Final Report

Document Information

Author	Francisco J. Cazorla, Roberto Gioiosa, Mikel Fernandez, Eduardo Quiñones
Contributors	Marco Zulianello, Luca Fossati
Reviewer	Jaume Abella
Keywords	

Table of Contents

1 Introduction

This document describes the *Multicore OS Benchmark* project done by BSC in collaboration with ESA under contract *RFQ-3-13153/10/NL/JK*. The document introduces the initial context and project specifications, summarizes the successive development stages, describes the current state of the technology and suggests future work directions. In this final report, special emphasis is given to the project outcomes and conclusions.

The objective of this document is to provide a general perspective of the Multicore OS Benchmark project:

- Description of the need for a project like this
- Problem definition
- Experimental Setup
- Results Obtained
- Conclusions

This document targets the Management and Technical staff from ESA/ESTEC, mainly those working with Real-Time Systems. It has a broader target than the other project deliverables that were more technical or project oriented.

1.1 Project Abstract

The European Space Agency (ESA) and the Barcelona Supercomputing Center (BSC) collaborated in an exploration project to define and develop a benchmark suite to exercise the new features of the Next Generation MicroProcessor (NGMP), the multicore architecture solution designed by ESA and developed by Gaisler, and the better understand thread interaction in shared multicore resources.

Through the execution of benchmarks coming from different domains (both real time and non-real time) in different configurations (e.g., 1-4 cores, different input sets), in addition to a set of specifically designed benchmarks, the project aims at increasing the understanding and the confidence of how to use multicore processors in future space scenarios, in which real-time and payload applications co-exist. The benchmark suite is, hence, designed to feature the following characteristics:

- Exercising the new NGMP: The NGMP provides different levels of resource sharing among the four available cores. The interaction of threads concurrently running on the four cores may induce execution time variability and, thus, timing issues. The benchmark suite aims at identifying and bounding such interactions and timing issues.
- Developing a benchmark able to mimic some characteristics of some ESA reference applications.
- Based on world well known benchmarks compose workloads that comprise real and non-real time applications and study inter-thread interaction on those workloads when running on the NGMP.

According to the ESA view, also confirmed by the state-of-the-art literature, future real-time, space systems will run a mix of real-time and payload (non real-time)

applications. This will allow minimizing space and power consumption, hence production and maintenance costs. To reflect this scenario, the benchmark suite provides two classes of benchmarks:

1. On-board control-like applications: These are real-time mission or safety critical applications. They demand functional and timing correctness, thus each command should be correctly executed before a given deadline. Solar panels on a satellite are controlled by this kind of applications.
2. On-board payload-like applications: These are non real-time applications that are run next to the real-time control applications. They do not pose time constraints but should provide a desired QoS. In an aircraft system, customer entertainment is a typical payload application.

The results of the project show how real-time and non real-time applications interact when co-running on the NGMP and on resource sharing yields the highest potential impact on applications' performance. Depending on the resource under consideration the maximum effect of interactions between tasks can range from 75% or up to more than 19x. The result of the study will help application developers at ESA to better design their applications so that they can control the effect of inter-task interferences on their applications. The benchmarks developed can be easily adapted to other multicore architectures.

1.2 Background of the CAOS team

This project has been carried out by members of the CAOS group at at the Barcelona Supercomputing Center (www.bsc.es/caos). The CAOS group has long experience in managing bilateral projects with industry and European projects. The CAOS group has participated in industrial projects with IBM and Sun Microsystems (now ORACLE) and the European FP7 Projects MERASA. At the time of writing this document, the group participates in the PROARTIS and parMERASA FP7 projects. The group is also participating in the VeTeSS ARTEMIS projects.

Some more details about these projects are provided next:

- HiPEAC Network of Excellence: BSC has been an active member of this Network of Excellence, participating in several clusters (projects) related to multithreaded hard real-time capable processors. In these clusters BSC has been collaborating with other European universities (University of Augsburg, University of Toulouse and University of Karlsruhe), SME (Rapita Systems LTD) and industry (Infineon).
- The MERASA FP7 project (www.merasa.org), in which ESA was involved as an industrial advisor, is highly related with this ITT. The project, which recently finished, investigated analyzable high performance hardware features for embedded hard real-time multi-core processors and the system software changes required to support those hardware features. MERASA put together important avionics companies such as Honeywell, which provided avionics applications and Airbus, which was part of the Industrial Advisor Board. Several key machinery (Bauer) companies and processor vendors (Infinenon and NXP) were in the industrial advisory board of MERASA as well.
- The parMERASA project (www.parmerasa.eu), in which ESA is involved as an industrial advisor, aims to develop a multi-core processor architecture that provides a predictable timing behaviour, a suitable system-level software,

software design guidelines for parallelising hard real-time applications, and tools for estimating and verifying the timing behaviour of such parallel applications. parMERASA put together important avionics, automotive and construction machinery companies such as Honeywell, which provided avionics applications, and Airbus as part of the Industrial Advisor Board (IAB); Denso, which provides automotive applications, and BMW, Mecel AB, Elektrobit Automotive and Infineon as part of the IAB; and Bauer, which provides automation applications.

- Finally, BSC leads the PROARTIS FP7 project, in which Airbus and ESA are also involved: the former as full partner and the providing avionics applications to perform case studies, and the latter as an industrial advisor partner. The PROARTIS project enables the BSC to explore a new set of hardware and system software proposals that enable probabilistic timing analysis and that overcome the limitations of statically analyzable real time systems.
- VeTeSS: Verification and Testing to Support Functional Safety Standards. The increasing complexity of the systems and the need for higher quality levels as a differentiating characteristic, have pushed European automotive industry towards releasing a new safety standard ISO-26262, which requires methodologies for automatic verification and testing of systems. The VeTeSS ARTEMIS project addresses this challenge by proposing a new complete and automatic methodology to verify and test components and systems against the ISO-26262 standard and allowing components and systems to be certified out-of-context, so that they can be used in any system or car matching their specifications without requiring a re-certification. VeTeSS consortium is formed by some of the main European companies in the automotive arena such as Infineon, NXP Semiconductors, Volvo Technology, Centro Ricerche FIAT, Catena, AVL List, TWT, e-AAM, Fico-Triad, ViF, SprintSoft, Rapita Systems, IKV++, QRTECH and Exida, and some of the main European public research institutions such as the BSC-CNS, Fraunhofer, Politecnico di Torino, University of Oxford, Technische Universität Wien and SP Technical Research Institute of Sweden.

As a result of some of the advances done in the embedded field BSC has filed two patents to the European Patent office and to the United States Patent and Trademark Office.

1.3 Document Structure

Section 2 describes the project's objectives and requirements, the current state of the art, and the expected output of the project. Section 3 describes the project's structure and details the project's work packages and tasks. Section 4 describes the characteristics of the target system, the tools and techniques used during the implementation of the project and the design and implementation of the benchmark suite, the mimicking applications and the analysis tools developed throughout the project. Section 5 describes the results of this activity. Finally, Section 6 presents the main conclusions of this activity.

2 Project Objectives

The objective of this activity is to define and develop a *benchmark suite*, representative of *reference ESA applications*, suitable to exercise the new NGMP multicore processor, so the level of confidence of the design of the NGMP increases. The benchmark suite will be capable to generate different inter-task interference scenarios that may arise in the NGMP processor, by stressing the different processor hardware shared resources. The benchmark suite developed has to be flexible enough such that different processor components can be stressed in different levels, e.g. configurable number of tasks running simultaneously inside the processor, CPU load, memory load, etc. The benchmark suite comprises benchmarks mimicking the behavior of *payload* applications and benchmarks mimicking *on-board control* applications, and it will run on both RTEMS and Linux operating system.

2.1 State of the art

Improving performance by means of processor's frequency increment or Instruction Level Parallelism (ILP) has reached a sudden stop caused by the unsustainable increase of power consumption. Multicore Processors (CMP) are increasingly being considered as an effective solution to cope with current performance requirements of Critical Real-Time Embedded (CRTE) systems, like those used in space domain. CMPs feature several processors inside a single die where the cores share different level of hardware resources, providing performance improvement by means of Thread Level Parallelism (TLP). By doing so, the performance of a CMP is improved offering a better performance/Watt ratio than a single core solution with similar performance, while maintaining a relatively simple processor design.

This trend has been followed by many processor vendors in different CRTE domains like the Aeroflex Leon4 Multicore processor [NGMP], implementing a four-core processor for space domain, the Freescale MPC5510 [MPC55] and MPC5668 [MPC56], both implementing a dual-core processor for automotive and avionics domain and Texas Instruments TMS570 [TMS57], implementing a dual-core processor for transportation safety applications.

CMP systems can be used in a wide range of application domains as they provide several advantages over single core systems:

- System with limited space (satellites, aircrafts, automobiles, etc.) benefit from the higher density of CMPs. That is, CMPs allows scheduling workloads composed of a higher number of applications than single-core processor. In addition, workloads can be composed of mixed criticality applications, i.e. safety and non-safety critical applications, maximizing the hardware utilization and so reducing cost, size, and weight and power requirements.
- Systems with limited power budget (satellites, mobile phones, etc.) take advantage of the shared hardware resources, increasing operation time, battery efficiency, etc.
- Applications with high computing demands can exploit the CMP performing a parallel implementation of their algorithm and so exploiting the TLP.
- CMPs facilitate the design of safety critical redundant systems. For example, if the system is operating in an environment in presence of cosmic radiation or alpha

particles where there is a high risk of soft errors, several instances (say 3) of the same application can execute on the same CMP, each running in a different core. Then, an arbiter can later select the result of the computations that are not effected by soft errors by looking at the applications that provide the same result.

In the space industry, CMP architectures may result especially beneficial as they achieve higher performance levels without increasing CPU clock frequencies which could lead, in turn, to an unacceptable error rate caused by electromagnetic interferences and cosmic radiations.

Unfortunately, though the advantages provided by CMPs are clear, the timing correctness of Critical Real Time Embedded (CRTE) systems is harder to be proven for multicore (CMP) processors than for single-core processors due to *inter-task interferences*. Inter-task interferences appear when two or more tasks access simultaneously a hardware shared resource, requiring an arbitration scheme to decide which task get the access granted. As a consequence, the execution time of a task may increase depending on the inter-task interferences generated by the other co-running tasks, and so it becomes extremely difficult or even impossible to perform a tight WCET analysis; and in CRTE systems it is essential to guarantee the timing correctness of the system, and for the most critical ones, strong arguments and proofs are needed to be able to prove correctness to certification authorities. CRTE systems need to prove that they operate correctly, satisfying all temporal constraints.

2.2 Requirements

This project requires expertise in two main areas: processor architecture and benchmarking. The former is required to understand the different processor components that may affect the execution time of a given application when running in a multicore processor environment. The latter is required to design a benchmark suite capable of stressing the specific processor components that affect the execution time of a given application.

Several boards where considered at the beginning of the project. The one that fits the budget and requirements of the project was the Xilinx ML510 development board implementing a Next Generation Multipurpose Microprocessor (NGMP) in its FPGA. The development board is described in [XML510SC, XML510RD, XML501ED]. There are two variants of the NGMP that can be implemented in this board. One with two cores each comprising Floating-Point unit (FPU) and another without FPU but with 4 cores. We chose the latter as it lead to scenarios with higher contention between threads on shared resources. The FP operations where done through specialized libraries using integer functional units.

A requirement on the board was to enable the reading of Performance Monitoring Counters (PMC). In general, PMC values can be generally obtained through one of the following methods:

- Internal measurements: performance counters values are obtained directly from the performance counter registers. These values can then directly be combined with other internal measurements (such as execution time or application-depend metrics). This option generally requires software components (i.e., operating system and/or runtime library support) that is not available for Linux on NGMP and was, thus, discarded.

- External measurements: performance counters values are obtained from the NGMP statistical unit through the JTAG debug interface. Besides being more general (it does not rely on the particular software running on the NGMP), this option introduces lower measurement noise, thus more reliable results. On the other hand, external measurements can only be performed with a low frequency (e.g., no more than 1 Hz) and need to be coordinated with other (eventual) internal measurements. As part of the project, we developed an offline infrastructure that combines external measurements obtained through the NGMP statistical unit with the internal measurements obtained by the benchmark framework running on the NGMP.

2.3 Expected Output

The main results expected of the projects are: (1) a comprehensive understanding of the architecture of the new NGMP processor; (2) a comprehensive understanding of how inter-task interferences affect the execution of control-like applications; and (3) a comprehensive understanding of how the benchmark suite has been designed to stress NGMP resources and how the benchmarks mimic ESA reference applications. This will facilitate the porting of the benchmark suite to future processor architectures that are of the interest of the ESA.

3 Project overview

The project is scheduled to have 12 month of duration and structured into 4 tasks/phases:

- The first phase comprised the Task 1 (2 months). During this phase, we identified the most relevant state of the art. This includes research papers, industry papers (white papers), patents and projects. We performed an initial identification of the hardware shared resources in NGMP systems based on the available NGMP documentation. It is important to remark that during the first phase, the board was not available. The main focus of the study was on benchmarking techniques and applications for multicore and real-time architectures that were taken as reference in the rest of the project.
- The second phase comprised the Task2 (4 months). During this phase refined the knowledge of the NGMP and the ESA reference application acquired during the first phase. The main objective of this phase was the development of a benchmark suite capable of running on-board control-like and payload-like applications and of stressing the different processor resources.
- The third phase comprised the Task 3 (4 months). During this phase, we performed a dry-run of the benchmark suite a FPGA implementation of the NGMP, available at this point. Based on the experiments performed, we analyzed performance and timing issues caused by application interaction and resource sharing.
- The last phase comprised the Task 4 (2 months). The objective of this phase was to port the benchmark suite to the NGMP implementation in ESA, a HAPS54 board. This task was agreed not to be done with the technical offer of this activity. During the last two month of the activity an extensive set of results for both Linux and RTEMS were obtained.

At the end of each Task2 and Task4 we have set a milestone, i.e. *Mid-term review (milestone MS) and Final-Review milestone MS2* respectively, in which strategic decisions will be taken.

3.1 Task 1: Study of the literature and requirements definition

During this task, we generated an initial list of the processor components of the NGMP that may affect the execution of an application, taking into account the requirements of the different application types, i.e. payload-like applications as well as on-board control applications. It is important to remark that each application type may stress different processor components, which in turn may affect the execution time of other co-running applications. For example, memory-intensive applications may generate a lot of traffic not only into the memory controller, but also into the processor bus. Thus, applications that use the same processor bus, like I/O-intensive applications, may be potentially delayed although not being of the same type.

Concretely the work performed is:1) Study of the Literature. 2) Collecting an initial list of the NGMP shared component and the execution profile of the most important

run-time features of payload and on-board control applications. 3) Study other benchmark suites such as Parsec or EEMBC.

This task has ParSec and EEMBC benchmark suites as inputs. As an output it has: (1) Analysis of the benchmark suites to use in the project, and (2) an analysis of the literature

3.2 Task 2: Benchmark implementation

In this task we developed the benchmark infrastructure and load that mimic ESA reference applications. The framework consists of several components:

1. The **Front-end** or **Generator**: this component runs on the host machine. It takes as an input the loads and their parameters and generates the necessary configuration files and executables to run on the NGMP target system.
2. The **Back-end** or **Executor**: this component is in charge or running the specified loads in a determined order and configuration. It consists of a general framework and several loads (benchmarks) executed as specified by the user during the build and configuration of the experiments. There is a back-end for Linux and one for RTEMS.
3. The **Analyzer**: this component collects and processes the data generated during the execution of an experiments and provides statistics used to analyze the output of the experiments. The input data comprise of the execution time (measured on the NGMP) and performance counters (obtained from the JTAG) of each load. This component runs on the host machine (offline analysis) and consists of bash, perl, python, and matlab scripts.

The benchmark framework is able to run both control-like and payload-like applications as shown in the introduction. In several cases, the porting of applications on NGMP (mainly Parsec applications) was not available and significant time was invested to port some of them (namely, X264, Blackscholes, Ferret, and Dedup).

