Exploring and Understanding Multicore Interference from Observable Factors

Benjamin Lesage[1], David Griffin[1], Iain Bate[1] and Frank Soboczenski[1]

Abstract:

Multi-core processors bring a wide variety of challenges to the development, maintenance and certification of safety-critical systems. One of the key challenges is to understand how tasks sharing the processing resource affect one another, and to build an understanding of existing or new platforms. Industry reports that interference can lead to large variations in execution times which can lead to a wide variety of problems including timing overruns. To support performance improvements, debugging and timing analysis, a framework is presented in this paper for reliably establishing the interference patterns of tasks using simple contenders. These contenders systematically manipulate the shared resources so the effect on interferences can be understood and analysed. The approach relies on guided exploration of the interference space and existing performance monitoring infrastructure. It has been implemented on a Tricore AURIX platform to analyse the behaviour of multiple real and kernel applications.

Keywords: Inter-core interferences, timing analysis, Shared resources, measurement-based, performance monitoring

1 Introduction

The drive for performance in modern systems has to face an increasing energy wall; merely increasing clock speeds to achieve higher performance is no longer a viable solution. On the other hand, multicore platforms offer improved performances through the use of multiple processing units on the same chip. Tasks are running concurrently on different cores while sharing off-core resources such as lower levels of the memory hierarchy or external IO channels. The introduction of resources shared by concurrent tasks introduce new sources of interferences. Accesses to a shared resource may suffer arbitration delay when multiple cores compete for said resource. The behaviour of a request may also be altered due to interferences, e.g. the eviction of a cache block by a concurrent task.

Preemption-related interferences, similarly to inter-core interferences, exhibit both direct and delayed effects after the occurrence of an interference event. Inter-core interferences however do not occur as a single point of interference, as opposed to a preemption, but interleave with the normal execution of a task. During inter-core interference analysis, this requires to more pessimistic assumptions where each request is assumed to be in conflict with a request from a co-runner [Al15], or complex models of the underlying architecture requiring precise knowledge of its behaviour [JHH15]. Various mechanisms have been

[1] Department of Computer Science, University of York, York, United Kingdom

proposed instead to limit or preclude interferences in time and space between concurrent cores, such as cache partitioning [SM08] or leaky bucket schemes [Ji13]. Such schemes require often costly support from the underlying platform. To identify the solution adapted to a specific task, a necessary step is the building of an understanding of its sensitivity to interferences and the underlying factors.

Modern architectures exhibit performance monitoring counters (PMCs) as a tool to evaluate the behaviour of a task. PMCs provide an estimate of the occurrences of low-level events on the platform. The information they expose can be used in particular to evaluate how often and how much a task makes use of particular resource, private or shared [An97], with minimal knowledge on the platform. As an example, consider an architecture with a single private instruction cache on top of a shared SRAM memory. Without further knowledge of the platform, one can safely assume that a task with a large observed miss rate is more likely to access the shared memory and be sensitive to inter-core interferences. Some factors are thus more pertinent than others to the execution time variations of a task. Moreover, there are typically more events to be counted than registers available to track those events.

This paper proposes an approach to identify the main factors of variability in the temporal behaviour of a task. The analysis operates without knowledge of the underlying platform and the implemented policies for shared resources. The technique exposes the PMCs which are tied to variations in the execution time of a task. Through systematic and reproducible exploration of the interference space, this allows the isolation of the effect of interferences. The selection of most relevant factors provides feedback on the sources of interferences that need to be tackled to improve the predictability of a task's behaviour and the robustness a task or the system as a whole w.r.t. interferences. Focused testing on the factors identified by the analysis can further provide for a partial multi-variate model of the temporal behaviour of a task in relation to said factors. We provide general guidelines on how synthetic contenders can exercise inter-core interference in the analysed system.

2 Related Work

Current state of the art techniques for the analysis of the effects of inter-core interferences focus on either taking into account possible effects or precluding them. The work of Altmeyer and al. [Al15] belongs to the first category. The authors propose a generic framework to compute the response time of task, taking into account the delays that might rise from the arbitration of accesses to the shared bus. While focused on a single shared resources, the technique needs to take into account a wide range of effects, such as memory refreshes, cache hits and misses, to build a reasonable model of interleaved accesses. More integrated approaches [JHH15] further improve the precision of the estimates but require an extensive knowledge of the underlying platform or complex models capturing a large portion of the system state.

Approaches such as partitioning divide the shared resource space into segments dedicated to the sole benefit of a single task. They focus on precluding interferences at the expense of limiting the resources available to each task [Al14; SK11; SM08]. Leaky bucket schemes [Ji13]

offer an alternative solution to reduce interferences. Accesses to a shared resource by a task are budgeted to limit the amount of interferences they might generate in a given time interval. In either case, the interference problem becomes an optimisation one to derive the budget allocated to each task in the system, satisfy the system's constraints, and maximise the use of the shared resource. Such techniques should therefore be applied with care, and their underlying assumptions and impact empirically validated. As an example, considerable interferences might still occur in partitioned caches due to shared miss handling status registers [VYF16].

