

Configuration of Parallel Real-Time Applications on Multi-Core Processors

Mohammad Samadi Gharajeh, Tiago Carvalho, Luis Miguel Pinho
School of Engineering, Polytechnic Institute of Porto, Porto, Portugal
{mmas, tdc, lmp}@isep.ipp.pt

Abstract—Parallel programming models (e.g., OpenMP) are more and more used to improve the performance of real-time applications in modern processors. Nevertheless, these processors have complex architectures, being very difficult to understand their timing behavior. The main challenge with most of existing works is that they apply static timing analysis for simpler models or measurement-based analysis using traditional platforms (e.g., single core) or considering only sequential algorithms. How to provide an efficient configuration for the allocation of the parallel program in the computing units of the processor is still an open challenge. This paper studies the problem of performing timing analysis on complex multi-core platforms, pointing out a methodology to understand the applications' timing behavior, and guide the configuration of the platform. As an example, the paper uses an OpenMP-based program of the Heat benchmark on a NVIDIA Jetson AGX Xavier. The main objectives are to analyze the execution time of OpenMP tasks, specify the best configuration of OpenMP directives, identify critical tasks, and discuss the predictability of the system/application. A Linux perf based measurement tool, which has been extended by our team, is applied to measure each task across multiple executions in terms of total CPU cycles, the number of cache accesses, and the number of cache misses at different cache levels, including L1, L2 and L3. The evaluation process is performed using the measurement of the performance metrics by our tool to study the predictability of the system/application.

Keywords—real-time systems, multi-core processors, timing analysis, parallel programming, OpenMP

I. INTRODUCTION

Real-time systems (RTS) are one of the essential components in industrial applications. The system software used in embedded systems, such as robotics, Internet of Things (IoT), and medical devices, needs to be programmed in such a way that real-time requirements are met. In real-time systems, tasks can be categorized into hard, soft, and firm. Hard tasks (e.g., flight control) have a strict deadline, thus missing a deadline can cause a system failure. In a soft task (e.g., data acquisition), missing a deadline causes a degradation of the system, but may not cause a critical problem in the system. Firm tasks are similar to soft, but there is no value in the output of the task after the deadline [1].

Due to the increasing performance requirements of real-time industrial applications, single-core processors are increasingly not capable of supporting many applications, leading to new modern high-performance processors. These processors may have complex architectures, spanning from a

few cores executing in shared memory to many-core processors with few tens of cores to thousands or more, mainly in clusters of shared memory cores and interconnected clusters with on-chip networking.

Timing analysis in RTS can be performed using three methods: static, measurement-based, and hybrid. Static timing analysis builds a timing model based on program structure and hardware characteristics. It is more appropriate for simple processors (e.g., single-core processors), as the timing model for these processors can be built more accurately.

Since static analysis may not be applicable for modern processors due to their complex architecture, measurement-based timing analysis is presented as a practical method, using multiple executions of the program, from hundreds to thousands, in an analysis phase. The analysis phase means that the program executes in the deployment phase before actual deployment. Hybrid timing analysis uses both static and measurement-based analysis, in which the workflow of the program is discovered using static analysis but individual components are measured using measurement-based analysis. This type of analysis is more appropriate for complex processors [2, 3].

Measurement-based methods rely on the instrumentation process to extract timing information. Instrumentation can change the timing behavior of the program. It influences the state of hardware during execution and the software during compilation. The impact of instrumentation on the program can be reduced by various techniques, such as splitting and measuring code blocks instead of measuring the whole program [2].

The performance of modern processors can be improved by parallel programming models (e.g., OpenMP) [4, 5]. However, the predictability of parallel programs in RTS is very challenging, particularly in what concerns timing analysis. Many works exist for timing analysis of real-time systems, but some (i) use static timing analysis based on traditional methods and platforms (e.g., sequential programming) that are primarily suitable for simple processors, (ii) apply measurement-based methods, but without considering parallel models to improve the system performance, or (iii) use hybrid timing analysis without using parallel platforms. A few works apply parallel models (e.g., OpenMP) with measurement-based timing analysis, which is the approach used in this work.

This paper therefore discusses the problem of performing timing analysis on complex multi-core platforms. To show the existing challenges, an OpenMP-based program of the Heat benchmark is measured to evaluate the effect of the number of threads on system performance and identify critical tasks. The

This work has been co-funded by the European commission through the AMPERE (H2020 grant agreement N° 745601) project.

program is measured with Linux perf [6] using the performance events, including total CPU cycles, L1 data cache, L2 data cache, and L3 data cache. The main contributions of the paper can be represented as:

- Discussing the importance of doing an execution time analysis which uses performance counters to understand the timing behavior of the application
- Devising a method to specify appropriate configurations of OpenMP directives (i.e., number of threads, tasks to measure, and number of cores)
- Identifying critical tasks to reduce the necessity for the measurement process at runtime
- Understanding the predictability of the system/application

