using the *sched_setaffinity* system call.

In order to quantify the impact of the OS processes on the platforms used in the study, we repeated each benchmark in isolation for 10,000 times and measured the impact of OS processes to the execution time of the benchmark. Our results show that the impact of interference with the OS processes to variation of benchmark execution time is below 1%.

4. EVALUATION

In this section, we show how the proposed methodology can be applied to determine the suitability of the three MT COTS architectures presented in the previous section for time-critical environments. This scenario is relevant when companies have to determine which MT COTS architectures are good candidates to be used in their future time-critical system. After using our methodology to select a subset of processors with appropriate distribution and characteristics of shared resources, the company can do a detailed analysis of each of the selected processors to provide stronger guarantees that it meets the requirements of the system.

4.1. Potential Execution Time Variation

We start by analyzing the potential slowdown that applications may experience because of the collision in shared processor resources. To that end, we run all resource-stressing benchmarks in isolation and in different workloads. By observing the slowdown of the benchmarks, we can quantify the potential slowdown a real application would experience due to resource sharing. The results are presented in Table II. Each entry of the table shows the slowdown that the benchmark under study (listed in the rows of the table) experiences when it is simultaneously executed with the stressing benchmarks (columns).

Atom Processor. When running on the Atom processor, all benchmarks in our resource-stressing benchmark suite have a co-runner that makes them experience significant slowdown, see Table II(a). In most of the experiments, the benchmark under study experiences the highest slowdown when it is simultaneously executed with one more instance of the same resource-stressing benchmark. We also observe that, in general, the detected slowdown is quite high (up to 15.3×) and that the variation of the slowdown for different benchmarks is also very high. The slowdown due to the sharing of pipeline resources is less than 2x. When co-running tasks collide in the cache memory, the slowdown is higher and it ranges up to 6.2×, 2.7×, 14.1×, and 15.3× due to collision in the L1 data cache, instruction cache, L2 cache, and memory bandwidth, respectively. The mem_bw stressing benchmark stresses memory bandwidth through shared L2 cache. Hence, the slowdown that the benchmark under study experiences when it is co-scheduled with mem_bw stressing benchmark is the consequence of cumulative collision in the L2 cache and the memory bandwidth.

As applications running on Atom processor may experience a significant slowdown because of interference in several resources, it is very difficult to estimate WCET of concurrently running time-critical tasks. Therefore, we conclude that the hyperthreading feature in the Atom processor should be avoided in time-critical environments that require predictability of application execution time. Disabling hyperthreading would lead to a more predictable execution time of the applications. However, this would make Atom a single-threaded processor that would defy all performance improvements of MT architectures.

Pentium D Processor. The results for the Pentium D processor are presented in Table II(b). We do not detect any slowdown because of the collision in execution pipeline resources (front-end of the pipeline, integer and FP execution units) or L1 data and instruction caches. These resources are private to each core, and, therefore, not shared among co-running tasks. Simultaneously-running applications only share the memory bandwidth. The slowdown we detect when the two mem_bw benchmarks simultaneously execute is very low, only 30%. We also detect interference between L2 and mem_bw benchmarks (10% slowdown). This is due to the fact that the L2 benchmark traverses an array whose size is equal to the size of the L2 cache. Although we try to access the whole cache equally, the replacement policy may put a higher stress on some sets. This causes the L2 benchmark to experience some L2 misses. The memory accesses that miss in the L2 cache collide with mem_bw in the memory bandwidth.

To sum up, the potential slowdown because of inter-task interference on Pentium D processor is fairly low (around 30%). In addition to this, the co-running tasks do not interfere in most of the processor resources. Thus, it is much easier to estimate WCET of co-running time-critical tasks. We conclude that the architectures like Pentium D are good candidates to be used in time-critical environments that require predictability of application execution time.

Table II. Possible Slowdown Because of the Collision in Processor Resources

		Pipeline front end	Pipeline back end (Execution units)						Cache memory			Off-chip resources
		nop	intAdd	intMul	intDiv	fpAdd	fpMul	fpDiv	L1 data	L1 icache	L2	mem_bw
Pipeline front end	nop	2.0	2.0	2.0	1.8	1.6	1.7	1.6	1.7	1.8	1.7	1.6
Pipeline back end (Execution units)	intAdd	2.0	2.0	1.2	1.2	1.2	1.0	1.0	1.0	1.1	1.0	1.0
	intMul	1.3	1.3	1.3	1.1	1.1	1.1	1.0	1.2	1.2	1.1	1.0
	intDiv	1.2	1.2	1.2	1.2	1.1	1.1	1.8	1.2	1.1	1.2	1.1
	fpAdd	1.1	1.2	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
	fpMul	1.0	1.0	1.0	1.1	1.0	1.0	1.0	1.1	1.0	1.1	1.0
	fpDiv	1.0	1.0	1.0	1.5	1.0	1.0	2.0	1.0	1.0	1.0	1.0
Cache memory	L1 dcache	1.0	1.0	1.2	1.0	1.0	1.1	1.0	6.2	1.0	3.6	3.6
	L1 icache	1.1	1.1	1.0	1.0	1.0	1.0	1.0	1.0	2.7	1.0	1.0
	L2	1.0	1.0	1.0	1.0	1.0	1.0	1.0	2.3	1.0	14.1	15.3
Off-chip resources	mem_bw	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	2.5	2.5

