Virtual Timing Isolation for Mixed-Criticality Systems

Johannes Freitag
Airbus, Munich, Germany
johannes.freitag@airbus.com

Sascha Uhrig
Airbus, Munich, Germany
sascha.uhrig@airbus.com

Theo Ungerer
University of Augsburg, Augsburg, Germany
theo.ungerer@informatik.uni-augsburg.de

Abstract

Commercial of the shelf multicore processors suffer from timing interferences between cores which complicates applying them in hard real-time systems like avionic applications. This paper proposes a virtual timing isolation of one main application running on one core from all other cores. The proposed technique is based on hardware external to the multicore processor and completely transparent to the main application i.e., no modifications of the software including the operating system are necessary. The basic idea is to apply a single-core execution based Worst Case Execution Time analysis and to accept a predefined slowdown during multicore execution. If the slowdown exceeds the acceptable bounds, interferences will be reduced by controlling the behavior of low-critical cores to keep the main application's progress inside the given bounds. Apart from the main goal of isolating the timing of the critical application a subgoal is also to efficiently use the other cores. For that purpose, three different mechanisms for controlling the non-critical cores are compared regarding efficient usage of the complete processor.

Measuring the progress of the main application is performed by tracking the application's Fingerprint. This technology quantifies online any slowdown of execution compared to a given baseline (single-core execution). Several countermeasures to compensate unacceptable slowdowns are proposed and evaluated in this paper, together with an accuracy evaluation of the Fingerprinting. Our evaluations using the TACLeBench benchmark suite show that we can meet a given acceptable timing bound of 4 percent slowdown with a resulting real slowdown of only 3.27 percent in case of a pulse width modulated control and of 4.44 percent in the case of a frequency scaling control.

2012 ACM Subject Classification Computer systems organization → Real-time systems, Computer systems organization → Embedded and cyber-physical systems, Computer systems organization → Reliability

Keywords and phrases multicore, hard real-time systems, timing isolation, safety-critical systems, mixed-criticality design and assurance

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2018.13

Funding This work was partially supported by the German Federal Ministry of Education and Research within the project ARAMiS II with the funding ID 01IS16025Q and the ARTEMIS Joint Undertaking under grant agreement 621429 (EMC2).

30th Euromicro Conference on Real-Time Systems (ECRTS 2018).
Editor: Sebastian Altmeyer; Article No. 13; pp. 13:1–13:23

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Several companies are seeking a new generation of autonomously piloted aircrafts for future mobility concepts. Vehicles like *Vahana*, *Pop-up*, *CityAirbus* [4], or Lilium Jet [18] will be ultra light-weight electrical helicopter-style vehicles providing a novel autonomous urban transportation concept. The avionic systems for this kind of aircraft need to implement most functionality available in current aircrafts while providing additional complex functionality for autonomous flying. Furthermore, the electronic systems must be optimized for weight and space in order to fit into this new generation of aircrafts.

A possible solution that enables the necessary integration of multiple avionic applications into less avionic computers is the use of (massive) multicore processors comprising eight or even more cores. Avionic systems show special requirements with respect to system reliability and availability because of their safety-critical nature.

Even though first ideas of the regulations on how to apply multicore systems in avionics are presented in the CAST-32 position paper and its follow-up CAST-32a [7], both authored from the Certification Authorities Software Team (CAST), concrete design details are still open. One of the major challenges in this context is the interference between applications since theoretically one application can compromise another one, at least in the timing domain. Accordingly, an essential requirement for certification is a clear and reliable isolation of safety-critical applications that needs to be demonstrated to the certification authorities.

One of the most important issues is the contention on the memory (sub-)system resulting from different applications on the cores since it has a major impact on the actual execution time of an application. This is based not only on queued accesses to the memory and interconnection systems but also on contention on shared caches.

For multicore systems, an approach to support execution of highly critical avionic (legacy) applications is the *Fingerprinting* technology presented in [11]. *Fingerprinting* continuously tracks the progress of an application by comparing the current state of execution to a virtual single-core execution of the same application. Unacceptable timing deviations caused by inter-core interferences can be mitigated by controlling the behavior of the non-critical cores. Furthermore, the approach used for slowing down the cores shall allow the most efficient possible usage of the other cores.

The contributions of this paper are

- an evaluation of the *Fingerprinting*'s accuracy,
- an analysis of the *Fingerprinting*'s (non-)intrusiveness on the main application,
- three possible approaches to influence the behavior of the low priority cores for interference reduction of the critical core,
- a complete external closed control loop (CCL) that guarantees virtual timing isolation between one main application and any other application running on a multicore system.

The remainder of this paper is organized as follows. The environment in which the approach applies as well as the relevant hardware configurations are presented in Section 2. Section 3 provides an overview of mature techniques and related work. The fingerprint technology is described in Section 4 while the actuators are presented in Section 5. Section 6 introduces the complete control loop. Sections 4 to 6 comprise individual evaluations. The paper concludes with Section 7 including an outlook on future work.

2 Setting the Scene

The avionic domain is a very defensive domain regarding novel technologies, mainly caused by possible safety issues. Hence, we focus on the use of multicores with only a single-core executing highly (safety) critical application (referred to as *main* application in the

following) while the others run applications with lower criticality. With respect to the timing requirements examined in this paper, this means the first core is executing applications with hard deadlines which must never be missed while the other cores run weakly hard [6], soft, or non real-time applications. Accordingly, we are proposing a technique that enables performance and timing guarantees for one core on the cost of the other cores' performance.

In this approach we focus on critical applications that are executed periodically which is typically the case for avionic applications. An example is an application which in every loop reads data as input, processes the data and creates an output while the complete procedure happens in a cycle of 5ms to 100ms. Any type of algorithm can be computed and the execution of different code depending on the input is possible in every loop. A lightweight operating system can schedule multiple applications with fixed time slicing (e.g. as in integrated modular avionics (IMA)). However, the critical application shall not exchange data with the low priority applications. The aforementioned restrictions do not apply for the low non-critical applications running on the other cores. However, no timing guaranties can be provided for these applications and it must be possible to change the timing behavior arbitrarily without crashing neither the high-critical nor the low-critical application.

In order to reuse legacy software, guaranteeing a required performance shall be non-intrusive. Moreover, modifications to an operating system (if any) shall be restricted to a minimum to not increase system complexity too much. As appropriate standards and best-practices propose extra circuits external to the processor system to increase system reliability and safety (e.g. mentioned in [7]), we target at using such an external device for guaranteeing performance and timing. In the optimal case, this timing isolation shall be done in addition to the original tasks of a watchdog system.

2.1 Basic Idea

Our virtual timing isolation approach tracks the main application's progress on the basis of characterized behavior of hardware event counters integrated inside the core of a multicore. Examples for suitable events are the number of executed instructions, cache misses, and executed floating point operations based on a given time period. Periodically reading and resetting such counters results in a curve that is characteristic for an executed application, more specifically, for the progress of that application. When comparing a recorded reference curve to the performance counter values measured online, the current progress with respect to the reference execution can be measured.

In case the performance drops down, our timing isolation system is able to thwart the other cores to reduce interferences. An integrated closed loop controller is responsible for this task.

The hardware setup consisting of a main multicore processor and the timing isolation system is shown in Figure 1. In our demonstrator system, a Xilinx FPGA is used for the timing isolation system and implements all functionality required for measuring and influencing the performance of the main application running on one core of the multicore. The two systems are connected by the trace channels of the multicore (high speed serial link) which need to be a bidirectional connection. The available Aurora interface fulfills this requirement and provides access to the internal debug unit. Hence, the FPGA can read the performance counters via the debug unit and any action for controlling the cores can also be performed by this debug unit.

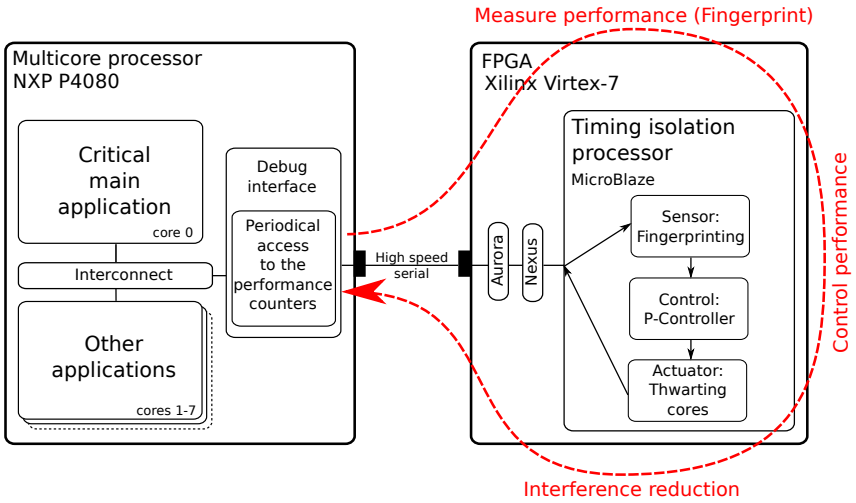


Figure 1 Hardware setup with a multicore processor under observation by the timing isolation system implemented in an FPGA.

