Assessing the Suitability of the NGMP Multi-core Processor in the Space Domain

Mikel Fernández[†]
mikel.fernandez@bsc.es

Roberto Gioiosa[†]
roberto.gioiosa@bsc.es

Eduardo Quiñones[†]
eduardo.quinones@bsc.es

Luca Fossati[‡]
luca.fossati@esa.int

Marco Zulianello[‡]
marco.zulianello@esa.int

Francisco J. Cazorla[†,*]
francisco.cazorla@bsc.es

[†] Barcelona Supercomputing Center
C/Jordi Girona, 31
08034 Barcelona (Spain)

[‡] European Space Agency
ESTEC, Postbus 299
2200 AG Noordwijk (The Netherlands)

[*] IIIA-CSIC
Campus UAB
08193 Bellaterra
Cerdanyola del Valles (Spain)

ABSTRACT

Multi-core processors are increasingly being considered as a means to provide the performance required by future safety-critical embedded systems. In this line, Aeroflex Gaisler has developed, in conjunction with the European Space Agency, the NGMP, a quad-core processor to be used in the future space missions of the Agency. Unfortunately, the use of multi-core processors in industrial domains is not straightforward since it poses various challenges on the timing behavior of the system. This is mainly due to the interferences tasks suffer when accessing hardware shared resources and which can affect their WCET. Although the effect of inter-task interferences in multi-core shared resources on real-time applications has received attention from academia, most of the solutions proposed require hardware changes. The lack of quantitative studies of the slowdown on applications' performance caused by inter-task interferences on real COTS multi-core processors, limit their use by industry.

As a first step to understand the effect of inter-task interference in real COTS processors, this paper evaluates the timing predictability properties of the NGMP. In particular, we measure the maximum variation on tasks' execution time due to inter-task interferences accessing NGMP's shared hardware resources. To that end, we use a set of specialized micro-benchmarks designed to stress specific processor shared resources. The results of this can be useful for developing interference-aware WCET estimation methodologies and scheduling algorithms for real-time applications running on embedded multi-core processors.

1. INTRODUCTION

The market for Critical Real-Time Embedded Systems (CRTES) has experienced an unprecedented growth in recent years, and is expected to grow in the foreseeable future [5][11]. Because of the competition on functional value, CRTES industry is faced with rising demands for greater performance and hence increased computing power, as well as to reduce the number of processing units used in the system [15]. Such high performance requirements could be met by designing more complex processors with longer pipelines, out of order execution, and higher clock frequency. However, using complex processor cores in CRTES designs is problematic because they could introduce timing anomalies [14] due to the their non-deterministic run-time behaviour. Moreover, the high energy requirements of such complex processors do not satisfy the low-power constraints and the severe cost limitations common in most embedded systems.

Another way to meet high performance requirements is by means of multi-core processors. Multi-cores offer better performance per watt than single-core processors, while maintaining a relatively simple processor design. Moreover, multi-core processors ideally enable co-hosting applications with different requirements (e.g. high data processing demand and stringent time criticality). Co-hosting non-safety and safety critical applications on a common powerful multi-core processor brings many advantages to the embedded system market, allowing to schedule a higher number of tasks on a single processor hence maximizing the hardware utilization while cost, size, weight and power requirements are reduced. This is especially important for the space industry where weight reduction is essential.

Even if multi-core processors may offer several benefits to embedded systems, their use is not straightforward. First of all, it is necessary to prevent that one application corrupts the state of other applications ensuring that low-criticality applications cannot affect high-criticality ones. This can be accomplished by providing software isolation and has been done within the space domain through the use of hypervisors [3]. CRTES also require guarantees on the timing predictability of the system. Unfortunately, multi-core processors are much harder to time analyse than single-core processors, because of inter-task interferences accessing hardware shared resources (shared bus, shared cache, main memory, etc.): Inter-task interferences appear when two or more tasks that share a resource try to access it at the same time. To handle this contention an arbitration mechanism is required, potentially affecting the execution time and WCET of running tasks. As a result, providing a meaningful timing anal-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EMSOFT'12, October 7-12, 2012, Tampere, Finland.
Copyright 2012 ACM 978-1-4503-1425-1/12/09 ...$15.00.

ysis becomes extremely difficult because the execution time of a task may change depending on the other tasks running simultaneously. Hence, understanding and controlling inter-task interferences is mandatory to provide time analysability, so that the activity of tasks has a bounded effect on the execution of the most critical ones.

In this paper we evaluate the suitability for CRTES of the latest multi-core processor used by the European Space Agency, the NGMP (Next Generation Multi-Purpose Microprocessor). The NGMP is a LEON4-based quad-core processor, developed by Aeroflex Gaisler together with the European Space Agency [2]. The NGMP has private data and instruction caches per core. Each core access to the shared L2 through the AMBA AHB processor bus [1]. The memory bandwidth is also shared by all cores. In particular, this paper focuses on providing accurate figures on the execution time variation introduced by four of the main sources of inter-task interferences: the AMBA AHB processor bus, the shared L2 cache, the shared memory controller and the write-through policy of the L1 data cache.

Approach: To reach these objectives, we use a set of specialized micro-benchmarks [9][10][16][25] designed to stress each of the processors resources considered in this paper. The micro-benchmarks are designed to stress a particular processor resource like the L1 data cache, or the bus. By means of those micro-benchmarks we identify in which hardware shared resources the interaction among tasks significantly affects their execution time. The higher the variability due to inter-task interferences the lower the suitability of the architecture to real-time environments. We do our analysis on both Linux and RTEMS [29] with a two fold-objective: increasing the confidence on the results obtained in our study and determining whether inter-task interferences effects are the same under both operating systems.

Our results show that applications may experience high execution time variations due to inter-task interferences when executed in the NGMP under both RTEMS and Linux. In particular, we observe the following:

- When several bus-hungry tasks try to access at the same time the AMBA AHB processor bus, they may suffer a slowdown of up to 1.83x in Linux and 1.95x in RTEMS.

- The combined effect of the interaction in the memory controller and the AMBA AHB processor bus introduces a slowdown on applications' performance of 2.6x for Linux and 3.4x for RTEMS.

- The combined effect of the three resources, i.e. AMBA AHB processor bus, L2 cache and memory controller makes the execution time of applications increase up to 4.3x for Linux and more than 9x for RTEMS.

- Finally, and more surprisingly, applications with high number of stores may suffer slowdowns higher than 19x. This is due to the fact that the first level data cache is write-through, so every single store has to go to L2, suffering significant slowdowns.

The different effect of inter-task interferences under Linux and RTEMS is due to the fact that under RTEMS tasks are less affected by the OS operation (noise), having higher performance when run in isolation. As a result, when the task runs with other micro-benchmarks its performance degradation is higher.