Radojković et al. [Ra12] evaluate the effect of inter-core interference from co-runners through empirical experiments. Their work demonstrates that shared resources can contribute to large variations in the temporal behaviour of a task. Their evaluation relies on resource stressing kernels, thus identifying the slowdown which may be induced by a specific component on the platform. We instead aim at identifying which components or combinations thereof contribute to variations in the behaviour of a specific task. This reduces the pessimism of the following analyses by focusing on the components known to impact the analysed task. Testing can then proceed by focusing on said components, e.g. allowing for the empiric validation of selected inter-core interference management methods. Our approach also relies on a wider exploration of the interference space as the worst-case scenarios do not stem from stressing a single resource.

3 Overview

Our approach to evaluate the impact of inter-core interferences on the execution of a task can be broken down into simple steps. Data is first collected by running the task of interest against selected competitor tasks while collecting data related to both execution time and as many performance counters as possible. All collected measurements capture the end-to-end behaviour of the analysed task; instrumentation primitives surround the analysed task. We then identify a set of representative factors to understand which factors drive variations in the execution of the analysed task, and whether or not they relate to inter- or intra-core effects.

Without prior knowledge of the usefulness of PMCs, it is necessary to build a small dataset with all PMCs in order to determine their usefulness. To this end in Section 6, Principal Components Analysis is used for an automatic feature selection phase, i.e. to find a set of PMCs which is capable of representing the variability of the data. The identification of the PMCs relevant to the analysed task can help direct later testing phases, to evaluate selected inter-core interferences management approaches or build a model of the analysed task. Further data collection phases and the exercised contenders can be focused on the factors known to contribute to variability in the behaviour of the analysed task. Without refining the selection of components exercised during analysis, more different types of contenders need to be considered to exercise all possible sources of interference. This is turn leads to more testing or less significant data available.

While the factor selection is platform-agnostic (§ 6), the available factors and their interpretation depend on the underlying system. The methods thus relies on platform-specific

instrumentation. Instrumentation and contenders are expected to respectively capture the available PMCs alongside timing information and exercise the different sources of variability in the platform. We discuss the requirements inherent to those steps in Section 5. To illustrate our approach, we focus in the following on an Infineon AURIX Tricore platform as presented in the next section.

4 Evaluation Platform

Our evaluation platform is composed of the OSEK/VDX compliant Erika Enterprise [En16] real-time operating system running on top of an Infineon AURIX Tricore TC27x [In14]. Figure 1 outlines the architecture of the AURIX platform. The AURIX cores have different capabilities and fulfil different roles in the system:

- Core 0: Error checking, Energy Efficient Tricore 1.6E core
- Core 1: Error checking, High Performance Tricore 1.6P core
- Core 2: No error checking, High Performance Tricore 1.6P core

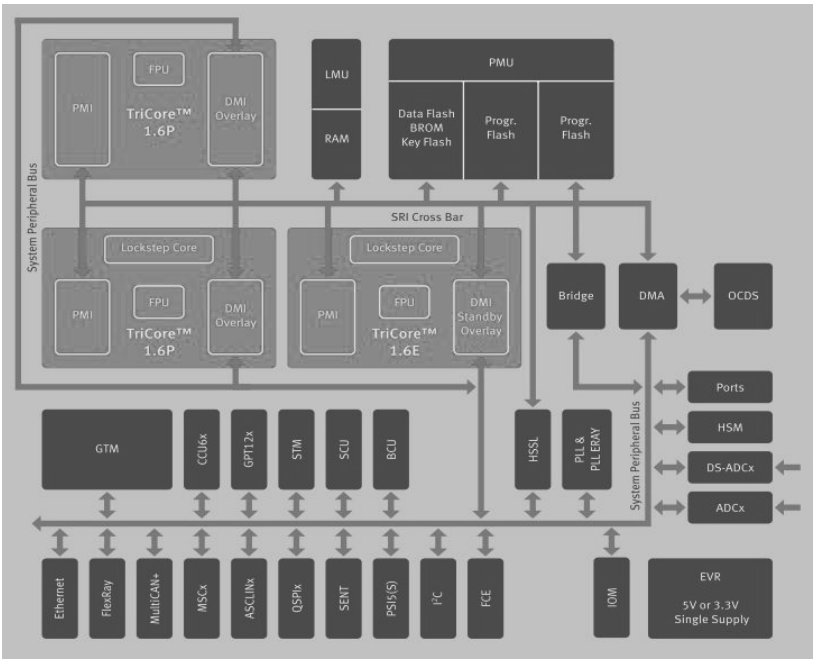

Fig. 1: Overview of the Infineon AURIX Tricore platform.

