Experiences on the characterization of parallel applications in embedded systems with Extrae/Paraver

Adrian Munera
adrian.munera@bsc.es
Barcelona Supercomputing Center

Sara Royuela
sara.royuela@bsc.es
Barcelona Supercomputing Center

Germán Llort
german.llort@bsc.es
Barcelona Supercomputing Center

Estanislao Mercadal
estanislau.mercadal@bsc.es
Barcelona Supercomputing Center

Franck Wartel
franck.wartel@airbus.com
Airbus Defence and Space

Eduardo Quiñones
eduardo.quinones@bsc.es
Barcelona Supercomputing Center

ABSTRACT

Cutting-edge functionalities in embedded systems require the use of parallel architectures to meet their performance requirements. This imposes the introduction of a new layer in the software stacks of embedded systems: the parallel programming model. Unfortunately, the tools used to analyze embedded systems fall short to characterize the performance of parallel applications at a parallel programming model level, and correlate this with information about non-functional requirements such as real-time, energy, memory usage, etc. HPC tools, like Extrae, are designed with that level of abstraction in mind, but their main focus is on performance evaluation. Overall, providing insightful information about the performance of parallel embedded applications at the parallel programming model level, and relate it to the non-functional requirements, is of paramount importance to fully exploit the performance capabilities of parallel embedded architectures.

This paper contributes to the state-of-the-art of analysis tools for embedded systems by: (1) analyzing the particular constraints of embedded systems compared to HPC systems (e.g., static setting, restricted memory, limited drivers) to support HPC analysis tools; (2) porting Extrae, a powerful tracing tool from the HPC domain, to the GR740 platform, a SoC used in the space domain; and (3) augmenting Extrae with new features needed to correlate the parallel execution with the following non-functional requirements: energy, temperature and memory usage. Finally, the paper presents the usefulness of Extrae to characterize OpenMP applications and its non-functional requirements, evaluating different aspects of the applications running in the GR740.

CCS CONCEPTS

• **Software and its engineering** → **Parallel programming languages**; **Empirical software validation**; *Software performance*; • **Computer systems organization** → **Embedded systems**; • **Hardware** → *Power estimation and optimization*; *Temperature monitoring*.

KEYWORDS

Embedded systems, parallel programming models, performance evaluation, analysis tools, OpenMP

ACM Reference Format:
Adrian Munera, Sara Royuela, Germán Llort, Estanislao Mercadal, Franck Wartel, and Eduardo Quiñones. 2020. Experiences on the characterization of parallel applications in embedded systems with Extrae/Paraver. In *49th International Conference on Parallel Processing - ICPP (ICPP '20), August 17–20, 2020, Edmonton, AB, Canada.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3404397.3404440

1 INTRODUCTION

The computing capabilities of current parallel embedded processor architectures (e.g., GR740 [14], Xilinx UltraScale+ [21], MPPA Coolidge [3], NVIDIA Jetson [22]) provide embedded systems with the level of performance needed to implement the most advanced functionalities, e.g., autonomous driving. This promotes the evolution of embedded software stacks from simple micro-controllers, where sequential execution is predominant, to complex parallel frameworks, where parallel programming models are key to fully exploit the performance capabilities of parallel architectures. In this context, fine-grained models such as OpenMP or Pthreads [27, 35, 41] are already being considered as appropriate solutions to leverage the potential of the newest embedded systems. Particularly, OpenMP has been shown to provide time predictability [30, 31] and correctness [26] capabilities at parallel programming model level.

Parallel programming models introduce challenges in current embedded software stacks, specifically in the tools used for guaranteeing the correct operation of the system. In this sense, parallelism affects the *functional* behavior of the system, regarding its operation in response to the inputs, and the *non-functional* behavior, referring to the operation of the system within the time, energy, memory, etc., budgets imposed by the environment. As a result, there is a need for correlating the behavior of parallel embedded applications with the impact parallelism has on their functional and non-functional requirements. This paper focuses on the correlation of parallel performance and non-functional requirements at the OpenMP parallel programming model level.

High-performance computing (HPC) systems already require the exploitation of parallel programming models in order to extract the best performance from highly parallel architectures. For this purpose, analysis frameworks targeting these systems, like Extrae [37] and Score-P [18], offer an accurate description of the parallel behavior, and allow relating this information with the parallel programming model. However, these tools mainly focus on performance analysis, and do not take into account non-functional requirements, such as energy, temperature and memory usage, which are critical for the correct operation of embedded systems. There is hence a need to complement the information provided by HPC tools with the information relevant in embedded systems.

Applying HPC to embedded systems raises further issues, mostly related with the tighter constraints of these systems. These are: (1) environments are often static, meaning that software is developed in ROM, evolve very little, and may last the lifetime of the electronic product (consider for instance, the systems controlling satellites in the space domain); and (2) the amount of software available for embedded systems is restricted, e.g., drivers offer reduced information of the system, compilers might not support rare architectures and libraries might not support rare operating systems. Furthermore, HPC analysis tools do not support the analysis of non-functional requirements, as their main objective is performance.

This paper contributes in the field of analysis tools for embedded systems in different aspects: (1) it analyzes the most common aspects to be considered when porting HPC analysis tools to embedded systems; (2) it illustrates the adaptation by porting Extrae, a well-known tracing tool from the HPC domain, to the GR740 board, a state-of-the-art system-on-chip (SoC) used in the space domain; (3) it extends the introspection capabilities of Extrae introducing new features to trace non-functional requirements, including temperature, energy and application memory usage, and allows correlating this information with the execution of the application at the parallel programming model level; and (4) it uses Extrae and Paraver [38], a wide-spread visualization tool also from the HPC domain, to analyze the behavior of two different applications parallelized with OpenMP in the GR740. This experience can be of great help for those adapting tools from HPC to embedded systems.

Overall, the extended capabilities of Extrae allow describing the performance behavior of OpenMP embedded applications at a parallel programming model level, and relating this information with several non-functional requirements. This contribution is paramount to exploit the benefits of parallelism in embedded systems and so facilitate the development of advanced functionalities.

2 RELATED WORK

Embedded applications usually must satisfy precise constraints to match the specifications of the system, including *functional*, i.e., operate correctly in response to the inputs, or fail in a predictable manner, and *non-functional*, i.e., satisfy the timing, energy, temperature, etc. requirements due to cyber-physical interactions of the system. Consequently, visualization technologies for non-functional requirements are wide-spread in embedded systems. ULINKplus Debug Adapter [17] is a debug and trace adapter that allows tracing events and timing information. Together with the μVision® IDE [16], it allows for visualizing power consumption, exceptions, variable changes, and operating system (OS) events (i.e., cycles per instruction, exceptions overhead, load/store cycles, and folded instructions). Similarly, J-Trace Debug Probe [28], a debug probe that, together with SystemView analyzer [29], a real-time recording and visualization tool for embedded systems, offers different views with OS information, such as task execution, task switches, and interruptions, based on a timeline. Although powerful, these options require specific hardware and are limited to Arm Cortex-based devices.

Hardware-independent solutions have been around for years, e.g., LTTng [4], an open source technology for software-based tracing in Linux, that can record several kernel-level activities, including scheduling events, system calls, interrupt requests (IRQs), memory management, and other kernel-level activities. There are also tools specific in the embedded market, such as RapiTime and RapiTask [34], RTOS scheduling analysis and visualization tools to help understanding the timing behavior of applications (e.g., worst-case execution time, execution time, response time), and spotting rare timing events such as race conditions, priority inversions or deadlocks. Finally, Tracealyzer [24], is a trace visualization tool for RTOS-based and Linux-based embedded systems that offers information about the run-time behavior, including task scheduling, timing and priorities, CPU load, and memory usage, among others.