We have evaluated the OpenMP-based heat benchmark on a Jetson AGX Xavier [7], in terms of the number of CPU cycles and cache-level metrics. Fig. 1 shows the CPU and GPU block diagrams of the Xavier board. As can be seen in Fig. 1(a), the CPU COMPLEX consists of four homogeneous dual-core NVIDIA Carmel CPU clusters based on ARMv8.2 with a maximum clock frequency of 2.26GHz. Each core has 64KB data L1 caches and 128KB instruction plus a 2MB L2 cache shared between a pair of cores. A 4MB L3 cache is also shared among the CPU clusters. Fig. 1(b) illustrates the main elements of the VOLTA GPU. It includes 512 CUDA cores and 64 Tensor Cores for up to 11 TFLOPS FP16 or 22 TOPS of INT8 compute, with a maximum clock frequency of 1.37GHz. Note that the GPU supports CUDA 10 with a compute capability of sm_72. It contains eight Volta Streaming Multiprocessors (SMs) with 64 CUDA cores and 8 Tensor Cores per Volta SM. Each Volta SM has a 128KB L1 cache, in which all the SMs share a 512KB L2 cache. The current work only uses the CPU cores of the architecture, future work will also consider the offload of computing tasks to the GPU fabric. Furthermore, our tool would be extended to support the measurement of programs on the GPU.

The rest of the paper is organized as follows. Section 2 presents background works, including the Heat benchmark, Linux perf, and measurement-based timing analysis. Section 3 describes the suggested approach for timing analysis. Section 4 evaluates the measurement results obtained and finally, the paper is concluded by Section 5.

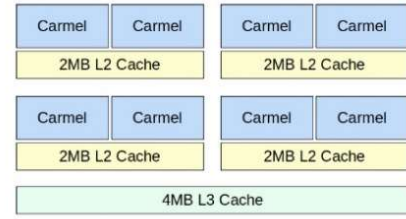
II. BACKGROUND

This section presents the background work related to the subject of this paper. It describes the Heat benchmark, the main features and applicability of Linux perf for the measurement process, and provides a review of measurement-based analysis of real-time applications.

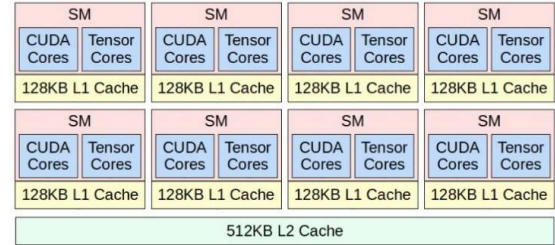
A. Heat Benchmark

The Heat benchmark [8] is a simulation that applies an iterative Gauss-Seidel method to solve the heat equation, a parabolic partial differential equation that explains the distribution of heat (or variation in temperature) in a given region over time.

The heat equation is of fundamental importance in a wide range of science fields [9]. In mathematics, it is considered as the parabolic partial differential equation par excellence. In statistics, it is related to the study of the Brownian motion. Furthermore, the diffusion equation is a generic version of the



(a)



(b)

Fig. 1. Block diagram of the Jetson AGX Xavier: (a) CPU COMPLEX; (b) VOLTA GPU.

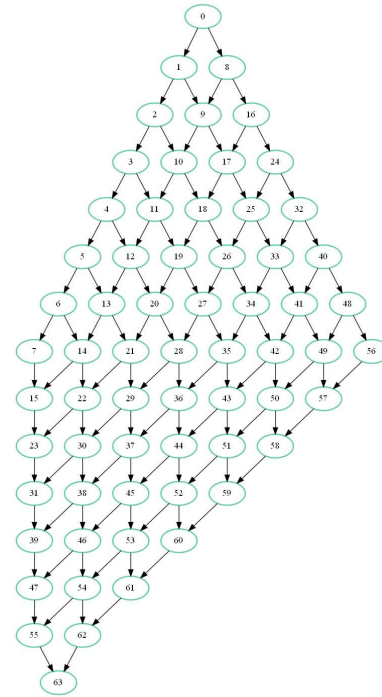


Fig. 2. TDG of the Heat benchmark.

heat equation and is related to the study of chemical diffusion processes. Fig. 2 shows the task dependency graph (TDG) of a parallel version of this benchmark, using the OpenMP tasking model, where the algorithm is decomposed in 64 tasks. The arrows represent data dependencies between tasks – although ready, a task may be suspended waiting for the availability of the output data of a sibling task.

B. Linux perf

Linux perf [6] is one of the basic and popular performance tools. After the original Pentium processor was released, patches supporting Linux for performance counters appeared soon. Although there are many features in the Linux

performance counter interfaces, only four classes are mostly used: Oprofile, Perfctr, Perfmon2, and perf event. Oprofile allows an ability to take the sample with arbitrary performance events (or a timer in the lack of performance counters) and offers profiles, frequency graphs, and stack traces. The kernel interface in Perfctr contains a feature for opening a device and accessing it with different calls. Perfmon2 is an extension of the original itanium-specific Perfmon, which adds several system calls with additional system-wide configurations performed by the pseudo-filesystem. The main advantage of the perf tool is that it is included in the Linux kernel, which is the main reason to employ it in all computing implementations.