This task takes as an input the outcome of Task 1, in addition to the GRMON software to connect the ML510 board to a host computer, RTEMS and Linux. As an output it provides: (1) the micro benchmark suite; and (2) the benchmark suite manual (D3).

3.3 Task 3: Benchmark dry-run

During this task, the benchmark suite developed in Task 2 was executed on a preliminary version of the NGMP implemented on a Xilinx ML510 evaluation board by Aeroflex Gaisler. The available NGMP design consists of 4 cores but does not feature FPUs (instead a software floating-point library is used to perform application's floating point operations).

This task takes as an input the outcome of Task 2, the Aeroflex Gaisler NGMP implementation on a Xilinx ML510 evaluation board, GRMon and an implementation of Linux and RTEMS for NGMP. As an output it provides (1) a Revised Benchmark suite; (2) a revisited Benchmark suite manual and (3) the final Report of the execution of the benchmarks on the board (D4).

4 Design and Implementation

4.1 The ML510 Board: setting it up and the tool chain

This section describes the steps followed to set up an Aeroflex Gaisler NGMP FPGA prototype design for Xilinx ML510 development board.

The test evaluation board is a Xilinx ML510 development board equipped with a Next Generation Multipurpose Microprocessor (NGMP) design from Aeroflex Gaisler. The development board is described in [XML510SC][XML510RD][XML510ED] but only a subset of the interface provided by Xilinx are enabled. A complete list of the available interface can be found in [GNGMP]**,** in this document we will focus on the following interfaces:

1. Ethernet: the first and only Ethernet MAC is connected to the top Ethernet connector.
2. UART: both UARTs are enabled and UART0 is the lower connector.
3. JTAG: the JTAG interface is connected and can be access through J9.

Among those interfaces only the Ethernet and the JTAG interfaces are connected to the debug link **[**GNGMP**].** The development board is connected to the workstation through one or more of these interfaces. The workstation is a normal PC running 32-bit Ubuntu Linux version 11.04. This configuration was selected after consulting with Aeroflex Gaisler. Although running MS Windows is possible, the JTAG debug interface require a 32-bit driver and compiling such driver on a 64-bit MS Windows is said to be non-trivial.

The main connection between the Xilinx ML510 development board and the workstation is through a Xilinx USB Cable II JTAG. The JTAG is connected to the J9 connector on the board and to a standard USB port on the workstation (NB: the USB connector must be able to supply enough energy to power on the JTAG). Using the JTAG on the workstation requires the installation of a device driver and some extra system configuration. The rest of this section describes these processes while section [GRMON] describes the use of the JTAG to connect to the board.

In order to boot and run applications on the NGMP development board several tools need to be installed and configured. Most of these tools are installed in the Workstation and the compiled and assembled as an image into the board. This section briefly describes such tools. A more detailed description can be found in [GLINUX]. The tools are available on Areoflex Gaisler [GAISLER] web site, the precise URLs can be found in [GLINUX]. Aeroflex Gaisler distributes a crosstool-NG toolchain based on GCC and fixed for the NGMP processor for both Linux and RTEMS. The Linux package also contains the configuration required to build a custom GLIBC.

The Linux toolchain installation steps are detailed in [GLINDRV] and consist of simply extract and install the package provided by Aeroflex Gaisler. The RTEMS cross-compiler (RCC) toolchain installation steps are described in [GRCC]. Although the document, as most of the RTEMS documentation, describes the installation of the RTEMS toolchain for LEON2/LEON3/ERC32, the process works for the NGMP processor as well.

This LEON Linux RAM loader combines the Linux kernel and the root file system image into a single image that can be loaded through the board device interface or a PROM/FLASH card.
GRMon is a tool provided by Aeroflex Gaisler to interact with the development board. The tool can be used in two ways:

1. Console interface: In this mode GRMon can be started from any terminal console. Throughout this document we use this mode.
2. Graphical interface: This mode requires the installation of an additional tool (GRMonRCP) also provided by Aeroflex Gaisler [GNGMP].

In order to run GRMon full version, a special USB protection key is required. This protection key is provided by Aerroflex Gaisler together with the development board. In order to make it running in Linux (MS Windows follows a similar process) a special package (HAS Sentinel) is required. The package is provided by Aeroflex Gaisler together with the board but installing the canonical Ubuntu 11.04 package is simpler and does not require extra configuration.
GRMon is mainly designed for interactive debugging purposes. As such, the latency of sending commands and receiving information to/from the NGMP board over the JTAG connection is not critical: the user issues commands and stop the processor while analyzing the results and, eventually, issues other commands. In our experimental environment, instead, we wish to automatically send commands and retrieve information. We explored with Gaisler several options, reported in the following:

1. Placing breakpoints before/after staring the loads: the problem is the symbols are not known when loading the ramdisk and we need to wait till we start the program in Linux. This option was not practical and, perhaps, impossible.

2. Application trapping: make the application trap, with an invalid opcode, and return the control to GRMon to read data. Unfortunately, GRMon catches either all exceptions or none (which would make GRMon trap at every Linux exception or interrupt).

3. Attach GDB to GRMon. This would work but it requires considerable work in order to automatize the process.

4. GRMon poll/runpoll command to monitor the performance counters during the execution of applications on the NGMP board. This would work but requires extra synchronization techniques, as GRMon, on the host workstation, and Linux, on the NGMP board, are not synchronized. In particular, this extra synchronization is needed between the data collected on the NGMP board (timing, application start/stop) and the data collected from GRMon (performance counters). We perform such synchronization offline, after the test is concluded. Note that coordinated start/stop of the performance counters at the exact moment on which the load start/stop, would require on-chip sampling. This, in turn, would require a Linux kernel module that interacts with the NGMP statistical unit (writing/reading to the statistical unit's I/O memory is protected and require operating system's privileges).

The final option we have chosen was option 4.

We leverage several features provided by GRMon:

1. Batch script: GRMon allows the user to provide an input file containing a list of commands to be executed in order by GRMon. This eliminates the need of issuing commands such as loading the disk image, initializing the statistical unit, etc. manually.
2. Logging: GRMon is able to log the output on an external file on the host workstation
3. Counter polling: This option allows the user to continuously poll the statistical unit once the executable loaded on the NGMP board starts. This functionality was not present in the version of GRMon delivered with the board (1.1.47) and has been added successively (version 1.1.50). Gaisler provided us with the updated software and manual.

The main issue we had with retrieving information from the NGMP board is the latency of the JTAG connection. Sampling the values of performance counters with a frequency higher than 1 Hz is not practical, as the latency of the JTAG cable would not sustain that frequency.

The MKPROM2 tool is required for creating PROM/FLASH images. This document does not describe the use of PROM/FLASH to autonomously start development board without a JTAG debug interface, as this was not used during the project. MKPROM2 is distributed by Aeroflex Gaisler [GLINBLD].
The compilation process is conveniently tied together in Linuxbuild. Linuxbuild tool performs the following steps:

1. Download additional packages from external web sites (such as the Linux kernel from [LINUX]). The step is required before proceeding to the compilation of the components and it is performed through the update option the first time the script is executed. Both the Linux kernel and the boot loader need to be selected for the update [GLINDRV]. Notice that the workstation needs to be connected to Internet for this step to work.
2. Configure each individual component. There are default configuration files in gaisler/configs that can be loaded through the “Save/Load Configuration” option in Linuxbuild. Loading a configuration file updates the configuration of all components. In particular, the following component can be configured at this stage: Toolchain, Linux kernel, Boot loader, Buildroot (creates the root File System that has to be compressed it into a Ramdisk) and MKPROM2
3. Compile the components and merge them into a single image.

Once all the packages are installed, the system can be configured and build using the configuration scripts. We used the default configuration file lb_config_ngmp1_v8fpu.tar.bz2, the only one provided for the NGMP processor. The final image will be available in the `output/` directory. Notice that, at this stage, components that have not yet been downloaded will be downloaded.

The current official Linux kernel [LINUX] supports LEON3 and LEON4 processors, thus there is not need of extra kernel patches for the processor. A separate Linux kernel driver is, however, required for the SpaceWire device [GLINDRV].
We used the default configuration file lb_config_ngmp1_v8fpu.tar.bz2, the only one provided for the NGMP processor. However, this configuration file is generic and does not take into account the two available designs (2 cores with FPU and 4 cores

without FPU). Our design is a 4-cores without FPU design, thus, the option FPU must be removed from the kernel configuration.
Theoretically, any Linux distribution with support SPARC can be used with the development board. As a matter of fact, however, there are not many. Aeroflex Gaisler provides patch for GCC to build custom images with buildroot. The image can be used as root file system mounted with MTD from a FLASH card of loaded with initramfs from the debug interface (the approach followed in this document).
The compiled Linux kernel must be loaded into a pre-defined address location in RAM. During the compilation process, this address must be provided statically. In the process described in this document we used the default memory address.
The Linux kernel and, in our case, the distribution ramdisk are wrapped together through the LEON Linux RAM Loader (`mklinuximg`). The resulting image can then be loaded through the debug interface or the PROM/FLASH card (see Section [**MKPROM2**]).
The image created with Linuxbuild (boot loader + Linux kernel + Ram Disk) can then be loaded with the command `load` from the GRMon console: and started with the command `run`.

4.2 The loads: microbenchmarks

The purpose of the micro-benchmarks is to put a constant load on a processor resource. They are, hence, specifically designed to reach that objective. Each benchmark stresses one of the main resources of the processor, i.e. Integer execution units, L1 data cache, L2 data cache, AMBA AHB Processor and Memory Bus and the Memory Controller. They are written in assembler and are composed of a partially unrolled loop to cause the maximum effect of inter-task interferences in NGMP resources. In the following subsections, different micro-benchmark families are presented.

4.2.1 CPU micro-benchmark (CPU)

The objective of this micro-benchmark is to see if programs not accessing the memory hierarchy suffer from inter-task interference. Its loop is composed of *add* instructions, which are CPU intensive. The structure of the micro-benchmark is the following:

```
for (i=0; i<it; i++) {
  a=a+b; b=a+b;
  a=a+b; b=a+b;
  // … repeated 508 more times
}
```

The assembler code for the main loop is composed of the following instructions: 512 *add*, 1 *subcc*, 1 *br*.

4.2.2 Load-instruction micro-benchmarks

Several load-instruction based micro-benchmarks have been prepared. With these benchmarks we want to determine how sensitive tasks accessing different memory hierarchy levels are to inter-task interference.
This family of micro-benchmark initialize memory in a way that then it will be accessible through indirect load instructions (pointer chasing), which allows longer sequences of instructions.
The structure of the initialization code of the matrix is the following. The *stride* and the *array_size* determine how often a benchmark hits/misses 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.

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

For this micro-benchmark the main loop is composed of *ld* instructions, which access different levels of the memory hierarchy depending on how we configure each micro-benchmark. In particular, the code of the micro-benchmarks is the following.

```
for (i=0,a=M; i<it; i++) {
  b=*a; a=*b
  b=*a; a=*b
  // … repeated 124 more times
}
```

The assembler code for the main loop is composed of the following instructions: 128 *ld*, 2 *cmp*, 2 *br*, 2 *nop*. The load-instruction based micro-benchmark family contains the following micro-benchmarks:

- **L1 Load hit (L1)**: 8KB data footprint, always hits the private L1 cache.
- **L2 Load hit ($L2_{40}$)**: 40KB data footprint, always misses the private L1 cache, always hits the shared L2 cache.
- **L2 Load full ($L2_{200}$)**: 200KB data footprint, always misses the private L1 cache, hits the L2 cache when no other concurrent processes are using it, misses the L2 cache when other concurrent processes are using it.
- **L2 Load miss ($L2_{miss}$)**: 8MB data footprint, always misses L2.

4.2.3 Store-instruction based micro-benchmark ($L2_{st}$)

The objective of this micro-benchmark is to check the effect of having a lot of store instructions in a program may affect its performance when running it together with other store-intensive programs. This is relevant because NGMP's L1 data cache implements a write non-allocate, write-through policies, so all store operations access directly to the shared L2 cache, thus increasing the risk of inter-task interference.
This benchmark uses *st* instructions to store a value in the memory position defined by a pointer plus an immediate value. The store micro-benchmark has a 40KB data footprint, which hits the L2 cache if no other concurrent processes are using it, and misses the L2 cache if other concurrent processes are using it.
The structure of the micro-benchmark is the following:

```
for (i=0; i<it; i++) {
  M[p+(imm+=stride)]=a;
  M[p+(imm+=stride)]=a
  // … repeated 506 more times
}
```

The value of *imm* in the structure description is updated after every store; we use a different immediate value for each store. Next we show a fragment of the loop after the unrolling has been applied, considering a stride of 32 bytes:

```
M[p] = a;
M[p+32] = a;
M[p+64] = a;
```

There is a limitation to the maximum value of the immediate, which is 4095 for the target architecture. As a consequence, if we use 32-byte stride, after 127 stores the *p* pointer has to be updated as shown next.

```
M[p+4032] = a;
M[p+4064] = a;
p += 4096;
M[p] = a;
M[p+4] = a;
```

The main loop of the store micro-benchmark is composed of the following instrictions: 3 *cmp*, 3 *br*, 1 *mov*, 1 *clr*, 508 *st*, 6 *add*.

We wrote our loop bodies in assembler, and this assembler code is marked as _volatile_ so the compiler will not try to change the unrolled loop. Other instructions might be generated by the compiler, such as subcc, addcc, cmp and br instructions.

To ensure no extra memory accesses in case a –O0 optimization level is used, induction variables are declared with the *register* keyword

The object code generated for every micro-benchmark has been checked and the overhead added by the compiler has been taken into account. For instance, in section 4.3.1 one *subcc* and one *br* instructions are in the loop body in addition to the 512 *add* instructions which were explicitly inserted by us. We unrolled the loops enough to make this overhead negligible.

4.3 The loads: Mimicking benchmark

In this section we describe the main characteristics of the ESA mimicking benchmark. This benchmark is meant to copy the main characteristics of several ESA applications taken as a reference.

4.3.1 Parameters to consider

As part of the deliverables presented in month 2, the consortium agreed on the main parameters required to mimic ESA applications: Memory footprint, execution time duration and number of lines of code.
Month 2 deliverables also provide some recommendations when designing the mimicking benchmark: 1) Usage of simple *c*-language statements so that the lines of assembly code per *c* line of code is limited; 2) Usage of simple conditions; and 3) Do not use many nested conditions.
Next we show a survey of the main parameters observed in the characterization given by ESA to BSC of the applications *AOCS* and *PacketWire*, each comprised of several threads. Both applications are executed independently from one another. The numbers from the two Tables below, where obtained on a single-core architecture, i.e. at any point in time there was only one thread running.

- In the AOCS characterization there were several threads, each of which was categorized into a group. The main groups are: Application, AOCS, System and Payload.

	Application	**AOCS**	**System**	**Payload**
CPU usage	17%	35%	11%	32%

Memory usage	22kB	450kB	1M	10M
Lines of code	Not provided	Not provided	Not provided	Not provided

- In the case of the PacketWire we group threads into the following groups: System asynchronous activities (e.g. handlers), system synchronous activities, AOCS, Platform and Payload.

	SysA	SysS	AOCS	Platform	Payload
CPU usage	5%-12%	6%-7%	3%-15%	4%-16%	1%-5%
Memory usage	<1kB	<1kB	<1kB	<1kB	<1kB
Lines of code	Not provided	Not prov.	Not prov.	Not prov.	Not prov.

4.3.2 Benchmark Design

We have designed an Automatic *Benchmark Generator* tool, (based on perl script), which is able to construct a benchmark that replicates the behavior of other applications based on the parameters defined above. Concretely, it allows configuring:

- Instruction mix: the amount of loads and stores,
- Lines of code: number of instructions inside the loop
- memory footprint and stride
- memory accesses as burst or evenly distributed,
- Duration: done through the number of iterations.