Table 1 Different cache configurations used in the evaluations.

Realistic	
L1	on
L2	off
L3	off
Memory	ext. SDRAM

Max. Interferences	
L1	off
L2	off
L3	off
Memory	ext. SDRAM

Max. Traffic	
L1	off
L2	off
L3	int. SRAM
Memory	int. SRAM

2.2 Hardware Configuration

The system under observation is a NXP P4080 which is an eight-core multicore processor based on the PowerPC architecture (see Figure 1). All cores are configured to run at a nominal frequency of 1.5 GHz. The processor comprises three caching levels where the L1 (separated instruction and data) and L2 (shared instruction and data) caches are private to each core while the 2 MB L3 cache is shared between all cores [23]. Furthermore, the processor provides two memory controllers from which only one is used for the evaluations. Cache coherency as well as cache stashing is disabled for isolation of the cores.

For our evaluations we used different configurations of the caches (see Table 1) to demonstrate the different technologies under appropriate conditions. In all configurations the private L2 cache is disabled because enabling it increases core-local caches and reduces interferences between cores (due to lower miss-rates). Moreover, L3 is never used as shared cache since this would complicate a possible WCET analysis. The *Realistic* configuration uses the local L1 instruction and data caches and the external SDRAM as main memory. *Max. Interference* also disables both L1 caches to generate the maximum accesses from the cores to the interconnect. Since all accesses target the external SDRAM, they show a comparatively long latency. The *Max. Traffic* configuration is similar to *Max. Interference* but uses an internal SRAM (L3 used as SRAM) instead of the external SDRAM. This configuration generates the highest traffic on the interconnect due to the low latencies of accesses.

The timing isolation processor is implemented as a soft-core micro-controller (Xilinx MicroBlaze) running at 125 MHz inside a Xilinx Virtex-7 FPGA (see Figure 1). The FPGA is attached to the debug unit of the P4080 via a high speed serial *Aurora* link (2.5 GBit/s).

Thus, the FPGA is able to extract information from the multicore processor without stopping the cores, trigger frequency scaling and halt and continue cores. Since messages from and to the debug interface are wrapped into the NEXUS protocol [1], we developed a Nexus IP block to speed up the process of extracting data in the FPGA. The software for monitoring and controlling the main applications progress is executed by the MicroBlaze.

2.3 Benchmarks

For the evaluation of the approach two different benchmarks are used to demonstrate the effectiveness in a worst case interference scenario (read/write opponent) and a more realistic scenario (TACLeBench benchmark suite).

2.3.1 Read/Write Opponent

In order to create a worst case interference scenario benchmark, as much data has to be stored/loaded to and from memory as possible. In the presence of caches, every access to the data should be a cache miss in the private caches in order to create interferences. Therefore, the distance between two consecutive memory accesses has to be at least the size of a cache line. In the case of the P4080 this is 64 Bytes in the L1 and L2. This technique is described in more detail in [21]. For the evaluation, an assembler program was developed that consists of a loop that only executes either load or store instructions with a distance of 64 Bytes. Thus, the code fits into the instruction cache but not in the data cache.

2.3.2 TACLeBench

TACLeBench [10] is a benchmark suite comprising five packages of algorithms which are commonly used in embedded systems. In this paper only 19 algorithms from the TACLeBench (version 1.9) sequential package are used because these algorithms do not fit completely in the private caches like the other benchmark packages. Examples of the sequential algorithms are encrypting, sorting, dijkstra, H.264 block decoding and image recognition. These programs can be compiled independent of standard libraries and operating systems which makes them easy to adapt to our test system. The code size of the individual algorithms ranges from 117 to 2710 SLOCs.

Similar to a real avionic application we executed the 19 benchmark algorithms successively in a loop. In order to simulate a program that acts different for different input parameters the order of the algorithms can be defined to be random for each run.

3 Related Work

Multicore systems in avionic applications are still not wide spread. One reason is the difficulty to obtain suitable Worst Case Execution Time (WCET) estimates since application performance can drop significantly if multiple cores (i.e. applications) are sharing bus and memory [20]. Furthermore, it is not possible to identify all interference channels on COTS multicore processors [2]. Therefore, a WCET analysis on possible worst case scenarios leads to a high WCET overestimation (estimated WCET compared to unknown realistic WCET) for current COTS MPSoCs. Hence, the performance gain of the multicore is neglected.

Predictability by processor design is studied for example in [25], [15], and [27]. Furthermore, there exist several approaches to limit or even control the interferences between high and low critical tasks on multicore systems in software to relax the worst case scenario

and, hence, improve WCET analysis results. These solutions focus on task or even thread granularity and are integrated into the scheduling of the system. The main idea of these approaches is counting e.g. bus accesses and limiting them by suspending the corresponding thread. Examples of such approaches are presented in [17], [21], [5], [26], [16], and [3]. An overview of these and other approaches is given in [14].

Even though these approaches are very interesting for newly developed applications, they are not suitable for combing multiple legacy single-core avionic applications on a multicore processor because the legacy applications or the underlying operating system would either have to be modified, which leads to a high effort in certification (because of increased system complexity), or restrict the applications in a way that the performance gain of the multicore is neglected.

A previous approach for characterizing an application's execution is presented in [9]. It is used in high performance systems to predict an application's future behavior and needs for adjusting architectural parameters for performance optimizations. It is not related to embedded real-time systems but successfully uses a similar, but intrusive, technology for tracking application's performance.

The use of feedback controllers in combination with real-time systems is not new. For example, a closed loop controller is used in [19] for dynamic resource allocation and power optimization of multicore processors. An example for closed loop control in a real-time scheduler is presented in [24] and [8] while a controller for thermal control of a multicore processor is introduced in [13]. However, all of these methods require intrusive measurements and no non-intrusive approach for controlling the interferences between cores by an external device has been presented in the past.

4 Progress Measurement using Fingerprints

Implementing a closed control loop requires a controlled system, a sensor, an actuator, and a control algorithm. The controlled system is the main application running on one core. As sensor element, we developed a *Fingerprinting* system that tracks the progress of an application transparently. The *Fingerprinting* is described in this section while the actuator and control elements are presented in Sections 5 and 6, respectively.

4.1 Fingerprinting

During the execution of an application a flow of instructions is executed. This flow is not homogeneous in terms of type of instructions (e.g. arithmetic, floating point, branch), source of the instructions (e.g. cache, internal scratchpad, external memory), and execution time of instructions (e.g. simple arithmetic, complex arithmetic, memory access). Accordingly, measuring for example the number of executed floating point instructions per time unit will lead to a characteristic curve of an application or a part of the application. If the application (or the relevant part of it) is executed several times the measured curves are very similar. For tracking the progress of a known application, its measured curve can be compared to the recorded reference curve of executed floating point instructions at any time.

In case an application executed on a multicore processor suffers from interferences with other applications on the shared memory hierarchy, its progress is slowed down. Slowing down the application will result in a stretched (in time) but shrunk (in the value range) curve. When comparing such a mutated measured curve with the original reference curve, the actual slowdown can not only be identified but also be quantified at any time during execution.

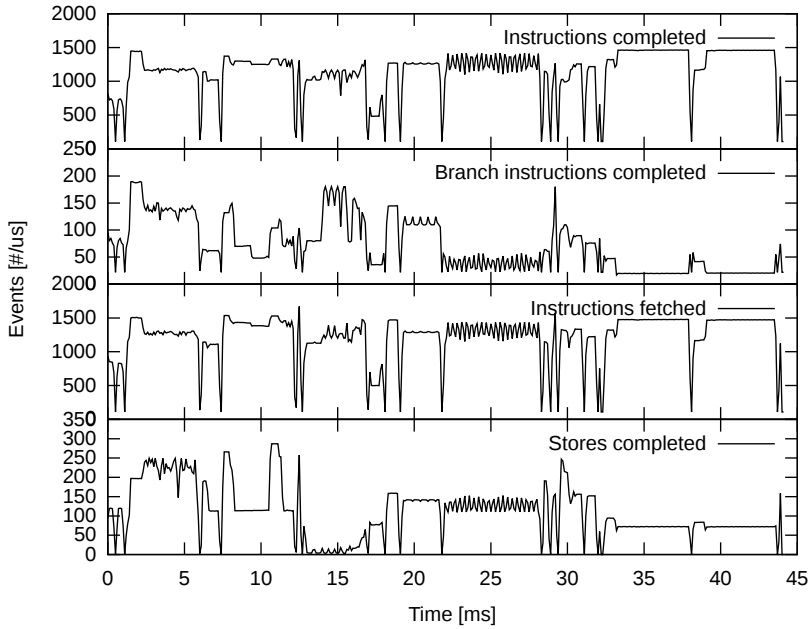


Figure 2 Measured curves of the four event counters: *Instructions completed* on top curve, *Instructions fetched* second, *Branches completed* third, and *Stores completed* in lowest curve when executing the sequential benchmarks of the TACLeBench benchmark suite.