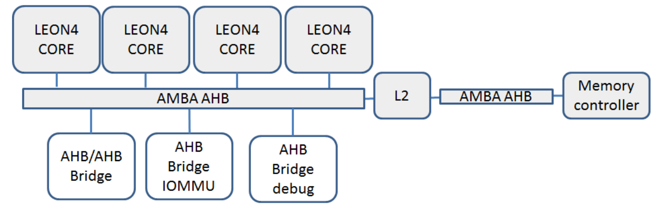

Figure 1: Block diagram of the part of the NGMP architecture studied in this paper

We confirm our results on the micro-benchmarks with real-time benchmarks, EEMBC Automotive, on which we observe a maximum slowdown of 5.5x. We identify the average number of stores per instruction as the main factor determining the sensitivity of an application to inter-task interferences in the NGMP architecture.

Contribution: Although the effect of inter-task interferences in multi-core hardware shared resources on real-time applications have received significant attention by the research community (e.g. [28] [15][19][6][23]), most of the works have focused on simulation environments or involve hardware changes. Currently, the lack of quantitative studies on inter-task interferences on real COTS multi-core processors, limit their potential use by industry. This paper represents an step in that direction: We evaluate, under heavy load conditions, the inter-task interference impact of NGMP shared resources on applications' execution time. On the one hand, this study is a first step towards providing safe WCET bounds to the execution time of applications in the NGMP. On the other hand, this can also help determining how applications have to be sheduled to reduce inter-task interferences effect on WCET bounds. Both these topics are left as future work.

The rest of this paper is organized as follows. In Section 2 we show introduce the NGMP architecture. In Section 3 we explain the micro-benchmarks we use in this study. Section 4 presents our experimental setup. Section 5 shows the main results we have obtained under Linux and Section 6 the ones we obtained under RTEMS. Section 7 discusses the applicability of the results of this study. Section 8 describes the related work. Finally, Section 9 summarizes the main conclusions of this study and future lines of research.

2. INTRODUCTION TO THE NGMP

The NGMP is a SPARC V8 quad-core processor, developed by Aeroflex Gaisler and the European Space Agency featuring the latest LEON core design, LEON4, which provides a significant performance increase compared to earlier LEON processors [8][4]. The LEON4 is a 32-bit 7-stage pipeline processor, comprising an always-taken branch predictor and private data and instruction caches of 16 KB each. Both the instruction and the data cahche have 32-byte lines and are 4-way associative. The data caches employs a write-through with no-allocate miss policy.

In the NGMP, each LEON4-core connects to a shared 256 KB L2 cache through an AMBA AHB processor bus [1] with a 128-bit data width and round-robin arbitration policy. The L2 cache uses the LRU replacement algorithm implementing a write-back, write-allocate policy. The L2

cache connects to the memory controller through a single memory channel shared by all cores (see Figure 1).

Three AHB bridges connect the AHB processor bus to other I/O and debugging specific AMBA buses: (1) a 128-bit to 32-bit unidirectional AHB-to-AHB bridge from debug bus to processor bus, (2) a 128-bit to 32-bit unidirectional AHB-to-AHB bridge from processor bus to slave I/O bus and (3) a 32-bit to 128-bit unidirectional AHB-to-AHB bridge with IOMMU from master I/O bus to processor bus. Although these three specific buses are not the focus of our study, they should be considered when providing timing analysis for the NGMP.

The NGMP provides a set of performance counters that enable collecting run-time information about certain events of the processors. In this study we have used the following performance counters: data and instruction cache misses, L2 cache misses, total number of executed instructions, number of memory operations, number of executed cycles, and processor AMBA bus usage.

3. MICRO-BENCHMARKS

In a multi-core architecture, the performance of one process tightly depends on the other processes running simultaneously and on their specific execution phases. Programs go through different execution phases in which the effect of inter-task interference may vary significantly. Evaluating all possible combinations of programs that might be running together and their different execution phases is not feasible.

Instead, in this paper we execute the application under analysis into different high-load inter-task interference scenarios. In order to build a basic knowledge of inter-task interference effects we developed a set of synthetic micro-benchmarks, each of them stressing a specific processor characteristic. Benchmarks are designed to cause high load on a specific hardware resource, allowing us to isolate independent behaviours of a specific shared resource. Furthermore, micro-benchmarks have been designed to provide higher flexibility and compatibility with other processor architectures due to their simplicity.

3.1 Stressing the cache hierarchy

In most shared-memory, symmetric multi-core architectures, like the NGMP, running tasks interact in the cache hierarchy. For this reason we have designed and implemented five micro-benchmarks which allow exercising the different hardware shared resource that form the cache hierarchy of the NGMP. Next, we define the characteristics of each micro-benchmark. Note that micro-benchmarks are independent tasks that do not share any data:

- *L1.* This micro-benchmark accesses a vector with a data footprint smaller than 16 KB, fitting completely inside the data cache. Hence, it does not stress the L2 of the AHB processor bus.
- $L2_{40}$. It accesses a vector with a data footprint of 40 KB, so most loads miss in data cache and hit in L2 cache. Hence, it stresses the data cache, the processor AHB AMBA bus and, to a lesser extent, the L2 cache.
- $L2_{200}$. It accesses a vector with a data footprint of 200KB with the purpose of generating L2 cache hits when run alone and L2 cache misses when run together with other L2 stressing micro-benchmarks. Hence, it stresses the data cache, the processor AHB AMBA bus and the L2 cache.
- $L2_{miss}$. It accesses a vector with a data footprint of 1 MB, generating systematic misses in the L2 cache. Hence, it stresses the data cache, the processor AHB AMBA bus, the L2 cache, the memory AHB AMBA bus and the memory controller.
- $L2_{st}$. This benchmark simply writes to a 40KB vector. Hence, it mainly executes store operations.

Table 1: code listing for $L2_{40}$, $L2_{200}$ and $L2_{miss}$

(a) Initialization of the array to be accessed by the micro-benchmark

```
for(cnt=0; cnt<array_size; cnt+=stride){
  if(cnt<array_size-stride)
    M[cnt] = (int*)&M[cnt+stride];
  else
    M[cnt] = (int*)M;
}
```

(b) Actual code of the micro-benchmark

```
a=&M[0];
for (i=0; i<it; i++) {
  b=*a; a=*b
  b=*a; a=*b
        ... // repeated 126 more times
}
```

All micro-benchmarks, except $L2_{st}$, which uses direct addressing, employ *pointer chasing* to access memory: in each location of the vector we store the address of the next memory location to be accessed. By doing so, no instructions are required to compute the memory address to be accessed. Moreover, the data inside the vector is stored such that the stride controls which particular cache level (L1 or L2) is accessed. It is also important to remark that micro-benchmarks are small enough to fit inside the instruction cache. In fact, in this study we do not consider the effect of the L1 instruction cache, we rather focus on the L1 data and the L2 cache.