Each core has access to a crossbar which connects a SRAM unit, flash memories, and external peripherals through a bridge. Interference across cores typically stem from concurrent accesses to either of those resources, e.g. congestion on the flash due simultaneous

requests requiring arbitration. In the following, we focus on interferences caused by either the SRAM and one of the code ROM. The default memory mapping in Erika does not map data into the secondary ROM or segments located in remote scratchpads.

Core 1 and 2 expose 12 PMCs, 9 in the case of Core 0, which have the capability to monitor performance metrics such cache hits/misses, executed branches, or stalls in the pipeline. Only 3 registers are available per core to track PMCs, and each PMC can only be tracked by a specific register. This restricts the set of PMCs which can be monitored during a single run. For example, there is no configuration of said registers which allows tracking both data and instruction cache misses for Core 1.

In the best case, to capture data on all PMCs from the AURIX, it would be necessary to run each test four times. This is undesirable in that it increases the amount of testing that is required from the user to understand the impact of interferences on a task. Other platforms may expose more PMCs which renders this approach infeasible for large sets of tests[3]. Redundancy between factors or a lack of correlations to interferences further reinforces the need to reduce the set of collected PMCs to a significant one. Consider for example a computationally intensive program mapped into its core local scratchpads. Monitoring its cache hits will not yield conclusive results.

5 Data Collection

Our approach relies on task-level instrumentation to capture end-to-end timing traces. The collected traces include both the execution time of the analysed task and related metrics pertinent to the behaviour of the platform. We thus rely on the PMCs exposed by the hardware. PMCs expose counts for specific events or latencies suffered by a task, e.g. the number of executed branches acts as an observable proxy for the path executed during an observation. Increases in stalls suffered by the load/store unit can be indicative of increased contention in the shared memory. Our implementation automates the whole trace collection process, allowing for the automated collection of traces capturing all available PMCs on the platform or a selection thereof.

The PCA requires the presence for each run of all available factors, PMCs on the target platform, to establish their relation to the execution time of the analysed task. The analysed task is ran multiple times, under the same inputs, merging the results obtained for identical runs but different configurations of PMCs. While we aimed at improving the reproducibility of the framework between runs, the platform still exhibit some uncontrolled sources of variability, e.g. uninitialised values on processor start. Those prevent the perfect reproduction of the runs of a task. We validated that the error between reproduced runs is both minimal and characterised as random noise, by fixing a performance counter and comparing its value across identical runs and varying PMCs configurations. The error is minimal ($< 5\%$). The use of the Wald-Wolfowitz [St06] further confirmed that it could be reasonably characterised as random noise and would not introduce systemic failings [St06].

[3] For example, the P4080 platform [Se], which we have also applied the technique to, exposes approximately 128 PMCs with 4 registers per core, and would require each experiment to be repeated 32 times.

A dedicated instrumentation buffer is used to log the timing and PMCs values on each core. A full instrumentation buffer interrupts all execution on the platform and triggers the collection of the data through the debugger interface. The same debug interface is used to configure the PMCs exercised during a set of runs. A single binary and test vector can thus be used to collect different PMCs. Instrumentation buffers are mapped into a debug memory segment, itself mapped onto local scratchpads during analysis or unmapped on a deployed system. Writes to the unmapped debug segment are simply discarded by the platform. The event instrumentation routines can therefore be kept in the deployed system. Each request for an instrumentation point is broadcast to all cores in the system to capture PMCs across all cores.

5.1 Synthetic contenders

We developed a set of synthetic contenders to drive the exploration of the possible inter-core interference configurations. The contenders aim to exert the variability inherent to the analysed task in reaction to inter-task conflicts. Knowledge about the potential sources of variability in a system is required to exert them in a significant way. As such, contenders are strongly platform-dependent. While contenders for one platform may not apply to another, similar principles apply, e.g. varying accesses across cache lines, cache sets or physical pages, interleaving sequences of reads or writes, etc.. The use of the complete system as deployed would help understand the main sources of variability in the analysed task, but may not highlight the impact of contention should it suffer a constant interference rate.

A single set of platform-wide contenders has been derived for the AURIX. Controlled accesses to the shared memory segments, through non-cacheable addresses, are used to generate interferences. Those are restricted to the Shared RAM and a segment of flash. Given the default memory mapping implemented by the OSEK/VDX-compliant Erika OS, a core is restricted to either its local scratchpads, the shared memory, or one of the flash segments. While the method is not restricted to a specific platform, taking into consideration its underlying restrictions helps reducing the configurations that need to be considered during testing. Contending accesses interleave with non-interfering one; accesses by a core to its scratchpad do not contribute to the overall inter-core interferences. Each contender loops upon a determined access sequence to generate a controlled, continuous amount of contention. To preclude any impact on the functional behaviour of the analysed tasks and preserve data coherency, there is no sharing of data between contenders and analysed tasks.