(a) Atom

		Pipeline front end	Pipeline back end (Execution units)						Cache memory			Off-chip resources
		nop	intAdd	intMul	intDiv	fpAdd	fpMul	fpDiv	L1 data	L1 icache	L2	mem_bw
Pipeline front end	nop	1.0	1.0						1.0			1.0
Pipeline back end (Execution units)												
Cache memory	L1 dcache											
	L1 icache											
	L2											1.1
Off-chip resources	mem_bw											1.3

(b) Pentium D

		Pipeline front end	Pipeline back end (Execution units)						Cache memory			Off-chip resources
		nop	intAdd	intMul	intDiv	fpAdd	fpMul	fpDiv	L1 data	L1 icache	L2	mem_bw
Pipeline front end	nop	1.0	1.0						1.0			1.0
Pipeline back end (Execution units)												
Cache memory	L1 dcache											
	L1 icache											
	L2										14.4	14.4
Off-chip resources	mem_bw										1.1	1.1

(c) Core2Quad: RS-nn distribution

		Pipeline front end	Pipeline back end (Execution units)						Cache memory			Off-chip resources
		nop	intAdd	intMul	intDiv	fpAdd	fpMul	fpDiv	L1 data	L1 icache	L2	mem_bw
Pipeline front end	nop	1.0	1.0						1.0			1.0
Pipeline back end (Execution units)												
Cache memory	L1 dcache											
	L1 icache											
	L2										1.3	1.3
Off-chip resources	mem_bw											

(d) Core2Quad: Rn-SS distribution

Core2Quad Processor. The Core2Quad processor has two levels of resource sharing [Čakarević et al. 2009], see Figure 3(c). In order to cover all possible scenarios, we analyzed the two benchmark distributions for the Core2Quad processor: (1) RS-nn, when the benchmark under study and the stressing benchmark share L2 cache, and (2) Rn-SS when the benchmark under study and stressing benchmarks do not share L2 cache, but only the memory bandwidth.

Results for the RS-nn distribution are presented in Table II(c). We detect a significant slowdown when the two benchmarks collide in the L2 cache. The L2 resource-stressing benchmark experiences a slowdown of 14.4x when co-running with one more instance of L2 or mem_bw benchmark. In addition to this, we detect a 10% slowdown because of collision in memory bandwidth.

Results for the distribution in which the benchmark under study does not share the L2 cache with stressing co-runners are presented in Table II(d). As for the Pentium D processor, we detect only a low interference in the memory bandwidth (30% slowdown). In addition to interference between mem_bw benchmarks, we also detect a slowdown when reference mem_bw is co-scheduled with two instances of the L2 benchmark. When two stressing L2 benchmarks share the L2 cache, they experience a lot of L2 cache misses that access the main memory and stress memory bandwidth, as the majority of the instructions miss in L2 cache. Therefore, although the L2 benchmark is not designed to stress bandwidth to the main memory, the two benchmark instances that share L2 cache are very bad co-runners for applications using memory bandwidth.

From the analysis of the Core2Quad processor, we see that the potential slowdown that the application experiences because of collision in processor resources with the co-running tasks also depends on the distribution of running benchmarks. When the two running benchmarks share L2 cache, they can experience a significant slowdown because of the collision in this resource. However, if the benchmarks are distributed in such a way that they share only the memory bandwidth, the slowdown we detect is low, around 30%. Therefore, architectures similar to Core2Quad processor are good candidates for systems with timing requirements, as long as time-critical applications do not share the L2 cache memory with co-running tasks.

4.2. Application Sensitivity to Resource Sharing

The results presented in the previous section show that applications simultaneously-running on a processor may experience high variations in execution time because of collision in shared processor resources.

Unlike stressing benchmarks, real applications do not use a single resource during their entire execution, but have different phases in which they use different resources. The usage patterns determine the actual effect that resource sharing has on the applications' execution time. In order to understand which resources are stressed by real applications, we execute them simultaneously with the resource-stressing benchmarks. We define the sensitivity of an application to a specific processor resource as the slowdown that the application experiences when it is co-scheduled with corresponding resource-stressing benchmark.