The first four micro-benchmarks have the same structure, see Table 1. They are mainly composed of loads (more than 95% of the total instruction count), that access a vector with a variable data footprint. The micro-benchmark code is contained inside a main loop.

In the initialization code, see Table 1(a), the *stride* and the *array_size* determine how often the micro-benchmark will hit/miss in each cache level: The stride is always set to prevent several accesses to the same cache line. The *array_size* is set to ensure that a benchmark hits/misses in a desired cache level. In the code of the micro-benchmark, Table 1(b), every execution of the loop body, called a micro-iteration, contains 128 loads, 1 cmp, 1 br and 1 nop instructions. The number of micro-iterations is so that the percentage of control operations of the loop is less than 5%.

The structure of the $L2_{st}$ micro-benchmark is a bit different, see Table 2. For this benchmark GCC inline assembler syntax is used: *%0* is replaced by *data*, a variable containing the value to be stored, and *%1* is replaced by *st_pointer*, a pointer to access the allocated array. The destination memory address is defined by a pointer plus an immediate value (*st_pointer* plus the numeric value in Table 2).

Table 2: code listing for $L2_{st}$ microbencmarks

(a) Initialization of the array to be accessed by the st microbenchmark

```
st_array    =
st_pointer = (int*)malloc(array_size);
```

(b) Actual code of the st microbenchmark

```
#define STRIDE 32
int data = 4;   // data may be any integer value
for (i=0; i<it; i++) {
  __asm__ __volatile__(
            "st %0, [%1]"          "\n\t"
            "st %0, [%1+32]"       "\n\t"
            "st %0, [%1+64]"       "\n\t"
                ...
            "st %0, [%1+4032]"     "\n\t"
            "add %1, 4064, %1"     "\n\t"
            "st %0, [%1]"          "\n\t"
            "st %0, [%1+32]"       "\n\t"
                ... // total of 508 stores
  :
  : "r"(data), "r"(st_pointer)
  );

  if (st_pointer+509*STRIDE >=
     st_array+array_size)
        st_pointer = st_array;
  else
        st_pointer +=STRIDE;
}
```

Each store instruction has a different immediate value, where each immediate equals the previous one plus the stride. In Table 2, a 32-byte stride is used. This methodology presents the limitation that the maximum allowed immediate value in the target architecture is 4095. This limitation is overcame by reseting the immediate value and increasing *st_pointer*. After completion of an iteration, *st_array* bounds are checked to make sure it does not overflow during the next iteration.

All micro-benchmarks were compiled with gcc with the -O2 option and their object code was verified in order to guarantee that the benchmarks retain the desired characteristics. Note that that all micro-benchmarks are independent processes so they do not share data, even when several copies of the same benchmark run at the same time.

4. EXPERIMENTAL SETUP

4.1 Experimental Methodology

All the experiments presented in this paper have been obtained using a ML510 embedded development platform. This platform contains a Virtex 5 FPGA that implements a preliminary design of the NGMP operating at 70Mhz. Due to FPGA space limitation, each core lacks the floating point unit. The NGMP under this implementation comprises a 64-bit data DDR2-800 memory interface with a channel frequency of 140 MHz. This frequency-ratio between the NGMP and the memory has important effect on our conclusions: the effect of memory controller interferences on the execution time of programs will be higher in the final implementation of NGMP than in our FPGA implementation. Unfortunately, at the time of carrying out this study, the only available implementation of an NGMP processor is on a FPGA.

We designed an Execution Infrastructure that allows the execution of workloads comprised of different applications and it is compatible with both operating systems, i.e. Linux and RTEMS. The Execution Infrastrcutre facilitates the study of Linux- and RTEMS-based applications under different inter-task interferences scenarios by measuring its execution time variation.

For the experiments on Linux we use Linux 2.6.36 operating system (OS). As our experiments were executed on a full-flegded OS, they were designed to provide reliable results and minimize the impact of the OS to our measurements [12, 21, 26]. To avoid task migration among different cores of a processor, we bound each benchmark to the corresponding core using the *sched_setaffinity* system call.

We also use RTEMS 4.10 provided by Aeroflex Gailser as part of the Development package for NGMP. For the experiments presented in this paper, RTEMS was configured as a multi-processor application with 4 cores, where each core was assigned a 8MB memory segment, 4MB of which were used as stack. Three global semaphores were used to synchronize cores, and a single global RTEMS partition was used to allocate output buffers. Each core uses a GPTIMER as a system clock generator, which is configured to provide millisecond resolution.

4.2 Application Workloads

In order to empirically evaluate the maximum application delay due to inter-task interferences, we have used two different types workloads: (1) *micro-benchmark workloads*, composed of only micro-benchmarks, and (2) *EEMBC workloads* composed of both EEMBC Autobench benchmarks [22] and micro-benchmarks. The former allows us to generate the worst possible delay *any* application might suffer due conflicts in the specific resource (or set of resources) exercised by the micro-benchmark. The latter allows us to see the effect that the specific resource (or set of resources) on a specific application, in this case an EEMBC benchmark.

The micro-benchmark workloads are formed by four different workloads that will continuously stress a given resource (or set of resources):

1. $L2_{40}$ workload to analyze processor AHB AMBA bus that connects cores to L2.

2. $L2_{miss}$ workload to analyze the memory controller and memory AHB AMBA bus that connects the L2 cache with the main memory and the processor AHB AMBA bus.

3. $L2_{200}$ workload to analyze the memory controller and memory AHB AMBA bus that connects the L2 cache with the main memory, the shared L2 cache and the processor AHB AMBA bus.

4. $L2_{st}$, $L2_{40}$, $L2_{200}$ and $L2_{miss}$ workloads to analyze the effect of write-through policy in the L1 data cache.

The EEMBC workloads are formed by multiple copies of the same benchmark, as well as one EEMBC benchmark and multiple micro-benchmarks. EEMBC Autobench suite is a well-known benchmark suite that reflects the current real world demands of embedded systems.

Table 3: L1 data cache miss ratio, L2 miss ratio and processor AHB AMBA bus utilization per process when we simultaneously run 1, 2 and 4 copies of the same micro-benchmark, each on a different core.