The previous tools can be very useful to understand the behavior of a system at an OS level, considering interactions among applications and kernel activity. But the low-level system measurements they provide are difficult to correlate, especially when parallel computing is involved. Overall, the tools lose sight of the global structure of the application where the interaction with the parallel runtime affects aspects like load-balance and synchronizations, which have a critical impact on performance.

3 THE IMPORTANCE OF CHARACTERIZING THE PARALLEL PROGRAMMING MODEL

On-chip parallelism found on modern embedded multi-core and many-core processor architectures delivers high performance and reduced energy consumption at a lower cost, essential for systems with constrained execution environments. Efficiently leveraging the performance of parallel architectures requires to parallelize applications. OpenMP [23] is a high-level directive-based parallel programming model considered the state-of-the-art solution for parallel tasking, thanks to its great expressiveness and evinced performance. It is actively being considered as a possible solution for exploiting fine-grained parallelism in embedded systems [1, 13, 35], by virtue of its delimited functional safety [26] and time predictability [30, 31]. Yet the OpenMP framework does not include a performance monitoring tool for characterizing the parallel execution[1], and the existing analysis tools for embedded systems lack the capability to express detailed per-thread performance information at the parallel programming model level. For this reason, this section (1) introduces the important features of the OpenMP model to be considered in the analysis tool; and (2) describes the existing techniques used in HPC analysis tools, motivating the use of Extrae/Paraver.

3.1 OpenMP: Characterization Aspects

OpenMP defines a relaxed fork-join execution model in which parallelism is spawned with the `parallel` directive, and distributed based on two different approaches: (1) the *threading* model, which defines an abstraction of user-level threads to work as proxies for

[1]OpenMP v.5.0 introduces OMPT (Chapter 4 of the specification [23]), an application program interface (API) for helping performance analysis of first-party tools.

the physical processors, and allows defining a rather structured parallelism with worksharing constructs such as `for`; and (2) the *tasking* model, which is oblivious of the physical layout focusing on exposing parallelism rather than mapping parallelism onto threads, and facilitates the definition of unstructured and highly dynamic parallelism with constructs such as `task` and `taskloop`.

Regarding synchronization, the thread-based model only allows for full synchronizations with the `barrier` construct. The tasking model allows, additionally, different forms of partial and fine-grain synchronization, including the `taskwait` construct, which forces waiting only for those tasks that are children of the current task, and the `depend` clause that, attached to a task, allows ordering the execution of tasks based on a data-flow model.

Figure 1 depicts the execution model of OpenMP with the thread model (left) and the task model (right). The picture shows the structured and unstructured nature of each model, respectively. While the former distributes work among threads (more or less evenly) based on the workload of each chunk and the scheduling configuration, the latter distributes work based on the amount of parallelism exposed in the application. In this case, the parallelism can be represented in the form of a Task Dependency Graph (TDG), where nodes are tasks and edges are dependencies between them.

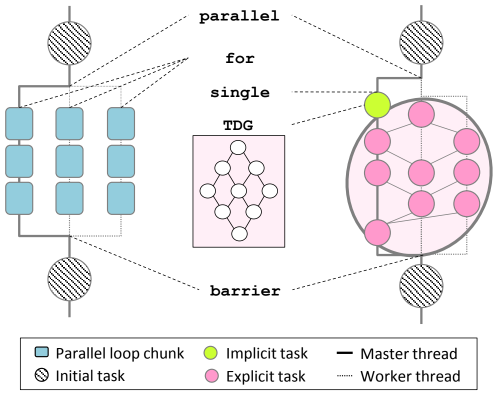


Figure 1: OpenMP execution model using (left) threads and (right) tasks.

Characterization-wise, each model poses different interesting challenges. In the thread model, it is important to be able to recognize how parallelism is spawned and joined to evaluate the overhead of synchronizations, and also understand how parallel loops are distributed among the existing resources to evaluate the load balance of the application. The tasking model, instead, introduces the possibility of asynchronous execution: tasks are instantiated when they are encountered (i.e., task input data is captured at that time) and can be executed immediately or deferred, depending on factors such as the scheduling policy, the available resources, and the readiness of the task (i.e., the task depends on previous tasks that have not finished yet). In both cases, understanding the amount of parallelism exposed by the application taking into account the different features of the two models at the parallel programming level is fundamental to efficiently exploit the thread-based and task-based parallelism. These and more features are considered in the evaluation presented in Section 5.

3.2 A Glance at Analysis Techniques

The mechanisms used in analysis tools for gathering information can be classified as follows:

- Based on the way data is gathered:
  - *Basic measurements*, like clock cycles or elapsed wall-clock time between two points, are easy to obtain, but come without information concerning the factors that explain them.
  - *Sampling* mechanisms, based on probing the program counter to identify the most time-consuming parts, can provide a better understanding of the structure of the application. However, the ability to characterize fine-grained tasks and outliers relies on a delicate trade-off between the sampling frequency and the overhead.
  - *Instrumentation* mechanisms, based on capturing the relevant activity of the application, provide a more precise picture of the actual program behavior.
- Based on how data is stored:
  - *Profiling* mechanisms can picture general information about the execution, but fail to connect that information to specific points in time because data is summarized in counters.
  - *Tracing* mechanisms store events in a timeline basis, hence are the most reliable to get an exact picture of the behavior of the program.

The level of information required at analysis time narrows down the mechanisms to be used. Particularly, our work aims at allowing a detailed description of the complete parallel execution. For this reason, the use of tracing mechanisms for storing data is fundamental. In this context, there are two main frameworks available for HPC systems: Score-P [18] and Extrae [37]. The former, Score-P, is a performance measurement infrastructure for profiling, event tracing, and online analysis of parallel applications, with a strong focus on user code instrumentation by the insertion of calls to the measurement system into the application, either fully automatically at compile-time, or with a certain amount of control handed to the software developer. Score-P makes use of a common plugin interface [25] to standardize the gathering of metrics, so additional metrics can be easily added. The traces generated with Score-P are used by a number of analysis tools, including Scalasca [9], Vampir [44], and TAU [33]. The latter, Extrae, is a dynamic instrumentation package for parallel programs that focuses on measuring the parallel activity of the runtime rather than on the syntactic structure of the application. This empowers analysis to work with codes without the need for previous knowledge or experience with them. Furthermore, Extrae for HPC neither requires access to the source code, nor recompiling, nor relinking, facilitating the analysis of already existing binaries. These characteristics as well as its support for OpenMP, among many other programming models, are the key aspects for which we select the Extrae measurement system for our studies. Next subsection describes important details about this tool.

3.3 Analysis Tools: Extrae/Paraver

Extrae [37] is an open-source tracing framework of the Barcelona Supercomputing Center (BSC) tool-suite that generates Paraver [38] trace-files. This package automatically instruments applications using the `LD_PRELOAD` mechanism to capture information from parallel programming models such as OpenMP, Pthreads, OpenCL, CUDA, MPI, OmpSs, and combinations of them. This information includes the activity of the parallel runtime (e.g. parallel loops in OpenMP), performance counters through PAPI [15], and source code references through *libunwind* and *GNU binutils* to respectively walk the call stack and fetch human-readable debugging information from the binary. The same mechanism is used to configure Extrae to instrument specific vendor implementations of a given language (i.e., APIs). Extrae also offers sampling mechanisms to get performance details for long non-instrumented regions.

The result of an instrumented run with Extrae is a Paraver trace [39], which consists of a sequence of time-stamped entries that record the actual execution of the application. There are three types of records: (1) `events`, representing punctual actions of the program (e.g., entry and exit of routines), (2) `states`, displaying activity for a period of time (e.g., scheduling or transferring data), and (3) `communications`, reflecting interactions between two processes/threads (e.g., task data dependencies). Paraver is the visualization tool of the BSC tool-suite that enables visual and numerical inspection of the trace through timelines, histograms and profiles, to get insight of the potential bottlenecks (examples of the views offered in Paraver are provided in Section 5). Since the format of the Extrae traces has not changed, the Paraver tool has not been modified. Hence, the tool is only used for evaluation purposes, to plot the traces generated with our modified version of Extrae.