The level of interference generated by a contender is expressed as and controlled by the portion of its instructions generating contention. Both the interference level and pattern exercised by a contender are set dynamically within user-defined bounds. An interference level is first randomly selected. Then a permutation of instructions is generated to select the conflicting ones. The code of each contender is stored in a local scratchpad such that it can be rewritten to enforce the selected interference pattern. The interference patterns exercised by a contender during an experiment can easily be reproduced.

Contenders run on all cores effectively acting as an idle task in the system. On each core a preemptive task with the lowest priority runs the main loop of a contender. Variation in the

observed interference patterns relies on periodic reconfiguration of the contenders, upon a signal from the analysed task. The process can proceed without explicit synchronisation between contenders and analysed task. This signal is triggered at the end of the periodic analysed task allowing for the reconfiguration to occur between activations of the task. Like the instrumentation routines, the primitives can thus be kept into the deployed system.

6 Feature Selection

Due to the impracticality of capturing vast amounts of data for each and every PMC, as well as a desire to focus on high-quality PMCs, it is necessary to reduce the number of observed PMCs[4]. The goal for this step is to identify the PMCs which are correlated, and then select a set of representative PMCs which can be captured in a single configuration while still describing the majority of the data. While it is inevitable that some detail in the data will be lost at this stage, the reduction in the amount of effort required to get a single data point enables more data to be collected.

In order to accomplish this, we use the technique of PCA [Jo02]. PCA is a technique which identifies correlations within a dataset by finding the *Principal Components* (PCs) of the data. Each PC describes one of the main axes of variance in the analysed dataset such that variations on each axis can be attributed to a specific set of factors. Correlated factors are thus captured as part of the same PC. For example, the main axis of variation on a data-sensitive application, its main PC, will include factors such as hits in the cache or stalls in the memory units. This is further illustrated in Figure 2, which shows the main axes of variations in a 2-dimensional dataset; PC1 captures the axis along which the majority of the variance in the data occurs.

Additional metrics are attached to the PCs and each factor inside a PC to measure their respective impact on the whole dataset and the PC. The PCs specify a loading on each factor that indicates the weighting that must be assigned to its observations such that they lie on the axis defined by the PC. In other words, the loading of a factor on a PC captures its correlation to the PC, how it evolves alongside the axis defined by the PC. A loading of near 0 indicates that the observations are not correlated on the PC, whereas loadings of 1 and -1 indicate perfect positive and negative correlation respectively, i.e. a factor which values respectively increase or decrease with values along the PC. Considering the same example of a memory intensive task, accesses to the bus from the task or contenders are likely to increase its execution time, thus being positively correlated to the principal components. Conversely, a decrease in executed integer instructions may be correlated to an increase in the memory traffic, and the task's execution time. In the Figure 2 example, x has a high loading in PC1, indicating a high degree of correlation to PC1, but a much lower loading on PC2.

PCs themselves have an overall magnitude assigned to them which can be seen as a proxy for their contribution to the variations in the dataset, i.e. the amount of variance in the dataset captured upon the axis of the PC. The right-hand side of Figure 2 shows how it is

[4] In statistical literature, this is commonly referred to as *dimensionality reduction* or *feature selection*.

possible to reduce the 2-dimensional dataset to a single dimension on which accounts for the majority of the variability.

A standard use of PCA is to identify which PCs do not significantly contribute to the overall distribution of a dataset using their respective loadings. For example, if a PC accounts for less than 10% of the variance in the entire dataset, as accounted for by its loading, then variation along that PC can be simply dismissed as sampling error and the PC ignored; this quickly and simply reduces the number of dimensions in the analysed data set. However, in this application it may be necessary to reduce the number of factors further, and select only the high quality PMCs which yield information on the observed interferences multiplier. Hence it is necessary to filter the PMCs further, such that only the highest quality PMCs are used.

The first step to finding the highest quality PMCs is to remove all PCs which are not correlated to the execution time of the task under analysis, as these are unlikely to yield useful information. This relies on the absolute loading of the analysed task's execution time on the PC to measure their correlation. In addition, components which explain a low amount of the overall variance are removed, as these represent factors which are unlikely to have a high impact on the result. The remaining components are weighted by the amount of variance they explain, through their respective loadings, and then the PMCs with the highest degree of correlation to these components are selected. For example, if 2 components A, B remain, of which A explains 60% of the variance and B 30%, then for each PMC that is selected correlated to B, two PMCs will be selected that correlate to A. The exact PMCs selected will be those with the highest correlation to the components A and B respectively.

It is also possible to add additional constraints to factor selection. Depending on the platform, and the amount of effort a user is prepared to undertake when gathering data, it may be desirable to place a restriction on the number of runs required to gather the data. This may not always be the case, or the user may not be able to expend the additional effort to capture them. Therefore, if the user wishes to impose additional constraints to reduce the burden of data gathering, these constraints should be formulated and applied at this stage. This is implemented by using a Integer Linear Programming (ILP) solver [Gu15] to perform the maximisation step, and so any ILP constraints can be used.