	Data cache miss (%)			L2 cache miss (%)			% of stores
No. of copies →	1	2	4	1	2	4	1
L1	0.01	0.01	0.01	0.01	0.01	0.01	0.04
$L2_{40}$	99.7	99.7	99.6	0.08	0.06	5.94	0.15
$L2_{200}$	99.5	99.0	98.1	31.5	88.4	98.6	0.20
$L2_{miss}$	99.2	99.0	98.1	100	99.6	98.5	0.36
$L2_{st}$	0.02	0.09	0.19	0.03	0.04	0.04	95.3

Note that in this paper we are not interested in scheduling aspects so all the experiments we run comprise workloads of 4 or less applications running simultaneously, i.e. at most one application per core.

4.3 Metrics

In order to evaluate the impact that inter-task interferences have on the execution time, the application under study is, in a first step, run in isolation, i.e. without suffering any interferences coming from other tasks. In a second step, the application is run simultaneously with other applications, i.e. within a workload. Finally, we measure the inter-task interferences in shared resources and their effect in execution time as the ratio between the execution time of the program in isolation and its execution time when it runs as part of a workload, as shown in Equation 1, where $Exec.Time_{wokload}$ is the execution of the application under consideration when it runs in multi-core mode as part of a workload.

$$Execution\ Time\ Slodown = \frac{Exec.Time_{wokload}}{Exec.Time_{isolation}} \quad (1)$$

Execution time is measured using the *gettimeofday*() call for Linux. On RTEMS a single *GPTIMER* was used, and a different timer assigned to each core.

Other metrics are also considered such as L1 and L2 cache misses, processor AHB AMBA bus utilization, number of load and store instructions executed, and total number of instructions executed. Average cache miss rates are measured per 100 instructions.

5. RESULTS UNDER LINUX

This section evaluates the execution time variability of both workloads, i.e. micro-benchmarks and EEMBC workloads, running under Linux.

5.1 micro-benchmark validation

In order to guarantee that micro-benchmarks generate significant inter-task interferences, it is fundamental to provide arguments about the fact the they significantly stress shared processor resources. To that end, we simultaneously executed several copies (1, 2 and 4) of each micro-benchmark, and measure the slowdown they suffer due to inter-task interferences using the performance monitoring support provided by the NGMP. For each benchmark we measure (1) the number of memory operations per instruction, (2) the miss rate in the data cache, (3) the miss rate in the L2 cache and (4) the processor AHB AMBA bus utilization. Table 3 summarizes the behaviour of each micro-benchmark.

As expected, the L1 micro-benchmark does not stress shared resources at all, having a data cache miss rate of almost 0. Instead, $L2_{40}$, $L2_{200}$ and $L2_{miss}$ access the processor bus frequently. As a result, the processor AHB AMBA bus utilization is almost 100% when running 4 copies of the benchmarks (this result is not shown in Table 3).

$L2_{40}$ does not stress much the memory controller, even four copies. In the worst case, $L2_{40}$ has 6% L2 cache miss rate since its data footprint is small enough to fit inside the L2 cache. $L2_{200}$ suffers some L2 misses in isolation that significantly increase when several copies are run (up to 88% with 2 copies and 99% with 4 copies). $L2_{miss}$ stresses always all shared resources, having roughly the same L2 cache miss rate (99%) regardless of the number of copies. Finally, the $L2_{st}$ benchmark executes store opeartions which, due to the write-through policy of the data L1 cache, are written into the L2 cache. As a result $L2_{st}$ is sensitive to L2 occupancy, suffering significant slowdowns when it runswith other L2 stressing benchmarks.

5.2 Processor AHB AMBA Bus

With this first experiment we want to determine the effect that interactions in the processor AHB AMBA bus have on program execution time. To that end, we use the $L2_{40}$ micro-benchmark that has a high data cache miss rate and hits in L2, as its data footprint is higher than the size of data cache but smaller than the L2. Moreover, when running up to four copies of $L2_{40}$, the overall data footprint, i.e. 160 KB, fits in the L2 cache, so in general it does not generate evictions in the L2 cache. As a result, under all core counts most access to cache goes to the L2 and the execution time slowdown experienced when several copies of $L2_{40}$ are run simultaneously is due to inter-task interferences in the processor AHB AMBA bus.

The first set of bars in Figure 2 shows the execution time slowdown of the $L2_{40}$ when we simultaneously run several copies of it in different cores. All values are normalized to $L2_{40}$ execution time when running in isolation. We observe that the worst delay due to sharing the AMBA bus is 12% when two tasks are executed and 83% when 4 tasks are executed. This slowdown is relatively moderate. These results are higher than those reported in [17]: 27% slowdown for a configuration comprising 4 cores. The may reason behind this difference is that in [17] the bus model used is simpler, having low latency and hence less effect on applications' performance.

5.3 Memory Controller and Processor AHB AMBA Bus

This experiment aims at determining the effect of inter-task interferences generated at the memory controller, the memory AHB AMBA bus and the processor AHB AMBA bus on program's execution time. We use the $L2_{miss}$ micro-benchmark that misses in L1 and L2 frequently, regardless

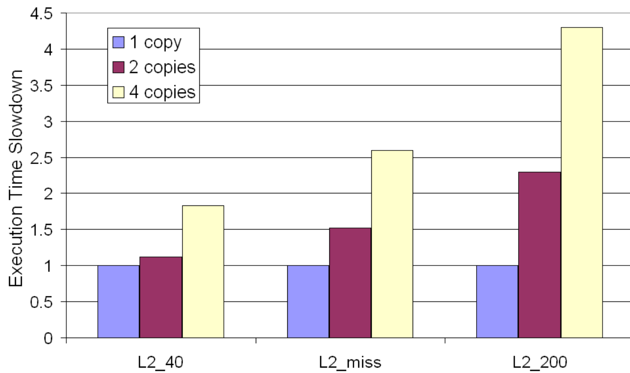

Figure 2: Execution time slowdown of $L2_{40}$, $L2_{miss}$ and $L2_{200}$ under different workloads

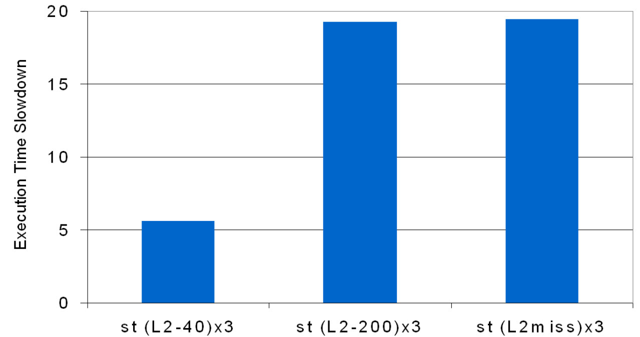

Figure 3: Execution time slowdown of $L2_{st}$ when run against $L2_{40}$, $L2_{200}$ and $L2_{miss}$ for Linux

of the workload in which it runs. This is so because its data footprint is much higher than the L2 cache cache size. Hence, regardless of the workload within it runs, the $L2_{miss}$ generates systematic L2 misses without taking advantage of the L2 cache at all.