The benefits of using Extrae and Paraver to evaluate the performance of parallel applications and architectures has been thoroughly proved in the context of HPC systems [19, 32, 43]. However, there exist a number of challenges to benefit from the virtues of Extrae in embedded systems: (1) adapt Extrae to work in constrained embedded systems, and (2) augment Extrae with the features needed to evaluate the non-functional characteristics of the system, so these can be related with the parallel programming model and the performance. This work contributes in these two aspects, as follows: (1) we have ported Extrae to the GR740 platform, a SoC used in the space domain, addressing the main factors to be tackled when porting an analysis tool to a restricted environment composed of a SPARC V8 architecture and a RTEMS operating system, and (2) we have extended Extrae as to record information about non-functional requirements including temperature, energy and memory consumption. The remainder of this paper introduces these contributions.

4 FROM HPC TO EMBEDDED SYSTEMS: ACCOMMODATING EXTRAE TO GR740

HPC systems are usually designed with symmetric and scalable computing nodes, where the software stack is supported by most compilers and chip vendors, and the system is frequently configured dynamically. Differently, embedded systems are much more heterogeneous, and the variety of boards and drivers makes the porting of tools difficult. Furthermore, they tend to be statically configured, so the applications are built together with the OS, as in the case of the GR740 running RTEMS. This makes difficult to use HPC tools in embedded systems. The next subsections describe (1) the GR740 platform, (2) different aspects relevant to the porting of Extrae to the GR740, and (3) new features implemented in Extrae[2].

[2]The modified version of Extrae is publicly available at https://github.com/bsc-performance-tools/extrae/tree/gr740.

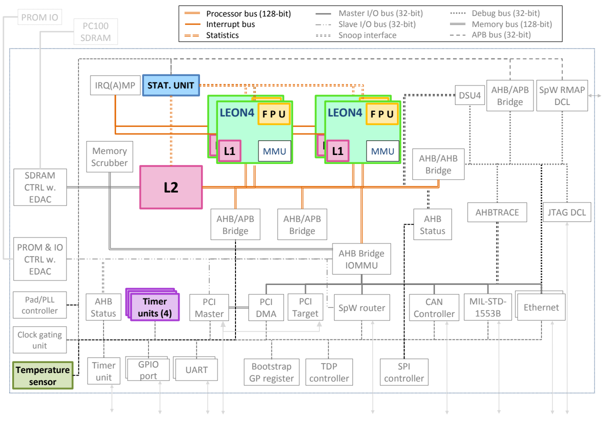


Figure 2: GR740 SoC micro-architecture.

4.1 The GR740 System-on-Chip

The GR740 is a radiation-hard system-on-chip designed as the European Space Agency's (ESA) Next Generation Microprocessor and developed by Cobham Gaisler. This SoC is targeted at high-performance general purpose processing, and is suitable for both symmetric and asymmetric multiprocessing.

Architecture. The GR740 device features a quad-core fault-tolerant LEON4 SPARC V8 processor with a frequency of 250MHz, each equipped with a double-precision IEEE-754 floating point unit and a 16 KB instruction and data caches. The SoC offers a 2MB write-back L2 cache with memory access protection (fence registers), and eight port SpaceWire router, 2x 10/100/1000 Mb Ethernet interfaces, and a 33MHz PCI initiator/target interface. Different dedicated AMBA[3] High-performance (AHB) buses for memory, processors, debug, and IO. Finally, some modules of interest are: (1) the LEON4 Statistics Unit, *L4stat*, used to count events in the LEON4 processor (the architecture offers up to 16 registers to store hardware counters, and the available counters are listed in the documentation of the board [2]), (2) the AHB bus, and (3) the Temperature Sensor Controller, a peripheral that provides an interface to the on-chip temperature sensor. Figure 2 shows an overview of the micro-architecture of the GR740. The colored parts are those that will be later involved in the evaluation, and include: the 4 cores, the FPUs, the memory caches L1 and L2, the four timer units, the statistics unit, the temperature sensor, and the AMBA High-performance Bus (AHB).

Operating System and Tools. The GR740 runs, among others, a Real-Time Executive for Multiprocessor Systems (RTEMS) Real-Time Operating System (RTOS). Cobham Gaisler offers the RTEMS LEON GNU Cross Compilation System (RCC) [8], a multi-platform development system including a RTEMS-5.0 C/C++ real-time kernel v5.1 with pre-compiled Board Support Packages (BSP) for Symmetric Multiprocessing (SMP) support. The fact that this OS was designed for uniprocessors, makes its adaptation to SMPs a challenge. The ESA has dedicated important efforts to the improvement and actual qualification of RTEMS SMP [36]. There are however some modules that still do not support SMP in the GR740, like the

[3]The *Advanced Microcontroller Bus Architecture* (AMBA) is an open standard for the connection and management of functional blocks in a system-on-chip.

statistics unit, the GPIO, and the SDRAM scrubber. This prevents the use of these modules when running multi-threaded applications, and instrumented applications will not be able to extract any information from them. Additionally, the GR740 incorporates the *L4stat* driver, which is used to count events in the LEON4 processor and the AHB bus. For the purpose of this work, we have used the GCC compilation toolchain, and the GR740 BSP with SMP support.

4.2 Adapting Extrae: Challenges and Solutions

This section introduces the relevant challenges addressed during the porting of Extrae to the GR740. This experience can be of great help for those adapting tools from HPC to embedded systems.

4.2.1 Intercepting calls in a static system.

Vanilla Extrae is loaded before the system libraries required by the application under analysis, including those responsible of the parallel execution, using a mechanism provided by the dynamic linker on Unix systems through the `LD_PRELOAD` environment variable. This mechanism allows intercepting calls to the runtime system (e.g., GNU's OpenMP runtime library) by defining the symbols to be intercepted. Hence, when a function call is intercepted by Extrae, it first takes performance measurements, and then bypasses the call to the real implementation of the symbol found with the *dlsym* method from *libdl*.

This mechanism is not available on statically linked binaries as it is the case of the GR740, in which the final executable loaded in the board includes the RTEMS OS and the application statically linked. As a result, we can no longer rely on the dynamic wrapping of runtime functions of Extrae. To recreate this behavior in a static environment, we propose two different solutions:

(1) The use of the *--wrap* linker flag, which allows defining a wrapper function for a specific symbol. The principle is that any undefined reference to *symbol* will be resolved to *__wrap_symbol*, and any undefined reference to *__real_symbol* will be resolved to *symbol*. This mechanism allows users to define the list of functions that are to be instrumented, boosting precision and so possibly reducing overheads. However, it can become a burden when the list is too long (e.g., instrumenting the libgomp features, which have tens of calls including runtime routines and API functions).
(2) The implementation of the wrapping logic in the compiler[4], so it can transform the calls to the runtime library into calls to Extrae. This mechanism is much more transparent to the user, and could still be tuned with input from the users if a flag, similar to *--wrap*, was added to the compiler.

In both cases, the Extrae API has to be modified: in the former option, implementing the wrapped functions with the name *__wrap_function* instead of *function*; in the latter option, implementing the wrapped functions with the names agreed between the compiler and Extrae. To solve this issue we are currently using the GNU linker feature for wrapping, accessed via the *--wrap* flag, because it does not require complex modifications, while providing the full functionality.