Many current MPSoC (e.g. based on ARM, PowerPC) include performance counters implemented in hardware which can be configured to increment every time a given event is raised. While the amount of events which can be configured is usually more than 100, the amount of counters that can be incremented simultaneously is small (around four to six) [22]. Therefore, the events that are suitable for tracking have to be selected.

Figure 2 presents event counter curves of the TACLeBench *sequential* benchmarks (see Section 2.3.2) for the four event types *Instructions completed*, *Instructions fetched*, *Branches completed*, and *Stores completed*. These event types were used throughout the whole paper. For the fingerprints it is not relevant which event types are selected as long as it produces a continues stream of measurement data (which is for example not the case for floating point instructions) and the selected event types result in different curves. The displayed curves originate from a bare metal execution on a NXP P4080. The characteristics origin from the following algorithms within the TACLeBench in the following order: adpcm, anagram, audiobeam, cjpeg_transupp, cjpeg_wrbmp, epic, fmref, g723, gsm, h264, huff, ndes, petrinet, rijndael, statemate. These algorithms for example include jpeg transformations (7th to 12th ms), h264 decodings (21th to 28th ms) and AES decryption (32rd to 38th ms). In the figure it is visible that the characteristics are different for the type of algorithms executed as well as the monitored events for the same algorithm.

4.2 Creation of a Fingerprint Model

The *Fingerprint* model is obtained by executing the main application several (thousand) times. The performance counter values of the selected events are recorded with the frequency defined by the algorithm running on the timing isolation processor ($100\mu s$ period in this case which is identical to the tracking frequency). Afterward, the recorded characteristics

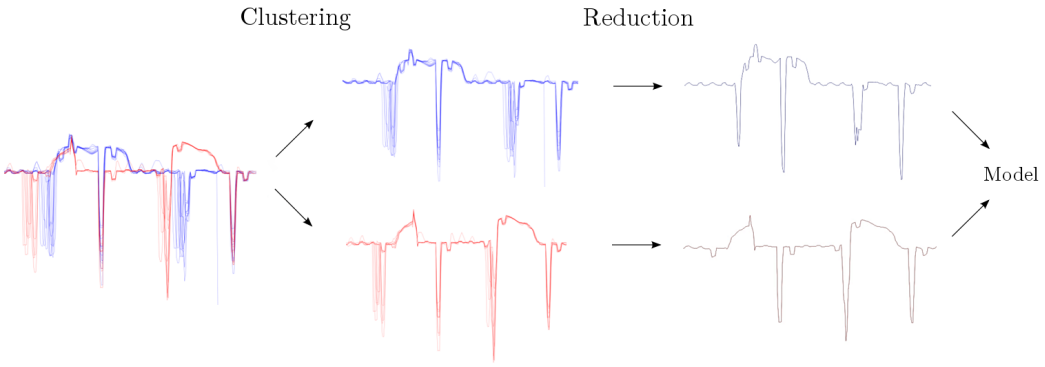


Figure 3 Generation process of the Fingerprint model. The raw data (left) is clustered by a k-means algorithm and is reduced to the median curve to build up the Fingerprint model.

are clustered using a *bisecting k-means* algorithm applying the distance function[1]

$$d(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^{n} [|x_i - y_i| > \delta_{max}] \tag{1}$$

with runtime measurement vector $\mathbf{x}$, centroid vector $\mathbf{y}$, length n of the pattern and the maximum difference between two data points δ_{max}. This distance function does not sum up the differences between each measurement point but it counts the number of samples with an error higher than the given δ_{max}. In comparison to the standard distance function this function is not sensitive to drops in the curve as displayed in Figure 3 on the left. In case of two curves where these drops are slightly shifted, the overall distance in the standard function would be big even though the rest of the curve fits perfectly. With the given distance function, errors bigger than δ_{max} are taken equally into account which better clusters the main streams within the recorded data.

The *bisecting k-means* algorithm was chosen because no predefined number of clusters has to be given. Instead, the number of clusters is resulting from a defined maximum distance d_{max}. Thus, only fingerprints with a distance $d \leq d_{max}$ to each other are in one cluster. The centroid is defined randomly for the first iteration of the *bisecting k-means* algorithm and refined in the subsequent iterations until the clusters reach their final states. The medians of the resulting cluster centroids are combined into a tree model, the *Fingerprint*. Figure 3 shows an example process flow of the Fingerprint generation. In the first step an overlay of the recorded curves is displayed which are clustered in the second step. Afterwards, the medians of the clusters are computed and the original curves are discarded. Finally, the tree model is created.

The *Fingerprint* is represented by a graph as shown in Figure 4. The root node is the starting point of the application[2]. A node describes a sequence of counter values characterizing a part of the applications execution. One node consists of at least the number of simultaneously compared samples during tracking (4 samples in the current implementation) and depending on the application, all the samples until to the end of the period of the

[1] Please note the Iverson brackets: $[P] = \begin{cases} 1 & \text{if } P \text{ is true;} \\ 0 & \text{otherwise.} \end{cases}$

[2] Note that the application must be a typical embedded application that is executed periodically.

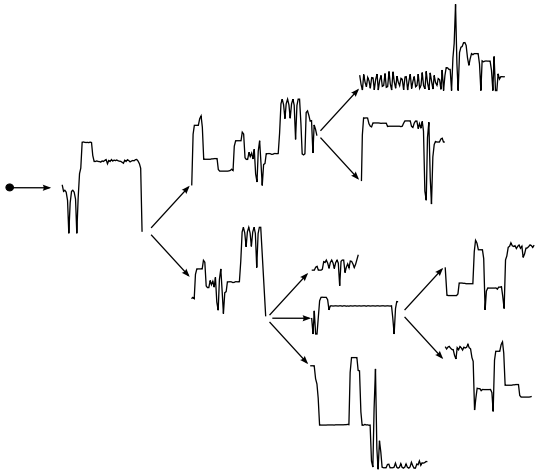


Figure 4 Example of a *Fingerprint* tree model.

application or the next branch. Each node has at least one successor node, except for the last node representing the end of execution in one iteration. In case of multiple successor nodes they represent (at least) two different paths of following characteristics. The split into multiple paths can be caused by different execution paths of the application or by different environmental conditions e.g., the first execution can have a different path than the following executions due to cold caches and warmed-up caches. It can also happen that different execution paths are not explicitly visible in the *Fingerprint* if these paths show similar characteristics as others.

4.3 Tracking the Progress of Applications in Real-Time

During actual execution time, the timing isolation system again reads the performance counter values. In this case, the values are compared to the stored *Fingerprint* model and the actual path along the tree is tracked. In contrast to the generation of the *Fingerprint* model which can be created off-line on a powerful compute node, timing is crucial for the tracking phase. The comparison between the measurements and the stored characteristic sequences has to be performed in real-time. Furthermore, it has to be done with the limited performance of a micro-controller as it would economically not make sense to observe the multicore with a high power single-core processor or even a multicore processor. In our setup the tracking is performed by a Microblaze which is a soft core micro-controller. For performance reasons the similarity is determined with a simple distance function

$$d(t) = \sum_{i=t-n}^{t} (x_i - F_i)^2 \tag{2}$$

with $\mathbf{x}$ runtime measurement vector, $\mathbf{F}$ Fingerprint vector, t discrete sampling time step from the beginning of the current period and n number of samples to compare. In the current implementation, in every iteration the recent four samples are compared. Comparison of only four samples is sufficient because the resulting similarity is accumulated over time. Thus, the decision whether the complete application run fits to the *Fingerprint* is not only based on the most recent comparison but on all the measurements.

Since measurements and *Fingerprint* never match exactly because of different execution environments (cache state, concurrent bus and memory accesses) or just because of jitter at

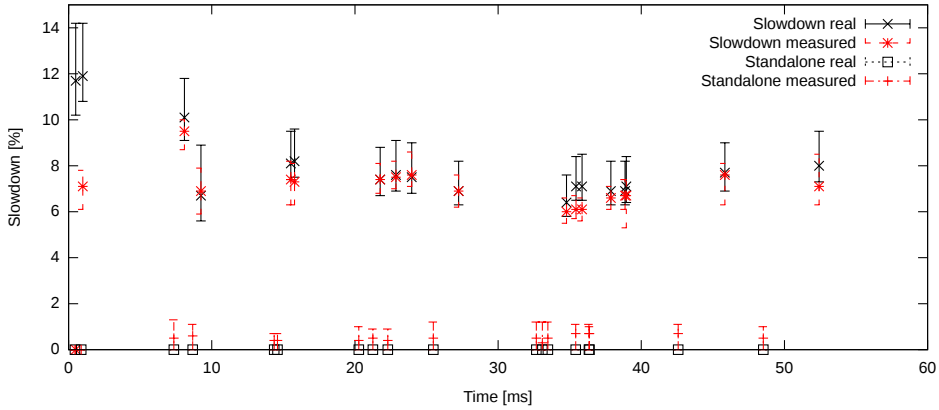


Figure 5 Quality of the quantification over the runtime of the TACLeBench suite in standalone and with seven opponent cores (*Slowdown*) using the *Realistic* cache configuration (L1 cache is enabled). The plotted dots represent the mean values while the bars reflect the minimum and maximum value measured.

the measuring points, the tracking algorithm is based on similarities, not on exact equality. This is of special importance at the nodes of the *Fingerprint* model because here the algorithm has to decide which path to continue.