The results presented in Table III show the sensitivity of applications to resource sharing. Each entry of the table shows the slowdown that the real benchmark, whose sensitivity we measure (listed in the rows of the table), experiences when it is simultaneously executed with different resource-stressing benchmarks (columns).

Atom. The results for the Atom processor are presented in Table III(a). We reach two conclusions: (1) All real benchmarks presented are sensitive to sharing most of the processor resources, and (2) The slowdown experienced by real applications when co-running with resource-stressing benchmarks is up to 22% for STAP Radar, 47% for

Table III. Sensitivity of the Real Benchmarks to a Collision in Processor Resources

	Pipeline front end	Pipeline back end (Execution units)						Cache memory			Off-chip resources
	nop	intAdd	intMul	intDiv	fpAdd	fpMul	fpDiv	L1 data	L1 icache	L2	mem_bw
STAP Radar	1.06	1.07	1.17	1.10	1.02	1.11	1.03	1.03	1.17	1.22	1.06
H264 encode	1.26	1.27	1.35	1.25	1.26	1.26	1.20	1.23	1.39	1.47	1.45
CoreMark	1.02	1.01	1.01	1.18	1.13	1.14	1.08	1.18	1.18	1.18	1.18

(a) Atom

	Pipeline front end	Pipeline back end (Execution units)						Cache memory			Off-chip resources
	nop	intAdd	intMul	intDiv	fpAdd	fpMul	fpDiv	L1 data	L1 icache	L2	mem_bw
STAP Radar	1.00	1.00						1.00			1.02
H264 encode											1.04
CoreMark											1.03

(b) Pentium D

	Pipeline front end	Pipeline back end (Execution units)						Cache memory			Off-chip resources
	nop	intAdd	intMul	intDiv	fpAdd	fpMul	fpDiv	L1 data	L1 icache	L2	mem_bw
STAP Radar	1.00	1.00						1.00		1.00	1.00
H264 encode										1.02	1.02
CoreMark										1.01	1.01

(c) Core2Quad: RS-nn distribution

	Pipeline front end	Pipeline back end (Execution units)						Cache memory			Off-chip resources
	nop	intAdd	intMul	intDiv	fpAdd	fpMul	fpDiv	L1 data	L1 icache	L2	mem_bw
STAP Radar	1.00	1.00						1.00		1.00	1.01
H264 encode											1.01
CoreMark											1.00

(d) Core2Quad: Rn-SS distribution

H264_encode, and up to 18% for the CoreMark benchmark. This is significantly lower than the slowdown we detect for resource-stressing benchmarks (see Table II(a)). This means that, although the underlying architecture has a strong potential to lead to high variation in the execution time of running applications, the particular real benchmarks that we used do not experience a significant slowdown.

Pentium D. The results for the Pentium D processor are presented in Table III(b). Real benchmarks running on the Pentium D are only sensitive to sharing the memory bandwidth. The slowdown that the real benchmarks experience is still lower than the slowdown of resource-stressing benchmarks (see Table II(b)), but the difference is low, below 30%.

Core2Quad. The results for the Core2Quad processor and the task distribution in which reference and stressing benchmarks share the L2 cache are presented in Table III(c). We detect no slowdown for the STAP Radar benchmark, and a very low slowdown (below 2%) when H264_encode and CoreMark are executed with L2 and mem_bw co-runners. This means that the processor has the potential for significant slowdown because of a collision in the shared L2 cache, but that the given set of benchmarks is insensitive to this resource. However, if the target applications change, the possible variation in the execution time due to interference in L2 cache is high.

We repeat the experiments on the Core2Quad processor for the distribution in which real applications and resource-stressing benchmarks do not share the L2 cache, see

Table III(d). We detect a very low slowdown (around 1%) when STAP Radar and H264_encode simultaneously execute with mem_bw stressing benchmark. Again, the slowdown that the real benchmarks experience is lower than the slowdown of resource-stressing benchmarks (see Table II(d)).

One conclusion we reach from the presented results is that the real benchmarks we use in the study show low sensitivity to resource sharing. The other way to understand the results is that the applications under study are under-utilizing some of the processor resources (e.g. the L2 cache), so a less aggressive processor could be used to provide similar performance. It is important to note that these conclusions apply to the benchmarks and processors we analyze in the study and not to the methodology we propose in the paper. The same methodology applied to different real benchmarks or processors could reach different conclusions.

4.3. WCET Estimation

In order to reduce costs, current and future real-time systems follow an integrated approach in which more functionality is executed on the same hardware. This requires hardware that provides higher performance, and that enables incremental timing verification. Incremental timing verification means that a user does not have to verify the timing behavior of all running applications each time a new component (application) is changed or added to the system [EMPRESS]. In that sense, the system is time composable if the WCET estimate of the tasks do not change if any of the tasks in the workload change. In this section, we show how our methodology helps with providing composable WCET estimations.