[4]Compiler automatic instrumentation through the *-finstrument-functions* is not considered because it would require deeper changes in Extrae, as well as extra work on the programmer to define the symbols that should not be instrumented, so the traces have a reasonable size.

4.2.2 POSIX dependence.

The Extrae original sampling mechanism has a tight dependency on the POSIX API, particularly of: (1) the *setitimer* function to define an interval timer to each process, i.e., a timer which generates a signal each time a specified time interval expires; and (2) the *ucontext* structure, to gather thread context information such as the program counter, as well as information about signals. These features are typically defined in *libc*, the standard library for C programming. There are several implementations of this standard, targeting different operating systems, e.g., *glibc* [11] for Linux, and *newlib* [42] for embedded systems, among others. Although the libraries common in HPC systems (i.e., *glibc*) include the features mentioned above, not all implementations satisfy this dependency. This is the case of RTEMS and the *newlib* version for the GR740.

To overcome the lack of the mentioned features, we propose the next solutions:

(1) Regarding the *setitimer* functionality, this could be implemented following two different approaches: (1) use the hardware timers available in the GR740 (purple boxes in Figure 2), through the RTEMS API (e.g., *rtems_timer_create*, *rtems_timer_fire_after*), and (2) use software interruptions with signal alarms defined in the POSIX API and available in *newlib* (e.g *alarm*, *ualarm*). The GR740 system handles hardware interrupts by a pseudorandom CPU, so there is no guarantee all cores can be interrupted at the same time using the four timers available in the SoC. On the other hand, software alarms do not queue in RTEMS, hence only one software alarm can be active per-process at the same time (newer calls to an alarm from threads of the same process will reschedule the alarm). Due to this behavior, the interrupted core must be in charge of reading the state of the other cores without stopping them. Since hardware interruptions tend to be faster, we decided to use the hardware timers for this purpose.
(2) Regarding the *ucontext* functionality, particularly to retrieve the information about the user functions, we propose two solutions: (1) implement the missing features in the corresponding *newlib* library (these could be ported from existing implementations such as FreeBSD [20]); and (2) use the information from the RTEMS *Thread_Control* structure associated to the thread executing the instrumented region. Seeking a trade-off between complexity and functionality, we have chosen the second option. This mechanism is however limited when the instrumentation occurs through interruptions (i.e., when sampling is enabled). This is so because only information about the user-function being executed on the thread handling the interruption can be restored. Nonetheless, even in this case, meaningful information about hardware counters can still be gathered for all threads.

4.2.3 Retrieving function names.

Extrae uses the GNU's Binary Utilities library collection [10], *binutils*, particularly the Binary File Descriptor library, *bfd*, and *libiberty* [12], to translate addresses into source code function names. This functionality is fundamental for the readability of the traces, hence, these libraries have to be ported to the target device. Luckily, Cobham Gaisler provides both the libraries and a cross-compiler (see Section 4.1 for details) that allows for generating the target version straightforwardly.

4.2.4 *Trace generation.*

Extrae stores events during the execution of the application, and flushes the full trace at once at the end of the program, avoiding interfering with the application as much as possible. The size of the traces is proportional to the number of times an instrumented event is executed, and the time the application is running. As a consequence, trace files might be huge. This entails several issues for the GR740 ecosystem:

(1) *File size.* The RTEMS OS defines in its headers the variable `IMFS_MEMFILE_DEFAULT_BYTES_PER_BLOCK`, to allow defining the maximum file size allowed in the system. This value is originally determined in a preventive manner, set to a small size in order to avoid possible memory fragmentation issues (a typical problem of file systems). The OS can be recompiled changing this value to tune it as desired.
(2) *Buffer size.* Currently, RTEMS only supports the NFS[5] (Network File System) version 2. This version limits the maximum size of an on-the-wire NFS read or write operation to 8KB (8192 bytes). To overcome this limitation, Extrae has been modified to write to disk the trace in smaller pieces so it can be safely written through NFS v2. Particularly, it reduces the maximum size of each *write* function call, so in case the maximum size is exceeded the data is divided in smaller pieces.
(3) *Heap size.* Although the heap memory is bounded to the total RAM of the device, it was artificially reduced by RTEMS in the GR740. This is so because the OS allocated an extra workspace area to store objects created by RTEMS itself, such as tasks, semaphores, and message queues. To overcome this limitation, we have configured the workspace area to be shared with the heap, adding the environment variable provided by RTEMS called `CONFIGURE_UNIFIED_WORK_AREAS`.

4.2.5 *Supporting hardware counters.*

The support for hardware counters depends on the target architecture (i.e., how many registers are built for storing hardware counters, if any) and the driver that implements the access and management of the counters. Extrae relies on PAPI, a portable interface in the form of a library available in most modern microprocessors, to gather hardware performance counters information. In this context, the best solution for porting the support for hardware counters would be implementing a new component in PAPI, which will use the target hardware counter driver (so Extrae remains unperturbed). However, PAPI is neither supported for RTEMS nor for SPARC V8.

Although Extrae only supports PAPI, it has been conveniently designed to easily add new software modules with the logic to gather hardware counters other than PAPI. To that end, RTEMS provides the *L4stat* driver, which allows to configure and access the different hardware counters of the *L4stat* hardware module[6], and assign each one of them to an event and a core. The module provides up to 16 configurable hardware counter slots that can be assigned to any core. In this scenario, we implemented hardware counters support for the GR740 adding an extra module on Extrae that uses the API provided by the *L4stat* driver instead of the PAPI API. The usage of the hardware counters is defined by the environment variable `EXTRAE_COUNTERS`, where the user can specify the id of the hardware events to be gathered.

4.2.6 *Statically defining the Extrae environment.*

Extrae is originally configured via the XML file defined in the `EXTRAE_CONFIG_FILE` environment variable. Migrating this mechanism to our system is costly because of three reasons: (1) it requires cross-compiling *libxml* for the GR740, (2) it adds a new dependency to Extrae, and (3) it forces reading an extra file through the NFS system at initialization phase. For these reasons, we have used some already defined environment variables to set up the configuration of Extrae from the application, and we have added new ones, i.e., `EXTRAE_SAMPLING_PERIOD`, for defining the period of the sampling mechanism, and `EXTRAE_PROGRAM_NAME`, for defining the prefix of the resulting intermediate trace files (it is essential that the latter matches the binary name, so binutils can find the binary copy and translate the program addresses properly).

4.3 New in Extrae: Relating Performance and Non-functional Information

Additionally to the porting, we have included three new functionalities in Extrae to correlate information about the performance of the parallel execution with the temperature, power consumption and the memory usage. Moreover, we have enhanced Extrae with a new feature for tracing communications in order to better visualize the relationship between the amount of parallelism exposed and exploited when the task-based programming model is used.

The GR740 temperature sensor. As shown in Figure 2, the GR740 SoC includes a sensor that provides information of the on-chip temperature. However, the board's software package does not include a specific driver to interact with this module. Instead, it implements a memory mapped peripheral that, according to the documentation [6], uses three registers: (1) a *control register*, used to initialize the peripheral, setting the desired frequency, enabling the alarm and enabling the clock, (2) a *status register*, used to store the current temperature, together with maximum and minimum read values, and a field that indicates when a new value is available, and (3) a *threshold register*, which holds the maximum temperature value that triggers the alarm bit in the *control register*. We have extended the Extrae hardware counters module to be able to manage this chip with a new event by defining the *initialize* and *read* functions for the specific chip.

Power consumption. The GR740 does not include any hardware performance counter to gather power consumption information. Instead, Cobham Gaisler offers a spreadsheet [7] for computing the core power consumption based on the configuration of the SoC, e.g. clock rate, and information about the execution of the application, e.g., CPUs usage. We have modified Extrae to include in the execution trace all the information needed to compute the power consumption based on the information given in the mentioned spreadsheet and the efficiency of the parallel execution.