A pseudocode implementation of the application of PCA in our approach to select n relevant factors is given in Algorithm 1. Once PCA has identified which PMCs must be collected, the main data collection process can now take place and is only required to record values for these PMCs, which reduces the burden of instrumentation.

Quality of the selected factors and contenders

One issue that may be encountered during feature selection is the selection of poor quality factors. This can happen if user constraints prevent high quality factors from being selected, or high quality features simply don't exist. If this is the case then the outcome of the algorithm may be a limited number of factors ($< n$) or factors with a low correlation to the execution time of the analysed task.

```
Function GetBestPMCs(dataset, n)
    P ← PCs of dataset given by PCA
    discard any c ∈ P not correlated with execution time
    discard any c ∈ P not accounting for a substantial amount of variance (e.g. < 10%)
    compute relative weighting of remaining c ∈ P
    select n PMCs maximising the sum of loadings subject to the weightings (and additional user
      constraints) by ILP solver
    return PMCs
end
```

Algorithm 1: Pseudo-code implementation of PCA as used to extract the best possible PMCs for instrumentation

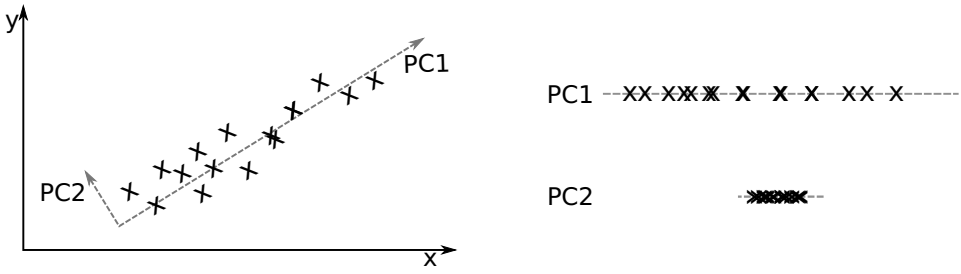

Fig. 2: Graphical Example of PCA

The use case for the analysis also drives requirements on the exercised contenders during the construction of the analysed dataset. If the observations focus on a subset of shared resources of interest, the selected factors then offers a relative classification of those resources which contribute the most to variations in the analysed task. Similarly for a conclusive feature selection, it is important for a contender to exercise various sources of interferences; focus on a single of the shared resources of interest, e.g. the shared memory, may lead to orthogonal variations of available PMCs hindering the analysis process.

Variability within the analysed dataset is an important factor to discriminate the PMCs that correlate to the behaviour of the analysed task. The exercised contenders should aim not only at generating worst-case interference scenarios but also produce a wide gamut of scenarios. Our feature selection focuses on factors correlated to the execution time of the analysed task. Therefore, variation on the inputs, executed paths, and observed scenarios is only captured by the analysis if it has a noticeable impact on the analysed task's temporal behaviour.

7 Evaluation

We evaluated our approach on various benchmarks deployed on the Erika OS running atop the AURIX platform, as described in Section 4. Two sets of benchmarks were investigated: simple examples from the Taclebench suite [Co] and three real-world applications. The

collection of end-to-end timing and PMCs values relies on the instrumentation of the analysed task, i.e. the insertion of a call to the instrumentation routine before and after calls to the analysed task. We use the Rapita Verification Suite [Ra] to that purpose. All benchmarks are set to run in a single periodic task concurrently with our synthetic contenders. Input vectors are either provided as part of the benchmark [Co] or randomly generated during our experiments. Similarly, explored interference levels and patterns are randomly generated for each execution before each execution of the analysed task. To enforce the occurrences of inter-core interferences, portions of the data manipulated by some benchmarks have been mapped into the shared memory.

7.1 Understanding the sources of execution time variability

We present the factors identified as relevant to the variability of a selection of benchmarks in Table 1. The temporal variability for a benchmark is captured by the ratio between the maximum and minimum observed execution time during our experiments (in the last column). The real-world benchmarks investigated were *missile-c*, a missile control program converted from Ada to C [Hi], and the *powerwindow* and *lift* benchmarks from the Taclebench [Co] suite. ALU, LSU, LU stalls represents stalls in the Arithmetic, Load/Store and Loop unit respectively. Each factor is prefixed by its measuring core.

The largest ratios between maximum and minimum execution times are observed as an example for *anagram*, *binarysearch*, *dijkstra*, and *missile_c*. Our analysis selects in such cases factors such as the number of executed branches, multiple issues, or stalls in the ALU, as most relevant to the observed variations. This suggests that variability in these benchmarks stems first and foremost from the observations capturing different execution paths; variations in the execution time of such tasks is not driven by the variations in inter-core interferences but changes in executed paths and input vectors. The analysis of the impact of inter-core interferences on those benchmarks should therefore distinguish between different execution scenarios. Considering *dijkstra* as an example, the path searching algorithm includes a short, special case if the source and target nodes are the same.