Because the selection of the future path is based on similarity, there is an uncertainty at the decision point. To make the tracking robust despite of this uncertainty, our *Fingerprinting* implementation is able to track multiple different paths in parallel. In case of further branches in the tree, the most probable paths are followed and less probable paths are dropped. This decision is based on the matching of the runtime values with the path until the decision point is reached. In the current implementation four paths can be tacked simultaneously.

When determining the similarity of an actual measurement sequence to the model, the actual values are compared to the original model. Furthermore, a slowed down version of the model is computed by shifting the original model which is also compared to the measured values. In case of a better fit of the slowed down version the delay of the actual execution is determined. For the following comparisons the complete model is shifted in order to fit to the measurements. This enables the algorithm to track the application also in case of a delayed execution, e.g. by bus and/or memory contention. A slowdown can not only be detected, it can also be quantified during the execution.

4.4 Precision of the Interference Quantification over Time

For the evaluation of the precision of the quantification algorithm we instrumented the TACLeBench suite. The instrumentation is inserted at the start and after every benchmark. Thus, 20 milestones are inserted into the 19 algorithms of the TACLeBench suite subsequently. Each instrumentation consists of a time measurement within the multicore processor, which is stored in the RAM of the P4080 for later readout, and a special trace message which is sent to the quantification FPGA. Therefore, the time measurements can be used to calculate the actual slowdown which can be compared to the interference quantification values at the time these messages are received.

For the comparison, the TACLeBench was first executed standalone in order to record the time measurements without slowdown as shown in Figure 5 “Standalone real”. All the different measurements are performed 100 times and the mean values are plotted. The

bars are indicating the minimum and maximum values. The cache configuration in this experiment is *Realistic* (L1 cache is enabled). At the same time the slowdown was measured by the FPGA displayed in the "Standalone measured" curve. Here it is visible that there is a slight overestimation in the interference quantification of around 1% in some cases.

In the second step, the benchmark was executed with seven *write* opponents causing an average total slowdown of around 8%. However, the slowdown over time varies as it is depending on the different algorithms and their memory access behavior. The actual slowdown is shown in Figure 5 "Slowdown real" while the slowdown detected by the interference quantification with fingerprints is labeled as "Slowdown measured". Overall, the real and the measured slowdown are matching very well. For example the final (at 52ms) actual average slowdown value is 8.0% compared to a measured slowdown of 7.1%. In the case of the "Standalone" execution, the final average overestimation of the slowdown is only 0.5%. In total, the average deviation of the average value of less than 1%. However, in the beginning of the run (first millisecond) the values do not match. This is due to the fact that the quantification algorithm takes a small start period of around 1 ms to align the measured curve with the curves in the model. However, once this alignment is fixed the matching is very responsive as can be seen in the figure. An evaluation on which slowdowns can be reliably detected and how fast these can be detected is presented in [11].

4.5 Non-intrusiveness of the Read-Out Process

In order not to create any further interference on the critical application, the read-out overhead of the progress tracking should be as small as possible (non-intrusive). However, as the *Fingerprinting* approach relies on the performance counter values (see Section 4.1) which reside inside the cores these values have to be accessed from outside the SoC. The extraction process including the possible interference channels are displayed in Figure 6. Every extraction is triggered by an external signal sent by the external timing isolation system.

Once an external signal is received, the first step (1 in Figure 6) inside the SoC is the transfer of the performance counter values to the memory-mapped *Performance Monitor Counter Capture Registers* as the performance counter registers inside the cores are not accessible by the debug interface. This transfer is triggered by a signal from the debug interface to the respective core. In the manual of the e500mc cores [22] it is not specified where the *Performance Monitor Counter Capture Registers* are located and how the transfer is implemented. However, measurements showed that this happens in a non intrusive way as no delay of a program executed on the core could be observed.

In the second step (2 in Figure 6) the *Performance Monitor Counter Capture Registers* are accessed by the *Triggered memory-mapped access* unit of the debug interface. This is implemented as a usual memory mapped access. Therefore, the interconnect is used to transport the data to the debug interface. This is a possible interference channel as the interconnect is also used by the cores when these access the memory, the shared cache or the I/O interfaces.

For a reliable tracking four 32bit performance counter values per core need to be extracted as mentioned in Section 4.1. Depending on the read out frequency the bandwidth needed on the interconnect varies. For an example extraction frequency of 1 MHz (1 μs period) a bandwidth of 128 Mbit/s per observed core is needed. However, if the performance counters of all cores shall be extracted in parallel at this frequency the resulting bandwidth is 1 Gbit/s. Even though NXP claims that the P4080 provides 0.8 Tbps coherent read bandwidth [23], interference is measurable even if only one core is observed.

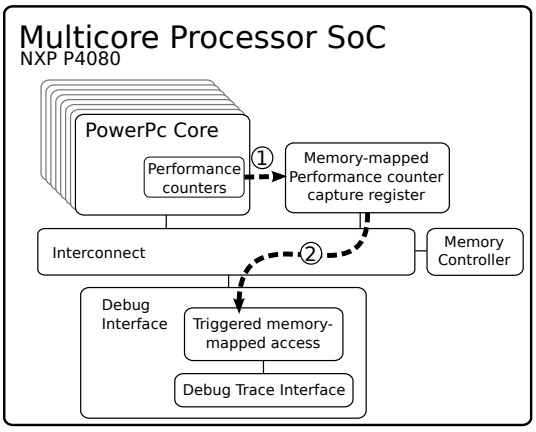


Figure 6 Possible interference channels on the example of an NXP P4080 SoC with eight cores.

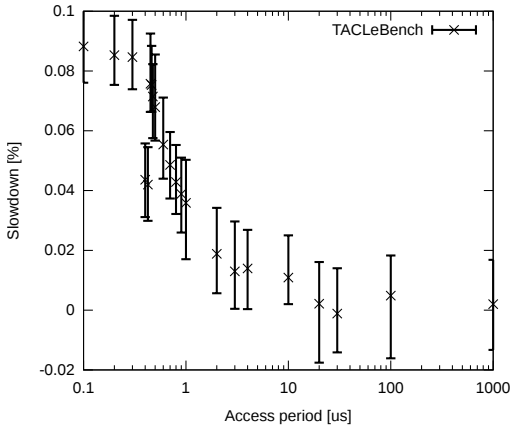


Figure 7 Slowdown of the TACLeBench execution on one core depending on the access period of the performance counter read out process.

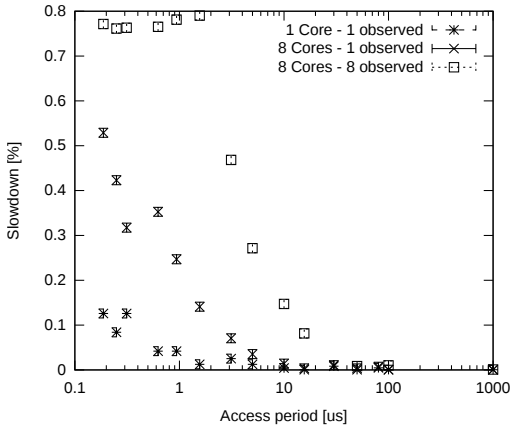


Figure 8 Slowdown of the read benchmark to the on-chip SRAM (L3 cache configured as SRAM) while the L1 and L2 caches are disabled.

The interference measured for the execution of the sequential TACLeBench benchmarks on one core while the remaining cores are idle is shown in Figure 7. The slowdown s is defined as

$$s(p) = \left(\frac{x(p)}{x_{unobserved}} - 1 \right) * 100 \tag{3}$$

with the access period p, the execution time without observation $x_{unobserved}$ and the execution time for a given period for reading out the PMC capture registers $x(p)$.

The bars are the respective observed min/max values. At some points the bars are below zero. This results from the fact that the execution time is varying even without disturbance from the read out process. These slight variations are a result of cache, interconnect and memory mechanisms.