In general, perf provides three events, including generalized events, event scheduling, and multiplexing. Generalized events are a common subset of performance events, such as the number of CPU cycles, cache accesses, and cache misses, that are available on most modern CPUs. Some of the events control hardware constraints and can only run on a specific subset of available counters with event scheduling. The multiplexing feature allows users to measure more simultaneous events than the hardware can physically support. In addition, this tool provides new hardware features, including Offcore Response, Uncore and Northbridge Events, Sampled Interfaces, Virtualized Counters, and AMD Lightweight Profiling.

C. Measurement-Based Timing Analysis

Measurement-based timing analysis has already been used in many works. In most cases, these works have used the traditional frameworks rather than using parallel models (e.g., OpenMP). This subsection addresses some of these works.

Diaz et al. [10] have combined deterministic and probabilistic jitter-bounding techniques to reliably deal with the contention on access to shared hardware resources and the variability in execution time caused by caches. This work uses a software randomization technique for cache-jitter management on the multi-core processors with multi-level caches. Moreover, it considers an upper-bound limit for multi-core contention management based on contention level of the contenders.

Li et al. [2] have addressed the possibility of adjusting between quality and cost through a block-isolation technique. This technique is used to decouple the execution of a block of code from the rest of the whole program. The size of blocks can be specified at different levels of granularity based on user requirements. It focuses on three objectives: (i) the consistency between the measured code and the original code in operation, (ii) the test-input data applied for the measurement process, and (iii) the state of the hardware prior to the measurement.

Gonzalez et al. [11] have studied the potentiality and feasibility of performance monitor hardware for embedded systems based on temporal situations. They have applied a framework to evaluate the interference that tasks receive from shared memory resources in multi-core real-time embedded systems. Furthermore, a statistical analysis is used for average and worst-case modeling of task execution times in different scenarios.

Iorga et al. [12] have analyzed the auto-tuning techniques, including random search, simulated annealing, and Bayesian optimization, that can be effective in generating better enemy

programs. The work presents a reliable methodology to measure empirical multi-core interference and an auto-tuning framework provided by a reliable measurement approach to find effective parameters for enemy tasks. Accordingly, an open-source tool is developed to automatically adjust the parameters of a set of ANSI-C enemy programs and increase the interference they cause on shared resources.

Fusi et al. [13] have addressed the performance validations and measurement-based practices for the prediction of real-time execution time. These techniques are considered for high-performance computing (HPC) applications to provide reliable and real-time guarantees. They randomly discover potential timing behavior of the application using Extreme Value Theory (EVT) through the randomization techniques, including code, stack, and heap. These techniques are used to estimate the rare and extreme execution times to derive probabilistic Worst-Case Execution Time (pWCET) curves.

Li et al. [14] have provided more flexibility in timing analysis of non-safety critical systems. They have presented a hybrid probabilistic timing analysis method to facilitate measurement-based probabilistic timing analysis (MBPTA) on software blocks/components. This method reliably executes the whole program by reducing the dependency between components and the dependency on measured execution times. It provides an exceedance probability for each possible execution time, rather than a deterministic value.

Mururu et al. [15] have presented a methodology to build non-determinism models for parallel applications based on hardware performance counters. These models try to suppress unanticipated behavior in production through the compiler optimization level. They are also provided for most of non-deterministic situations to reproduce rare condition bugs during the debugging process. One of the main goals of this work is to reduce non-determinism status of the system for increasing the coverage of runtime conditions through a fixed number of permutations.

Davis et al. [16] have addressed the quantitative measurement of the effects of data management on performance in MapReduce-based applications on two many-core systems when running on HPC systems. They have specifically assessed the effect of power capping on efficiency metrics (e.g., runtime and power consumption) over time in data-intensive applications on top of a MapReduce over MPI framework. This work uses the PAPI library to identify ideal situations for running the mini applications based on dataset characteristics, systems characteristics, and the implementation of the MapReduce programming model.

Munera et al. [17] have discussed the analysis tools for embedded systems in different aspects, especially considering the most common features when applying HPC analysis tools for these systems. Accordingly, they have described the adaptation feature by porting Extrae, a well-known tracing tool for the HPC domain, to the GR740 board, a system-on-chip (SoC) utilized in the space domain. Furthermore, the capabilities of Extrae are evaluated in identifying non-functional requirements, including temperature, energy, and application memory usage, along with running the application at the parallel programming model level. This work has also used Paraver, a visualization tool for the HPC domain, to evaluate the behavior of two different applications parallelized with OpenMP on the GR740.

Reghenzani et al. [18] have addressed probabilistic real-time theory, in particular MBPTA, based on the applications running on an HPC infrastructure. They have discussed the benefits and restrictions of this assessment when applying it to HPC and the practicable advantages for heterogeneous platforms. This work identifies the barriers to the exploitation of probabilistic real-time in HPC systems in which heterogeneous computing may be useful to solve such problems. The qualitative and quantitative features are evaluated, the benefits of probabilistic real-time hypotheses for different resource managements are studied, and experimental results carried out on a real HPC cluster are analyzed in detail.