[5]The NFS system is needed to permanently store the traces because RAM is erased after the execution of the application.

[6]The *L4stat* module of the GR740 is clockgated by default to save energy consumption. To set it up, (1) the module has to be enabled manually using the SoC debug tool GRMON, and the command line interface *grcg*, which configures the clockgating module, and (2) the applications must define the variable `CONFIGURE_DRIVER_AMBAPP_GAISLER_L4STAT` in order for the OS to initialize the driver adequately.

Memory consumption. Memory consumption is a critical aspect in embedded systems due to the typically restricted constraints they have. Particularly, stack and heap analysis are fundamental to ensure the system's stability and reliability. In this sense, we have implemented new support in Extrae to analyze the use of the stack and heap memories in the GR740. This implementation relies on RTEMS, and uses information extracted from the *Thread_Start_information* structure contained in the *Thread_Control* associated to each executing thread to retrieve the state of the stack, and information from the *Heap_Control* structure, particularly the *area_begin* and *area_end* fields, to retrieve the state of the heap. This new features allow two evaluations: (1) relating the application and the parallel programming model to the use of memory, and (2) bounding the amount of memory used by the application, both partially and globally.

Task communication. The workloads of embedded applications are smaller to those considered in HPC. As a result, the exploitation of fine-grain parallelism is fundamental in embedded systems, especially when exploiting irregular parallelism with the OpenMP task-based model. With the objective of better understanding if the amount of parallelism exposed by the application is efficiently exploited by the underlying architecture or, on the contrary, there are not sufficient resources to exploit it, we use an experimental API of Extrae that allows for combining records. It is based on a structure, called `extrae_combined_events_t`, that holds a number of events and communications occurring at the same point in time. In the OpenMP tasking model, communications can be task creations, tasks dependencies, and task synchronizations, among others. We focus on task dependencies, which define a data-flow model among tasks that can be characterized as a Task Dependency Graph (TDG), generated at run-time (regular implementations), or at compile-time [40]. The TDG allows analyzing an application in terms of the amount of parallelism exposed, defined by the width of the graph. The information about the time a task has to wait although its dependencies are already solved, provides information about the amount of parallelism exploited, so the longer the time the less parallelism is being exploited.

5 EVALUATION

This section shows the potential of the Extrae/Paraver tool-suite, including the proposed modifications presented in Section 4.3. This analysis has been performed in two applications parallelized with OpenMP and executed on the GR740. It is important to remark that the purpose of this section is not to perform a detailed analysis of a particular application, but rather showing in an educative way, how Extrae is useful to correlate the performance of the application at the OpenMP parallel programming model level with information about non-functional requirements, particularly temperature, power consumption and memory usage.

5.1 Experimental Setup

Evaluations have been conducted on the GR740 [14] (see details in Section 4.1), and Extrae has been ported to the board as explained in Section 4.2. Furthermore, we have selected the following representative applications[7]:

Image Processing, Proc this application from Airbus Defence and Space (ADS) receives a stream of images purportedly from an optical instrument. The application performs infinite iterations of a loop that first reads a new image, and then applies a series of pipelined phases over it. Each phase might use information computed from previous phases, and even from previous iterations (previous images). In the parallel version, four phases (the ones consuming >90% of the total execution time) have been parallelized using the OpenMP thread model, concretely using parallel loops.

Matrix Factorization, LU This benchmark implements an LU decomposition, typically used for numerical solution of linear equations. The application decomposes a four-dimensional sparse matrix in four steps: *lu0*, *fwd*, *bdiv*, and *bmod*. We use two different parallel versions of the benchmark: one using parallel loops, *LU-for*, and other using tasks and dependencies for synchronization *LU-tasks*.

5.2 Performance Analysis with Extrae

This section uses Extrae and Paraver to analyze the performance and the non-functional requirements, and relates this data with the OpenMP parallel programming model in the GR740.

5.2.1 *Memory consumption.*

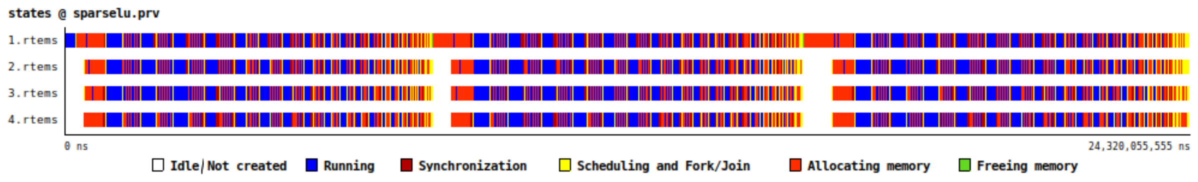
To evaluate the new feature introduced in Extrae to analyze the memory consumption, including information about the stack and the heap, we consider the *LU* application parallelized with the worksharing-loop construct.

Figure 3 relates the state of the application (3a), including the fork/join and synchronization operations of the worksharing-loop construct, and the call to `malloc` and `free`, with different views of the memory, including stack (3b), amount of dynamically allocated memory using malloc-like calls (3c), and the heap (3d). The *stack* view reveals that the master thread, running in core 1, uses more stack. This is because this thread holds the structures that are common to all threads in the application. Although the variations among the other threads and parts of the code are not significant, this view allows to upper bound the use of stack in each part of the application, which ranges between 1000 and 2000 bytes. The *dynamic allocation* view (already provided by Paraver), allows to know the amount of memory allocated and deallocated in each moment. Furthermore, the newly introduced *heap* view, allows for upper bounding the heap use and, together with the previous view, characterizes the usage of the heap. This information is of paramount importance due to the stringent memory constraints of embedded systems, specially when dynamic memory is allowed.

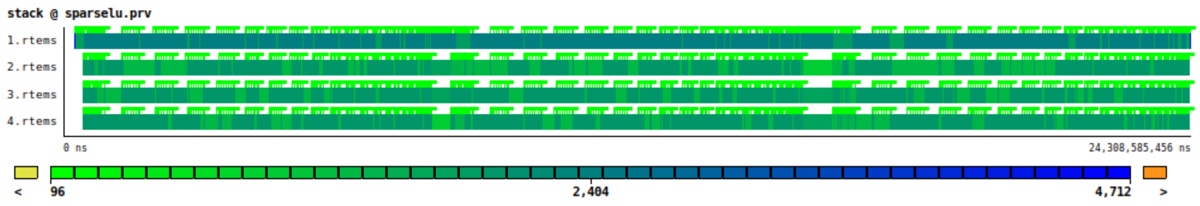
5.2.2 *Temperature and power consumption.*

This section focuses on the evaluation of the new features introduced in Extrae to analyze: (1) the temperature (using the GR740 temperature sensor) and (2) the power consumption (using the CG spreadsheet) of the SoC. For

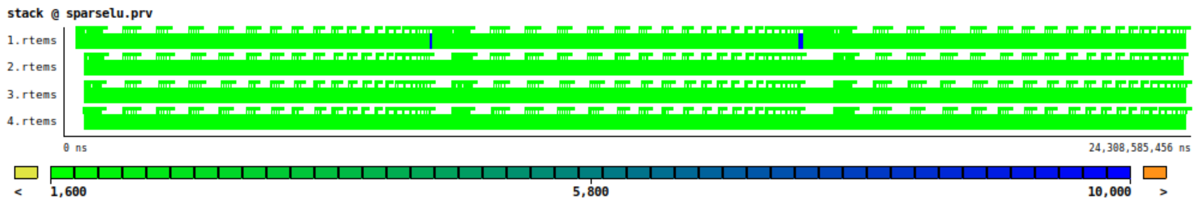
[7]The sources of the *Proc* application are not distributed due to confidentiality issues. The sources of the *LU* benchmark are very similar to those provided in the Barcelona OpenMP Task Suite (BOTS) [5].