The tool generates a C program composed of a memory initialization part and a loop with assembly code which will take most of the overall execution time. This loop performs the configured amount of load and store instructions. Every instruction, other than load and store is an add instruction, which do not access the memory hierarchy. Load instructions inside the loop traverse an array using pointer chasing, while store instructions traverse a different array referencing it with a pointer plus an immediate. The allocated memory depends on the amount of load and store instructions. For example, for a 200KB data footprint, with 10% loads and 20% stores, the array traversed by the store instructions will be twice as big as the one traversed by the load instructions, so 133KB for stores and 67KB for loads.
In the example below, we set the following configuration in the script:

```
my $DATA_FOOTPRINT      = 200*1024;
my $STRIDE              = 32;
my $ITERATIONS          = 1000;
my $LD_PERCENT          = 10;
my $ST_PERCENT          = 20;
my $INST_PER_ITER       = 100;
my $EVEN_DISTRIBUTION   = 1;
```

This will generate a C program with a 200KB array, accessing memory with a 32 byte stride, a loop body with 100 instructions 10% of them being loads and 20% of them being stores, and distributed evenly though the code. A fragment of the resulting code can be seen below:

```
for (i = 0; i < 1000; i++) {
        __asm__ __volatile__(
                "add %2,  1, %2"        "\n\t"
```

```
"add %2, -1, %2"        "\n\t"
"st %2, [%3+32]"        "\n\t"
"add %2,  1, %2"        "\n\t"
"add %2, -1, %2"        "\n\t"
"add %2,  1, %2"        "\n\t"
"add %2, -1, %2"        "\n\t"
"st %2, [%3+64]"        "\n\t"
"add %2,  1, %2"        "\n\t"
"ld [%0], %1 "          "\n\t"
"add %2, -1, %2"        "\n\t"
"add %2,  1, %2"        "\n\t"
```

If `$EVEN_DISTRIBUTION` is set to 0, then the operations of each type are executed consecutively, the following program will be generated:

```
for (i = 0; i < 1000; i++) {
        __asm__ __volatile__(   "ld [%0], %1"   "\n\t"
                                "ld [%1], %0"   "\n\t"
                                "ld [%0], %1"   "\n\t"

[...]
                                "st %0, [%1]"   "\n\t"
                                "st %0, [%1+32]"        "\n\t"
                                "st %0, [%1+64]"        "\n\t"
[...]
                                "add %0,  1, %0"        "\n\t"
                                "add %0, -1, %0"        "\n\t"
                                "add %0,  1, %0"        "\n\t"
```

After each iteration some checks are done to ensure store instructions will not access out of bounds in next iteration. These check takes only 4 assembly instructions, and are subtracted from the number of add instructions in the loop. The percentages of load and store instructions are not affected by this.
This C code can be compiled with GCC. A –O2 optimization setting is suggested.

4.3.3 Setups

Based on this generic benchmark we generated several configurations to mimic the ESA application behavior. Regarding the memory footprint, we can differentiate 4 cases: memory footprint fits in L1, partially fits in L1, fits in L2 and does not fit in L2. All designed mimicking benchmarks have 30% memory operations from which 70% are loads and 30% stores. There was no characterization of the percentage of load and stores of ESA applications. We have used these figures, based on previous analysis of other benchmark suites

- MckBench_1KB: This benchmark fits in L1 data cache (memory footprint of 1KB). This provides the same data cache footprint than all thread of PacketWire and the *Application* thread in AOCS.
- MckBench_22KB: This benchmark partially fits in L1 data cache (memory footprint of 22KB). This covers the *Application* thread in AOCS.
- MckBench_450KB: This benchmark fits in L2 data cache. This covers the *AOCS* thread in AOCS.
- MckBench_1M: This benchmark does not fit in L2 data cache
    - This covers the *System* and *Payload* thread in AOCS.

Once the slowdown factors due to inter-task interferences are obtained when these mimicking benchmarks run in the NGMP, it is normalized based on the *cpu_usage* to see the actual affect on the execution time of the application:

*Actual_slowdown = cpu_usage * observed_slodown*

4.4 The loads: CoreMark and EEMBC

The EEMBC (Embedded Microprocessor Benchmark Consortium) is a non-profit association which aims to develop embedded benchmarks for processor evaluation. EEMBC targets telecom/networking, digital media, Java, automotive/industrial, consumer, office equipment products, and two generic benchmark suites specifically developed for targeting multi-core processors and processor cores.
The EEMBC benchmarks suites that are of interest for this project are the one mimicking control applications: *AutoBench* and *CoreMark*.
MultiBench has been also analysed, although discarded to be used in this project. MultiBench is a suite of embedded benchmarks that allows processor and system designers to analyze multi-core architectures and platforms. Unfortunately, MultiBench encompasses workloads from the networking, consumer, and office automation domains, not including control applications.

4.4.1 EEMBC AutoBench

AutoBench is a suite of benchmarks that allow users to predict the performance of microprocessors and microcontrollers in automotive, industrial, and general-purpose applications. It is composed of 16 benchmark kernels including:

- *Generic workload tests*. These tests include bit manipulation, matrix mapping, a specific floating-point tester, a cache buster, pointer chasing, pulse-width modulation, multiplication, and shift operations (typical of encryption algorithms).
- *Basic automotive algorithms*. These tests include controller area network (CAN), tooth-to-spark (locating the engine's cog when the spark is ignited), angle-to-time conversion, road speed calculation, and table lookup and interpolation.
- *Signal processing algorithms*. These tests include algorithms which are becoming increasingly important for sensors used in engine knock detection, vehicle stability control, and occupant safety systems. They include Fast Fourier Transforms (FFT and iFFT), a finite impulse response filter (FIR), an Inverse Discrete Cosine Transform (iDCT), and an Infinite Impulse Response (IIR) filter.

The characteristics of each benchmark kernel are the following:

- Each kernel is composed by a single function (*t_run_test*) which contains the control loop. The control flow only contains a single path, allowing to improve the coverage of WCET analysis when using WCET tools such as RapiTime.
- The number of iterations of this loop is determined by an input parameter (*-I*).
- The input data required by each benchmark is hard-coded inside the source code.
- The total size of the working set of each benchmark kernel is 32KB for data and 4KB for code.

The size of the working set may not be enough to stress certain hardware shared resources of the NGMP such as the AMBA bus or the L2 shared cache. The size of the instruction and data caches of the LEON4 are 16 KB each.

All benchmark kernels contain the file same structure. The most important ones are *bmark_lite.c* and *algotst.c*:

- *Bmark_lite.c.* It contains the *main* function, which calls the *t_run_test.*
- *Algotst.c*. It contains the hard-coded input data used by the control loop. The data is stored in an array called *inp[bench]ROM*, being *bench* the type of data contained in the array. For example, in case of *aifftr01* the array is called *inpSigROM,* while in case of *a2time01* the array is called *inpAngROM.* In order to enlarge the input data considered by the benchmark it is required to enlarge this structure.

Lite versions of EEMBC have been compiled as control benchmarks. Lite EEMBC are not interactive and can be configured for not printing any output. This is perfect for our execution infrastructure, as the interaction with the system is minimized.
EEMBC implementing floating point operations have not been considerd as the target board does not implement an FPU. These benchmarks are: a2time, aifftr, aiifft, basefp, idctrn, iirflt, matrix and tblock.
The following table shows a list of EEMBC benchmarks used and the number of configured iterations for each one of them. The number of iterations is fixed to an amount that makes the benchmark run for at least 30 seconds.

EEMBC	Iterations
aifirf	1M
bitmnp	40K
cacheb	10M
canrdr	10M
pntrch	50K
puwmod	10M
rspeed	6M
ttsprk	300K

4.4.2 CoreMark

CoreMark benchmark suite (composed of only one benchmark) has been designed specifically to test the functionality of a processor core. CoreMark contains 3 types of functionalities, each of which is implemented with a different set of functions.

- *Matrix-related functionality*: multiply matrices, add matrices, add constant to a matrix.
- *Linked list management operations*: add, remove, insert, find, ….
- *Finite State Machine operations*

In each run of the CoreMark all these functions are executed. Depending on the input parameters the time each of these functions run can vary.

Several input arguments (parameters) can be given to the CoreMark benchmark. This is configured through the *make* file. To add compiler flags from the command line, we can use XCFLAGS e.g.

```
make XCFLAGS="-g -DMULTITHREAD=4 -USE_FORK=1".
```

Some of the main parameters are:

- ITERATIONS: By default, the benchmark will run between 10-100 seconds. To override, use ITERATIONS=N

- Several copies of the benchmark can be run in parallel. To that end use `make XCFLAGS="-g -DMULTITHREAD=4 -DUSE_FORK=1"`
- Several implementations are available to execute in multiple contexts by using `make XCFLAGS="-DUSE_FORK=1`. We can use fork or posix threads api (`-DUSE_PTHREAD`).
- Method used to allocate data, `MEM_METHOD = MEM_STATIC, MEM_MALLOC or MEM_STACK`

The CoreMark executable itself also accepts different parameters.

- 1st - A seed value used for initialization of data. Could be any value.
- 2nd - A seed value used for initialization of data. Must be identical to first for iterations to be identical.
- 3rd - A seed value used for initialization of data. Any value should be at least an order of magnitude less then the input size, but bigger then 32.
- 4th - Number of iterations (0 for auto : default value). Special, if set to 0, iterations will be automatically determined such that the benchmark will run between 10 to 100 secs
- 5th - Reserved for internal use.
- 6th - Reserved for internal use.
- 7th - For malloc users only, ovreride the size of the input data buffer.

The buffer size for the algorithms must be defined via the compiler define TOTAL_DATA_SIZE. TOTAL_DATA_SIZE must be set to 2000 bytes (default) for standard runs. The default for such a target when testing different configurations could be make XCFLAGS="-DTOTAL_DATA_SIZE=6000 -DMAIN_HAS_NOARGC=1".

We compiled the CoreMark benchmark suite on Linux on an Intel architecture. We vary some of the input parameters and determine the effect of this variation on the duration of each function. This characterization was not done on the target architecture because this task would consume too much time.

ARGUMENTS (input parameters)							Functions										
Random	Random	Random	Iterations	Reserved for internal use	Reserved for internal use	Data Size	matrix_mul_matrix_bitextract()	core_state_transition()	matrix_mul_matrix()	ee_isdigit()	crcu8()	matrix_sum()	core_list_reverse()	core_list_mergesortv()	core_list_find()	core_bench_state()	**TOTAL PERCENTAGE**
0x0	0x0	0x66	2K	7	1	16K	32	29	14	11	4	3	0	0	0	0	**93**
0x0	0x0	0x66	2K	7	1	2K	5	28	4	4	33	0	16	0	0	0	**90**
0x0	0x0	0x66	2K	7	1	32K	39	25	0	17	0	2	0	0	0	0	**83**
0x0	0x0	0x66	2K	7	1	2K	13	21	8	8	39	0	0	0	0	0	**89**
0x127	0x127	0x127	2K	7	1	2K	0	23	4	11	21	0	11	5	5	5	**85**

4.4.3 CoreMark configurations

To execute the CoreMark benchmark we have selected to compile a single binary file and then use different arguments to call it. Two different configurations have been used:

- Coremark 8KB (c8): 8KB data footprint. Does fit in the private L1 cache. The number of iterations is forced to 300 to make the execution time on the target board take approximately 30 seconds. The following command line arguments have been passed to the coremark executable: 0x127 0x127 0x127 300 7 1 8192
- Coremark 32KB (c32): 32KB data footprint. Does not fit in the L1 cache. The number of iterations is forced to 15 to make the execution time on the target board take approximately 30 seconds. The following command line arguments have been passed to the coremark executable: 0x127 0x127 0x127 15 7 1 32767

4.5 The loads: ParSeC

PARSEC is a benchmark suite that features multithreaded workloads. This is a presentation of the basic features of the 2.0 version, including the different workloads and their functionality, the program directory tree, the different build configurations, the available input sizes and the script that is provided for ease of use.

Parsec does not officially support SparcV8 (it does though support SparcV9). In addition to this, the NGMP does not implement an FPU, so floating point instruction are not supported either. Porting the Parsec to SparcV8 took significant effort and we

were not able to compile all of the benchmarks. In addition to this problem, some Parsecs presented input data which was too big for our platform. Finally, blackscholes, dedup, ferret, and x264 were successfully compiled.
The execution of Parsec is done with the parsecmgmt script. For this project we need to execute the binary file directly as we have limitations in the size of the image file loaded to the board, and also limitations in the developed infrastructure, which expects an executable file to be called and not a script. This has forced us to analyze the script to find out the command line arguments we had to use.
Problems were found when trying to execute two of the compiled Parsecs in the NGMP:

- Dedup: prints the following error: "Memory allocation failed". The cause to this error could be having too little memory available.
- Ferret: Uses a data base as an argument and some parameters associated to this database as arguments. It also presents some auxiliary programs which are possibly to manage or create this database. It was not possible to find out the way it works, so ferret is not used as a benchmark.

Blackscholes and x264 were used as payload benchmarks. The following commands were used to build them:

```
parsecmgmt -a build -p x264 –c hfleonv8-cross-gcc
parsecmgmt -a build -p blackscholes –c hfleonv8-cross-gcc
```

Binary files are generated in pkgs/ [apps|kernels]/ <benchmark_name>/ inst/ hfleonv8-linux/ bin. The following table shows the input data set and the parameters used to run them.

Parsec	Data set	Parameters (N=number of threads)
blackscholes	sim_small	N inputs/sim_small /dev/null
x264	sim_small	--quiet --qp 20 --partitions b8x8,i4x4 --ref 5 --direct auto --b-pyramid --weightb --mixed-refs --no-fast-pskip --me uhm --subme 7 --analyse b8x8,i4x4 -o /dev/null inputs/sim_small --threads N

The input file is selected to make benchmarks run for longer than 30 seconds. Sim_dev takes few seconds to execute, and sim_medium takes more than 5 minutes on the target architecture, so sim_small was used as input set.

4.6 Execution infrastructure

Our framework is composed of one front-end and two back-ends targeting Linux and RTEMS. The execution infrastructure (EI) is comprised of a set of scripts, C and C++ programs that allows anyone connected to the host machine to run experiments on NGMP and collect results. These scripts enable us to speedup the process of doing experiments and generating results.
Note that the framework is not focused on task scheduling. It does not deal with task priorities either. The user provides the scheduling to follow in each core (i.e., which sequence of tasks will be executed in each core). Figure 2 shows a general view of our framework

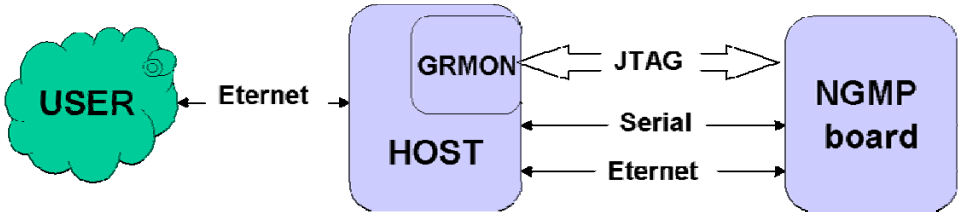


Figure *1 .General view of the execution infrastructure*

The framework allow the execution on Linux or on RTEMS. Two different workflows exist to allow this dual behaviour, but most of the programs and scripts are common for both OS.

4.6.1 Linux workflow

Three main scripts comprise the execution infrastructure for Linux, see Figure 2. The first two form the front-end and the third the back end.

- *runbench*: This is the common front-end. It processes the user parameters, compiles the loads (benchmarks) and gets the executables that will be run on the NGMP.
- *script_grmon*. It interacts with the GRMON to read performance monitoring counters.
- *runbench_ngmp*: It runs on the NGMP. It spawns threads, run loads in each core.