Zhou et al. [19] have extended HPCToolkit performance tools [20], developed by Rice University, to provide a scalable measurement process and analyze GPU-accelerated applications running on NVIDIA, AMD and Intel GPUs. HPCToolkit collects call path profiles and illustrates them with a graphical user interface that presents both profile and trace views. Consequently, the following features are added to the tool: (i) a GPU-independent measurement framework to monitor the performance of GPU cores, (ii) wait-free queues for an efficient cooperation between application, runtime, and tool threads, (iii) the measurement of necessary performance metrics based on PC sampling and instrumentation, (iv) compact sparse representations of the metrics, (v) a combination of distributed-memory parallelism and multithreading for the aggregation of global assessment metrics across many profiles, and (vi) detailed information to perform a performance optimization.

Czappa et al. [21] have presented a modular method to construct the performance models of software components during the design of parallel frameworks. The main objectives of this work are to reduce conceptual complexity of the construction process and allow performance models to be aggregated together using well-defined rules in the form of formal composition operators. Moreover, composition operators are defined for the design patterns pipeline, task pool, and MapReduce to be applied in the modular construction method.

III. THE SUGGESTED APPROACH

In this paper we suggest a measurement-based approach to (i) assess the execution time of OpenMP tasks, (ii) demonstrate the importance of performing an execution time analysis (which should be more than just looking at execution time), (iii) specify appropriate configurations of OpenMP directives (i.e., number of threads, tasks to measure, and number of cores), (iv) identify critical tasks for reducing the need to measure at runtime, and (v) understand the predictability of the system/application under different conditions. Linux perf is applied to measure total CPU cycles, the number of cache accesses, and the number of cache misses at different cache levels, including L1, L2 and L3. The evaluation process is done based on the effect of the number of threads on total cycles and cache miss rate. This section describes the approach in more detail.

A. Methodology

The main objectives of the suggested technique are to understand the timing behavior of the application and guide to select an efficient configuration for the allocation of the

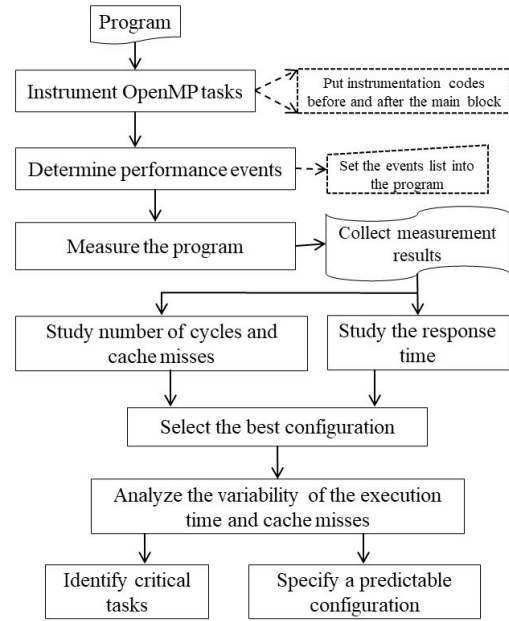


Fig. 3. Methodology of the suggested approach.

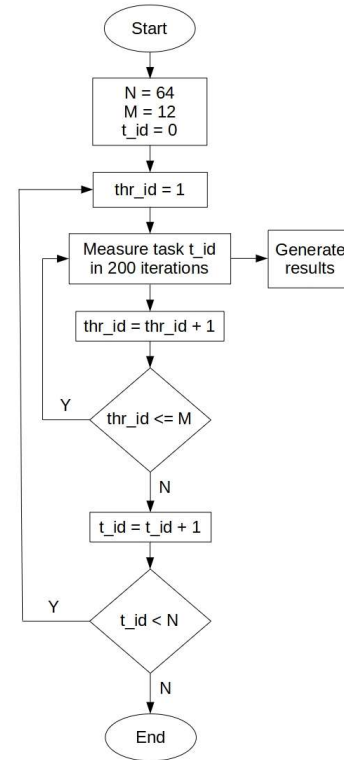


Fig. 4. Workflow of the suggested measurement process.

parallel program in the platform computing units. Moreover, it is intended to identify the “critical” tasks, which are defined as the tasks that have higher execution time variability – the goal is to reduce the runtime overhead, by reducing the number of tasks that needs to be monitored after deployment. We have applied our tool to measure the parallel tasks on the platform and used the coefficient of variance for analyzing the measurement results.