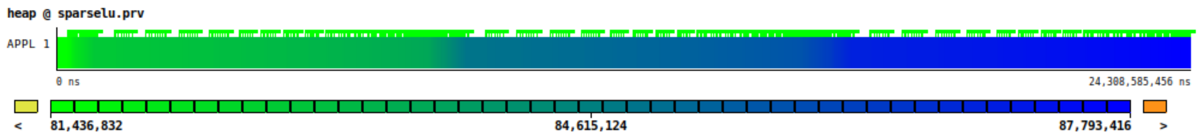
(a) Paraver *parallel functions* view.



(b) Paraver *stack* view.



(c) Paraver *dynamic (de)allocation* view.



(d) Paraver *heap* view.

Figure 3: Analysis of the memory usage for the *LU-for* benchmark.

this purpose, we consider the *LU* application parallelized with the worksharing-loop construct.

Figure 4 shows two views of Paraver, extracted from the same execution of the LU benchmark, containing two types of Extrae events: (top) the *work-sharing view*, showing when threads are executing a worksharing-loop construct, and (bottom) information collected from the *temperature sensor* (as explained in Section 4.3). In the top trace, red color indicates the core is working and white color indicates the core is idle. The bottom trace plots the values as a function line, and the y-axis shows the values of the function. As the combined traces reveal, the temperature ranges between 53 and 54 degrees Celsius when parallel execution of the four cores stabilize. Furthermore, as spotted with circles, when the parallel execution cannot exploit all four cores, this has a downward impact in the temperature. In some cases, however, as shown in the shadowed circles, the temperature is not reduced. This is because the temperature is stabilizing, and even though for a period of time some cores are idle, the overall context of the execution forces the temperature up.

Figure 5 shows two different Paraver views of the same Extrae trace used previously for analyzing the temperature: (top) the *useful computation view*, showing in blue the periods of time a core is executing (i.e., actual time the CPU is working, not considering, for example, the time a thread is blocked waiting for synchronization), and (bottom) the *power consumption view*, showing the values of power consumption in mW for each period of the trace (as explained in Section 4.3). As emphasized in the trace, the power consumption is directly related with the number of cores running at the same time. The view clearly shows that most of the time, the consumed energy is 1045,28mW (top line). To get an average value of the power consumption, we use the *efficiency histogram* offered with Paraver, which allows extracting the profile shown in Table 1. This table summarizes the percentage of time each core is idle/running, together with the total percentage and its average. This information, used in the Cobham Gaisler spreadsheet, gives an average power consumption of 944.5mW, as anticipated in the trace.

5.2.3 Task communications. To evaluate the amount of parallelism exposed in the TDG with respect to the actual parallelism exploited, we consider the *LU* application parallelized with the tasking model. Figure 6 represents the TDG of the application. The width and the height of the TDG are defined by the dependencies between tasks: the height is determined by the critical path, and the width is determined by the amount of parallelism exposed. Applications

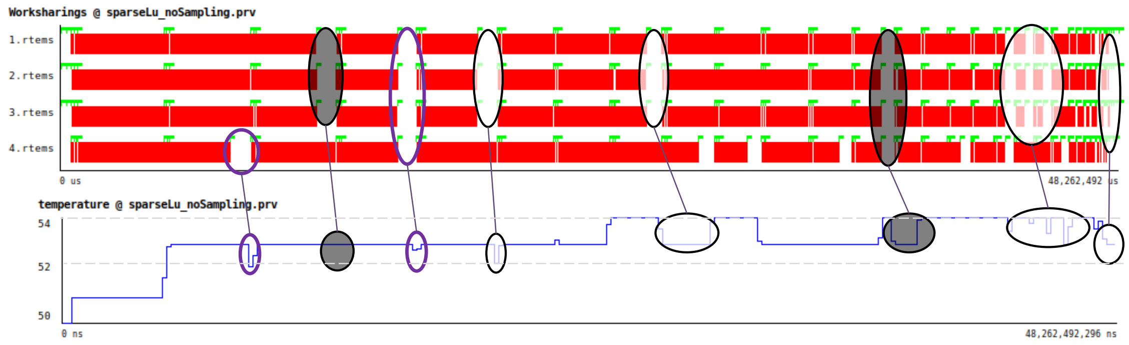


Figure 4: Analysis of the temperature for the *LU-for* benchmark with Paraver: (top) useful parallel execution view, and (bottom) temperature view.

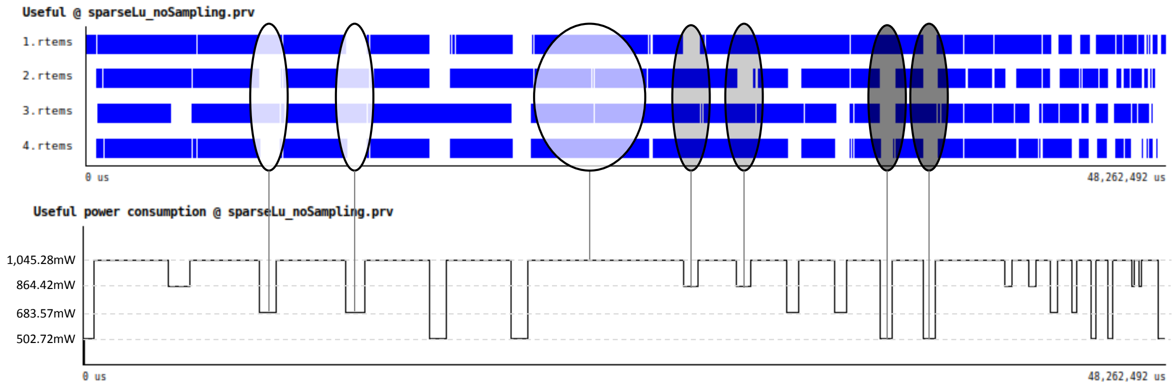


Figure 5: Analysis of the power consumption for the *LU-for* benchmark with Paraver: (top) useful computation view, and (bottom) power consumption view.

Table 1: Paraver efficiency profile for trace in Figure 5.

	CPU idle	*CPU Running*
1.rtems	11.53%	88.47%
2.rtems	14.08%	85.92%
3.rtems	11.44%	88.56
4.rtems	18.67%	81.33%
Total	55.72%	344.28%
Average	13.96%	86.07%

with a wider TDG might use better the processing resources, as more parallel work can be done. Ideally, the width is maintained across all layers of the TDG. Despite this, the width and the number of resources available are tightly related: there is no benefit of having more resources than the actual parallelism exposed (TDG width), and there is also no benefit on exposing more parallelism than actual resources.

Figure 7 shows a Paraver view of the LU benchmark with the `task` constructs filter (each color represents a different task, matching the colors in Figure 6) and the communications shown as yellow lines. The flatter the communication lines, the longer the time a task has to wait although its dependencies are already solved (notice that the description shown in Paraver includes the amount of time spent in the communication, as well as its type). We conclude that the architecture does not fully exploit the parallelism exposed by the LU benchmark: The GR740 features 4-cores, while the maximum amount of parallelism exposed is 9, as shown by the width of the TDG presented in Figure 6.