When directly extending the the standard measurement-based approach (used in single-threaded processors) to MT architectures, the application under study should be executed with different sets of co-running tasks (in different workloads). The longest observed execution time of the application in any workload would then be used to estimate the WCET. We show that this approach does not properly quantify the impact of inter-task interaction to application execution time, and that can lead to an underestimation of WCET.

In the standard measurement-based analysis (classical approach) the application under study is executed in different workloads. In our case, as we analyze three real benchmarks, we can run all possible workloads. In Table IV, we present the slowdown that the benchmarks under study (listed in the rows of the table) experience when they execute with different stressing benchmarks (columns). In the last column of the table (our approach), we also present the slowdown of the benchmark that is computed by measuring benchmark sensitivity to inter-task interference in shared processor resources. This is the maximum slowdown that the benchmark experiences when it is co-scheduled with different resource stressing benchmarks (see Section 2.4.2).

Atom. The results for the Atom processor are presented in Table IV(a). For all three benchmarks, STAP Radar, H264 encode, and CoreMark, the maximum slowdown detected using the classical approach is exceeded in experiments with resource-stressing benchmarks. The difference between the slowdown measured by the classical approach and the slowdown measured using our methodology ranges up to 14% (CoreMark benchmark).

Pentium D. The results for the Pentium D processor are presented in Table IV(b). In this case, we detected no slowdown when pairs of real benchmarks executed on the processor. However, when we execute STAP Radar, H264 encode, and CoreMark with resource stressing benchmarks, we detected a slowdown of up to 2%, 4%, and 3%, respectively. Again, the maximum slowdown detected when using the classical approach was exceeded in experiments with resource-stressing benchmarks.

Table IV. Comparison of Classical Measurement-Based Timing Analysis and Our Approach

	Classical approach				Our approach
	STAP Radar	H264 encode	CoreMark	Max	
STAP Radar	1.21	1.20	1.19	**1.21**	**1.22**
H264 encode	1.39	1.41	1.42	**1.42**	**1.47**
CoreMark	1.03	1.03	1.04	**1.04**	**1.18**

(a) Atom

	Classical approach				Our approach
	STAP Radar	H264 encode	CoreMark	Max	
STAP Radar	1.00	1.00	1.00	**1.00**	**1.02**
H264 encode	1.00	1.00	1.00	**1.00**	**1.04**
CoreMark	1.00	1.00	1.00	**1.00**	**1.03**

(b) Pentium D

	Classical approach				Our approach
	STAP Radar	H264 encode	CoreMark	Max	
STAP Radar	1.00	1.00	1.00	**1.00**	**1.00**
H264 encode	1.00	1.00	1.00	**1.00**	**1.02**
CoreMark	1.00	1.00	1.00	**1.00**	**1.01**

(c) Core2Quad: RS-nn distribution

	Classical approach				Our approach
	STAP Radar	H264 encode	CoreMark	Max	
STAP Radar	1.00	1.00	1.00	**1.00**	**1.01**
H264 encode	1.00	1.00	1.00	**1.00**	**1.01**
CoreMark	1.00	1.00	1.00	**1.00**	**1.00**

(d) Core2Quad: Rn-SS distribution

Core2Quad. The results for the Core2Quad processor are presented in Tables IV(c) and (d). In both task distribution, RS-nn and Rn-SS, we detect no slowdown when workloads composed of real benchmarks execute on the processor. When H264 encode and CoreMark share the L2 cache with resource stressing benchmarks (see Table IV(c)) we detect slowdown of 2% and 1%, respectively. When STAP Radar and H264 encode share memory bandwidth with stressing benchmarks, the detected slowdown is 1%, see Table IV(d).

Real applications have different phases in which they stress different processor resources. Also, the stress that real applications put on each processor resource is not the highest possible stress of that resource. Therefore, experiments in which several real applications simultaneously execute on the processor are unlikely to capture the worst possible inter-task interference. Resource stressing benchmarks put high stress on specific processor resources. Running real applications with resource stressing benchmarks will detect inter-task interference in shared processor resources and properly quantify the impact of this interference on execution time. To summarize, measuring the interference between real applications and resource stressing benchmarks can significantly improve measurement-based methods for estimation of WCET in MT COTS processors.