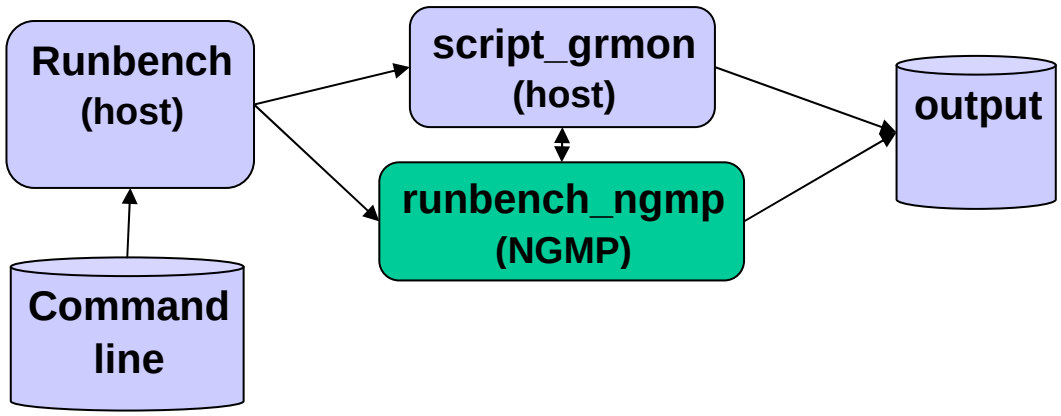


Figure 2. Three main scripts that compose our Linux execution infrastructure

These main scripts do several functionalities as shown in the next detailed diagram (Figure 3):

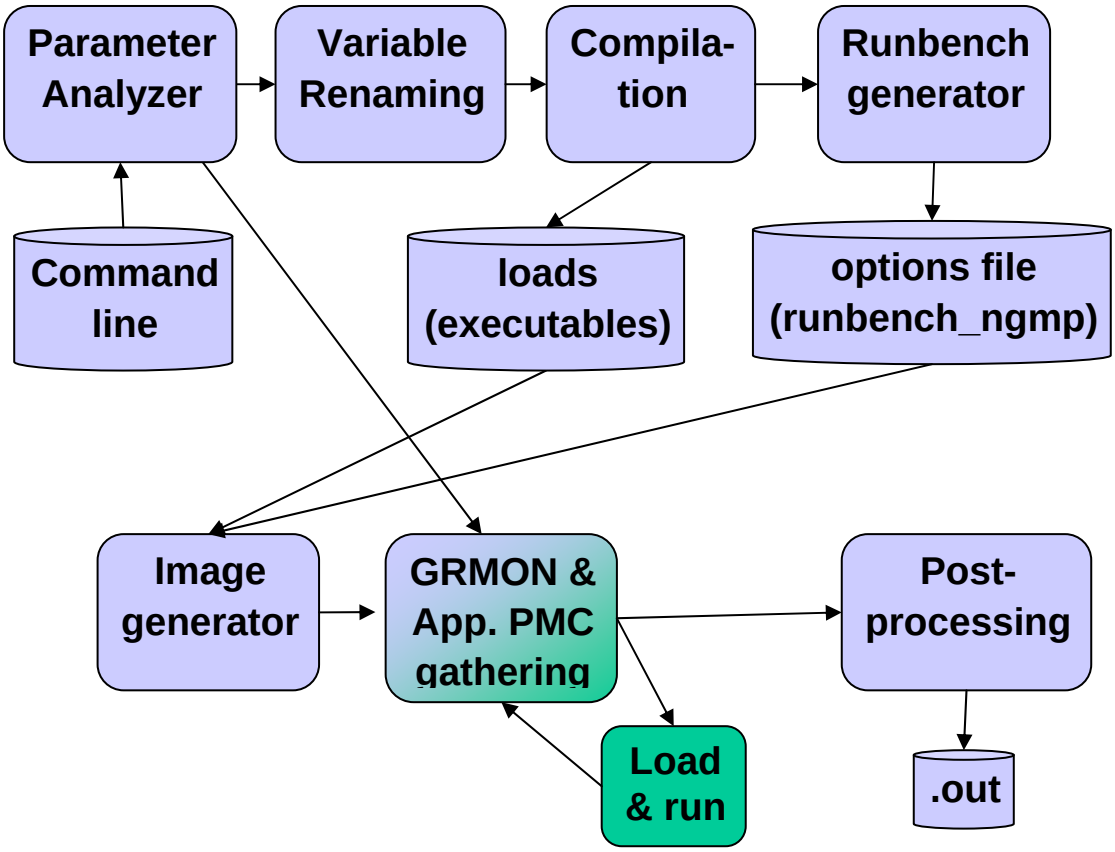


Figure 3. Components of the different scripts for Linux

4.6.2 RTEMS workflow

Three main scripts comprise the execution infrastructure (see Figure 4). The first two form the front-end and the third the back end.

- *runbench*: This is the common front-end. It processes the user parameters, compiles the loads (benchmarks) and gets the executables that will be run on the NGMP.
- *script_grmon*. It interacts with the GRMON to read performance monitoring counters.
- *Makefile*: Compiles generated source files for RTEMS and generates the bootable RTEMS images.

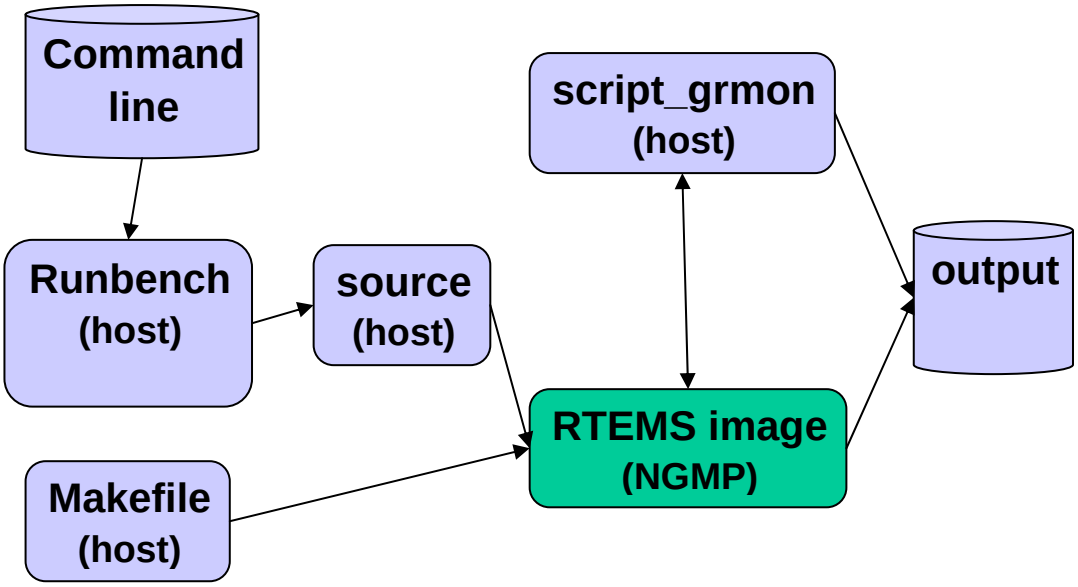


Figure 4. Three main scripts that compose our execution infrastructure for RTEMS

These main scripts do several functionalities as shown in the next detailed diagram (Figure 5):

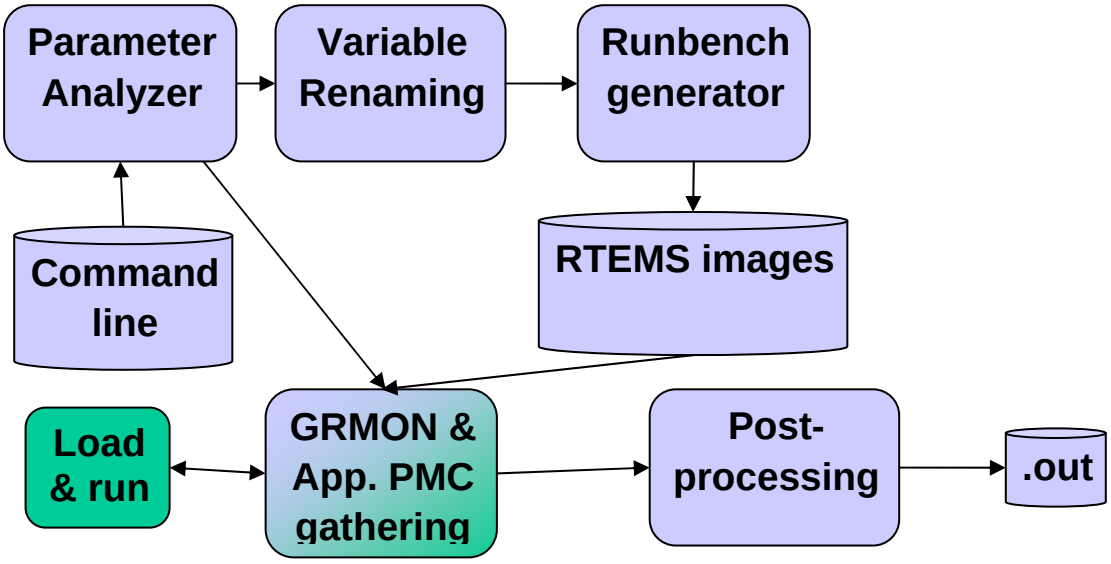


Figure 5. Components of the different scripts for RTEMS

4.6.3 Parameter analyzer (Linux and RTEMS)

The parameter analyzer is in charge of parsing the command line given by the user and checking it is semantically correct. It will report an error otherwise. It creates a set of data structures that will be query by the other scripts in order to understand the parameters set by the user.

Next we show which the parameters of the framework are. In defining it, we use the following notation:

- $\{\}_N$ parameter that repeats at least N times
- [] optional parameter

- <> generic name

Elements in lists are comma separated.

```
./run_bench
   -lname <name>
   -lpmcs <list_pmcs>
   {-lsrcfile   <filename>   {-lvar   <varname>   -lvalue
<value>}1 }0
   [-lmakefile <mfile>
    -lmake_ofile <mofile>]
   -lexecfile <efile>
   [-loptions "<argument list>"]
   -lexec_ofile <eofile>
   -lbind <bind_list> }1
   [-lperiodicity <period>]
   [-pf <parameters_file>]
```

The user can specify the commands in the command line or through a parameter file by using the option -pf <parameters_file>.
The user specifies a set of loads to run. For each load, the user may also specify some source files to generate that load and new values for some new variables. The execution framework changes the value of the desired variables in the source files of the load and recompiles it. The user has to specify:

- load name: each load starts with this parameter and the rest of options follow. Several loads may be defined.
    - -lname <name>
- Source files and variables to rename and new values. This parameter is optional. Several source files can be defined, and for each one of them, several variable/value pairs may be defined.
    - {-lsrcfile <filename> {-lvar <varname> -lvalue <value>}1 }0
- makefile to execute that automatically generate the executable file. This parameter is optional and can be skipped if the executable is provided instead of the source code. This option is only used in Linux. For RTEMS a specific Makefile is provided, which is used after the RTEMS code is generated.
    - [-lmakefile <mfile>]
- On Linux, the executable file that will be generated by the makefile or that will already exist if the makefile is not provided. On RTEMS, name of the function that will be called.
    - -lexecfile <efile>
- The file in which the output of the compilation process will be put. Optional, and may only be defined if a makefile has been configured.
    - [-lmake_ofile <mofile>]
- The file in which the output of the executable (load) will be put
    - -lexec_ofile <eofile>
- On Linux, the arguments that will be passed to the executable file. On RTEMS, function parameters to be passed to the configured function. This is optional, no arguments will be passed if this is left unconfigured.

    - –loptions “<argument list>”

- The core (or cores) where the load will be bound. In the 0-3 range, separated by commas.
    - -lbind <bind_list>
- The PMCs that will be read. Identified with the same name used by GRMON and separated by commas.
    - -lpmcs <list_pmcs>
- A load may be configured as periodic, meaning it will recurrently be executed at a periodic rate. Period is specified in milliseconds.
    - -lperiodicity <period>

To define the parameters in the parameter file, the same syntax applies. Each option has to be put in a different line without the heading dash.

4.6.4 Variable renaming (Linux and RTEMS)

For each input file this script changes the values of the given variables. It generates the values of the variables in each file have been changed.
It opens each srcfile, changes the value of <varname> in that file by the given one. The format of the line where parameters are found has to be one of the following:

- <typename> <varname> = <value>;
- <typename> <varname> = “<value>”;
- #define <constname> <value>

4.6.5 Compilation (Linux)

As an input this script receives the makefile and the executable file. Also the files in which to show the error and output messages of the make file.
It spawns a new process that executes make and print the result of the compilation process. Finally it checks that the executable has been generated

4.6.6 Runbench option generator (Linux)

The option generator creates an options file containing the data structures with the parsed input parameters. This file (or files) will be used by runbench_ngmp on the board to run the loads. They have opt extension and are loaded by runbench_ngmp.

4.6.7 RTEMS Runbench program generator (RTEMS)

A RTEMS program is automatically generated from the configured parameters. It is a C file that is ready to be compiled using the provided Makefile and RTEMS 4.10 provided by Gaisler. When compilled, the bootable RTEMS images will be generated. To configure RTEMS loads, keep in mind that the lexecfile parameter is used as the function name to be called, and not as the executable file. Thus *lexec*_ofile option is ignored, and the loptions is used not as command line arguments but as function parameters to the function specified by lexecfile. If more than one parameter (option) is specified, they will be passed as an integer array to the function. For example, the following configuration:

lexecfile ub_l2_200
loptions “200 50”

Will generate a code similar to this:

```
int param[] = {200, 50};
ub_l2_200(param);
```

Currently only int parameters are allowed as function parameters.

4.6.8 Linux Runbench Image generator (Linux)

The image is generated using Gaisler provided tools, specifically the linuxbuild tool. The basic configuration had to be changed so the Linux image generated uses software emulated floating point instead of using floating point instructions.
In this stage the executable files generated in the compilation phase and the options file containing all the load data are added to the operating system image.

4.6.9 runbench_ngmp (Linux)

Once the generated image is loaded into the board using the J-Tag interface, Linux is booted and runbench_ngmp executed. Runbench_ngmp will execute the configured loads according with the values stored in the opt files stored by Runbench. Runbench_ngmp is invoked from a start-up script.

4.6.10 Script_GRMON (Linux/RTEMS)

PMCs are read by GRMON during the execution of each micro-benchmark on the board. Two log files with all the information are generated: one with PMC data and the other one with timing data. The information in these two files is combined to obtain meaningful results during the result analysis phase.

4.6.11 Scripts for result analysis

A series of scripts have been developed to analyze the results obtained. These scripts are made to work analyze PMC data and timing data. Each one of these analysis require different kinds of experiments to be used.

4.6.11.1 PMC analysis

To collect PMC data our initial intention was to generate an interruption to trap back to GRMON and get the values. This approach was not applicable because GRMON does not allow configuring which interruptions were going to be handled by Linux and which ones were going to be handled by GRMON. At the end, and following Gaisler’s recommendations, counters are polled once per second. More information about GRMON and different methods studied for reading PMCs can be found in previous versions of document D4 “*Report of the Execution of the Benchmarks on the Board*”. Also, some PMCs were not functioning until November 2011, until Gaisler corrected a bug and flashed the bug fix into the board’s memory card.

4.6.11.1.1 Methodology

The scripts read the values of the PMC data generated during the execution of the load and provide the mean of each counter for each core. For this kind of analysis the loads comprise a single execution of a process in each core to avoid the overhead generated by the operating system (see figure 6). That interval has to be long enough to allow several PMC reads to happen. In our experiments, the duration of the loads is of 30 seconds or higher.