Fig. 3 show the steps of the proposed methodology. All the tasks are measured separately using performance counters based on the events list. From the measurements,

```

#pragma omp task
{
    /* Start the measurement process for t_id */
    /* Get the monitor of the current thread */
    uint32_t thread_num = omp_get_thread_num();
    xavier_monitor_t monitor = monitors[thread_num];

    if ((task_no == -1) || (task_no == t_id)) {
        /* Start measuring the events */
        xavier_reset(&monitor);
        xavier_enable(&monitor);
    }

    code();

    /* Terminate the measurement process for t_id */
    if ((task_no == -1) || (task_no == t_id)) {
        /* Stop measuring the events */
        xavier_disable(&monitor);

        /* Get measurement results of the events */
        xavier_result_t results;
        xavier_read(&monitor, &results);

        Store measurement results();
    }
}

```

Fig. 5. Instrumentation codes of the OpenMP task using Linux perf.

configurations are filtered to only analyze those that provide a response time less than a deadline. The remaining ones are analyzed based on number of cycles and cache misses in different levels, including L1, L2 and L3. The methodology provides a procedure to study execution time of the application to determine the best number of cores and threads for the configuration of the platform and OpenMP runtime. Furthermore, it clarifies the variability and configuration of the system that it is more predictable (less variability in the execution time and/or cache misses). Future work will also consider static mapping of OpenMP tasks to threads.

B. Measurement With Linux perf

The program is measured with Linux perf. Fig. 4 shows the workflow of the measurement process based on the number of tasks and the number of threads. The workflow is started by initializing the variables. N indicates the number of tasks, M represents the number of threads, t_id indicates the identifier of the current task, and thr_id represents the thread number. The measurement process is carried out in such a way that all the tasks are measured under different numbers of threads in the range of [1, 12].

Fig. 5 describes the instrumentation codes for the measurement of each OpenMP task with perf. It explains the instrumentation codes to organize the measurement process of each OpenMP task. The monitoring process starts before the first part and terminates after the last one. It is carried out for each thread separately using the current thread's identifier determined by `omp_get_thread_num()`. If the `task_no` variable is considered equal to -1, all the tasks will be measured concurrently; otherwise, the task t_id will be measured individually. Finally, measurement results of the task(s) in the current step are stored in the dedicated accumulators. Note that, in this work, the tasks are measured separately for efficiency reasons (the number of events that can be monitored and stored concurrently is limited).

IV. EVALUATION

In this section, we show a set of experiments conducted for the Heat benchmark, considering two system configurations of the Xavier platform: one using only 4 active cores and another one with 8 cores. For the 4 active cores version, a configuration of the Xavier is used that switches off two of the clusters. The evaluation process has been done in terms of time, CPU cycles, and cache miss rate for L1, L2 and

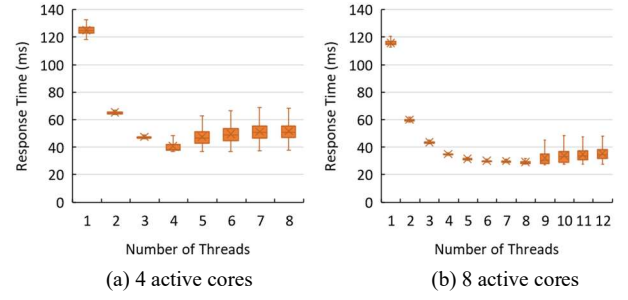


Fig. 6. Response time of the parallel region of the Heat benchmark, when running in the Xavier platform with 4 and 8 active cores.

L3, where the two last performance counters are uncore events. The OpenMP implementation of Heat program has a maximum parallelism level of 8, and it is measured using multiple performance counters. Both core and uncore events are used to measure the tasks separately. The miss rate for each cache level is calculated as:

$$Miss\ rate = \frac{Num_{dcm}}{Num_{dca}} \quad (1)$$

where Num_{dcm} is the number of data cache misses and Num_{dca} is the number of data cache accesses. Furthermore, the coefficient of variance (CV) is calculated as follows:

$$CV = \frac{Standard\ Deviation}{Mean} \quad (2)$$

This gives us an approximation factor of how spread out the values are related to the mean. Fig. 6 shows a set of boxplots for the response time of the Heat program, for both Xavier configurations, when running the program with different number of maximum OpenMP threads (OMP threads). The results show that the performance of the application improves when the number of active cores is high. For instance, using just 4 threads, the best of Fig. 6(a), it is still faster when using 8 cores (Fig. 6(b)). We can also observe that the response times in Fig. 6(b) are more stable up to 8 threads. This means that with 8 cores we have less variance in the system, which leads to a more predictable system.

When analyzing the system with 4 cores, the best number of threads used was 4 threads, which gave the overall best response time. For 2 and 3 threads, despite being slower, we have less variance and thus provide better predictability. This means that if we aim for a more predictable system we might consider the execution, for instance, with 3 threads, with a small percentage loss of performance. In contrast, for 8 cores, we would consider the best number of threads as 8 threads, which provides the best overall response time. When considering the stability of the system, here we have a higher number of cases that are stable and have close performance. Considering a deadline of 40ms for instance, the configuration might consider a range between 4 to 8 threads, mainly to their stability, as 9 to 12 threads are more unstable (i.e., has more variance and provide higher max value that goes beyond the synthetic deadline).