As we explained in Section 2.1, the design of a worst-stressing benchmark is infeasible in practice. As a part of future work, we plan to understand how the sensitivity

Table V. The Timing Analysis of the Core2Quad Processor: The Improvement of the Mixed Stressing Workload

	Homogeneous stressing workload			Mixed stressing workload			
	L2 L2 L2	mem_bw mem_bw mem_bw	Max	L2 L2 mem_bw	L2 mem_bw mem_bw	mem_bw L2 L2	mem_bw L2 mem_bw
STAP Radar	1.05	1.06	1.06	1.05	1.05	1.05	1.05
H264 encode	1.07	1.07	1.07	1.08	1.07	1.06	1.07
CoreMark	1.02	1.03	1.03	1.04	1.04	1.02	1.04

of real applications to collision in different processor resources independently can be combined to estimate the worst possible slowdown that the application may experience because of interference with tasks co-running on the MT COTS processor. This value could be used to compute a good estimate of the application WCET independently from the set of co-running tasks.

4.4. Mixed Stressing Workloads

When the analyzed MT COTS processor comprises more than two cores or hardware contexts, the application under study can be co-scheduled with different sets of resource-stressing benchmarks. In these experiments, the slowdown that an application experiences is a combination of collision in different processor resources, so this approach improves the estimation of application worst case slowdown.

In our study, Core2Quad is the only processor that supports simultaneous execution of more than two tasks. In order to test mixed criticality stressing workloads, we execute real benchmarks with homogeneous stressing workloads (three instances of L2 or mem_bw benchmark) and with workloads that combine L2 and mem_bw benchmarks. In these experiments, we use only L2 and mem_bw because they are the only benchmarks that stress shared resources of the Core2Quad processor (see Table II). The results of the experiments with mixed stressing workloads are presented in Table V. Each entry of the table shows the slowdown that the benchmark under study (listed in the rows of the table) experiences when it is simultaneously executed with different stressing workloads (columns). For two out of three real benchmarks under study, H264 encode and CoreMark, the slowdown caused by mixed stressing workload exceeds the highest slowdown of homogeneous workloads.

Overall, we show that running workloads composed of real applications may not be sufficient to determine which processor is a better candidate to be used in time-critical environments. In addition to analyzing the measured slowdown experienced by a given set of applications, it is also important to understand the potential slowdown that simultaneously-running applications can experience because of collision in shared resources. We show that the slowdown that applications experience because of collision in shared resources may be low if the applications are insensitive to these resources, even if the potential slowdown is very high.

4.5. Additional Considerations

System Level Timing Analysis. Once the presented methodology is used to quantify the slowdown that an application may experience due to inter-task interferences and WCET of the application is estimated, it is necessary to consider system level issues such as sharing of OS services, process preemption, context switching cost, or task scheduling, and to do a response time analysis. The impact of system level issues

on an application WCET, and the response time analysis are out of the scope of the presented study.

Hybrid WCET Analysis. Several studies propose hybrid WCET analysis as a method for the timing analysis of real-time systems. Hybrid WCET analysis is the combination of static program analysis and the measurement-based techniques [Kirner et al. 2004; Deverge and Puaut 2005; Schaefer et al. 2006; Wenzel et al. 2008]. First, the hybrid WCET analysis statically analyzes the application code. As the analysis of all possible execution paths of real industrial programs is complex or even infeasible, hybrid WCET analysis divides the program into mutually exclusive segments and analyzes each segment separately. For each program segment, the analysis determines the different possible execution paths of the program and generates sets of input data that force the execution of each path. Later, the program is executed on real hardware for different input data sets provided by static analysis. For each set of input data, the user measures the execution time of the corresponding segment of the code. Finally, the WCET of the whole program is computed based on the measured execution time of different program segments. In hybrid WCET analysis, the measurements are performed on real hardware, so a detailed model of the architecture under study is not required. This is its main advantage when compared to the static WCET analysis. To the best of our knowledge, current hybrid WCET analysis studies are focused on WCET estimation of applications running on single-threaded processors. As a part of our future work, we plan to use the presented methodology to detect the possible slowdown of different program segments and to extend the current hybrid WCET analysis to MT COTS architectures.

In all the experiments presented in the paper, the slowdown that the benchmarks experience is measured from the entry until the termination of the benchmark execution (*end-to-end* measurements). However, the presented methodology can be easily adjusted to focus on different program segments, so it can be used in hybrid WCET analysis or when only some program segments have time-critical requirements. In this case, the set of resource-stressing benchmarks and the set of experiments would remain the same: the application under study should be executed in isolation and with the resource-stressing co-runners. The only difference is that, in this case, instead of measuring the execution time (slowdown) of the whole application, the user should instrument only the program segments under study. Methods for low-overhead instrumentation of different application segments have already been proposed [Rieder et al. 2007].

5. RELATED WORK

Despite the benefits that MT COTS processors may offer in embedded real-time systems these architectures are still not widely used in real-time systems because the timing analysis is too complex. To the best of our knowledge, our study presents the first systematic approach for measurement-based timing analysis of time-critical applications running on MT COTS architectures. Several studies and projects analyze collision in shared hardware resources among tasks co-scheduled on MT architectures and the impact of this collision on WCET analysis.