The second set of bars in Figure 2 shows the execution time slowdown that $L2_{miss}$ suffers when it runs with other copies of $L2_{miss}$. We observe that the access to the main memory introduces an execution time variation up to 1.5x and 2.6x when considering 2 and 4 $L2_{miss}$ micro-benchmarks respectively. These slowdowns due to interactions in the memory controller and the AHB AMBA bus are similator to those reported in [20], in which authors measured a WCET slowdown of 3x due to interferences in the access to memory.

By comparing the first and second set of bars in Figure 2 we can conclude that the main memory introduces significantly higher execution time variation that the AMBA AHB processor bus.

5.4 Memory Controller, L2 cache and Processor AHB AMBA Bus

In this experiment we want to determine the overall effect of inter-task interferences generated in each of the memory hierarchy components, i.e. the memory controller, the L2 cache and the processor AHB AMBA buses. To that end, we consider the $L2_{200}$ micro-benchmark with a data footprint of 200 KB. When running in isolation the $L2_{200}$ almost always misses in the data cache and hits in L2, taking profit of the L2 cache. However, when running within a workload composed of four copies of $L2_{200}$, the micro-benchmark does not take advantage of the L2 cache as each task may evict data from other tasks.

The third set of bars in Figure 2 shows the execution time slowdown that $L2_{200}$ suffers when we run it simultaneously with other copies of $L2_{200}$. We observe that the inter-task interferences generated by memory hierarchy components, introduce an execution time variation up to 2.3x and 4.3x when considering 2 and 4 $L2_{200}$ micro-benchmarks respectively. By comparing the second and third set of bars in Figure 2 we conclude that in the worst case L2 cache interferences, makes the execution time vary from 2.6x to 4.3x, being still the memory controller and the bandwidth to memory the shared resources that introduces the highest execution time variation.

5.5 Effect of Store operations

Store operations introduce higher execution time variability than load operations due to the write policy implemented in the L1 data and L2 caches of the NGMP, i.e. write-through and write-back respectively. This makes every store instruction to always access to the AMBA AHB processor bus and the L2 cache, even if the data footprint of the program fits the L1. As a result, stores are delayed due to interferences accessing the AMBA AHB processor bus and the L2 cache and, in addition, stores create additional traffic on the bus. Moreover, if a store evicts a dirty line in L2 it is stalled until the data is copied to main memory.

In order to evaluate the impact of interferences caused by cache stressing benchmarks on store intensive benchmark ($L2_{st}$), we consider three instances of $L2_{40}$, $L2_{200}$ and $L2_{miss}$. Figure 3 shows the execution time slowdown of $L2_{st}$ when running it simultaneously with 3 instances of $L2_{40}$, $L2_{200}$ and $L2_{miss}$, respectively labelled as *st($L2_{40}$)x3*, *st($L2_{200}$)x3* and *st($L2_{miss}$)x3*.

In case of the *st ($L2_{40}$)x3* workload, the slowdown suffered by $L2_{st}$ is 5x, mainly due to conflicts on the AMBA AHB Bus, as well as and dirty cache lines. $L2_{st}$ has a data footprint similar of $L2_{40}$ which make the overall footprint to fit in L2 cache (40 KB x 4). However, performance degradation increases up to 19x when running it with $L2_{200}$ and $L2_{miss}$ due to the fact that stores that miss in L2, may evict dirty lines. This cause the store operation to be stalled until the dirty line is written back to main memory and the new line is brought into the cache, before the store writes its value on the new line.

This shows that, even if the data footprint of the store micro-benchmark fits in the data cache, the fact that it has to access to the L2 and hence use the AMBA AHB Processor bus on every store operation, as well as memory access at every dirty cache line eviction, make it quite sensitive to other benchmarks using the bus.

We conclude that programs with a high density of store instructions may suffer high slowdowns even if they fit in data cache, mainly if they are run concurrently with programs using the L2 cache or the AMBA processor bus extensively.

5.6 Effect of Intra-Core Resources

While previous sections evaluated the impact of using the shared resources of the cache hierarchy, this section evaluates the impact of not using them.

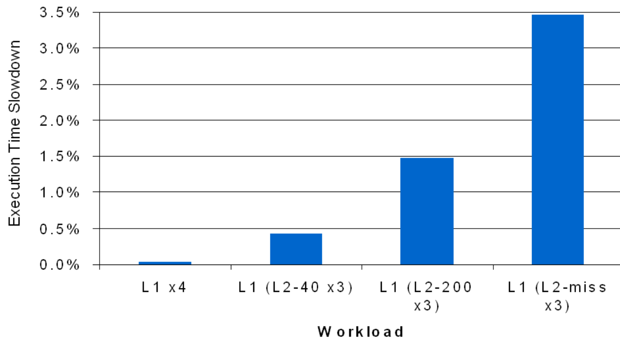

Figure 4: Execution time slowdown of L1 when running within several workloads composed of 3 L1, L2$_{40}$, L2$_{200}$ and L2$_{miss}$

Figure 4 shows the execution time slowdown of the L1 benchmark when running it within several 4-application workloads, taking as a baseline its execution time running in isolation. In the experiment, L1 runs with three copies of L1, L2$_{40}$, L2$_{200}$ and L2$_{miss}$, labeled as L1 x4, L1 (L2$_{40}$ x3) L1 (L2$_{200}$ x3) and L1 (L2$_{miss}$ x3) respectively.

We observe no noticeable slowdown, having a maximum execution time variation of less than 3,5% when running within a workload composed of the L2$_{miss}$ micro-benchmarks.

5.7 Impact of inter-task interferences on EEMBC

In this section we consider the EEMBC AutoBench to confirm the results we obtained in previous section with micro-benchmarks. We run each EEMBC simultaneously with several copies of the different micro-benchmarks described in previous sections.

Figure 5 shows the execution time slowdown of different EEMBC benchmarks when running each with two copies of the same benchmark (labeled as x2), with four copies (labeled as x4), with three copies of the L2$_{40}$ (labeled as L2$_{40}$ x3), with three copies of the L2$_{200}$ (labeled as L2$_{200}$ x3) and with three copies of the L2$_{miss}$ (labeled as L2$_{miss}$ x3).