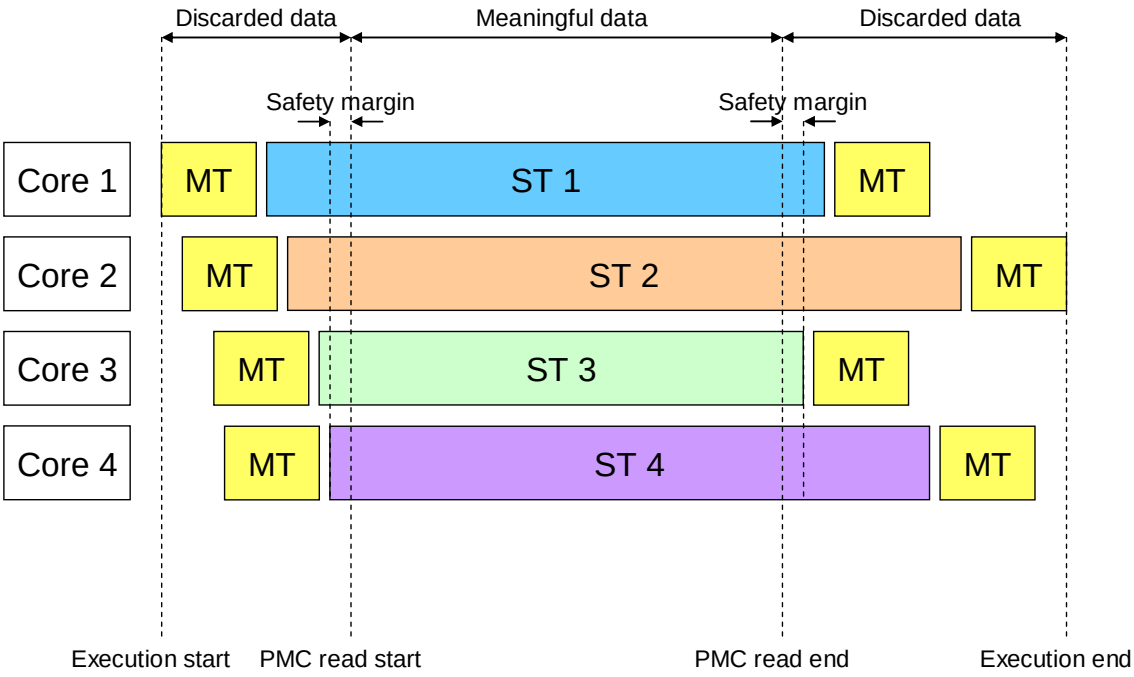


Figure 6. PMC analysis

4.6.11.1.2 Scripts

PMCs are read and reported by GRMON through the runpoll command. GMRON logs this data and timestamps it. Meanwhile, runbench_ngmp runs the loads on the board and reports their start and end time. This time is provided by Linux's gettimeofday() and does not match the time logged by GRMON. A set of scripts are provided to match these times and generate compact output files:

- *parse-time.py* / *parse-time-rtems.py*: the execution time reported by the OS run on the board for each experiment is saved in a .log.dat file in a more convenient format, ready to be read by Matlab as an input file.
- *parse-experiments.py*: PMC data output by GRMON is stored in .log.dat files in a more convenient format, ready to be read by Matlab.
- Matlab scripts (*run.m*): Matlab reads the .log.dat files generated by the preivous Python scripts and generate core0.log, core1.log, core2.log, and core3.log files, where for each experiment (row) 16 columns are set (4 configured PMCs per each core) with each PMC averaged. Run.m script should be modified to change the directory where the input files are located, as well as the experiment list.

4.6.11.2 Timing Analysis

4.6.11.2.1 Methodology

For measuring inter-task interference effects on timing, loads are configured as a series of repetitions of the same set of applications. The scripts read the reported execution times of each repetition, remove non-concurrent execution data and provide the execution mean time for each one of the cores (see Figure 7). The duration of STs

has to be long enough to make MT overhead (<10ms) negligible. The data extracted from the experiment in Figure 6 would be the execution time of ST1 (one repetition), ST2 (one repetition), ST3 (one repetition), and ST4 (mean of two repetitions). In reality, experiments are repeated more than 2 or 3 times, and hence we obtain more data to calculate more accurate statistics.

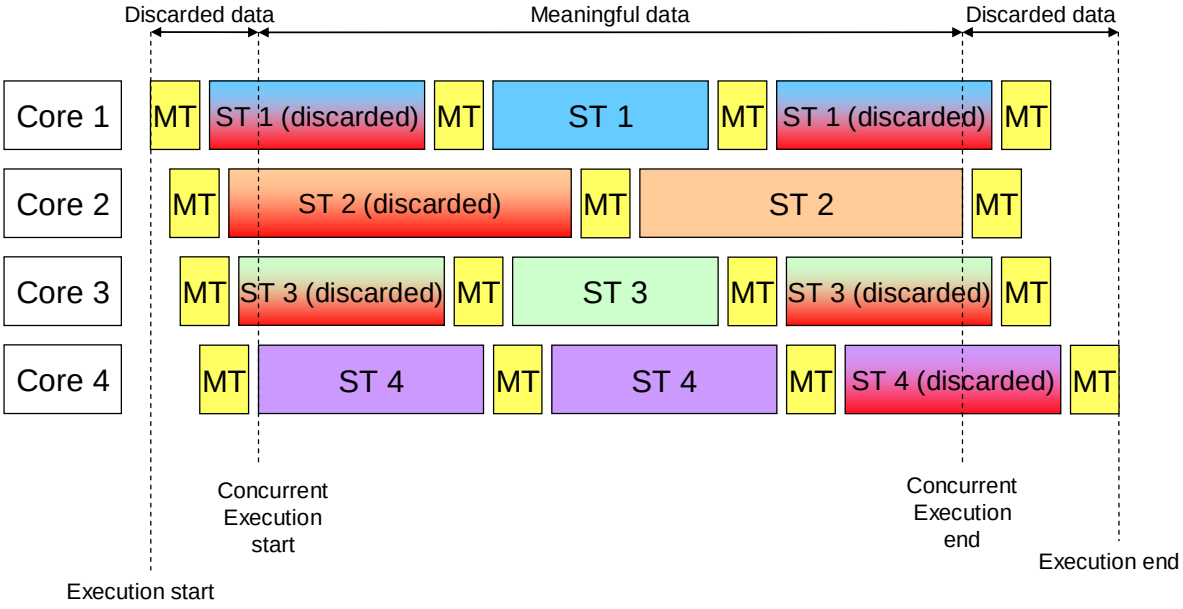


Figure 7. Measuring the effect of inter-task interferences on timing

4.6.11.2.2 Scripts

Timing data can be extracted from the log files using the *timing.sh* shell script provided.

4.6.11.3 Periodic task with different opponent in each activation

Our framework also allows to measure the deadline miss rate for a given periodic control application when it runs as a part of a workload with other applications. Note that, our framework does not focus on task scheduling but is the user providing the schedule of the tasks to execute. Our framework does not deal with task priorities either and tasks are not preemptable.
In order to do that, we bind the control application to a given core, and the payload applications to the other cores. In D4 we have explored two scenarios:

- 1 control application and 1 payload application
- 1 control application and 3 payload applications

In reality other scenarios with 3 or 4 control applications can be explored as well.
The user provides a given deadline for the control task. Then the execution of the control application and the payload applications is done as show in Figure 8. In this depicted scenario the control application runs against a different set of arriving and leaving payload applications. If a task finishes in the middle of a given period, the next instance of the task is delayed and started at the next period boundary.

Once the execution is done, a simple script is used to compare the different execution times of the control application again the user given deadline as follows.

```
hits = 0;
misses = 0;
if ( execution_time_isolation * allowed_increase > execution_time_i)
      hits=hits+1;
else
      misses=misses+1.
```

The total deadline hit rate is given by: hits/(hits+misses)%. In the evaluation section we show the results we have obtained for some scenarios in which we use EEMBC as control applications.

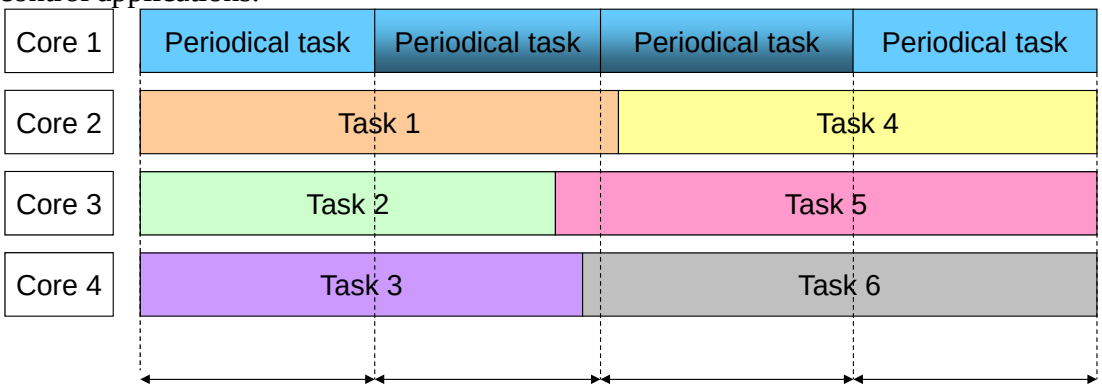


Figure 8. This set up could be used to check whether the periodic task will miss the deadline or not.

5 Evaluation and Discussion

5.1 Metrics

The main metric we want to take into account is the slowdown tasks suffer in the NGMP due to inter-task interferences. To that end, our execution environment allows running different 'workloads' in the desired cores, and periodically reading PMCs to derive the impact that inter-task interferences may have on the execution time of different programs. In particular, we measure data and instruction cache misses, L2 cache misses, number of load and store instructions, AMBA AHB bus utilization, and total number of instructions executed per second.

In order to compute the execution time slowdown of a task, we first run the program under study in isolation and then as part of a workload. With this, we compute inter-task interferences as the ratio between the average counter value for runs in isolation and for runs as part of a workload. The result is averaged from the PMC values polled each second.

We have prepared 4 different types of workloads:

- Workloads composed of Micro-benchmarks: They help bounding the maximum variation tasks may suffer due to inter task interferences. In all cases, as reference execution time we have the execution time of each micro-benchmark when running it in isolation. We run different sets of micro-benchmarks and compute the execution time variation of each of them: quadruples. For instance (L2 L1 ADD MULT) and (L1, L1).
- Workloads composed of CoreMark and Micro-benchmarks: They help bounding the maximum inter-task interference that a typical control task, modeled by CoreMark, may suffer. We study the execution time variation of CoreMark using an 8KB data footprint and a 32KB data footprint when running it concurrently with micro-benchmarks.
- Workloads composed of EEMBC AutoBench and micro-benchmarks: Similar to the previous case, they help bounding the maximum impact of inter-task interferences on EEMBC benchmarks.
- Workloads composed of EEMBC AutoBench and Parsec: These two benchmarks model the typical control task and the typical payload task. This test helps us understanding the effect of inter-task interference with reference applications, but does not help knowing the maximum inter-task interference level.

5.2 Results on Linux

5.2.1 Micro-benchmarks only executions

We designed several experiments showing the effect of inter-task interferences on:

- (1) AMBA AHB Processor Bus, which connects all cores to the L2.
- (2) Memory bandwidth and both the AMBA AHB Processor and Memory Buses.
- (3) Memory bandwidth, L2 cache and both the AMBA AHB Processor and Memory Buses.

- (4) Data cache Write-through policy effect on stores under high load.

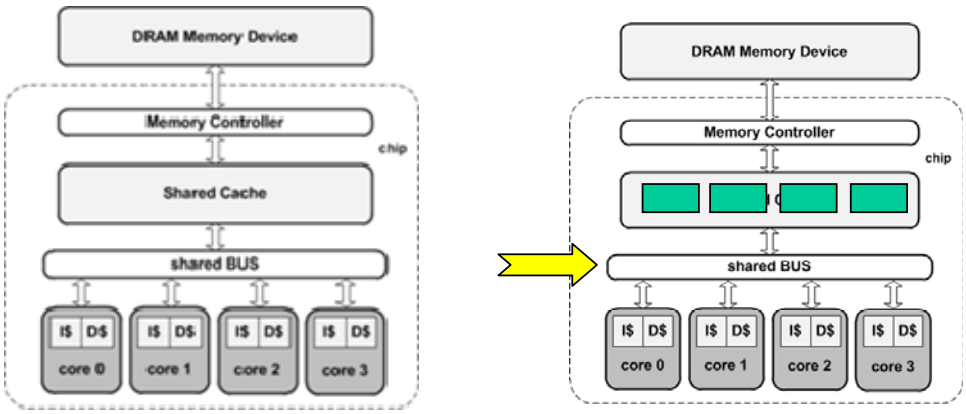
Finally, we provide some results that show the importance of task scheduling in minimizing inter-task interferences.

5.2.1.1 Overhead of the AMBA AHB Processor Bus

In this experiment we determine the effect that interactions in the AMBA AHB Processor Bus may have over program execution time. To that end, we implemented the $L2_{40}$ micro-benchmark that:

- When running in isolation it always misses in L1 and hits in L2. This is done by having a data footprint higher than 16KB that is the size of Data cache and properly accessing it.
- When running up to four copies of this benchmark, it is the case that all the data of the 4 copies fits in L2, see *Figure 9*. To that end, we make data footprint to be smaller than 256/4=64KB. In particular we choose a footprint of 40 KB.

With this experiment, the difference between the execution in isolation and the execution of several copies of this benchmark will be mainly due to interferences in the AMBA AHB Processor Bus. Notice, that regardless of the number of copies the number of hits per thread is the same, so are the number of access to the bus. Hence, the total number of AHB accesses and of L2 hits linearly increase with the number of copies of the benchmark executed.



(a) high-level view of the NGMP processor

(b) Resource stressed by the micro-benchmarks

Figure 9. Experiment to stress the AMBA bus

Figure 9 shows the amount of data cache and L2 misses when running the $L2_{40}$ micro-benchmark in isolation, two instances of the $L2_{40}$ each running in a different core, and four instances of the $L2_{40}$ each running in a different core. We observe that Data cache misses stay stable almost at 100%, and L2 cache misses stay low near 0%.

	L2 miss per load			DC miss per load		
	1	2	4	1	2	4
L2-40	0.08%	0.06%	5.94%	99.73%	99.74%	99.60%

Figure 10. L2 and data cache misses when running L2-40 on 1 core, 2 cores or 4 cores.

Figure 11 shows the results obtained for this experiment. We observe that the worst delay due to sharing the AMBA AHB Processor Bus is 12% when two tasks are

executed and 83% when 4 tasks are executed. This results are a bit higher than reported in MERASA project, where results where around 27% for 4 cores [PQCG+09]. The may reason ways that in [PQCG+09] the bus model used was simpler and hence has lower latency and less effect on applications.

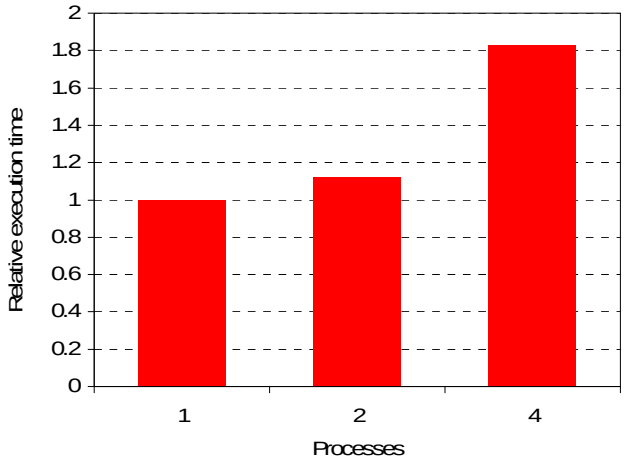


Figure 11. Experiment to stress the AMBA bus

5.2.1.2 Overhead of the memory bandwidth and the AMBA AHB Processor and Memory Buses

In this experiment our focus is on determining the effect that interactions in the memory controller and in both the AMBA AHB Processor and Memory Buses may have on program's execution time. With that aim we implemented the $L2_{miss}$ micro-benchmark that:

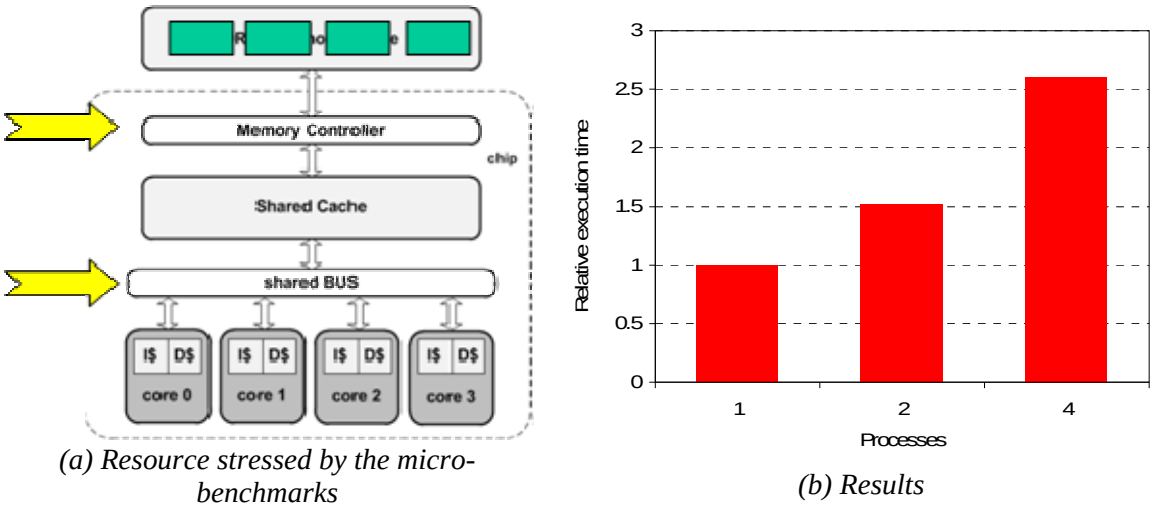
- When running in isolation it always misses in data cache and L2 cache. This is done by having a data footprint higher than 256KB that is the size of L2 cache and properly accessing it. In particular we choose a footprint of 8 MB.
- When running up to four copies of this benchmark, each copy misses the L2, see Figure 5(a), so no additional L2 misses happen per process. Note that all 4 copies are independent processes so they do not share data or instructions (beyond the code of some shared libraries).