Schliecker et al. [2008] and Pellizzoni et al. [2010b] analyze the delay of memory access in systems where several simultaneously-running tasks share the main memory. The proposals require a detailed profiling of application memory access pattern and a deep understanding of the memory arbitration policy. Although both proposals can be applied only in systems where the main memory is the only shared resource (authors assume non-shared caches), we believe that these studies are a very good starting

point to better understand the possible use of multithreaded processors in time-critical systems.

Cullmann et al. [2010] analyze the design of future multithreaded processors for time-critical systems. The authors show that some processor designs make the timing analysis infeasible and suggest design principles for making multithreaded architectures predictable. This analysis is complementary to our study. Based on the theoretical analysis, the authors give guidelines for the design of predictable architectures, while we present the measurement-based approach to determine if a given architecture is a good candidate for time-critical systems.

Several projects propose hardware solutions to deal with the inter-task interferences on WCET in MT architectures [Across; Hansson et al. 2009; Genesys; Merasa; Pret; Tta; Predator]. These projects make a wide range of proposals. Some suggest preventing inter-task interferences by assigning each task a subset of resources and not allowing other user tasks to use the resources. This can be implemented by splitting the hardware resource temporally or spatially. Other proposals suggest actually allowing tasks to share hardware resources and defining the boundaries of this interaction so that the maximum effect of the interaction on the WCET is known. Although these proposals are different, the common factor is that they all propose changes in hardware to reach their objectives. In this paper, our objective is to show how, without any change, MT COTS processors can assess the challenges and the requirements imposed in a real-time environment, mainly time predictability.

Several studies focus on Measurement-Based Timing Analysis (MBTA) for single-threaded architectures. The studies propose an improvement of the accuracy and coverage of MBTA by using static code analysis. Schaefer et al. [2006] propose measuring execution time at basic block level and using this data to estimate WCET of the whole program. Deverge and Puaut [2005] propose using structural testing methods to generate input data for experiments used in measurement-based WCET analysis. Kirner et al. [2004] use static program analysis to generate test data that cover different execution paths. Authors also propose a decomposition of program paths into smaller parts (subpaths) and using an independent measurement-based analysis for each subpath. Finally, the WCET estimate of the whole program is calculated based on the execution time of each subpath. Wenzel et al. [2008] present a similar approach: they propose a decomposition of the program into segments and doing a timing analysis for each segment. The authors also propose an approach for good program segmentation–one that balances the number of program segments with the average number of paths per segment. Rieder et al. [2007] analyze different approaches for measuring the execution time of program segments. The authors propose an external Runtime Measurement Device and suggest the integration of this device into the analysis framework that automatically collects the data needed for measurement-based WCET analysis. All the above studies propose improvements of end-to-end measurement-based timing techniques for single-threaded processors. In this paper, we extend measurement based timing analysis for multithreaded processors and show how it can be used to determine which architecture is more suitable for systems with timing requirements.

6. CONCLUSIONS

COTS processors are increasingly being considered in the design of systems with timing requirements. MT COTS architectures are of special interest due to good performance-per-watt ratio, high performance opportunities, and their suitability for embedded architectures in which several functions are integrated into the same processor.

Unfortunately, despite the benefits that MT COTS may offer in embedded real-time systems, the time-critical market has not yet embraced a shift toward these architectures. The main challenge with MT COTS architectures is the difficulty when

predicting the execution time for simultaneosly running time-critical tasks. Providing a timing analysis for real industrial applications running on MT COTS processors becomes extremely difficult because the execution time of a task, and hence its WCET depends on the interference with co-running tasks in shared processor resources.

In this paper, we have shown that the measurement-based timing analysis used for single-threaded processors cannot be directly extended for MT COTS architectures. Running workloads composed of real applications may not be sufficient to capture the possible slowdown that applications experience due to interferences. This is due to the fact that the applications used may be insensitive to interference in processor shared resources. We propose a methodology that quantifies the slowdown that a task may experience because of collision with co-runners in shared resources of MT COTS processor. To that end, we developed a set of specific resource-stressing benchmarks and propose a measurement-based approach to determine the possible slowdown caused by inter-task interferences. The two main applications of our methodology are: (1) The methodology helps to determine which architecture among different MT COTS processors is more suitable to be used in time-critical environments, (2) Our methodology shows the potential variation in execution time a task in a workload may experience if any of the co-runner changes.

We also presented several case studies in which we analyze three MT COTS architectures with different degrees of shared resources. We show that, for a given workload composed of several real-time benchmarks, all three types of architecture show low interference among co-running tasks and stable execution time. However, our method shows that potential variation in the execution time of applications is different for each architecture under study, and that not every one of the three architectures are good candidates to be used in time-critical systems.