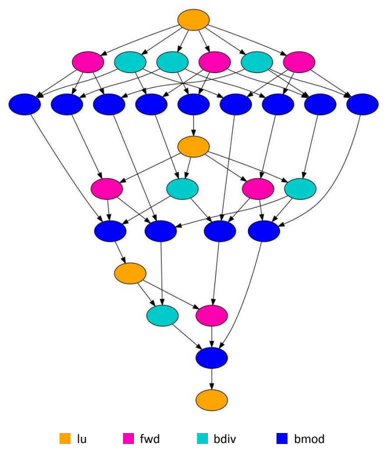
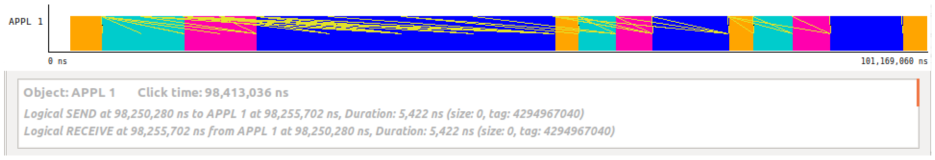
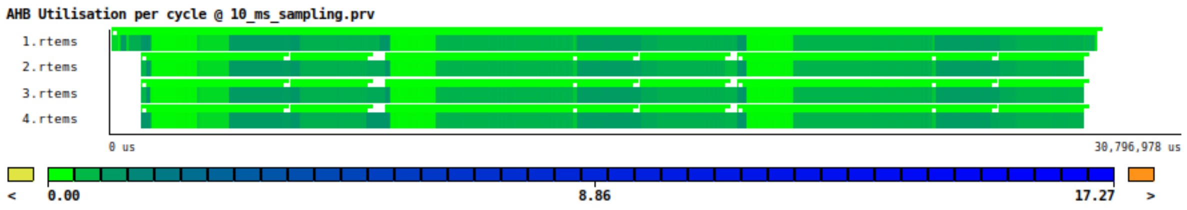


Figure 6: TDG of the *LU-tasks* benchmark.

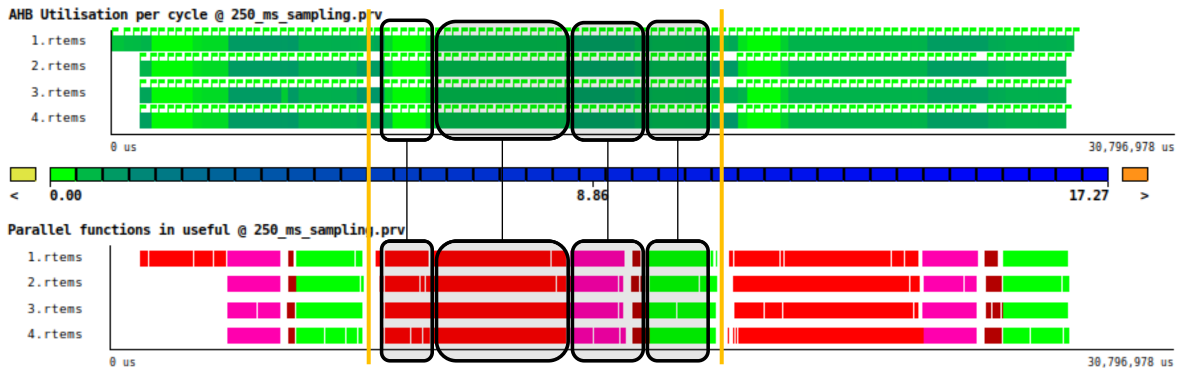


Object: APPL 1 Click time: 98,413,036 ns
Logical SEND at 98,250,280 ns to APPL 1 at 98,255,702 ns, Duration: 5,422 ns (size: 0, tag: 4294967040)
Logical RECEIVE at 98,255,702 ns from APPL 1 at 98,250,280 ns, Duration: 5,422 ns (size: 0, tag: 4294967040)

Figure 7: Analysis of the parallelism exposed and exploited for the *LU-tasks* benchmark with Paraver: user tasks view with communications.



(a) Sampling period: 10 ms.



(b) Sampling period: 250 ms. AHB utilization per cycle (top) and parallel user functions (bottom).

Figure 8: Analysis of the AHB utilization per cycle for different sampling periods for the Proc benchmark with Paraver.

5.2.4 The AMBA bus with sampling. This section further evaluates the analysis capabilities of Extrae, by using the new sampling mechanism, based on hardware interruptions of the Timer modules available in the GR740 architecture, when executing the *Proc* application. We take advantage of our new implementation using the L4stat unit in the GR740 in order to analyze, based on sampling, the utilization of the AMBA bus connecting the processors and the L2 cache (see details of the AHB bus in Figure 2). This is a specifically interesting component of many embedded systems because it may be a bottleneck for communications among cores, as well as for accessing the L2 level cache. Moreover, the AMBA bus is widely used on a range of ASIC and SoC parts, allowing to apply this analysis in many embedded processor architectures.

Figures 8a and 8b (top) show the same *AHB Utilization per Cycle* view for two different sampling periods: 10ms and 250ms respectively. The flags indicate the events, and the different colors are a gradient expressed in the legend below the images: light green color corresponds to values close to 0%, while dark-green colors are above 0.5%. The trace with a period of 10ms does not add much information compared to that of 250ms, while adds much more overhead. Hence, for this application a period of 250ms is reasonable. Furthermore, adding the *parallel functions* view, where each color represents a different user task (user-function names are anonymized due to confidentiality issues) allows us to relate each parallel function with the use of the AHB processors bus.

6 CONCLUSIONS

The need for analysis tools for embedded systems that are able to correlate parallel performance with non-functional requirements at the parallel programming model level has been thoroughly motivated in this paper. As a consequence, the paper represents a step forward in the introduction of parallel programming models into embedded systems by virtue of its contributions: (1) the analysis of the constraints of embedded systems to be considered when implementing/porting analysis tools for such systems; (2) the porting of Extrae, a well-known tracing tool for HPC systems, to the GR740 SoC; (3) the introduction of new features in Extrae/Paraver to trace non-functional requirements (i.e, temperature, energy and memory consumption);and (4) the evaluation of the new capabilities of Extrae/Paraver when analyzing applications in the GR740

SoC. The work done exposes that the requirements for adapting vanilla Extrae to a embedded system are a POSIX API, a standard C library, and a GCC or LLVM toolchain. Other features like the NFS could be replaced to other similar options like TFTP if it is not available in the target system. Regarding the extra features added to Extrae, each one has a different applicability depending on the environment: (a) the temperature and power measurements are applicable to only the GR740 with any OS, (b) the head and stack memory measurements are applicable to any system with RTEMS, and (c) the task communication instrumentation is applicable to any system.

Overall, we conclude that the enhanced version of Extrae can provide insightful information about the parallel performance of embedded applications and correlate this information with the non-functional requirements of the system. More importantly, the combined use of Extrae and Paraver allow relating the different levels of information (i.e., hardware, operating system and parallel programming model) in a very user-friendly interface, making the debugging and analysis process easier.

ACKNOWLEDGMENTS

This work has been partially funded from the HP4S (High Performance Parallel Payload Processing for Space) project under the ESA-ESTEC ITI contract № 4000124124/18/NL/CRS.

REFERENCES

[1] Barbara Chapman, Lei Huang, Eric Biscondi, Eric Stotzer, Ashish Shrivastava, and Alan Gatherer. 2009. Implementing OpenMP on a high performance embedded multicore MPSoC. In *International Symposium on Parallel & Distributed Processing*. IEEE, 1–8.

[2] Cobham Gaisler. 2019. Quad Core LEON4 SPARC V8 Processor Data Sheet. https://www.gaisler.com/doc/gr740/GR740-UM-DS-2-3.pdf

[3] Benoit Dupont de Dinechin. 2015. Kalray MPPA®: Massively parallel processor array: Revisiting DSP acceleration with the Kalray MPPA Manycore processor. In *2015 IEEE Hot Chips 27 Symposium (HCS)*. IEEE, 1–27.

[4] Mathieu Desnoyers and Michel R Dagenais. 2006. The LTTng Tracer: A Low Impact Performance and Behavior Monitor for GNU/Linux. In *OLS (Ottawa Linux Symposium)*. 209–224.