To better understand the stability of the execution, we present in Fig 7 the CV for five performance counters, in the different execution setups. The top row represents the results when running the Xavier platform with 4 active cores, and the bottom row when running with 8 active rows. Each column represents an execution using a certain number of OpenMP threads, more specifically 2, 4 and 8. Each chart represents the CV in the ordinate and the task id in the abscissa. Each line in

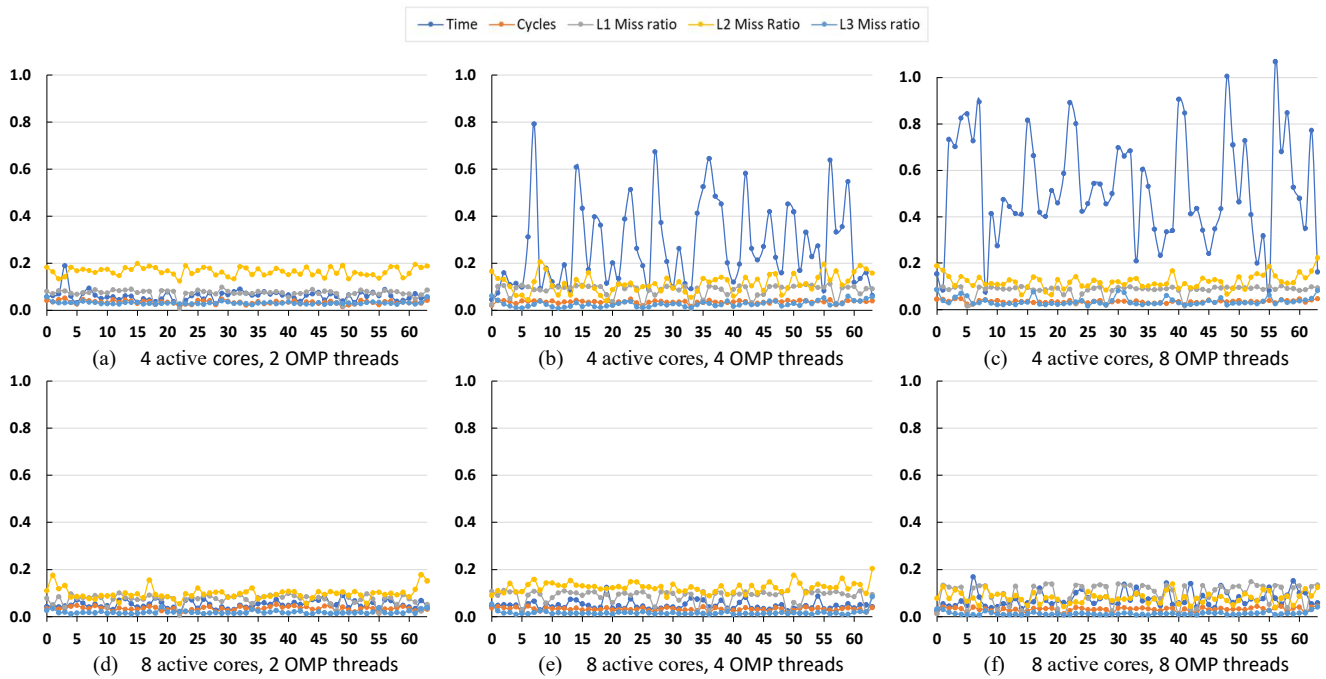


Fig. 7. Coefficient of Variance (represented in Y-axis) for each task pertaining to the Heat program (each one represented in the X-axis) for five types of performance counter measurements (each represented as a line of the chart). The first row represents the results when running the system with 4 active cores, and the second row with 8 active cores. Each column represents the execution when using 2, 4 and 8 OpenMP threads, respectively.

the chart shows a specific performance counter, which color is depicted in the top legend of Fig 7, illustrating the variation of that performance counter in that specific task. Regarding the timing analysis, critical tasks behave in an anomalous way, so that it is not possible to estimate the execution time (or WCET) of the tasks. In offline mode, all tasks are measured using the measurement process to determine the WCET. But in online mode, critical tasks are measured continuously to measure their execution time at runtime.

In overall all the performance counters, except execution time, have a CV factor less than 0.2, with most of the times being less than 0.1 (specially for 8 active cores) and the L2 Miss Ratio is also not as stable as the other performance counters. Regarding the cache misses, the results were more or less expected, as the values of cache misses are very low due to this being a regular application. In the future work, we expect to perform this study with more irregular patterns to evaluate the main differences compared to regular patterns.

Running the application with 8 active cores is in fact more stable than running with 4 active cores, with a CV factor always less than 0.2x. In fact, we saw in Fig. 6(a) that the most stable configurations for 4 active cores were using only 2 and 3 OpenMP threads, which a cost in performance. Using 8 active cores allows to keep results more stable if we use up to 8 threads.