ACKNOWLEDGMENTS

The authors want to thank the anonymous reviewers for their insightful feedback on this work.

REFERENCES

ACROSS. ARTEMIS CROSS-domain architecture. http://www.across-project.eu.

Andritsopoulos, F., Papastefanos, S., Georgakarakos, G., and Doumenis, G. 2007. Reliable multicast H.264 video streaming for surveillance applications. In *Proceedings of the IEEE 18th International Symposium on Personal, Indoor and Mobile Radio Communications*.

Atom Z530. 2009. Intel® Atom™ processor Z5xx series. http://download.intel.com/design/processor/datashts/319535.pdf.

AUTOSAR. AUTomotive open system architecture. http://www.autosar.org.

Baker, T. 2002. Lessons learned integrating COTS into systems. In *COTS-Based Software Systems*, Lecture Notes in Computer Science, Springer.

Butenhof, D. R. 1997. *Programming with POSIX Threads*. Addison-Wesley Professional.

Čakarević, V., Radojković, P., Verdú, J., Pajuelo, A., Cazorla, F. J., Nemirovsky, M., and Valero, M. 2009. Characterizing the resource-sharing levels in the UltraSPARC T2 processor. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-42)*.

Chevalier, F. L. and Maria, S. 2006. STAP processing without noise-only reference: requirements and solutions. In *Proceedings of the International Conference on Radar*.

Core2Quad. Intel® Core™2Quad extreme processor QX9000 and Intel® Core™Quad processor Q9000 series datasheet. http://www.intel.com/design/processor/datashts/318726.htm?wapkw=(datasheet+q9000).

CoreMark. The embedded microprocessor benchmark consortium benchmark suite. http://www.coremark.org.

Cullmann, C., Ferdinand, C., Gebhard, G., Grund, D., Maiza, C., Reineke, J., Triquet, B., and Wilhelm, R. 2010. Predictability considerations in the design of multi-core embedded systems. In *Proceedings of the Symposium on Embedded Real Time Software and Systems (ERTS)*.

DETTI, A., LORETI, P., BLEFARI-MELAZZI, N., AND FEDI, F. 2010. Streaming H.264 scalable video over data distribution service in a wireless environment. In *Proceedings of the International Symposium on a World of Wireless, Mobile and Multimedia Networks*.

DEVERGE, J.-F. AND PUAUT, I. 2005. Safe measurement-based WCET estimation. In *Proceedings of the 5th International Workshop on Worst-Case Execution Time Analysis*.

DOUCETTE, D. AND FEDOROVA, A. 2007. Base vectors: A potential technique for microarchitectural classification of applications. In *Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture*.

DURANTON, M., YEHIA, S., DE SUTTER, B., DE BOSSCHERE, K., COHEN, A., FALSAFI, B., GAYDADJIEV, G., KATEVENIS, M., MAEBE, J., MUNK, H., NAVARRO, N., RAMIREZ, A., TEMAM, O., AND VALERO, M. The HiPEAC vision. High performance and embedded architecture and compilation. http://www.HiPEAC.net.

EEMBC. The embedded microprocessor benchmark consortium benchmark suite. http://www.eembc.org.

EMPRESS. Incremental verification and validation practices, EMPRESS public deliverable. http://www.empress-itea.org/.

FRITTS, J. E., STEILING, F. W., AND TUCEK, J. A. Mediabench II video: Expediting the next generation of video systems research. http://mathcs.slu.edu/~fritts/papers/fritts_spie05_mbvideo.pdf.

GENESYS. GENeric Embedded SYStem Platform. http://www.genesys-platform.eu.

GEREFFI, G. 1999. *A Commodity Chains Framework for Analyzing Global Industries*. Duke University.

GIOIOSA, R., PETRINI, F., DAVIS, K., AND LEBAILLIF-DELAMARE, F. 2003. Analysis of system overhead on parallel computers. In *Proceedings of the ACM/IEEE Conference on Supercomputing*.

HANSSON, A., GOOSSENS, K., BEKOOIJ, M., AND HUISKEN, J. 2009. Compsoc: A template for composable and predictable multi-processor system on chips. *Trans. Des. Automat. Electron. Syst.*

HIGGINS, K. J. 2004. Video IP project boosts networks profile. *Netw. Comput*.

INTEL CORPORATION 2011. *Intel® 64 and IA-32 Architectures Software Developer's Manual*.

KIRNER, R. AND PUSCHNER, P. 2008. Obstacles in worst-case execution time analysis. In *Proceedings of the 11th IEEE International Symposium on Object Oriented Real-Time Distributed Computing (ISORC '08)*. 333–339.