With this experiment, the difference between the execution in isolation and the execution of several copies of this benchmark will be mainly due to sharing the memory bandwidth. To access the memory controller, both buses are also used, but as we can see comparing Figure 11 and *Figure 13 13* (b), the bottleneck is in the memory controller. Notice, that regardless of the number of copies of L2-miss the number of misses in L2 is the same, so are the number of accesses to memory. Figure 12 shows the percentage of data cache and L2 misses when running $L2_{miss}$ micro-benchmark in isolation, two instances of itself, and four instances of itself.

	L2 miss per load			DC miss per load		
	1	2	4	1	2	4
L2-miss	100.00%	99.59%	98.51%	99.20%	98.89%	98.08%

Figure 12. L2 and data cache misses when running L2-miss on 1 core, 2 cores or 4 cores.

We observe that the ratio is roughly the same regardless of the number of copies. So the effect on this benchmark is not on the number of L2 misses it suffers but on the time to solve each L2 miss due to contention on the memory controller.



(a) Resource stressed by the micro-benchmarks

(b) Results

Figure 13. Experiment to stress the memory bandwidth and AMBAAHB Processor and Memory buses

Figure 13(b) shows the execution time slowdown obtained when running $L2_{miss}$ in isolation (1), with two simultaneous copies of $L2_{miss}$ (2) and with four simultaneous copies of $L2_{miss}$ (4), in both cases each running in a different core. We observe that the worst delay due to sharing the memory bandwidth is 50% when two tasks are executed and 160% (2.6x) when 4 tasks are executed. These results are in accordance with the results obtained in [PQV+09].

Two Considerations have to be made when analyzing the results:

- Our NGMP evaluation board runs at 70MHz while the memory interface works at 140MHz. This is due to the fact that the NGMP is implemented in an FPGA. In reality, the processor will have higher frequency than memory, so the effect of interferences in memory will be much higher.
- This benchmark shows the worst possible effect of interferences sharing the memory bandwidth. In reality, programs will not constantly access to memory but will alternate CPU and memory phases, so the effect of interferences will be smaller.

5.2.1.3 Overhead of the memory bandwidth, the L2 cache, the AMBA AHB Processor and Memory buses

In this experiment we determine the effect that inter-task interferences arised in the whole memory hierarchy, i.e. the memory controller, the L2 cache and both the AMBA AHB processor and memory buses, may have over program execution time. To that end, we implemented the $L2_{200}$ micro-benchmark that:

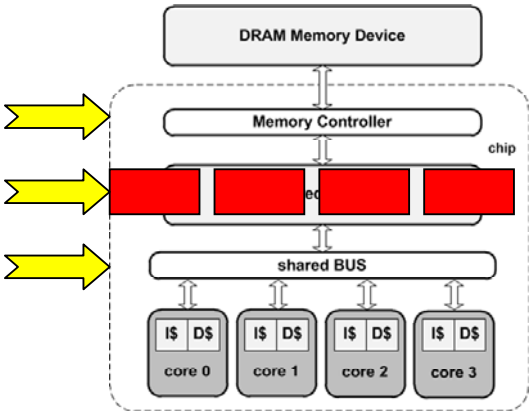
- When running in isolation it misses in data cache and hits in L2. This is done by having a data footprint higher than 16KB which is the L1 cache size, and smaller than 256KB, which is the size of L2 cache, and properly accessing it.
- When running up to four copies of $L2_{200}$ benchmark, it is the case that the data of all the copies does not fit in L2, see *Figure 15*(a). To that end, we make

data footprint to be higher than 256/2=128KB. In particular we choose a footprint of 200 KB.

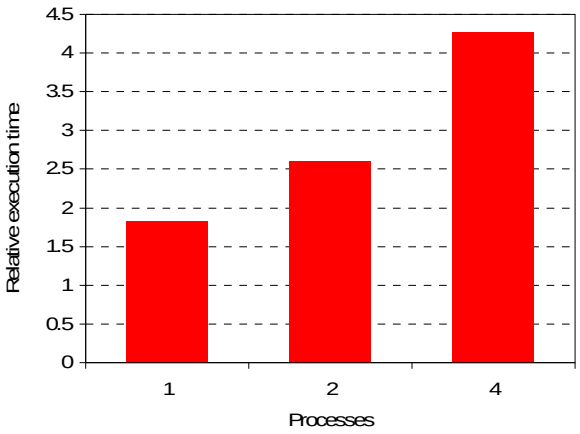
With this experiment, the difference between the execution in isolation and the execution of several copies of this benchmark will be mainly due to sharing the L2 cache and memory bandwidth. To access the L2 and the memory controller, the AMBA AHB Processor and Memory buses are also used, respectively, but as we can see comparing *Figure 11* and *Figure 15*(b) the bottleneck is in the L2 and the memory controller. Notice, that one copy of the benchmark running alone has most of its data in the L2 cache, so it does not have to access memory frequently. Performance degradation is caused by the sharing of the L2, which causes all processes to go to memory to get the data. Therefore, the number of accesses to main memory is increased from few when running in isolation to almost always when running more than one copy. Figure 14 shows the amount of data cache and L2 misses when running this micro-benchmark in isolation, two instances of itself, and four instances of itself. We can see a stable behavior on missing both L1 cache, and an increasing number of misses in L2, caused by the different instances competing for the shared resources.

	L2 miss per load			DC miss per load		
	1	2	4	1	2	4
L2-200	31.45%	88.42%	98.55%	99.53%	98.98%	98.09%

Figure 14. L2 and data cache misses when running L2-miss on 1, 2 and 4 cores.



(a) Resource stressed by the micro-benchmarks



(b) Results

Figure 15. Experiment to stress the memory bandwidth, the L2 cache, and the AMBA bus

Figure 15(b) shows the results obtained for this experiment. We observe that the worst delay due to sharing the memory bandwidth and L2 cache is 2.3x when two tasks are executed and 4.3x when 4 tasks are executed.

The considerations shown in the previous subsection about memory frequency hold here as well.

5.2.1.4 Write-through policy effect on stores under high load

The L1 data cache in the NGMP implements a write-through policy. This means that store instructions will always access directly to the L2 cache to store the data there, and will update the data cache as well only if the accessed cache line resides in it. This has important implications on inter-task interference, as the L2 cache is always accessed, even if the data footprint of the program fits the L1. In this section we study the interferences caused by memory stressing programs on store intensive programs.

A micro-benchmark composed by more than 95% *store* instructions and a 40KB data footprint was prepared based on the parsec script used to mimic ESA applications. We call this micro-benchmark $L2_{st}$, and it was executed concurrently with memory hierarchy stressing micro-benchmarks. In particular, 3 instances of $L2_{40}$, $L2_{200}$ and $L2_{miss}$ respectively have been used as interfering programs.

Figure 16 shows that $L2_{40}$ and $L2_{200}$ and $L2_{miss}$ cause a big slowdown, up to almost 20*x*, in the execution of the store micro-benchmark. 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 make it quite sensitive to other benchmarks using the bus.

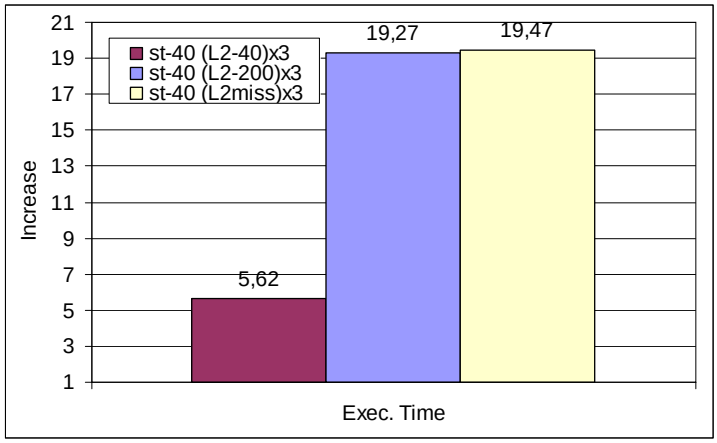


Figure 16. Increase in execution time of the store benchmark under different workloads

In addition to AMBA bus contention, another source of slowdown is that when $L2_{st}$ has to store a value to an L2 cache line owned by another process, the line has to be stored to main memory, if it is dirty, before being replaced.

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

5.2.2 Executions of Mimicking benchmarks

In this section we show the sensitivity of the mimicking benchmarks to inter-task interferences. In order to do so, we execute each of the three benchmarks we designed with our framework (MckBench_1KB, MckBench_22KB, MckBench_450KB and MckBench_1M) against the microbenchmarks stressing the bus, the L2 cache and the main memory ($L2_{st}$, $L2_{40}$, $L2_{200}$). Results are shown in Figure 17. Though we observe some difference in the slowdown, with the benchmarks with high data footprint with high slowdown, the main factor affecting its performance is the percentage of loads they have, which is common for all them.

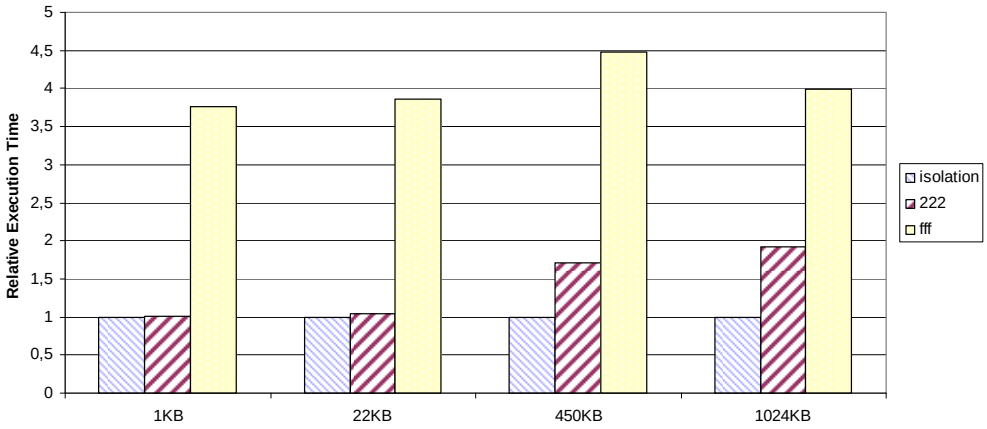


Figure 17. Increase in execution time of the mimicking microbenchmarks under different workloads

5.2.3 Executions of CoreMark with Micro-benchmarks

This section evaluates the execution time variation of the CoreMark benchmark, considering a data footprint of 8 KB and 32 KB (labeled as c8 and c32 respectively), when running each configuration within a workload composed of different micro-benchmarks: $L2_{40}$, $L2_{200}$ and $L2_{miss}$.

Figure 18 shows the execution time variation of c8 when running it within a workload composed of c8 and three copies of c8, $L2_{40}$, $L2_{200}$ and $L2_{miss}$ (labeled as c8-(c8x3), c8-($L2_{40}$x3), c8-($L2_{200}$ x3) and c8-($L2_{miss}$ x3) respectively), taking as a baseline the execution time of c8 running in isolation.

We observed that, when running multiple copies of c8, the execution time does not vary at all as the data footprint of c8 fits inside the data cache. However, when running it with $L2_{40}$, $L2_{200}$ and $L2_{miss}$ the execution time increases 7%, 51% and 51% respectively. These increments are far from the ones observed in previous sections, i.e. 83% due to the processor bus and 4.3x due to processor bus, L2 cache and main memory. The reason behind this difference will be explained in the next section.

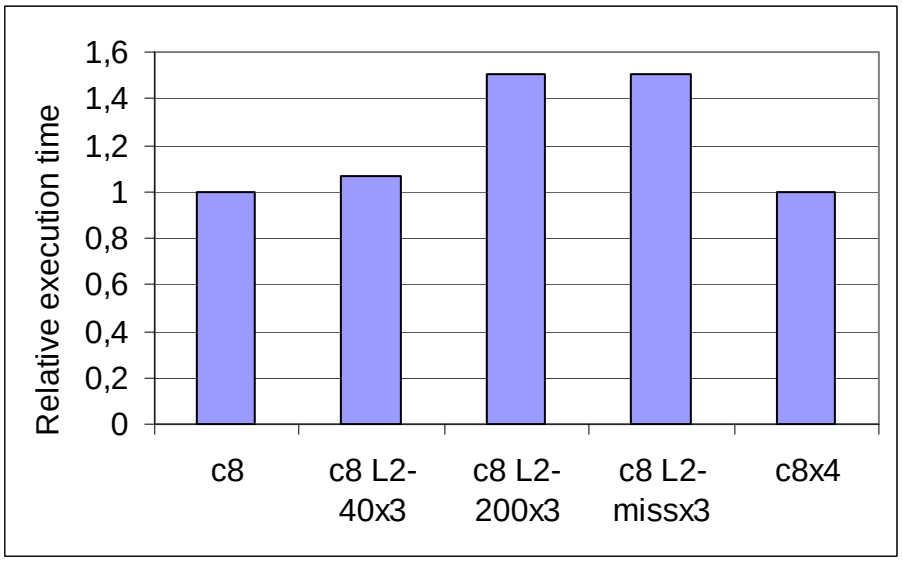


Figure 18. Execution time of CoreMark 8KB in isolation and with different opponents.

Figure 19 shows the execution time variation of c32 when running it within a workload composed of c32 and three copies of c32, $L2_{40}$, $L2_{200}$ and $L2_{miss}$ (labeled as c32-(c32 x3), c32-($L2_{40}$ x3), c32-($L2_{200}$ x3) and c32-($L2_{miss}$ x3) respectively), taking as a baseline the execution time of c32 running in isolation.
Similarly to c8, running multiple copies of c32 does not make the execution time vary as the data footprint of c32 is still too small to collision into the processor bus. However, when running it with $L2_{40}$, $L2_{200}$ and $L2_{miss}$ the execution time increases up to 2%, 31% and 31% respectively.

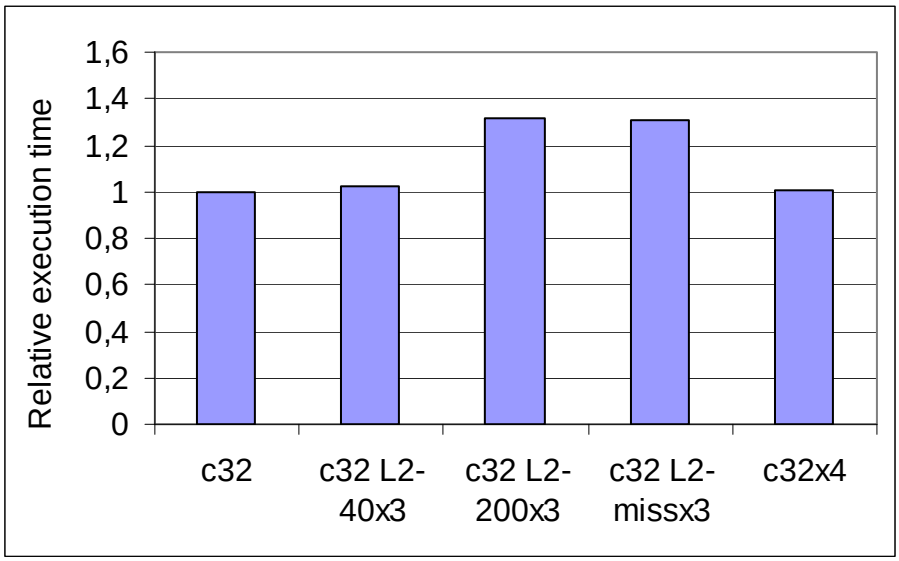


Figure 19. Execution time of CoreMark 32KB in isolation and with different opponents.