It can be recognized that the slowdown is very small (around 0.09%) even at access frequencies of 10 Mhz. For access periods larger than 20 μs the interference is not measurable. The result is identical for the case that the performance values are extracted from the core

executing the benchmark as well as from any core that is in idle mode. Even though the interference is decreasing with a higher access period there are two measurements that are lower than expected (around 0.4 μs). We assume this is because of a synchronization of the memory accesses of the TACLeBench with the memory mapped accesses of the debug interface.

The intrusiveness analysis using the TACLeBench benchmarks shows the potential impact on a real application. However, in order to determine the worst case interference of the read out process an application was developed that almost only performs memory accesses. Furthermore, the subsequent memory accesses read/write data from/to addresses with 64 byte distance which is the size of one cache line. Additionally, the L1 and L2 caches are disabled while the L3 cache is used as SRAM memory (*Maximum Traffic* cache configuration). Therefore, every load instruction initiates a transaction in the interconnect which is considered as the worst case.

The slowdown of this application is displayed in Figure 8 for three configurations. In the first configuration the application runs on one core while the remaining cores are idle. In the second and third configuration the application is executed on all eight cores simultaneously. The extraction process is performed on one core or all the cores. The slowdown is determined similar to the TACLeBench analysis with Equation 3 but normalized to the eight core execution without reading the counter registers. Thus, the slowdown resulting from the inter-core interference is eliminated.

The measured slowdown is not significantly higher compared to the TACLeBench analysis for the one core execution. However, when all the eight cores are used for execution, the slowdown is around six times higher in case only one core is observed which is still a very low slowdown. The higher interference for the eight core execution results from the utilization of the interconnect from the cores. For access periods larger than 20 μs (50 kHz) the interference is again not measurable. In case all the cores are observed simultaneously the interference is much higher. As expected, the interference is around eight times higher compared to the case where only one core was monitored. The slowdown reaches a maximum at around 1.02 μs and the slowdown is not increasing with decreasing access periods. At this point the maximum speed of the triggered memory mapped access of the debug interface is reached. However, for extraction periods above 50 μs the interference is also not measurable.

5 Performance Control by Interference Reduction

The presented *Fingerprinting* is used as the sensor element of the closed control loop. In this section the *actuator* which influences the performance of the other cores and, hence, the interferences is explained. A simple technique for reducing the interferences with the main application is halting and resuming the opponent cores based on a threshold. For example, whenever the slowdown of the main application rises over 5% the other cores are halted and continued once the slowdown drops under 5%. This actuator is effective and stops the interferences of the opponent cores but it is heavily intrusive for the tasks running on the other cores and might have severe effects depending on the executed application. Apart from the main goal of isolating the timing of the critical application a subgoal is also to efficiently use the other cores. Therefore, two more advanced and less intrusive actuators are presented in this section. These actuators are *pulse width modulated interferences* and *frequency scaling*. We present an extended evaluation of *pulse width modulated interferences*[12] and the new approach using a *frequency scaling* methodology.

5.1 Pulse Width Modulated Interferences

Modern multicore systems like the P4080 provide means to halt and continue cores individually. Whenever a core is halted, the clocks are still running, but the core is not fetching or executing instructions [22]. Thus, no accesses to the memory are performed and the interference is stopped. Both actions can be triggered by messages on the back channel of the trace interface, i.e. by writing to control registers. This means that the timing isolation processor (see Figure 1) is able to control the activity of the cores individually from an external device.

To provide not only a binary (on/off) way of setting the performance of the cores, we implemented a (software-based) Pulse Width Modulated (PWM) enabling/disabling of the individual cores, according to the signals from the closed loop controller in short time ranges. We have chosen a PWM period of $1ms$ which is equal to 10 times the $100\mu s$ required to track the application performance. Hence, we can reduce the performance of cores competing with our main core in steps of 10%. During the duty cycle all the opponent cores are active, while for the rest of the period the cores are stopped.

5.2 Frequency Scaling

A slowdown of an interfering low priority core can also be done by a frequency downscaling. This is highly depending on the processor architecture. For example, the number of different clocks and the divider steps are varying between processors. In the case of the P4080 only four possible configurations can be selected: Two different clock sources and to each clock source a divider (divide by two) can be applied. However, in order to have a maximum frequency range, the highest frequency selectable clock frequency is 1.5GHz while the lowest frequency is 800MHz. Therefore, the possible frequencies are 400MHz, 750MHz, 800MHz and 1.5GHz which can be configured for each core individually. This configuration is also possible during runtime from an external device via the debug interface. Since, it is possible to scale down the frequency of an individual core the execution on that core can be slowed down and thus, the interferences with the main core is reduced.

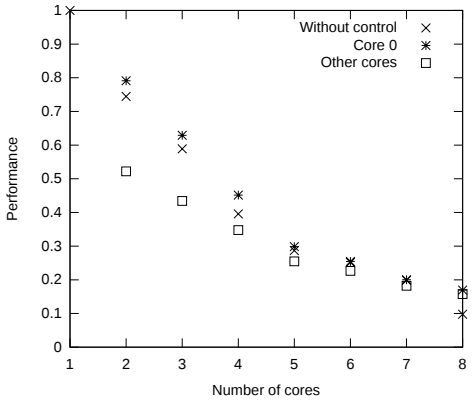
5.3 Evaluation of PWM and Frequency Scaling

For the evaluation of the effectiveness of both the PWM and the frequency scaling approach we used the TACLeBench and the read/write algorithm in three different scenarios:

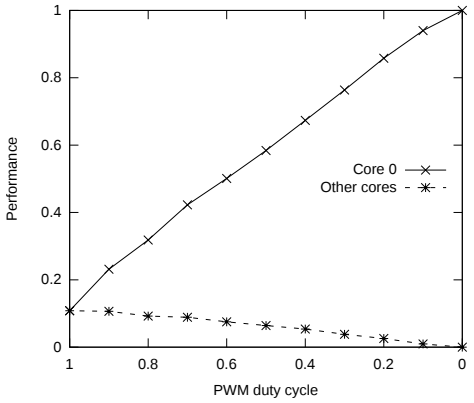
1. Read with seven read opponents (Figure 9): shows the worst case interference scenario,
2. TACLeBench with seven write opponents (Figure 10): shows a realistic application (this benchmark is application oriented and generates realistic traffic on the shared interconnect and memory and profits from local data caches) with worst case opponents,
3. TACLeBench with seven TACLeBench opponents (Figure 11): shows a realistic application on core 0 with realistic opponents.

All the scenarios were evaluated in two different cache configurations: *Realistic* (L1 is enabled) and *Maximum Interferences* (no caches enabled).

For the evaluation of the frequency scaling the performance of the applications was measured without frequency scaling of the opponent cores in the first step. All cores were running with 1.5GHz which is labeled in the figures as *Without control*. In scenario one and three the performance is identical for all applications on all core as the applications are identical. In scenario two the *Without control* performance is depicted individually. In a second step the opponent cores are set to 400MHz which is the minimum configurable speed for the P4080 when the maximum speed is configured to 1.5 GHz. The performance of core 0



(a) Frequency scaling for a varying number of opponents. The opponents are scaled down to 400 MHz while core 0 is running at 1.5 GHz.



(b) PWM halt and resume of the opponent cores for a varying duty cycle while seven opponent cores are running.

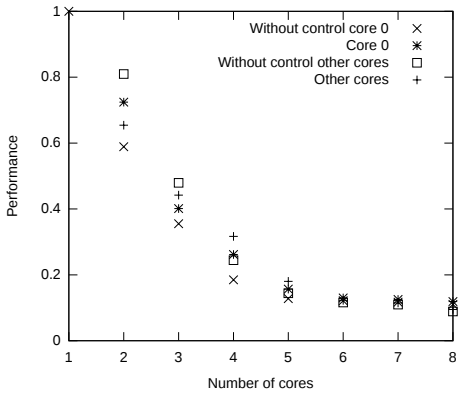
Figure 9 Frequency Scaling and PWM efficiency with the read algorithm (see chapter 2.3.1) on core 0 as well as read opponents on the other seven cores. These measurements were conducted with the *Maximum interference* cache configuration (all caches disabled). A separate analysis (not shown here) indicated that the curves are very similar for enabled local caches as the read algorithm is designed to cause maximum stress on the interconnect and does not take advantage of caches.

as well as the performance of the other cores was measured. As expected, it is visible that the performance of the other cores drops while the performance of core 0 is increased. However, the amount is highly depending on the scenario and cache configuration. The measurements were taken for a varying number of opponent cores. For example, in case of *Number of cores* is four, core 0 is running with 1.5GHz, cores 1 to 3 are running with 400MHz and cores 5 to 8 are idle in the controlled case. In the case of only one core, there is obviously no data plotted for *Core 0* and *Other cores* but this data point was used to normalize the measurements.