The identification of factors from other cores as relevant, e.g. C2 Data Memory Stalls for *compressdata* or *matmult*, is an indicator of the sensitivity of an application to variations in the behaviour of concurrent tasks as triggered by our contenders. The strong contribution of factors such as memory stalls further identifies those tasks as data intensive applications. On *compressdata*, variations due to contentious data accesses further trumps those due to small variations in the execution path. Further testing should therefore focus on contenders exercising the shared memory to exploit variability in these benchmarks.

The results of the application of the analysis to different benchmarks and cores also points towards an asymmetry on the platform. Tasks running on Core 1 are more likely to be impacted by contenders on Core 2 than on Core 0; factors from Core 2 are more often selected on their own for benchmarks running on Core 1 than factors from Core 0. This may stem from lower contention levels from the energy efficient Core 0, or asymmetry in the arbitration of accesses to the shared resources.

Tab. 1: Representative factors identified for each benchmark.

Benchmark	Core	Select Factors		Max/Min runtime ratio
MÄLARDALEN				
adpcm_encoder	C0	C0 Data Memory Stalls	C0 ALU Stalls	1.00276
		C0 Executed branches	C2 LSU Stalls	
adpcm_encoder	C1	C1 Data Memory Stalls	C1 Executed branches	1.00412
		C2 Data Memory Stalls	C2 ALU Stalls	
anagram	C1	C1 Data Memory Stalls	C1 LSU Stalls	3.4214
		C1 Multiple instructions issue	C1 Executed branches	
binarysearch	C0	C0 Data Memory Stalls	C0 ALU Stalls	1.85784
		C0 Executed branches	C2 Data Memory Stalls	
binarysearch	C1	C0 Data Memory Stalls	C1 Data Memory Stalls	1.9
		C1 ALU Stalls	C1 Executed branches	
bitcount	C0	C0 Data Memory Stalls	C0 ALU Stalls	1.2179
		C0 Executed branches	C1 LSU Stalls	
bitcount	C1	C1 ALU Stalls	C1 LSU Stalls	1.22467
		C1 Executed branches	C2 LSU Stalls	
codecs_dcodrle1	C0	C0 Data Memory Stalls	C0 LSU Stalls	1.43177
		C0 Executed branches	C2 Data Memory Stalls	
codecs_dcodrle1	C1	C1 Data Memory Stalls	C1 LSU Stalls	1.53956
		C1 Multiple instructions issue	C1 Executed branches	
compressdata	C0	C0 Data Memory Stalls	C0 Executed branches	1.55328
		C1 Data Memory Stalls	C2 Executed branches	
compressdata	C1	C0 Data Memory Stalls	C1 Data Memory Stalls	1.63359
		C1 Executed branches	C2 Executed branches	
countnegative	C0	C0 Executed branches	C1 Data Memory Stalls	1.25762
		C1 LSU Stalls	C2 Data Memory Stalls	
countnegative	C1	C1 LSU Stalls	C1 Multiple instructions issue	1.06209
		C1 Executed branches	C2 LSU Stalls	
dijsktra	C0	C0 ALU Stalls	C0 Executed branches	1577.23
		C1 LSU Stalls	C2 LSU Stalls	
dijsktra	C1	C1 Data Memory Stalls	C1 LSU Stalls	147.27
		C1 Multiple instructions issue	C1 Executed branches	
duff	C0	C0 Data Memory Stalls	C0 ALU Stalls	1.51672
		C1 Data Memory Stalls	C2 Data Memory Stalls	
duff	C1	C0 Data Memory Stalls	C1 Data Memory Stalls	1.51595
		C1 ALU Stalls	C2 Data Memory Stalls	
matmult	C0	C0 Data Memory Stalls	C1 Data Memory Stalls	1.32437
		C2 Data Memory Stalls	C2 Multiple instructions issue	
matmult	C1	C1 Data Memory Stalls	C1 LSU Stalls	1.31893
		C2 Data Memory Stalls	C2 Multiple instructions issue	
ndes	C0	C0 Data Memory Stalls	C0 ALU Stalls	1.09869
		C0 LSU Stalls	C1 Data Memory Stalls	
ndes	C1	C1 Data Memory Stalls	C1 ALU Stalls	1.112
		C1 LSU Stalls	C2 Data Memory Stalls	
qurt	C0	C0 Data Memory Stalls	C0 ALU Stalls	1.35984
		C0 Executed branches	C2 Data Memory Stalls	
qurt	C1	C1 Data Memory Stalls	C1 ALU Stalls	1.33716
		C2 Data Memory Stalls	C2 Executed branches	
rijndael_encoder	C0	C0 Data Memory Stalls	C2 Data Memory Stalls	1.03466
		C2 Multiple instructions issue	C2 Executed branches	
rijndael_encoder	C1	C1 Data Memory Stalls	C1 LSU Stalls	1.0344
		C2 Data Memory Stalls	C2 Multiple instructions issue	
statemate	C0	C0 Data Memory Stalls	C0 LSU Stalls	1.00501
		C0 Executed branches	C2 LSU Stalls	
statemate	C1	C1 LSU Stalls	C1 Multiple instructions issue	1.00509
		C1 Executed branches	C2 LSU Stalls	
st	C0	C0 Data Memory Stalls	C0 ALU Stalls	1.10449
		C1 Data Memory Stalls	C1 Multiple instructions issue	
st	C1	C1 Data Memory Stalls	C2 Data Memory Stalls	1.13961
		C2 Multiple instructions issue	C2 Executed branches	
REAL-WORLD EXAMPLE				
lift	C0	C0 Data Memory Stalls	C0 ALU Stalls	1.46235
		C0 Executed branches	C2 Multiple instructions issue	
lift	C1	C1 ALU Stalls	C1 LU Stalls	
		C1 Multiple instructions issue	C1 Executed branches	1.48444
missile_c	C0	C0 Data Memory Stalls	C0 ALU Stalls	24.3645
		C0 LSU Stalls	C0 Executed branches	