We start running workloads comprised of copies of the same EEMBC benchmark (x2 and x4). We do so, instead of putting different benchmarks in the same workload, because as shown in [25] the synchronized start of several instances of the same program shows higher slowdown than the parallel execution of different programs. Anyway, we observe that the inter-task interaction between several copies of the same EEMBC is very low.

However, when running each EEBMC with micro-benchmarks, the execution time increases significantly due to inter-task interferences. When running EEMBC together with 3 instances of the L2$_{40}$ benchmark, which have a 120KB data footprint in total, the execution time increases up to 60% in case of the cacheb. Such an slowdown is even higher when running EEMBC with L2$_{200}$ and L2$_{miss}$, observing an execution time slowdown of up to 5.5x (in case of cacheb and canrdr).

We have used PMC data to find the reason behind this behavior. Figure 6 shows the amount of load instructions (labeled as ld), the second column shows the amount of store instructions (labeled as st), the third column shows the sum of load and store instructions (labeled as ld+st), and the fourth column shows the bus utilization (ahbuse).

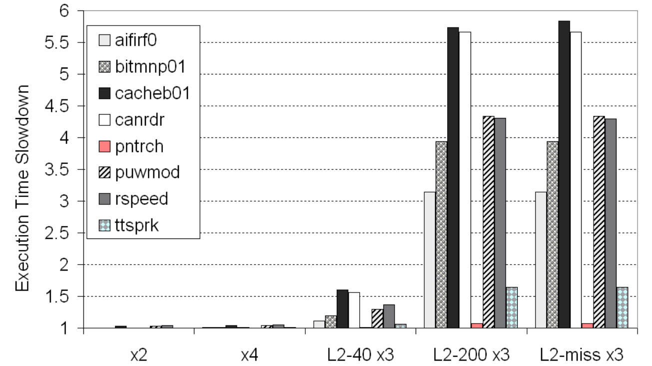

Figure 5: Execution time slowdown of EEMBC AutoBench when running them together with micro-benchmarks

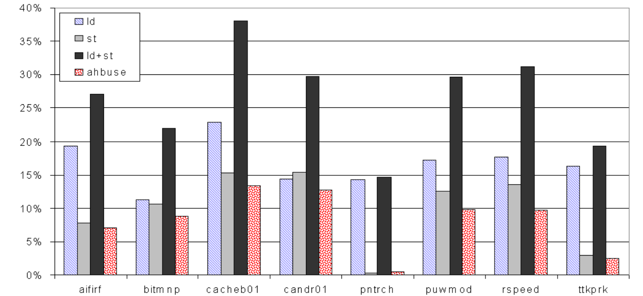

Figure 6: Characterization of EEMBC benchmarks

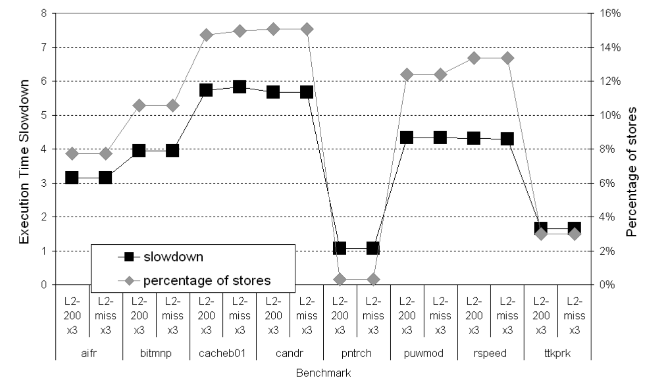

Figure 7: Correlation between percentage of stores (right x-axis) and slowdown (left x-axis) for all studied EEMBC benchmarks, when run together with 3 instances of either L2$_{200}$ or L2$_{miss}$.

We observe that the EEMBC with more load and store instructions and the ones with a higher bus utilization (*cacheb* and *candr*, followed by *puwmod* and *rspeed*) are the ones suffering from the highest inter-task interference. Specifically, there is a high correlation between the density of store instructions and the slowdown. This correlation is shown in Figure 7. The primary y-axis shows the Execution Time slowdown and the secondary y-axis the percentage of stores of each benchmark.

The grey line shows the percentage of stores per instruction. That is $\frac{Store\ Count}{Instruction\ Count} * 100\%$. The small varia-

tions are due to small noise that the system introduce in the measurements. The grey line measures the slowdown suffered by the different EEMBCs in each configuration. From the picture, we observe a clear possitive correlation between slowdown and store percentage. As a result, we conclude that store instruction count is the main source of inter-task interferences in the NGMP for the observed benchmarks.

We also run each EEMBC against workloads composed of different combinations of the $L2_{40}$, $L2_{200}$, $L2_{miss}$ and $L2_{st}$ benchmark. In particular, we bind each EEMBC to a core and run it with all possible combinations of the 4 micro-benchmarks in the other three cores, for a total of 20 experiments: AAA, BBB, CCC, DDD; AAB, AAC, AAD, BBA, BBC, BBD, CCA, CCB, CCD, DDA, DDB, DDC; ABC, ABD, ACD, BCD. Where A, B, C and D represent a different micro-benchmark. The slowdown observed in all cases for all EEMBC is less than the maximum reported in Figure 7.

6. RESULTS UNDER RTEMS

Though the use of adapted versions of COTS OS has recently been observed in the real-time industry, special-purpose OS, such as RTEMS, are still predominant. Hence, to increase the applicability of our study we have carried out under RTEMS the same study we did for Linux

Similarly to the experiments prepared for Linux, the main metric we take into account for RTEMS is the slowdown tasks suffer in the NGMP due to inter-task interferences. The Execution Infrastructure allows the the same type of experiment we have on Linux to be run on RTEMS: it allows running different 'workloads' binding tasks to the desired cores, periodically reading PMCs.

6.1 Validation

We analysed each micro-benchmark running simultaneously with multiple copies of itself (1, 2 and 4), using the performance monitoring support provided by the NGMP under RTEMS. The results are shown in Table 4.

By comparing Table 3 and Table 4 we observe that in general microbenchmarks have similar behavior. The main difference is $L2_{200}$ which under RTEMS has a much 'sharper' behavior: a single copy of $L2_{200}$ almost does not have any L2 cache miss under RTEMS, while under Linux the L2 miss rate is 31%. We checked that the object code executed in both cases is almost the same, and that the main difference in the behavior lies on the 'noise' introduced by the Operating System.

6.2 Results

Figure 8 shows the slowdown that $L2_{40}$, $L2_{200}$ and $L2_{miss}$ suffer with respect to its execution in isolation when they are run under different workloads.

In the case of $L2_{40}$, which bounds the maximum inter-task interference caused by the AMBA bus, results are very similar to the ones obtained on Linux.