[5] Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Eduard Ayguade. 2009. Barcelona OpenMP Tasks Suite: A set of benchmarks targeting the exploitation of task parallelism in OpenMP. In *International Conference on Parallel Processing*. IEEE, 124–131.

[6] Cobham Gaisler. 2020. GR740 Data Sheet and User's Manual. https://www.gaisler.com/doc/gr740/GR740-UM-DS-1-10.pdf

[7] Cobham Gaisler. 2020. GR740 Power Calculator Spreadsheet. http://gaisler.com/doc/gr740/GR740powercalculator.xlsx

[8] Cobham Gaisler. 2020. RTEMS Cross Compilation System (RCC). https://www.gaisler.com/index.php/products/operating-systems/rtems

[9] Markus Geimer, Felix Wolf, Brian JN Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. 2010. The Scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience* 22, 6 (2010), 702–719.

[10] GNU. 2020. Binutils. https://www.gnu.org/software/binutils/.

[11] GNU. 2020. The GNU C library (glibc). https://www.gnu.org/software/libc/.

[12] GNU. 2020. Libiberty. https://gcc.gnu.org/onlinedocs/libiberty/.

[13] Toshihiro Hanawa, Mitsuhisa Sato, Jinpil Lee, Takayuki Imada, Hideaki Kimura, and Taisuke Boku. 2009. Evaluation of multicore processors for embedded systems by parallel benchmark program using OpenMP. In *International Workshop on OpenMP*. Springer, 15–27.

[14] Magnus Hijorth, Martin Aberg, Nils-Johan Wessman, Jan Andersson, Remy Chevallier, Russel Forsyth, Rolad Weigand, and Luca Fossati. 2015. GR740: Rad-hard Quad-core LEON4FT System-on-chip. In *DAta Systems in Aerospace*.

[15] Innovative Computing Laboratory, University of Tennessee. 2019. Performance Application Programming Interface, PAPI. https://icl.utk.edu/papi/index.html

[16] Keil, An Arm® Company. 2019. μVision® IDE. http://www2.keil.com/mdk5/uvision/

[17] Keil, An Arm® Company. 2019. ULINKplus Debug Adapter. http://www2.keil.com/mdk5/ulink/ulinkplus/

[18] Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen Malony, et al. 2012. Score-P: A Joint Performance Measurement Run-time Infrastructure for Periscope, Scalasca, Tau, and Vampir. In *Tools for High Performance Computing 2011*. Springer, 79–91.

[19] Germán Llort, Antonio Filgueras, Daniel Jiménez-González, Harald Servat, Xavier Teruel, Estanislao Mercadal, Carlos Álvarez, Judit Giménez, Xavier Martorell, Eduard Ayguadé, et al. 2016. The Secrets of the accelerators unveiled: tracing heterogeneous executions through OMPT. In *International Workshop on OpenMP*. Springer, 217–236.

[20] Marshall Kirk McKusick, George V Neville-Neil, and Robert NM Watson. 2014. *The design and implementation of the FreeBSD operating system*. Pearson Education.

[21] Nick Mehta. 2013. Xilinx ultrascale architecture for high-performance, smarter systems. *Xilinx White Paper WP434* (2013).

[22] Sparsh Mittal. 2019. A Survey on optimized implementation of deep learning models on the NVIDIA Jetson platform. *Journal of Systems Architecture* (2019).

[23] OpenMP ARB. 2018. OpenMP Application Program Interface, version 5.0. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf.

[24] Percepio. 2019. Tracealyzer. https://percepio.com/tracealyzer/

[25] Robert Schone, Ronny Tschuter, Thomas Ilsche, Joseph Schuchart, DanielHackenberg, Wolfgang E. Nage. 2017. Extending the Functionality of Score-P Through Plugins: Interfaces and Use Cases. *Tools for High Performance Computing* (2017).

[26] Sara Royuela, Alejandro Duran, Maria A Serrano, Eduardo Quiñones, and Xavier Martorell. 2017. A Functional Safety OpenMP* for Critical Real-Time Embedded Systems. In *International Workshop on OpenMP*. Springer, 231–245.

[27] Michael Schmid, Florian Fritz, and Juergen Mottok. 2019. Parallel Programming in Real-Time Systems. In *32nd International Conference on Architecture of Computing Systems*. VDE, 1–7.

[28] SEGGER. 2019. J-Trace Streaming Trace Proves. https://www.segger.com/products/debug-probes/j-trace/

[29] SEGGER. 2019. SystemView. https://www.segger.com/products/development-tools/systemview/

[30] Maria A Serrano, Alessandra Melani, Roberto Vargas, Andrea Marongiu, Marko Bertogna, and Eduardo Quinones. 2015. Timing characterization of OpenMP4 tasking model. In *2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*. IEEE, 157–166.

[31] Maria A Serrano, Sara Royuela, and Eduardo Quiñones. 2018. Towards an OpenMP specification for critical real-time systems. In *International Workshop on OpenMP*. Springer, 143–159.

[32] Harald Servat, Germán Llort, Juan González, Judit Giménez, and Jesús Labarta. 2015. Low-Overhead Detection of Memory Access Patterns and Their Time Evolution. In *Euro-Par 2015: Parallel Processing*. 57–69.

[33] Sameer S. Shende and Allen D. Malony. 2006. The TAU Parallel Performance System. *International Journal of High Performance Computing Applications* 20, 2 (2006), 287–311.

[34] Rapita Systems. 2019. Products. https://www.rapitasystems.com/products

[35] Giuseppe Tagliavini, Daniele Cesarini, and Andrea Marongiu. 2018. Unleashing Fine-grained Parallelism on Embedded Many-core Accelerators with Lightweight OpenMP Tasking. *IEEE Transactions on Parallel and Distributed Systems* 29, 9 (2018), 2150–2163.

[36] The European Space Agency. 2017. RTEMS-SMP Improvement for LEON multicore. https://www.esa.int/Enabling_Support/Space_Engineering_Technology/Software_Systems_Engineering/RTEMS-SMP_Improvement_for_LEON_multicore

[37] BSC Performance Tools. 2019. Extrae. https://tools.bsc.es/extrae

[38] BSC Performance Tools. 2019. Paraver. https://tools.bsc.es/paraver

[39] BSC Performance Tools. 2019. Paraver Trace File Description. https://tools.bsc.es/doc/1370.pdf

[40] Roberto E. Vargas, Sara Royuela, Maria A Serrano, Xavi Martorell, and Eduardo Quiõnes. 2016. A lightweight OpenMP4 run-time for embedded systems. In *2016 21st ASP-DAC*. IEEE, 43–49.

[41] Angelo Nery Crestani Vieira, Paulo Silas Severo de Souza, Wagner Santos dos Marques, Marcelo Silva da Conterato, Tiago Coelho Ferreto, Marcelo Caggiani Luizelli, Arthur Francisco Lorenzon, Antonio Carlos S Beck Filho, Fábio Diniz Rossi, and Jorji Nonaka. 2019. The Impact of Parallel Programming Interfaces on the Aging of a Multicore Embedded Processor. In *International Symposium on Circuits and Systems*. IEEE.

[42] Corinna Vinschen and Jeff Johnston. 2020. Newlib. https://sourceware.org/newlib.

[43] Michael Wagner, Germán Llort, Estanislao Mercadal, Judit Giménez, and Jesús Labarta. 2017. Performance Analysis of Parallel Python Applications.. In *ICCS*. 2171–2179.

[44] Matthias Weber, Ronny Brendel, Michael Wagner, Robert Dietrich, Ronny Tschüter, and Holger Brunst. 2017. Visual Comparison of Trace Files in Vampir. In *Programming and Performance Visualization Tools*. Springer, 105–121.