7.2 Exploring the impact of inter-core interferences

We focus in the following on the *matmult* benchmark running on Core 0. The application comprises a single path computing the multiplication of two matrices mapped into the main memory. Variability in the temporal behaviour of the benchmark is thus mostly related to inter-core interferences and accesses to the shared main memory, as captured by our method in Table 1. Using our framework, we collect observations under different configurations of interference levels, i.e. the portion of accesses to the shared memory, from each core. The collected execution times are then normalised over the execution time for *matmult* in isolation, without contenders. The median of the observed execution times for each inte rference level are presented in Figure 3.

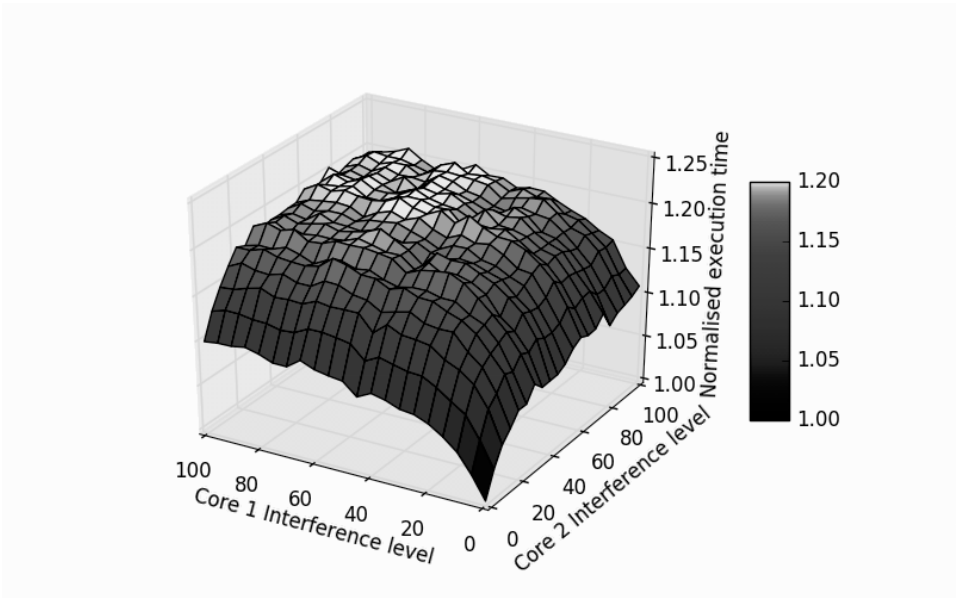

Fig. 3: Normalised execution time for *matmult* under inter-core interferences

As expected, the execution time of the *matmult* benchmark increases alongside the interferences generated by the synthetic contenders. This is however not a strictly increasing curve; high execution times are more likely to be observed when about 80% of Core 1 memory accesses hit the shared memory, and at least 60% of Core 2 do. This reinforces the observation that the arbitration policy on the AURIX shared memory may be asymmetric.

Furthermore, maximising the interferences generated by other cores does not guarantee maximising the impact on the analysed task as contending cores start interfering with themselves, restricting their maximum bandwidth to the main memory, and each other. Similarly, focusing solely on the impact of a single source of interferences at a time, e.g. Core 1, does not lead to maximised observed execution time. To understand the impact of interferences, we advocate the need to explore a wide variety of configurations, in terms of types of interferences but also strength of those interferences.