5.2.4 Executions of EEMBC with Micro-benchmarks

In this section we consider the EEMBC AutoBench as a control-like application. In order to evaluate the maximum slowdown caused by inter-task interferences on EEMBC, we run it simultaneously with the micro-benchmarks described in previous sections.

5.2.4.1 Execution time

The execution time slowdown observed when executing EEMBC with different workloads is computed considering the the execution time of each EEMBC running in isolation. Figure 20 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).
Note that when running EEMBC benchmarks with several copies of themselves (x2 and x4) the execution time does not increase. When we run EEMBC 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 is may be increased up to 60% in case of the *cacheb*. Such an increment is even much 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*). However, the execution time slowdown of *pntrch* is only 7%. The reason to this behavior is described in next sections.

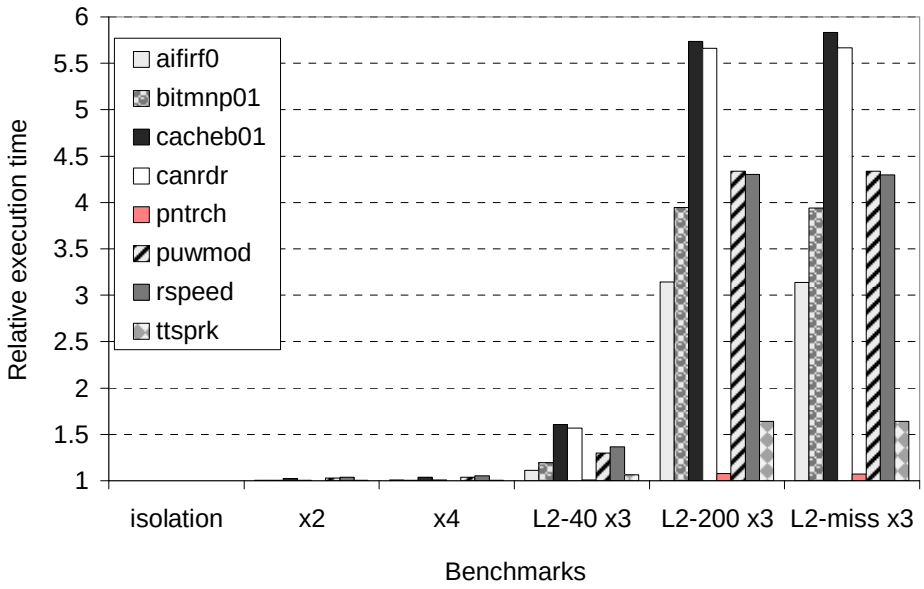


Figure 20. Execution time slowdown of EEMBC AutoBench when running them together with micro-benchmarks, taking as a baseline the execution time in isolation.

5.2.4.2 Characterization

We have used the PMCs data obtained to characterize each EEMBC benchmark in terms of instruction mix and AMBA AHB Processor bus used. Figure 21 shows the results of this characterization. For each EEMBC, the first column 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).

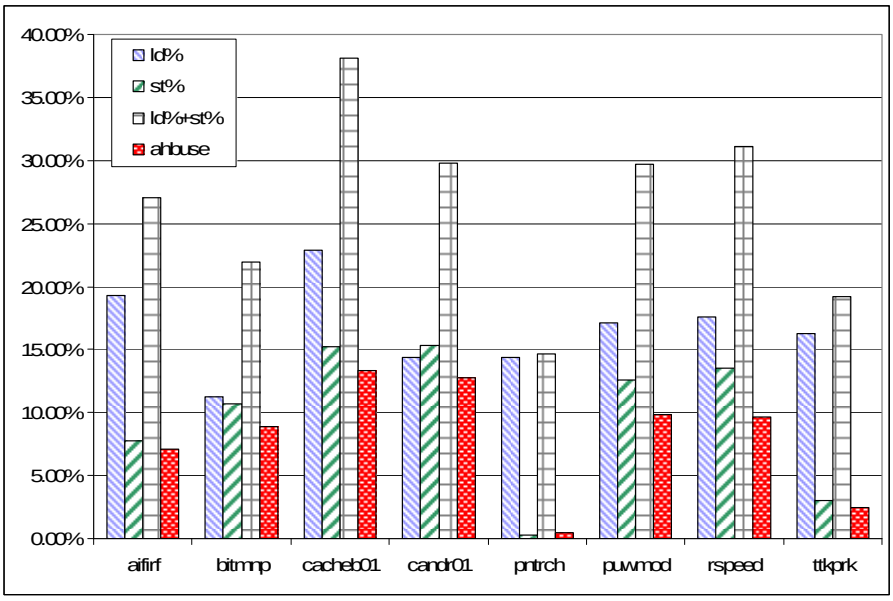


Figure 21. Characterization of EEMBC benchmarks.

We can see 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. Therefore, the observed execution time slowdown of each of the EEMBC benchmarks is caused not only by its data footprint, but also by the instruction mix of the benchmark. Specifically, a very high correlation has been found between the density of store instructions and the slowdown. This correlation is shown in Figure 22.

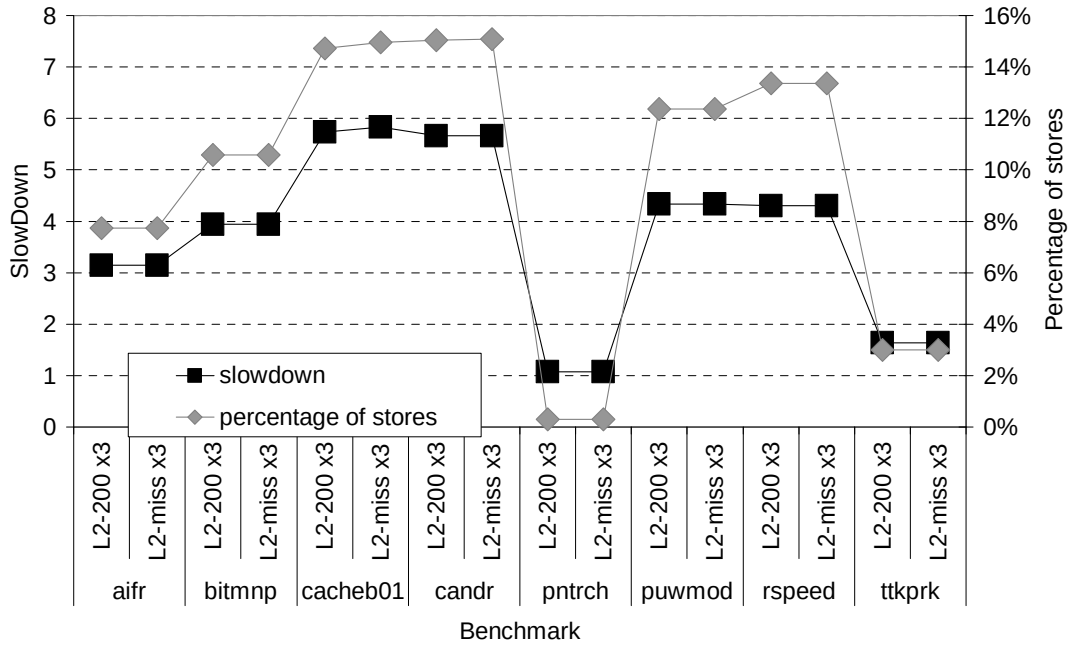


Figure 22. Correlation between percentage of stores (right y-axis) and slowdown (left y-axis) for all studied EEMBC benchmarks, when run together with 3 instances of either $L2_{200}$ or $L2_{miss}$.

As a result, we conclude that control applications with more store instructions suffer of higher inter-task interference. Payload applications with a data footprint higher than the L2 size cause higher inter-task interference.

5.2.5 Executions of EEMBC with Parsec

In this section Parsec is used to mimic payload applications. We used Parsec to test the behavior of EEMBC benchmarks when run concurrently with typical payload applications. This experiment is different to the one presented in the previous section as it is not intended to bound the execution time, but to obtain an estimation of what the average execution time for EEMBC when executed concurrently with a payload application.

We run the EEMBC presented in the previous section together with Blackscholes and x264 in different combinations:

- EEMBC in isolation (baseline).
- EEMBC with a multithreaded version of the Parsec (3 threads).
- Two EEMBC with a multithreaded version of the Parsec (2 threads).

In *Figure 23* the relative execution time of each EEMBC benchmark when executed in each different scenario is presented.

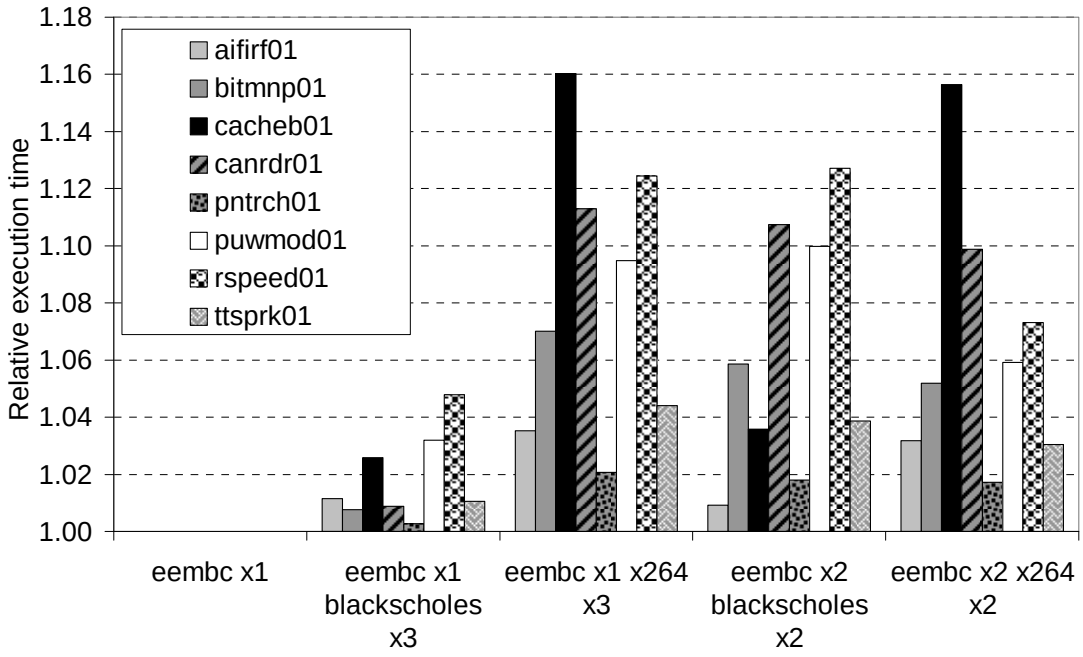


Figure 23. Relative execution time of EEMBC benchmarks when executed concurrently with Blackscholes and x264, in different parallel execution setups.

The worst case of performance degradation is 16% for *cacheb*, the most memory intensive EEMBC, when executed concurrently with x264. As expected, the observed slowdown is lower than the 5.8x boundary found for EEMBC in section 5.2.4.

It is important to remark that the purpose of the parsec is not to stress a given processor resource as it is the case of our micro-benchmarks, hence, it is obvious that the overhead introduced on the control applications is smaller.

The main corollary of these results is that depending on the instruction mix and memory footprint of the control application their time compos ability is affected by the other running applications. This will be extended in the conclusion section.

5.2.6 Periodic task with different opponent in each activation

In this Section we present the results for an scenario in which we select a given application as control application and run it against a different set of payload applications, measuring how often the control application hits its deadline. We have set three scenarios of increasing tightness for the control application. In the first scenario we assume that the deadline is only 30% bigger than the execution time of the control application in isolation, in the second it is 80% and in the thirds it is 200% bigger.

We have explored two thread-count scenarios:

- 2-thread configuration: 1 control application and 1 payload application
- 4-thread configuration: 1 control application and 3 payload applications

As control application we have used *cacheb* and *canrdr* as they show high sensitivity to resource sharing. As payload applications we have used the ParSec benchmarks and the $L2_{40}$ and $L2_{200}$ microbenchmarks.

In the case of the 2-thread configuration we run each of these benchmarks against a recurring sequence of payload applications *Blackscholes*, *x26*, $L2_{40}$ and $L2_{200}$.

In the case of the 4-thread configuration we run each of the *cacheb* and *candrd* against 3 threads, each executing a recurring sequence of paylod appliaction *Blackscholes*, *x26*, $L2_{40}$ and $L2_{200}$.

The results for *cacheb* and *canrdr* are shown in Figure 24 and 25 respectively. We observe that for the scenario in which the deadline is only 30% bigger than the execution time in isolation the percentage of hit deadlines is high for *cacheb* only if one extra thread is running is 97%, in the case there are several threads running the hit rates decreases to 85%. As the deadline increases (80% and 200%) the hit rate increases accordingly. *Canrdr* shows higher sensitivity and hence less hit deadlines.

	Scenario			**Total runs of the control application**
	Scenario 1 (30%)	**Scenario 2 (80%)**	**Scenario3 (200%)**	
1 cntrl –1 payload app	0,97	0,98	0,98	156
1 cntrl –3 payload apps	0,85	0,85	0,85	111

Figure 24. ***Deadline hit rate for*** **cacheb** ***for the 2- and 4 –thread configuration***

	Scenario			**Total runs of the control application**
	Scenario 1 (30%)	**Scenario 2 (80%)**	**Scenario3 (200%)**	
1 cntrl –1 payload app	0,00	0,81	1,00	156
1 cntrl –3 payload apps	0,00	0,89	0,89	111

Figure 25. Deadline hit rate for *canrdr* for the 2- and 4 –thread configuration

5.3 Results on RTEMS

Similarly to the experiments prepared for Linux, the main metric we want to take into account for RTEMS is the slowdown tasks suffer in the NGMP due to inter-task interferences. The execution environment prepared allows the same type of

experiment we have on Linux (allows running different 'workloads' in the desired cores, and periodically reading PMCs), and measuring data and instruction cache misses, L2 cache misses, number of load and store instructions, AMBA AHB bus utilization, and total number of instructions executed per second.
For RTEMS we have focused on workload, composed of micro-benchmarks as they help bounding the maximum variation tasks may suffer due to inter task interferences. In all cases, as reference execution time we have the execution time of each micro-benchmark when running it in isolation. We run different sets of micro-benchmarks and compute the execution time variation of each of them.

5.3.1 Micro-benchmarks only executions

5.3.1.1 Overhead of the AMBA AHB Processor Bus

In this experiment we want to determine the effect that interactions in the AMBA AHB Processor Bus may have over program execution time. To that end, we used the $L2_{40}$ micro-benchmark. Figure 26 shows the amount of data cache and L2 misses when running the $L2_{40}$ micro-benchmark in isolation, two instances of the $L2_{40}$ each running in a different core, and four instances of the L240 each running in a different core. We observe that Data cache misses stay stable almost at 100%, and L2 cache misses stay low near 0%.

			percentage of L2 misses per Id			percentage of D misses per Id		
			1	2	4	1	2	4
L2-40	L2-40x2	L2-40x4	0.25%	0.24%	0.16%	99.53%	99.49%	99.56%

Figure 26. *L2 and data cache misses when running L2-40 on 1, 2 and 4 cores.*

Figure 27 shows the results obtained for this experiment. We observe that the worst delay due to sharing the AMBA Bus is 32% when two tasks are executed and 97% when 4 tasks are executed. This results are a bit higher than the ones reported on Linux, probably because of the less overhead caused by the Operating System; RTEMS does not interfere the base line case and thus it is better, making the execution time of two and 4 concurrent micro-benchmarks relatively worse.