Therefore, the main differences between CV on the 4-core and 8-core system configurations are that:

- CV in the 8-core configuration is overall lower than that of the 4-core system
- Time counter is less stable on the 4-core system, with L2 showing a slighter destabilization
- All the counters are stabler in the 8-core system, especially Time, cycles, L1 Miss Ratio and L3 Miss Ratio

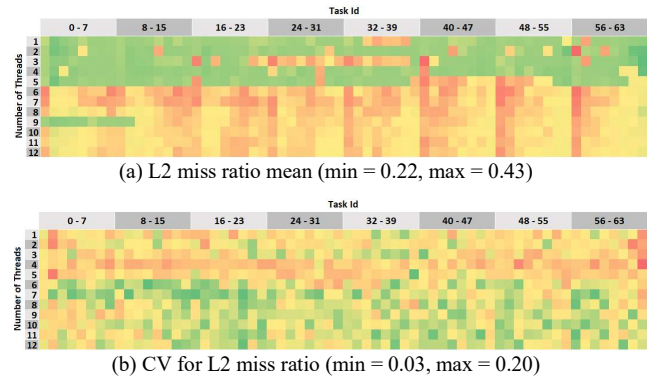


Fig. 8. CV and mean of L2 cache misses for each task executed, for different number of OpenMP threads, when running the Xavier platform with 8 active cores. Green means better, red for worst result.

Now, to further analyze the behavior of the cache misses among the different number of threads used, we show in Fig. 8 the miss ratio for L2 for the 8-core system. We selected this level of cache since this was the one that have shown the lesser stabilization among the different cache levels. The figure shows two heatmaps for cache misses, one for the average (Fig. 8(a)) and another one for the CV (Fig. 8(b)), among the different OpenMP tasks (horizontally), for a different number of threads (vertically). We show the results for all the tasks to provide an overview of the performance of each task. In Fig. 8(a), the greener the point is the lower the miss ratio and the redder it becomes, more cache misses occur for that task/number of threads combination. For Fig. 8(b), the greener the point is the less variance there is, while the redder it is, the results vary more for that combination. For both images, the in-between values are depicted in a gradient scale between green, yellow, and red colors.

The results show that, in average, the cache misses worsen (i.e., increases) as the number of threads increases. However, we can see that, in terms of CV, the values become more stable with a higher number of threads. This means that, despite

increasing the number of misses, we are able to build a configuration that provides more stable results and therefore a more predictable system.

We can conclude that the results related to the level of parallelism of 8 show that 8 threads (or less) on 8 cores is more performant and more predictable (i.e., measurements are more stable). In contrast, 4 threads on 4 cores are less predictable at any certain time of the application. So, we might have 8 tasks in parallel to execute, but only 4 cores to execute. By reducing the maximum number of threads up to 2 or 3 threads, we achieve more stable results, at the cost of performance but with a more predictable system. In addition, an interesting note is that more cache misses may actually lead to a more predictable system. In this case, although the execution time is higher, the variance is lower.

This shows how important it is to perform a timing analysis that goes beyond looking at the execution time of a task and response time of the TDG. Understanding the behavior of the system, in terms of timing, performance counters and stabilization (and other possible factors) allows one to create configurations and strategies that efficiently adapt the application to the target system.

V. CONCLUSIONS

Timing analysis is one of the main non-functional requirements in real-time systems. The response time of these systems on simple processors (e.g., single-core) can be estimated using static analysis. In contrast, complex processors (e.g., multi-core) cannot be evaluated temporally using static techniques due to their complex architecture (and black-box structure), so other techniques are needed (e.g., measurement-based analysis). Particularly, when parallel programming models (e.g., OpenMP) are used to improve the performance in these processors. This paper suggested a measurement-based approach to determine the best configuration of parallel applications on multi-core processors based on timing analysis. This approach was applied to an OpenMP-based program of the Heat benchmark, which was measured with Linux perf using performance counters. The measurement process was performed based on the effect of the number of threads on total CPU cycles and miss rate of multiple cache levels, including L1, L2 and L3. The goals were to analyze the execution time of OpenMP tasks, find out the best configuration of the OpenMP-based program (i.e., number of threads, number of cores, and tasks to measure), identify critical tasks to only measure them at runtime, and improve the predictability of the system/application.