8 Conclusion

This paper introduces a feature selection approach to understand the main source of variability in an application. Our approach draws the relation between the temporal behaviour of an application and the observable factors on the platforms through the performance monitoring infrastructure. We focus on the impact of inter-core interferences stemming from the use of shared resources by concurrent tasks. To exercise a sufficient level of variability, we rely on synthetic, configurable contenders to exercise different interference patterns, sources, and levels.

We implemented our approach on the OSEK/VDX-compliant Erika OS running atop the Tricore AURIX TC277x platforms. Our framework allows the joint collection of timing and PMCs information, under user-controlled interference ranges. The evaluation demonstrates that the method is able to classify the main sources of variability in different categories of applications, from control code to more data-centric kernels. Using such a simple kernel, we further illustrated the importance of variability in the test conditions to highlight variability and the worst-case configurations in the analysed task.

We evaluated our process on other platforms, such as the Freescale P4080, and plan to expand to other architectures or sources of interferences. Our approach requires only minimal knowledge of the underlying platform. Namely, potential sources of interferences need to be identified and contenders designed to exercise them. Work is further required to interpret the meaning behind the PMCs selected by the analysis. However, similar design principles still apply across platforms, such as varying access patterns, data-centric kernels, etc., and similar performance monitoring infrastructure are available.

Acknowledgments

This work was partially funded by EU FP7 IP PROXIMA (611085), and the UK EPSRC Project MCCps (EP/P003664/1). EPSRC Research Data Management: No new primary data was created during this study.

References

[Al14] Altmeyer, S.; Douma, R.; Lunniss, W.; Davis, R. I.: Evaluation of Cache Partitioning for Hard Real-Time Systems. In: 2014 26th Euromicro Conference on Real-Time Systems (ECRTS). July 2014.

[Al15] Altmeyer, S.; Davis, R. I.; Indrusiak, L.; Maiza, C.; Nelis, V.; Reineke, J.: A Generic and Compositional Framework for Multicore Response Time Analysis. In: Proceedings of the 23rd International Conference on Real Time and Networks Systems. RTNS, 2015.

[An97] Anderson, J. M.; Berc, L. M.; Dean, J.; Ghemawat, S.; Henzinger, M. R.; Leung, S.-T. A.; Sites, R. L.; Vandevoorde, M. T.; Waldspurger, C. A.; Weihl, W. E.: Continuous Profiling: Where Have All the Cycles Gone? ACM Trans. Comput. Syst. 15/4, Nov. 1997.

[Co] Contributors: Taclebench Benchmark Suite.

[En16] Enterprise, E.: ERIKA Enterprise | Open source RTOS Osek/VDX Kernel, 2016, URL: http://erika.tuxfamily.org/drupal/.

[Gu15] Gurobi Optimization, I.: Gurobi Optimizer Reference Manual, 2015, URL: http://www.gurobi.com.

[Hi] Hilton, A.: SPARK Missile Guidance Simulator.

[In14] Infineon: Aurix (TM) Family TC27xT Documentation, 2014, URL: www.infineon.com/aurix.

[JHH15] Jacobs, M.; Hahn, S.; Hack, S.: WCET Analysis for Multi-core Processors with Shared Buses and Event-driven Bus Arbitration. In: Proceedings of the 23rd International Conference on Real Time and Networks Systems. RTNS, ACM, New York, NY, USA, 2015.

[Ji13] Jing, W.: Performance Isolation for Mixed Criticality Real-time System on Multicore with Xen Hypervisor, MA thesis, Uppsala University, Department of Information Technology, 2013.

[Jo02] Jolliffe, I.: Principal component analysis. Wiley Online Library, 2002.

[Ra] Rapita Systems: Rapita Verification Suite, https://www.rapitasystems.com/.

[Ra12] Radojković, P.; Girbal, S.; Grasset, A.; Quiñones, E.; Yehia, S.; Cazorla, F. J.: On the Evaluation of the Impact of Shared Resources in Multithreaded COTS Processors in Time-critical Environments. ACM Transactions on Architecture and Code Optimization 8/4, 34:1–34:25, Jan. 2012.

[Se] Semiconductor, F.: EREF: A Programmer's Reference Manual for Freescale Embedded processors.

[SK11] Sanchez, D.; Kozyrakis, C.: Vantage: scalable and efficient fine-grain cache partitioning. In: SIGARCH Computer Architecture News. Vol. 39. 3, ACM, pp. 57–68, 2011.

[SM08] Suhendra, V.; Mitra, T.: Exploring locking: partitioning for predictable shared caches on multi-cores. In: Design Automation Conference (DAC). 45th ACM/IEEE. June 2008.

[St06] Stephens, L. J.: Schaum's Outlines: Beginning Statistics. McGraw-Hill, 2006.

[VYF16] Valsan, P. K.; Yun, H.; Farshchi, F.: Taming Non-Blocking Caches to Improve Isolation in Multicore Real-Time Systems. In: IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS). Apr. 2016.