For $L2_{miss}$, the micro-benchmark that bounds the combined effect of the AMBA bus and the memory controller, results are again slightly worse, yet very similar to the results obtained on Linux.

For $L2_{200}$, the micro-benchmark that bounds the combined impact of L2 cache, AMBA bus, and memory controller, we observe a much bigger performance degradation than we observed on Linux (5x slowdown on 2 cores, 9x slowdown on 4 cores). The reason for this is that on RTEMS, the baseline $L2_{200}$ run in isolation causes very few L2 misses, thanks to the small memory footprint of the operating system.

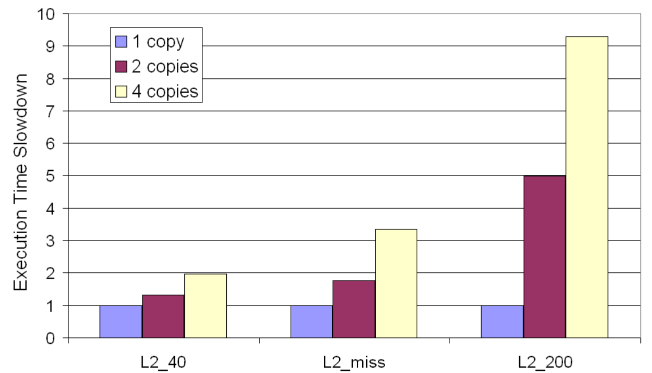

Figure 8: Execution time slowdown of $L2_{40}$, $L2_{200}$ and $L2_{miss}$ under differerent workloads in RTEMS

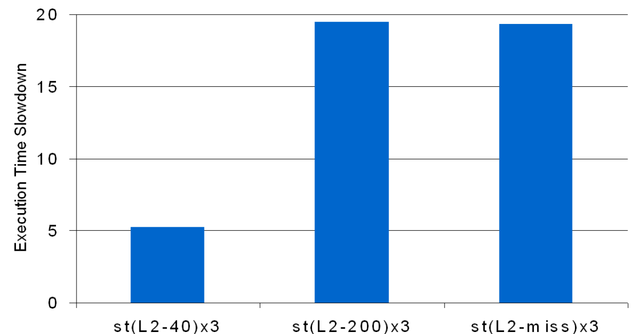

Figure 9: Execution time slowdown of $L2_{st}$ when run against $L2_{40}$, $L2_{200}$ and $L2_{miss}$ under rtems

Finally, in Figure 9 we observe the execution time slowdown of $L2_{st}$ when in runs in different workloads with 3 copies of $L2_{40}$, $L2_{200}$ and $L2_{miss}$. We observe that the slowdown is quite similar to the one obtained under Linux.

7. DISCUSSION

In this section we discuss some of the main applications of the results of this study.

Timing Verification of NGMP-based real-time systems. Verification is the process used to check that the requirements of a system are satisfied. The verification can be classified into functional verification and timing verification; the former checks that the system is functionally correct while the latter verifies that timing constraints are met. For industry it is of primary importance to keep the costs of such verification low. In Integrated Architectures [13], currenlty used in automotive and avionics, a key design principle in order to contain the cost of timing verification is to guarantee that there is no interaction between the different functions sharing the resources. To that end, at functional level, it is necessary to provide functional isolation, such that a bug/misbehavior in a function does not affect the others. At timing level, it is necessary to provide timing isolation, such that the timing behavior of a task is not affected by the others. Incremental qualification [13] relies on each software and hardware component exhibiting the property of

Table 4: **L1 data and L2 cache miss ratio, and processor AHB AMBA bus utilization per process of all micro-benchmarks, running simultaneously 1, 2 and 4 copies under RTEMS.**

	Data cache miss (%)			L2 cache miss (%)			% of stores
No. of copies →	1	2	4	1	2	4	1
L1	0	0	0	0	0	0	0.07
$L2_{40}$	99.5	99.5	99.6	0.25	0.24	0.16	0.21
$L2_{200}$	99.0	99.1	99.4	0.31	99.1	99.1	0.21
$L2_{miss}$	98.9	99.1	99.2	99.0	99.1	99.2	0.61
$L2_{st}$	0	0	0	0	0	0.06	96.9

time composability. Such property dictates that the timing behavior of an individual component does not depend on the rest of the components, thus allowing the system to be composed of individually analyzed components. Time composability also helps reducing system integration cost.

In the case of the NGMP we observe that the main software features affecting time composability are (1) the percentage of store instructions and (2) in case the application has low number of stores, whether its data fits in the L1 data cache. Given an application with high percentage of stores, even if it fits in the L1 cache, we found it very sensitive to the overall workload of the system: small changes in the other application's behavior may significantly affect the execution time (up to a 19x slowdown). This because store operations always access a shared resource, the processors main bus. Such behavior seriously compromise time composability.

Software design. For application developers the main conclusion is to reduce the number of stores of their applications. Obviously, this is intrinsic to the functionality of the application and hence it can be difficult to change it. Otherwise, in order to ensure time composability, those store-intensive applications have to be scheduled in isolation or it has to be ensured that any other application that may simultaneously run on the other cores fit their data cache so they do not introduce traffic in the AHB processor bus or use the L2 cache.

Hardware design. At hardware level, a write-back policy for the L1 data cache may be considered as it will significantly reduce the overhead on applications execution time due to inter-task interferences. This will introduce several challenges that would need to be addressed. (1) In the implementation of the consistency protocol, as MESI/MOESI or directory-based protocols will be needed, and they are consistently more complex than snooping-based ones. (2) If write-back schemes were used, there would be some data for which only one copy would exist in the system, located indeed in the L1 cache. And the current implementation of the NGMP features error detection only in the L1 cache; to maintain adequate protection from errors (frequent in space, the target environment for the NGMP) error correction schemes would have to be implemented in the L1 cache, potentially increasing the latency of read/write operations in such cache, thus lowering the maximum frequency of the overall system.

8. RELATED WORK

In this work we have used micro-benchmarks as the main tool to bound the effect of inter-task interferences in hardware shared resources. Micro-benchmarks have been also in prior studies in high-performance and real-time systems [9][10][16][25].

For hard real-time systems there have been several studies focused on new designs for multicore processors to make them more time predictable. This includes the cache and the processor bus [17] and the memory controller [18][7]. Also, there have been a series of projects working on the same direction [15][19][24][27][6]. Many of these works propose changes in the design of the multicore processors in order to make them more predictable. In this study, instead, we start from an existing processor architecture and provide software-only mechanisms to increase time predictability by bounding the effect of inter-task interferences in hardware shared resources.