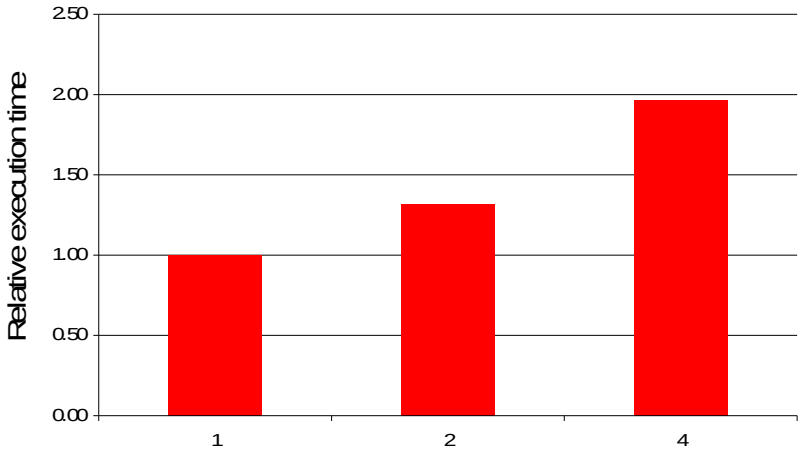


Figure 27. Experiment to stress the AMBA bus

5.3.1.2 Overhead of the memory bandwidth and the AMBA AHB Processor and Memory Buses

In this experiment our focus is determining the effect that interactions in the memory controller and in both the AMBA AHB Processor and Memory Buses may have on program's execution time. With that aim we used the $L2_{miss}$ micro-benchmark. Figure 28 shows the percentage of data cache and L2 misses when running $L2_{miss}$ micro-benchmark in isolation, two instances of itself, and four instances of itself.

			percentage of L2 misses per Id			percentage of D misses per Id		
			1	2	4	1	2	4
L2-miss	L2-missx2	L2-missx4	98.96%	99.09%	99.16%	98.91%	98.97%	99.03%

Figure 28. L2 and data cache misses when running L2-miss on 1, 2 and 4 cores.

We observe that the ratio is roughly the same regardless of the number of copies. So the effect on this benchmark is not on the number of L2 misses it suffers but on the time to solve each L2 miss due to contention on the memory controller.

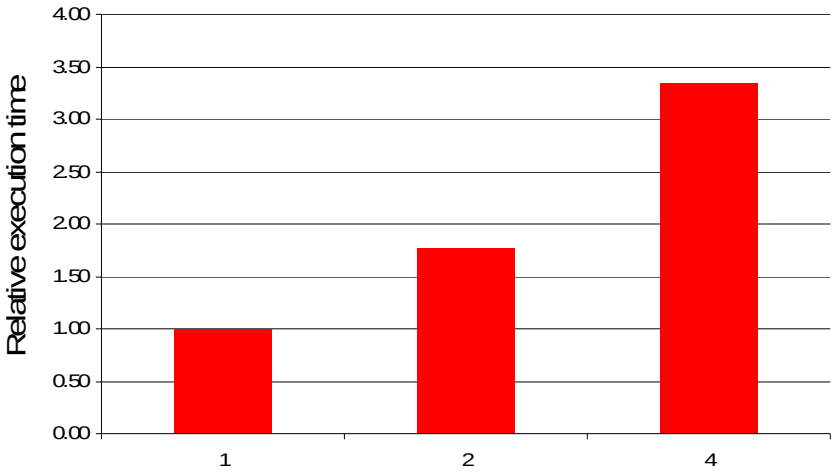


Figure 29. Experiment to stress the memory bandwidth and AMBAAHB Processor and Memory buses

Figure 29 shows the execution time slowdown obtained when running $L2_{miss}$ in isolation (1), with two simultaneous copies of $L2_{miss}$ (2) and with four simultaneous copies of $L2_{miss}$ (4), in both cases each running in a different core. We observe that the worst delay due to sharing the memory bandwidth is 77% when two tasks are executed and 235% (3.35x) when 4 tasks are executed. These results show again higher inter-task interference than on Linux, caused by a better base line case (due to less interference from the operating system).

5.3.1.3 Overhead of the memory bandwidth, the L2 cache, the AMBA AHB Processor and Memory buses

In this experiment we want to determine the effect that inter-task interferences arised in the whole memory hierarchy, i.e. the memory controller, the L2 cache and both the AMBA AHB processor and memory buses, may have over program execution time. To that end, we used the $L2_{200}$ micro-benchmark. Figure 30 shows the amount of data cache and L2 misses when running this micro-benchmark in isolation, two instances of itself, and four instances of itself. We can see a stable behaviour on missing both

L1 cache, and an increasing number of misses in L2, caused by the different instances competing for the shared resources.

			percentage of L2 misses per Id			percentage of D misses per Id		
			1	2	4	1	2	4
L2-200	L2-200x2	L2-200x4	0.31%	99.05%	99.14%	99.42%	98.95%	99.03%

Figure 30. L2 and data cache misses when running L2-miss on 1,, 2 and 4 cores.

Figure 31 shows the results obtained for this experiment. We observe that the worst delay due to sharing the memory bandwidth and L2 cache is 5x when two tasks are executed and 9.5x when 4 tasks are executed.

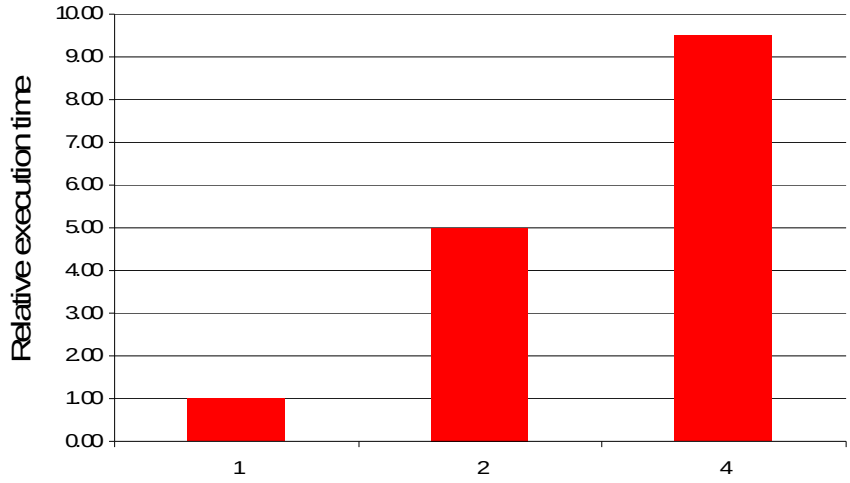


Figure 31. Experiment to stress the memory bandwidth, L2 cache, and AMBA bus

Contrary to the observed on Linux, the amount of L2 misses when $L2_{200}$ is run in isolation is almost 0. This makes the base line much better in terms of performance and thus makes the performance degradation caused by inter task interference (see *Figure 31*) much more noticeable than the observed on Linux.

5.3.1.4 Overhead of due to write-through policy

In Figure 32 we observe the execution time increment of $L2_{st}$ when in runs in different workloads with 3 copies of $L2_{40}$, $L2_{200}$ and $L2_{miss}$. We observe that the increment is quite similar to the one obtained under Linux, see Figure 16.

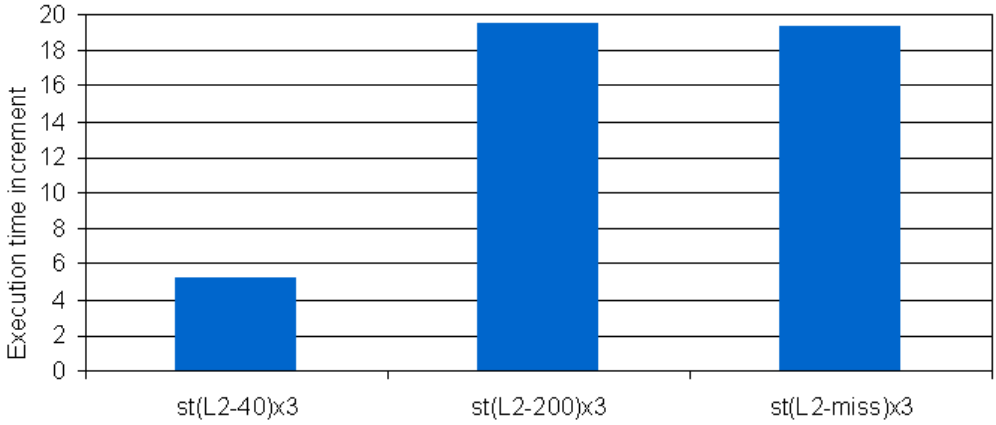


Figure 32. Increase in execution time of the store benchmark under different workloads

6 Conclusions

Several experiments have been run on the NGMP to tests its multi-tasking capabilities. Some tests have provided boundaries in the maximum impact that inter-task interference may have on the execution time on Linux and RTEMS. Other tests have bounded the effect of inter task interference in the applications used as representative of control applications. Finally, other experiments have found how the NGMP behaves when loaded with typical mixed control and payload applications, modelled by EEMBC and Parsec. Execution time results can be summarized as:

- CPU intensive tasks: little effect (at most) has been observed due to inter task interferences.
- Memory intensive tasks which do not execute store instructions: up to 4.3x slowdown in Linux and 9x in RTEMS depending on the level of inter-task interference:
  - 83% if interference is only in the AMBA AHB processor bus. (95% in RTEMS)
  - 2.6x if interference is in the AMBA AHB processor and memory buses and the memory controller (3.4x in RTEMS)
  - 4.3x if interference is in the AMBA AHB processor and memory buses, L2 cache, and memory controller (9x in RTEMS).
- Memory intensive tasks with a lot of store instructions: up to 20x slowdown, depending on the utilization of L2 and the AMBA AHB bus for both Linux and RTEMS
- CoreMark: up to 50% slowdown depending on the data footprint of CoreMark:
  - 8KB CoreMark: up to 50% slowdown.
  - 32KB CoreMark: up to 31% slowdown.
- EEMBC AutoBench. up to 5.83x slowdown depending on the instruction mix of the EEMBC:
  - *cacheb* and *canrdr*, with 15% stores: up to 5.83x and 5.67x slowdown.
  - *pntrch*, with 0.30% stores: up to 8% slowdown.
- Effect of Parsec on EEMBC: up to 16% slowdown depending on the instruction mix of the EEMBC:
  - *cacheb* and *rspeed*, with 15% and 13% stores: up to 16% and 12% slowdown.
  - *aifirf*, *pntrch*, with 7% and 0.30% stores: less than 2% slowdown.

6.1 Timing Verification of NGMP-based real-time systems: impact of our study

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 industries is of primary importance to keep the costs of such verification low.
In Integrated Architectures 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 relies on each software and hardware component exhibiting the property of time composability. Such property dictates that the timing behavior of an individual component does not change by the composition, i.e. composing the system. Time composability also alleviates System Integration. It is important to remark, that to be time composable it is also required to be time analyzable so the requirements of hard real-time systems of being both timing correct and to contain the verification costs can be ensured by providing time composability.

In the case of the NGMP we observe that the main application factor that may affect time composability are (1) the percentage of store the application has and (2) if the application has few number of stores, whether it fits in the first level data cache. Given an application with high percentage of stores, even if it fits in the first level cache, it may be the case that for the particular workload under which this application is run it does not suffer a significant execution overhead (slowdown). However, small changes in the other applications in the workload may significantly affect the execution of the application (up to 20x slowdown). Even if the other application has a footprint that fits in cache, the fact that the upgraded versions increase their number of access to cache, seriously affect the application with stores. This seriously compromise time composability.
Corollary 1: 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 run in isolation or it has to be ensured that any other application that may run on the other cores fit their data cache so they do not introduce traffic in the AHB bus.
Corollary 2: For the hardware designers a piece of advice could be to consider a write-back policy for the L1 data cache as it will significantly reduce the overhead on applications execution time due to inter-task interferences. This, of course, will introduce challenges 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. Moreover 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. 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.

With applications with few stores, but with high L2 cache footprint it is also hard to provide time composability on the NGMP. Time composability is subject to the fact that the other applications with whom the L2 cache-hungry application may potentially be run are not cache hungry as well. If these assumptions cannot be taken,

then it is not recommended running control L2-cache hunger control applications with other control or payload applications.

6.2 Future work

There are several lines that can be explored based on the results of this initial study.

- Exploring hardware support for timing isolation: In the documentation of the NGMP, some features are explained to provide isolation in the L2 cache between threads. For instance, in the L2 one or more ways can be configured to be locked, hence not to be replaced. This isolation hardware support may be interesting to explore to reduce some of the aspects of the NGMP jeopardizing timing composability: Each thread can be provided a separate partition of the L2 cache, preventing inter-task evictions in L2. Even if this solution is in place, the overhead due to the inter-task interactions are required. It should be studied the design of the AMBA AHB bus and determine how it can be adapted to reduce/bound this inter-task interactions.
- Explore the I/O path: In this study we have explore the *path* to memory from the processor: data cache, main AMBA AHB processor and memory buses, L2, memory. In addition to this main path, there are other paths that use other AMBA APB buses in the NGMP, to handle I/O traffic. Given the importance of I/O traffic for space applications, the slowdown that can be suffered in this path should be explored.
- Scheduling: In the literature there are many works dealing with scheduling and schedulability analysis for multicore processors. A common assumption is that threads receive an even part of the resources. That is, threads receive 1/N of the resources of the processors where N is the number of cores. From the results of this study it is clear that this assumption cannot be done in the NGMP. In fact, the characteristics of each thread determines the percentage of the shared resources, mainly AMBA bus and L2, they receive. Hence, an study of how to make scheduling in the NGMP taking into account inter-task interferences is required to effectively use the NGMP in safety-critical space domains.

References

PQCG+09 Marco Paolieri, Eduardo Quiñones, Francisco J. Cazorla, Guillem Bernat and Mateo Valero. "*Hardware Support for WCET Analysis of Hard Real-Time Multicore Systems*". ISCA 2009.

PQV+09 Marco Paolieri, Eduardo Quinones, Francisco J. Cazorla and Mateo Valero. *"An Analyzable Memory Controller for Hard Real-Time CMPs". In* IEEE Embedded Systems Letters, 2009. Volume 1, Issue 4

EEMBC J. Poovey. Characterization of the EEMBC Benchmark Suite. North Carolina State University, 2007.

E40C Space engineering – Software – ECSS-E-ST-40C – http://www.ecss.nl/

MERASA Multicore Execution of Hard Real-Time Applications (MERASA), European Union Framework (FP7) research programme – http://www.merasa.org

MPC55 http://www.freescale.com/files/32bit/doc/fact_sheet/MPC5510FS.pdf

MPC56 Freescale dual.core 32-bit Qorivva MPC5668G http://www.freescale.com/webapp/sps/site/taxonomy.jsp?code=MPC56XX

NGMP Next Generation Multi Purpose microprocessor – ESTEC/Contract No. 22279/09/NL/JK

GLINUX Aeroflex Gaisler, LEON Linux 2.6 Development, December 2010

GLINDRV Aeroflex Gaisler, GRLIB Linux Drivers User's Manual, November 2010

GAISLER Aeroflex Gaisler, www.gaisler.com

GRCC Aeroflex Gaisler, RCC User's Manual, October 2010

LINUX Linux kernel, www.kernel.org

GLINBLD Aeroflex Gaisler, Building the LINUX kernel for LEON, December 2010

XML510ED Xilinx, ML510 Embedded Development Embedded Platform Develop: User Guide, December 2008

XML510SC Xilinx, ML510 Schematics (rev. C)

XML510RD Xilinx, ML510 Reference Design: User Guide, June 2009

GNGMP Aeroflex Gaisler, NGMP FPGA Prototype Design for Xilinx ML510 Development Board: Next Generation Multipurpose Microprocessor, March 2010

TMS57 Texas Instrument TMS570 dual-core ARM Cortex R4F based