KIRNER, R., PUSCHNER, P., AND WENZEL, I. 2004. 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)*.

KIRNER, R., WENZEL, I., RIEDER, B., AND PUSCHNER, P. 2005. Using measurements as a complement to static worst-case execution time analysis. In *Intelligent Systems at the Service of Mankind*, Vol. 2, UBooks Verlag.

MERASA. Multi-core execution of hard real-time applications supporting analysability. http://www.merasa.org.

MEZZETTI, E. AND VARDANEGA, T. 2011. On the industrial fitness of WCET analysis. In *Proceedings of the 11th International Workshop on Worst-Case Execution Time Analysis (WCET '11)*. C. Healy, Ed., OCG, Austrian Computer Society.

NATALE, M. D. AND SANGIOVANNI-VINCENTELLI, A. 2010. Moving from federated to integrated architectures in automotive: The role of standards, methods and tools. *Proc. IEEE*.

PAOLIERI, M., QUIÑONES, E., CAZORLA, F. J., BERNAT, G., AND VALERO, M. 2009a. Hardware support for WCET analysis of hard real-time multicore systems. In *Proceedings of ISCA '09*.

PAOLIERI, M., QUINONES, E., CAZORLA, F. J., AND VALERO, M. 2009b. An analyzable memory controller for hard real-time CMPs. In *Embedded System Letters*.

PELLIZZONI, R., SCHRANZHOFER, A., CHEN, J.-J., CACCAMO, M., AND THIELE, L. 2010a. Worst case delay analysis for memory interference in multicore systems. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '10)*.

PELLIZZONI, R., SCHRANZHOFER, A., CHEN, J.-J., CACCAMO, M., AND THIELE, L. 2010b. Worst case delay analysis for memory interference in multicore systems. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '10)*.

PENTIUM, D. 2007. Intel® Pentium® D processor 900 Sequence and Intel® Pentium® processor extreme edition 955, 965. http://www.intel.com/Assets/PDF/datasheet/310306.pdf.

PETRINI, F., KERBYSON, D. J., AND PAKIN, S. 2003. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q. In *Proceedings of the ACM/IEEE Conference on Supercomputing*.

PREDATOR. PREDATOR consortium. http://www.predator-project.eu.

PRET. Precision timed (PRET) machines. http://chess.eecs.berkeley.edu/pret.

PUSCHNER, P. AND BURNS, A. 2000. A review of worst-case execution-time analysis. *J. Real-Time Syst. 18,* 2/3, 115–128.

RADOJKOVIĆ, P., CAKAREVIĆ, V., VERDÚ, J., PAJUELO, A., GIOIOSA, R., CAZORLA, F., NEMIROVSKY, M., AND VALERO, M. 2008. Measuring operating system overhead on CMT processors. In *Proceedings of the 20th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD '08)*.

RIEDER, B., WENZEL, I., STEINHAMMER, K., AND PUSCHNER, P. 2007. Using a runtime measurement device with measurement-based WCET analysis. In *Proceedings of the International Embedded Systems Symposium (IESS'07)*.

SCHAEFER, S., SCHOLZ, B., PETTERS, S. M., AND HEISER, G. 2006. Static analysis support for measurement-based WCET analysis. In *Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*.

SCHLIECKER, S., NEGREAN, M., NICOLESCU, G., PAULIN, P., AND ERNST, R. 2008. Reliable performance analysis of a multicore multithreaded system-on-chip. In *Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis (CODES+ISSS '08)*.

TTA. Time-triggered architecture. http://www.vmars.tuwien.ac.at/projects/tta.

UNGERER, T., CAZORLA, F., SAINRAT, P., BERNAT, G., PETROV, Z., ROCHANGE, C., QUINONES, E., GERDES, M., PAOLIERI, M., AND WOLF, J. 2010. Merasa: Multi-core execution of hard real-time applications supporting analysability. *IEEE Micro*.

WATKINS, C. AND WALTER, R. 2007. Transitioning from federated avionics architectures to Integrated Modular Avionics. In *Proceedings of the 26th Digital Avionics Systems Conference (DASC '07)*.

WENZEL, I., KIRNER, R., RIEDER, B., AND PUSCHNER, P. 2008. Measurement-based timing analysis. In *Proceedings of 3rd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*.

WILHELM, R., ENGBLOM, J., ERMEDAHL, A., HOLSTI, N., THESING, S., WHALLEY, D., BERNAT, G., FERDINAND, C., HECKMANN, R., MITRA, T., MUELLER, F., PUAUT, I., PUSCHNER, P., STASCHULAT, J., AND STENSTRÖM, P. 2008. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst*.

Received July 2011; revised October 2011 and December 2011; accepted January 2012