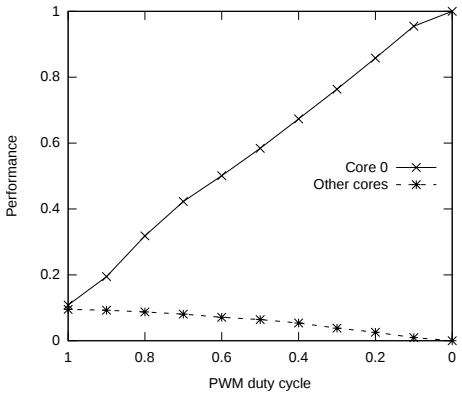
During the evaluation of the PWM approach all eight cores are utilized while the duty cycle of all seven opponent cores is varying from 0 to 100% in steps of 10%. As result, the execution time for the main application as well as the opponent applications is measured.

The results for scenario one with the read algorithm on the main core and up to seven read opponents is shown in Figure 9. It can be observed that the frequency scaling is not able to reduce the interferences from the opponent core enough to keep core 0 at a performance level higher than 90%. The increase of performance in comparison to the uncontrolled case is only around 4% for one opponent core (number of cores equal to two) and is completely negligible for seven opponent cores (number of cores equal to eight). This effect is the same for both cache configurations and can be explained by the cache behavior of the algorithm. Even though the opponent cores are executing instructions with one fourth of the speed, the memory interface is still jammed by the opponents because the memory is even slower. In contrast to that, the results for the PWM approach shown in Figure 9b reveals that even in the case of running seven opponents in parallel, the performance of the main application can be fully recovered. This shows that in the worst case the frequency scaling is not sufficient but the PWM approach can control the performance of the main application at the cost of heavily slowing down the opponents.

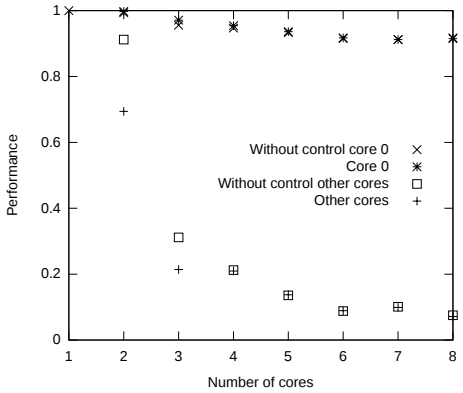
The more realistic scenario of TACLeBench with seven write opponents is shown in Figure 10. In contrast to the other scenarios there are four curves displayed for the frequency scaling instead of three. This is because there are different applications running on core



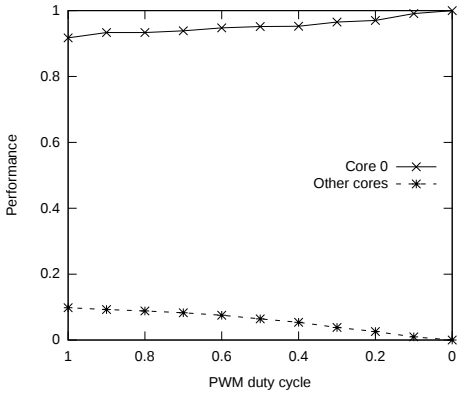
(a) Frequency scaling of the opponents to 400 MHz. No caches enabled.



(b) PWM halt and resume of seven opponents. No caches enabled.

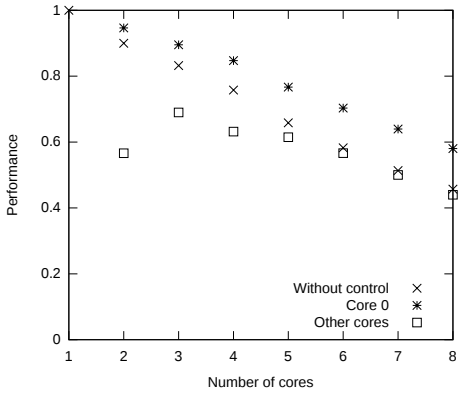


(c) Frequency scaling of the opponents to 400 MHz. L1 cache enabled.

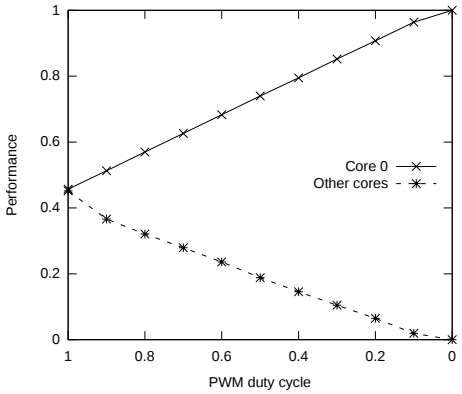


(d) PWM halt and resume of seven opponents. L1 cache enabled.

Figure 10 Frequency scaling and PWM efficiency with TACLe on core 0 and write opponents on the other cores. For Figure **a** and **b** the *Maximum interference* cache configuration applies while for Figure **c** and **d** the *Realistic* cache configuration (L1 cache enabled) was used.



(a) Frequency scaling of the opponents to 400 MHz.



(b) PWM halt and resume of seven opponents for a varying duty cycle.

Figure 11 Frequency scaling and PWM efficiency with TACLe on core 0 and TACLe opponents on the other cores. The measurements were taken with the *Maximum interference* cache configuration.

0 and the other cores which are evaluated separately. In the case of no caches the results are similar to the results in the *read/read* scenario. However, if the L1 cache is enabled the performance of the TACLeBench does not drop below 90% even with seven *write* opponents. The effect of the frequency scaling is not significant because of the cache behavior of the *write* algorithm like in the *read/read* scenario.

For the scenario of TACLeBench with seven TACLeBench opponents the results are displayed in Figure 11. It is visible that the frequency scaling has a significant effect on the performance of the application on core 0. Especially in the case of one and two opponents (two and three cores in Figure 11a) the frequency scaling increases the performance to over 90%. However, even though a performance increase of around 15% compared to the uncontrolled case is visible in the eight core case, frequency scaling is not sufficient for advancing the performance to a level of over 90%. Additional measurements (not shown in the figure) show that in the case of the *Realistic* cache configuration the loss in performance of the TACLe benchmark on core 0 is negligible and the performance of core 0 with seven opponents is still 99%.

Concluding this evaluation, frequency scaling is less efficient for improving performance of core 0 compared to PWM. On the other hand, frequency scaling affects applications running in parallel to core 0 less than PWM.

6 Closed Control Loop

We used two algorithms as control element, a simple threshold-based one and a proportional controller. Both techniques affect all concurrent cores synchronously. The threshold-based algorithm disables the concurrent cores when the slowdown of the main application exceeds a given threshold and enables the cores again when the slowdown falls below the same threshold again. The second technique uses a proportional controller with an actuator based on the PWM activity control and the frequency scaling, respectively, as described in the previous section.

As mentioned in Section 5.1, it is possible to control the cores individually which allows for idling only the low priority cores which create a high traffic on the shared resources. In order to detect the individual core interference, one performance counter in every low priority core has to be read periodically in addition to the performance counters in the critical core. This performance counter is configured with the event *bus interface unit accesses*. Thus, it counts the actual number of L2 cache misses which creates contention. The non-critical cores with the highest number of *bus interface unit accesses* can than individually be slowed down by one of the proposed techniques.

However, for this evaluation of the effectiveness of the control loop we used the TACLe benchmarks as main application and the aforementioned *Write* algorithm as opponents running on seven cores in parallel. We set a maximum slowdown of 4% as target performance of the main application compared to the standalone execution. Individual core interference detection is irrelevant in that case as all opponent cores are running the same application (worst case interference). Therefore, the low priority cores are equally slowed down while the critical application is untouched.

The results of the evaluation are shown in Figure 12, 13 and 14. In the figures, the progress of the TACLeBench over time is displayed in the upper part and the measured slowdown over time in the lower part. The upper part presents the number of executed instruction per μs. For comparison, the standalone (no opponent applications) and the uncontrolled (seven opponent applications without control) executions are displayed. The

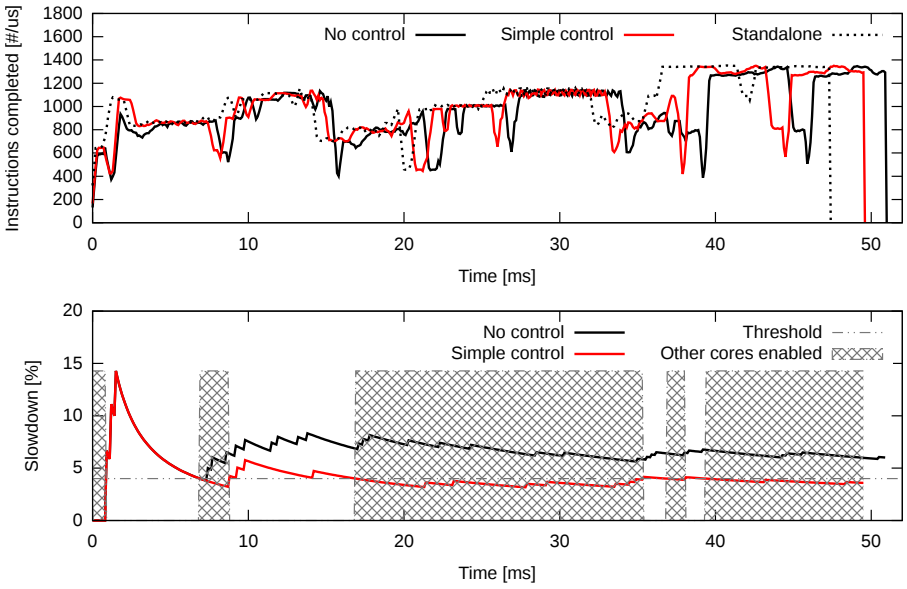


Figure 12 TACLeBench performance over time without control and with applied simple threshold controller.