9. SUMMARY AND CONCLUSIONS

In this paper we have evaluated the maximum variation that inter-task interferences may introduce in the execution time of applications when running in the NGMP multi-core processor, the latest multi-core processor used by the European Space Agency. Bounding the effect of inter-task interferences is of paramount importance to provide meaningful WCET estimations in safety critical systems.

Concretely, this paper provides accurate data on the impact of interferences arisen in the processor AHB AMBA bus, the L2 shared cache, the memory controller and the write-through policy in the data cache under both Linux and RTEMS. To do so, we use a set of specialized micro-benchmarks designed to stress each of the processor resources close to the worst-case scenario. Our results show that, when considering a workload composed of four micro-benchmarks, the interactions in the processor AHB AMBA bus may increase the execution time up to 1.83x (1.95 for RTEMS); the combined effect of the interaction in the processor bus and the memory controller may slowdown the execution time of applications up to 2.6x (3.4 for RTEMS); the combined effect of the processor bus, the L2 cache and the memory controller may can slowdowns of up to 4.3x (9x for RTEMS). Finally, the effect due to the write-through policy in the first level data cache can be as much as 19x for microbenchmarks and more than 5x for EEMBC benchmarks. For EEMBC we have also identified the average number of stores per instruction as the main factor determining the sensitivity of an application to inter-task interferences in the NGMP architecture.

Overall, the main software features affecting time composability are (1) the percentage of store instructions and (2) if the application has few number of stores, whether it fits in the first level data cache. We have proposed several hardware/software directions to improve the time composability of the NGMP. We expect that our in-depth evaluation of the effects of inter-task interferences in the NGMP for both Linux and RTEMS will benefit the real-time software community to develop interference-aware WCET estimation

techniques and scheduling algorithms for NGMP and real multi-core processors in general.

Acknowledgements

This work has been mainly funded by the European Space Agency under Contract 4000102623. Eduardo Quiñones is partially funded by the Spanish Ministry of Science and Innovation under the grant Juan de la Cierva JCI2009-05455. The authors thank Jiri Gaisler and Jan Ardenson for their help with the ML510 platform.

10. REFERENCES

[1] *AMBA Bus Specification. http://www.arm.com/products/system-ip/amba/amba-open-specifications.php.*

[2] *ESA contract: 22279/09/NL/JK.*

[3] *ESA contract 4200023100, System Impact of Distributed Multi-core Systems.*

[4] *NGMP Preliminary Datasheet Version 1.6, August 2011 http://microelectronics.esa.int/ngmp/LEON4-NGMP-DRAFT-1-6.pdf.*

[5] *ARC Advisory Group. Process Safety System Worldwide Outlook. Market Analysis and Forecast through.* 2012.

[6] ACROSS. ARTEMIS CROSS-Domain Architecture. http://www.across-project.eu.

[7] Benny Akesson, Kees Goossens, and Markus Ringhofer. Predator: a predictable SDRAM memory controller. In *CODES+ISSS*, USA, 2007. ACM.

[8] Jan Andersson, Jiri Gaisler, and Roland Weigand. Next generation multipurpose microprocessor. In *DASIA*, 2010.

[9] C. Boneti, F. J. Cazorla, R. Gioiosa, A. Buyuktosunoglu, C.Y. Cher, and M. Valero. Software-controlled priority characterization of power5 processor. In *35th International Symposium on Computer Architecture (ISCA)*, 2008.

[10] V. Cakarevic, P. Radojkovic, J. Verdu, A. Pajuelo, F. J. Cazorla, M. Nemirovsky, and M. Valero. Characterizing the resource-sharing levels in the ultrasparc t2 processor. In *42nd International Symposium on Microarchitecture (MICRO)*, 2009.

[11] P. Clarke. *Automotive chip content growing fast, says Gartner (9/6/2010).* http://www.eetimes.com/electronics-news/4207377/Automotive-chip-content-growing-fast.

[12] R. Gioiosa, F. Petrini, K. Davis, and F. Lebaillif-Delamare. Analysis of system overhead on parallel computers. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, 2003.

[13] Henning Butz. Open Integrated Modular Avionic (IMA): State of the Art and future Development Road Map at Airbus Deutschland. Department of Avionic Systems at Airbus Deutschland GmbH.

[14] T. Lundqvist and P. Stenstrom. Timing anomalies in dynamically scheduled microprocessors. In *RTSS*, 1999.

[15] MERASA. *EU-FP7 Project: www.merasa.org.*

[16] Jan Nowotsch and Michael Paulitsch. Leveraging multi-core computing architectures in avionics. *Ninth European Dependable Computing Conference (EDCC 2012)*, 2012.

[17] Marco Paolieri, Eduardo Quinones, Francisco J. Cazorla, Guillem Bernat, and Mateo Valero. Hardware support for WCET analysis of hard real-time multicore systems. In *ISCA*, Austin, TX, USA, 2009.

[18] Marco Paolieri, Eduardo Quinones, Francisco J. Cazorla, and Mateo Valero. *An Analyzable Memory Controller for Hard Real-Time CMPs* . Embedded System Letters (ESL), 2009.

[19] parMERASA. *EU-FP7 Project:http://www.parmerasa.eu/.*

[20] Rodolfo Pellizzoni, Andreas Schranzhofer, Jian-Jia Chen, Marco Caccamo, and Lothar Thiele. Worst case delay analysis for memory interference in multicore systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, 2010.

[21] F. Petrini, D. J. Kerbyson, and S. Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, 2003.

[22] Jason Poovey. *Characterization of the EEMBC Benchmark Suite.* North Carolina State University, 2007.

[23] PRET. Precision Timed (PRET) Machines. `http://chess.eecs.berkeley.edu/pret`.

[24] PROARTIS. Probabilistically analyzable real-time systems. feb 2010. http://www.proartis-project.eu/.

[25] Radojković, Petar, Girbal, Sylvain, Grasset, Arnaud, Qui nones, Eduardo, Yehia, Sami, and Cazorla Francisco J. On the evaluation of the impact of shared resources in multithreaded cots processors in time-critical environments. *ACM Trans. Archit. Code Optim.*, 8(4):34:1–34:25, January 2012.

[26] P. Radojković, V. Cakarević, J. Verdú, A. Pajuelo, R. Gioiosa, F. Cazorla, M. Nemirovsky, and M. Valero. Measuring Operating System Overhead on CMT Processors. In *SBAC-PAD '08: Proceedings of the 2008 20th International Symposium on Computer Architecture and High Performance Computing*, 2008.

[27] T-CREST. *EU-FP7 Project:http://www.t-crest.org/.*

[28] `http://www.predator-project.eu`.

[29] `http://www.rtems.org/`.