REFERENCES

- [1] L. M. Pinho, E. Quinones, A. Marongiu, *High-performance and time-predictable embedded computing*. Aalborg: River Publishers, 2018.
- [2] H. Li, P. De Meulenaere, K. Vanherpen, S. Mercelis, and P. Hellinckx, "A hybrid timing analysis method based on the isolation of software code block," *Internet of Things*, vol. 11, article ID 100230, 2020.
- [3] A. K. Singh, P. Dziurzynski, H. R. Mendis, and L. S. Indrusiak, "A survey and comparative study of hard and soft real-time dynamic resource allocation strategies for multi-/many-core systems," *ACM Comput. Surv.*, vol. 50, no. 2, pp. 1–40, 2017.
- [4] M. Aldinucci, V. Cesare, I. Colonnelli, A. R. Martinelli, G. Mittone, B. Cantalupo, C. Cavazzoni, and M. Drocco, "Practical parallelization of scientific applications with OpenMP, OpenACC and MPI," *J. Parallel Distr. Com.*, vol. 157, pp. 13–29, 2021.
- [5] F. Montagna, G. Tagliavini, D. Rossi, A. Garofalo, and L. Benini, "Streamlining the OpenMP Programming Model on Ultra-Low-Power Multi-core MCUs," in *Proc. of Int. Conf. on Architecture of Computing Systems*, pp. 167–182, 2021.
- [6] V. M. Weaver, "Linux perf_event features and overhead," in *Proc. of the 2nd International Workshop on Performance Analysis of Workload Optimized Systems, FastPath*, vol. 13, pp. 80, 2013.
- [7] NVIDIA Jetson AGX Xavier Delivers 32 TeraOps for New Era of AI in Robotics. NVIDIA, 2018. Accessed on the 9th of October 2021. <https://developer.nvidia.com/blog/nvidia-jetson-agx-xavier-32-teraops-ai-robotics/>
- [8] Heat benchmark. Barcelona Supercomputing Center, 2019. Accessed on the 23rd of April 2022. <https://pm.bsc.es/gitlab/openmp/examples/heat/>
- [9] S. Micu, I. Roventa, and M. Tucsnak, "Time optimal boundary controls for the heat equation," *J. Funct. Anal.*, vol. 263, no. 1, pp. 25–49, 2012.
- [10] E. Diaz, M. Fernandez, L. Kosmidis, E. Mezzetti, C. Hernandez, J. Abella, and F. J. Cazorla, "MC2: Multicore and cache analysis via deterministic and probabilistic jitter bounding," in *Proc. of the 22nd Ada-Europe International Conference on Reliable Software Technologies, Vienna, Austria*, vol. 10300, pp. 102–118, June 12–16, 2017.
- [11] A. M. Gonzalez, Y. Bouchebaba, and L. Santinelli, "Multicore shared memory interference analysis through hardware performance counters," in *Proc. of the 10th European Congress on Embedded Real Time Software and Systems (ERTS 2020), Toulouse, France*, hal-02446031, January 2020.
- [12] D. Iorga, T. Sorensen, J. Wickerson, and A. F. Donaldson, "Slow and steady: Measuring and tuning multicore interference," in *Proc. of the 2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), Sydney, NSW, Australia*, pp. 200–212, April 21–24, 2020.
- [13] M. Fusi, F. Mazzocchi, A. Farres, L. Kosmidis, R. Canal, F. J. Cazorla, and J. Abella, "On the use of probabilistic worst-case execution time estimation for parallel applications in high performance systems," *Mathematics*, vol. 8, no. 3, article ID 314, 2020.
- [14] H. Li, K. Vanherpen, P. Hellinckx, S. Mercelis, and P. De Meulenaere, "Towards a Hybrid Probabilistic Timing Analysis," *Advances in Science, Technology and Engineering Systems Journal*, vol. 6, no. 1, pp. 1360–1368, 2021.
- [15] G. Mururu, K. Ravichandran, A. Gavrilovska, and S. Pande, "Generating Robust Parallel Programs via Model Driven Prediction of Compiler Optimizations for Non-determinism," in *Proc. of the 49th International Conference on Parallel Processing-ICPP (ICPP '20), Edmonton, Canada*, pp. 1–12, August 17–20, 2020.
- [16] J. Davis, T. Gao, S. Chandrasekaran, H. Jagode, a. Danalis, J. Dongarra, P. Balaji, and M. Taufer, "Characterization of Power Usage and Performance in Data-Intensive Applications Using MapReduce over MPI," *Parallel Computing: Technology Trends*, vol. 36, pp. 287–298, 2020.
- [17] A. Munera, S. Royuela, G. Llort, E. Mercadal, F. Wartel, and E. Quinones, "Experiences on the characterization of parallel applications in embedded systems with extrae/paraver," in *Proc. of the 49th International Conference on Parallel Processing-ICPP (ICPP '20), Edmonton, Canada*, pp. 1–11, August 17–20, 2020.
- [18] F. Reghenzani, G. Massari, and W. Fornaciari, "Timing Predictability in High-Performance Computing With Probabilistic Real-Time," *IEEE Access*, vol. 8, pp. 208566–208582, 2020.
- [19] K. Zhou, L. Adhianto, J. Anderson, A. Cherian, D. Grubisic, M. Krentel, Y. Liu, X. Meng, and J. Mellor-Crummey, "Measurement and analysis of GPU-accelerated applications with HPCToolkit," *Parallel Comput.*, vol. 108, article ID 102837, 2021.
- [20] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "HPCToolkit: Tools for performance analysis of optimized parallel programs," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 685–701, 2010.
- [21] F. Czappa, A. Calotoiu, T. Hohl, H. Mantel, T. Nguyen, and F. Wolf, "Design-time performance modeling of compositional parallel programs," *Parallel Comput.*, vol. 108, article ID 102839, 2021.