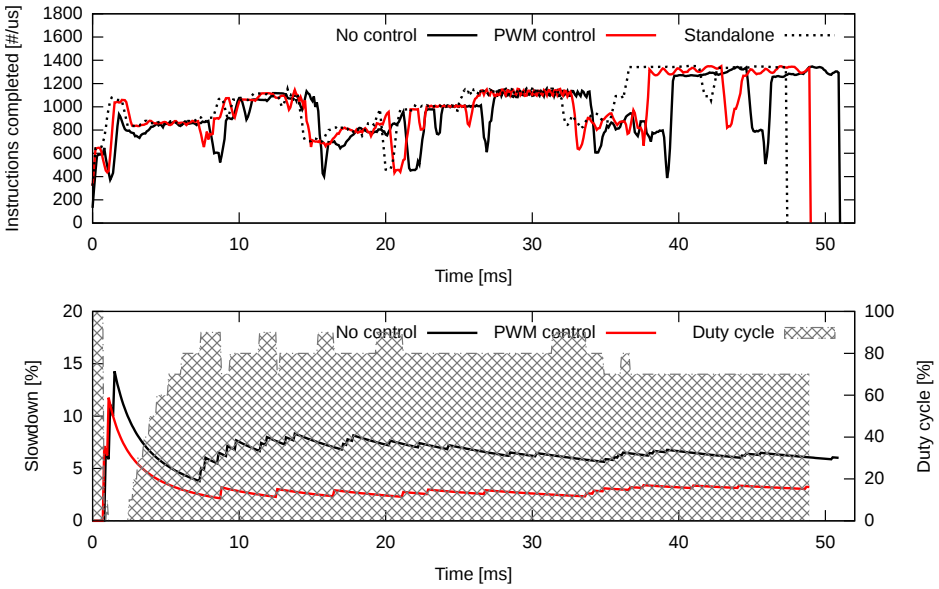


Figure 13 TACLe performance over time without control and with applied PWM controller.

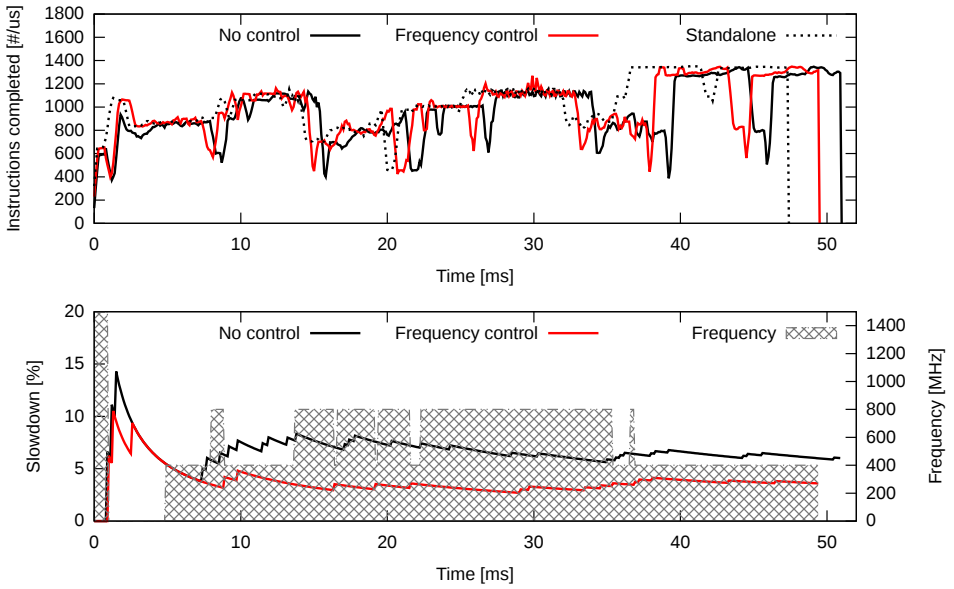


Figure 14 TACLe performance over time without control and with applied frequency scaling controller.

uncontrolled execution takes about 8% longer than the standalone run. The diagrams in the lower part of the figures represent the slowdown of the main application as tracked by the *Fingerprinting*. Since the tracking of progress is based on discrete steps, the performance reductions are manifested in sharp steps. The following phases of smooth performance increases are caused by relative distribution of a slowdown over a longer time, i.e. a one-time delay at the start of the application of 5% is reduced over the total execution time to a much lower slowdown.

In Figure 12 the results of the threshold controller are displayed. The dotted line represents the threshold (4%) i.e. the maximum target slowdown of the main application. The gray shaded boxes identify the times when the other seven cores are active. No grey shading means that the other cores are disabled by the control mechanism. It is visible that the opponent cores are disabled whenever the measured slowdown is higher than the threshold value which keeps the total slowdown in the end at a measured slowdown of 3.59%. The actual total slowdown is 4.44% (measured by comparing the times it took for executing the benchmark in the standalone and controlled case) which means an underestimation of the slowdown by the *Fingerprinting* and an exceedance of the threshold by less than 0.5%. During the total run of one TACLeBench benchmark the opponents are executed for 67% and halted for 33% of the time.

The behavior of the PWM controller is shown in Figure 13. The duty cycles of the competing cores are set according to the measured slowdown. A slowdown of less than 2% allows full performance for all cores, a slowdown above 7% leads to complete disabled competing cores. Between 7% and 2%, the duty cycles are adjusted in 10% steps from 10% to 90% (one step per half percent of slowdown). The grey shaded areas represent the duty cycles of the PWM core activation signal. As can be observed, the 4% target slowdown of the main application is reached after completion (3.23% measured while the actual slowdown was 3.27%). Moreover, the active phases of the competing cores are much longer in time but less intensive. Since we are using a PWM signal, this means that the cores are active for

many but smaller periods. With this PWM control, the seven *bad guys* get 74% of the cores' performance while the main application still meets the performance requirements which is an advantage of 7% over the threshold based actuator. Moreover, actual slowdown of the main application is better than the targeted acceptable threshold of 4%.

The frequency scaling approach is displayed in Figure 14. The possible frequencies of the opponent cores are 400MHz, 800MHz and 1.5GHz. Furthermore, the core can be halted. Similar to the PWM approach a slowdown of less than 2% allows full performance for all cores, a slowdown above 7% leads to completely disabled competing cores. Between 7% and 2%, the frequencies are adjusted in linear intervals. The grey shaded areas represent the frequencies of the opponent cores. The slowdown of the main application is reduced with a total measured value of 3.60% (real slowdown: 4.44%). However, this was not possible by only scaling down the cores. During the period of high interference in the beginning of the execution the opponent cores had to be halted for a sufficient reduction of the interferences. An assessment of the cores processing time compared to the aforementioned approaches does not make sense in this case. The frequency scaling of a core cannot be compared with halting and continuing a core because the performance of a scaled down core is highly dependent on the instructions executed.

The scenarios one and three show slight violations in actual slowdown compared to the target threshold. This is because of an underestimation of the slowdown by the *Fingerprinting* caused by the technologies latency. Adding a safety margin when defining the acceptable bounds could help here.

7 Conclusion and Future Work

This paper presents a virtual timing isolation of one main application running on one core from all other cores of a multicore processor. The proposed technique is based on hardware external to the multicore and completely transparent to the main application. The basic idea is to apply a single-core execution based Worst Case Execution Time analysis and to accept a predefined slowdown during multicore execution. If the slowdown exceeds predefined acceptable bounds, interferences will be reduced by thwarting other cores to keep the main application's progress inside the bounds.

We evaluated the accuracy of the transparent tracking of the application's progress (*Fingerprinting*), the effectiveness of different thwarting techniques, and the performance of a complete closed control loop using a simple P-controller. The latter shows that it is possible to transparently enable an application staying within given timing bounds even though there are a maximum of seven opponents flooding the shared interconnect with traffic. Evaluations indicated a slight underestimation of the application's slowdown which could be compensated by adding a safety margin. Determining a suitable range for this safety margin is part of future work.

It is planned to extend the thwarting in order to affect only cores driving high traffic on the interconnect instead of all competing cores. This can be reached by evaluating the interconnect accesses of the other cores to identify cores with high influence. Moreover, a combination of frequency scaling and PWM driven thwarting would be interesting for more effective and fine-grained interference control. Also using a more complex control algorithm like a full PID controller could be useful to increase performance of competing cores. The target of future research will be enabling more than one core running hard real-time applications.

References

1 The Nexus 5001 Forum - Standard for a Global Embedded Processor Debug Interface, 2012.

2 Irune Agirre, Jaume Abella, Mikel Azkarate-Askasua, and Francisco J Cazorla. On the Tailoring of CAST-32A Certification Guidance to Real COTS Multicore Architectures. In *12th IEEE International Symposium on Industrial Embedded Systems (SIES)*, 2017.

3 Ankit Agrawal, Gerhard Fohler, Johannes Freitag, Jan Nowotsch, Sascha Uhrig, and Michael Paulitsch. Contention-Aware Dynamic Memory Bandwidth Isolation with Predictability in COTS Multicores: An Avionics Case Study. In Marko Bertogna, editor, *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, volume 76 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 2:1–2:22, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ECRTS.2017.2.

4 Airbus. Future of urban mobility. 2018. http://www.airbus.com/innovation/urban-air-mobility.html.

5 S. Bak, G. Yao, R. Pellizzoni, and M. Caccamo. Memory-aware scheduling of multicore task sets for real-time systems. In *2012 IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 300–309, Aug 2012. doi:10.1109/RTCSA.2012.48.

6 G. Bernat, A. Burns, and A. Liamosi. Weakly hard real-time systems. *IEEE Transactions on Computers*, 50(4):308–321, Apr 2001. doi:10.1109/12.919277.

7 Certification Authorities Software Team (CAST). Position Paper CAST-32A: Multi-core Processors. November 2016. URL: https://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_papers/media/cast-32A.pdf.

8 Tommaso Cucinotta, Fabio Checconi, Luca Abeni, and Luigi Palopoli. Self-tuning Schedulers for Legacy Real-time Applications. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 55–68, New York, NY, USA, 2010. ACM. doi:10.1145/1755913.1755921.

9 Evelyn Duesterwald and Sandhya Dwarkadas. Characterizing and predicting program behavior and its variability. In *In International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 220–231, 2003.

10 Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Bo Sørensen, Peter Wägemann, and Simon Wegener. TACLeBench: A benchmark collection to support worst-case execution time research. In Martin Schoeberl, editor, *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*, volume 55 of *OpenAccess Series in Informatics (OASIcs)*, pages 2:1–2:10, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

11 Johannes Freitag and Sascha Uhrig. Dynamic interference quantification for multicore processors. In *2017 IEEE/AIAA 36th Digital Avionics Systems Conference (DASC)*, pages 1–6, Sept 2017. doi:10.1109/DASC.2017.8101991.

12 Johannes Freitag and Sascha Uhrig. Closed Loop Controller for Multicore Real-Time Systems. In Mladen Berekovic, Rainer Buchty, Heiko Hamann, Dirk Koch, and Thilo Pionteck, editors, *Architecture of Computing Systems – ARCS 2018*, pages 45–56, Cham, 2018. Springer International Publishing.

13 Yong Fu, Nicholas Kottenstette, Chenyang Lu, and Xenofon D. Koutsoukos. Feedback Thermal Control of Real-time Systems on Multicore Processors. In *Proceedings of the Tenth ACM International Conference on Embedded Software*, EMSOFT '12, pages 113–122, New York, NY, USA, 2012. ACM. doi:10.1145/2380356.2380379.

14 S. Girbal, X. Jean, J. Le Rhun, Daniel Gracia Pérez, and M. Gatti. Deterministic platform software for hard real-time systems using multi-core COTS. In *2015 IEEE/AIAA 34th*

Digital Avionics Systems Conference (DASC), pages 8D4–1–8D4–15, Sept 2015. doi:10.1109/DASC.2015.7311481.

15 Kees Goossens, Martijn Koedam, Andrew Nelson, Shubhendu Sinha, Sven Goossens, Yonghui Li, Gabriela Breaban, Reinier van Kampenhout, Rasool Tavakoli, Juan Valencia, Hadi Ahmadi Balef, Benny Akesson, Sander Stuijk, Marc Geilen, Dip Goswami, and Majid Nabi. *NoC-Based Multiprocessor Architecture for Mixed-Time-Criticality Applications*, pages 491–530. Springer Netherlands, Dordrecht, 2017. doi:10.1007/978-94-017-7267-9_17.

16 N. Kim, B. C. Ward, M. Chisholm, C. Y. Fu, J. H. Anderson, and F. D. Smith. Attacking the One-Out-Of-m Multicore Problem by Combining Hardware Management with Mixed-Criticality Provisioning. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–12, April 2016. doi:10.1109/RTAS.2016.7461323.

17 Angeliki Kritikakou, Christine Rochange, Madeleine Faugère, Claire Pagetti, Matthieu Roy, Sylvain Girbal, and Daniel Gracia Pérez. Distributed Run-time WCET Controller for Concurrent Critical Tasks in Mixed-critical Systems. In *Proceedings of the 22Nd International Conference on Real-Time Networks and Systems*, RTNS '14, pages 139:139–139:148, New York, NY, USA, 2014. ACM. doi:10.1145/2659787.2659799.

18 Lilium GmbH. Lilium Home Page. 2018. https://lilium.com/.

19 M. Maggio, H. Hoffmann, M. D. Santambrogio, A. Agarwal, and A. Leva. Power Optimization in Embedded Systems via Feedback Control of Resource Allocation. *IEEE Transactions on Control Systems Technology*, 21(1):239–246, Jan 2013. doi:10.1109/TCST.2011.2177499.

20 J. Nowotsch and M. Paulitsch. Leveraging multi-core computing architectures in avionics. In *2012 Ninth European Dependable Computing Conference*, pages 132–143, May 2012. doi:10.1109/EDCC.2012.27.

21 Jan Nowotsch, Michael Paulitsch, Daniel Buhler, Henrik Theiling, Simon Wegener, and Michael Schmidt. Multi-core interference-sensitive wcet analysis leveraging runtime resource capacity enforcement. In *ECRTS*, pages 109–118. IEEE Computer Society, 2014. doi:10.1109/ECRTS.2014.20.

22 NXP Semiconductors. *e500mc Core Reference Manual*, 2013. Rev. 3.

23 NXP Semiconductors. *P4080 QorIQ Multicore Communication Processor Reference Manual*. NXP Semiconductors, rev 2 edition, 2014.

24 D. R. Sahoo, S. Swaminathan, R. Al-Omari, M. V. Salapaka, G. Manimaran, and A. K. Somani. Feedback control for real-time scheduling. In *Proceedings of the 2002 American Control Conference (IEEE Cat. No.CH37301)*, volume 2, pages 1254–1259 vol.2, May 2002. doi:10.1109/ACC.2002.1023192.

25 Martin Schoeberl, Sahar Abbaspourseyedi, Alexander Jordan, Evangelia Kasapaki, Wolfgang Puffitsch, Jens Sparsø, Benny Akesson, Neil Audsley, Jamie Garside, Raffaele Capasso, Alessandro Tocchi, Kees Goossens, Sven Goossens, Yonghui Li, Scott Hansen, Reinhold Heckmann, Stefan Hepp, Benedikt Huber, Jens Knoop, Daniel Prokesch, Peter Puschner, André Rocha, and Cláudio Silva. T-crest: Time-predictable multi-core architecture for embedded systems. *Journal of Systems Architecture*, 61(9):449–471, 2015. doi:10.1016/j.sysarc.2015.04.002.

26 Lui Sha, Marco Caccamo, Renato Mancuso, Jung-Eun Kim, Man-Ki Yoon, Rodolfo Pellizzoni, Heechul Yun, Russel Kegley, Dennis Perlman, Greg Arundale, and Richard Bradford. Single Core Equivalent Virtual Machines for Hard Real-Time Computing on Multicore Processors. Technical report, 2014.

27 T. Ungerer, C. Bradatsch, M. Gerdes, F. Kluge, R. Jahr, J. Mische, J. Fernandes, P. G. Zaykov, Z. Petrov, B. Böddeker, S. Kehr, H. Regler, A. Hugl, C. Rochange, H. Ozaktas, H. Cassé, A. Bonenfant, P. Sainrat, I. Broster, N. Lay, D. George, E. Quiñones, M. Panic,

J. Abella, F. Cazorla, S. Uhrig, M. Rohde, and A. Pyka. parMERASA – Multi-core Execution of Parallelised Hard Real-Time Applications Supporting Analysability. In *2013 Euromicro Conference on Digital System Design*, pages 363–370, Sept 2013. doi:10.1109/DSD.2013.46.