CISTER
Research Center in
Real-Time & Embedded
Computing Systems

Conference Paper

The variability of application execution times on a multi-core platform

Vincent Nélis
Patrick Meumeu Yomsi
Luís Miguel Pinho

CISTER-TR-160608

The variability of application execution times on a multi-core platform

Vincent Nélis, Patrick Meumeu Yomsi, Luís Miguel Pinho

CISTER Research Center

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8321159

E-mail:

http://www.cister.isep.ipp.pt

Abstract

It is a known fact that processes running concurrently on different cores in a multicore environment interfere with each other on the processor shared resources. The contention on these shared resources considerably slows down the execution on every core since sometimes the cores must stall while their requests to access the resources are being served. But by how much the execution may be slowed down due to this interference? In this paper we answer this question with numbers coming from experimentation. That is, we quantify the magnitude of the impact of the interference on the execution time by running programs taken from the TACLeBench benchmark suite, a popular benchmark suite in the real-time research community, on the first generation of Kalray manycore processor family, the MPPA-256 (the development board) that goes by the code name “Andey”.

The variability of application execution times on a multi-core platform

Vincent Nélis, Patrick Meumeu Yomsi and Luís Miguel Pinho

CISTER/INESC-TEC, ISEP, Polytechnic Institute of Porto
{nelis, pamyo, lmp}@isep.ipp.pt

Abstract

It is a known fact that processes running concurrently on different cores in a multicore environment interfere with each other on the processor shared resources. The contention on these shared resources considerably slows down the execution on every core since sometimes the cores must stall while their requests to access the resources are being served. But by how much the execution may be slowed down due to this interference? In this paper we answer this question with numbers coming from experimentation. That is, we quantify the magnitude of the impact of the interference on the execution time by running programs taken from the TACLeBench benchmark suite, a popular benchmark suite in the real-time research community, on the first generation of Kalray manycore processor family, the MPPA-256 (the development board) that goes by the codename "Andey".

1998 ACM Subject Classification C.3 SPECIAL-PURPOSE AND APPLICATION-BASED SYSTEMS.

Keywords and phrases Execution time variability, timing analysis, WCET estimates, multi-cores, many-cores.

Digital Object Identifier 10.4230/OASIcs.xxx.yyy.p

1 The problem of inter-core interference

Determining the worst-case execution time (WCET) of a software application has always been a major problem in the design of real-time systems. Those WCET estimates are at the base of the whole stack of higher-level analyses defined to characterize the timing behaviour of the system and verify its timing requirements. Computing estimates that are as close as possible to the actual maximum execution time is crucial. Under-estimating the application execution times during the analysis phase may result in designing an over-utilized system that does not meet its timing requirements, whereas over-estimating them may result in an over-dimensioned (costlier) system of which the resources are under-utilized and thus wasted.

In multi-core architectures, the problem of finding WCET upper-bounds is further exacerbated by the high number of resources shared between the cores. In such platforms running processes may execute concurrently on different processor cores but any of their accesses to a memory (to fetch an instruction or data) traverses multiple layers of arbitration in which the request may contend with others, emitted by processes running on other cores. Contrasting with single-core architectures, on multi-cores the time during which a core stalls waiting for a memory request to be served is a significant component of the overall execution time of a program.

The research community has addressed the additional problem of estimating the time-penalty caused by inter-core interference in various ways. One methodology consists in estimating the worst-case interference that a program may incur at runtime and inflate the individual WCET upper-bounds accordingly. This preserves the original analysis flow used

16th International Workshop on Worst-Case Execution Time Analysis.
Editors: Billy Editor and Bill Editors; pp. 1–10

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

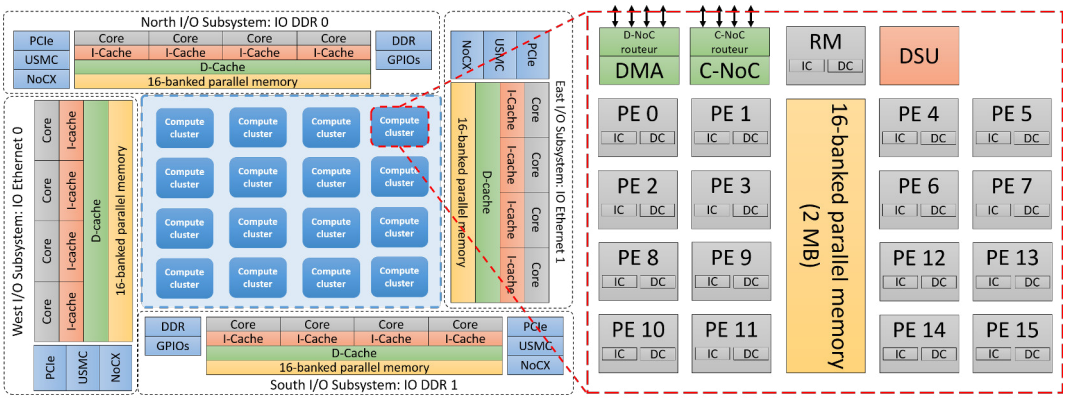

Figure 1 Outline of the architecture of the Kalray MPPA-256 (Andey) manycore processor.

in single-core systems, in which individual WCET are estimated for every software program and then fed as input into the higher-level schedulability analyses. Other works take into consideration that the interference between processes effectively depends on the scheduling decisions taken for these processes. In those works the estimated maximum interference is accounted for at the schedulability level. Other initiatives acknowledge that the interference between processes may be way too high on multi-cores and thus focus on how to temporally isolate the processes to nullify, or at least mitigate, the interference between them.

Related work and contribution: It is important to understand that the present work does not aim at measuring or determining the *worst-case* execution time of a given program. For that particular problem, the interested reader may consult [7] for an overview of the state-of-the-art solutions. Broadly speaking there are three main methodologies to estimate the worst-case execution time of a software program: static, measurement-based, and hybrid analyses, with their respective advantages and drawbacks discussed in [5]. More recently, the research community has been focusing on statistic methods as well (see [4] for an interesting starting point). In this work we performed a set of experiments to *measure and quantify the variability in the execution time of a program due to resource contention when run concurrently with other programs on a multi-core processor*. That is, we like to point out the magnitude of the time-overhead due to the interference on the processor shared resources. To do so, we have run a collection of programs taken from various application benchmark suites on the Kalray MPPA-256 development board. We have run the selected programs numerous times, always over the same set of input data, and varied the execution conditions.

2 Our test-case platform and applications

Platform settings: Our experiments have been carried out on the first generation of Kalray manycore processor family, the MPPA-256 that goes by the codename “Andey” [3] (we used the development board). It is a clustered many-core platform composed by an array of 16 clusters and 4 I/O subsystems, themselves connected by two NoCs (illustrated in Fig. 1). Cores are grouped in clusters connected by a Round-Robin (RR) arbitrated Network-on-Chip (NoC) in a 2D-torus topology [2]. Each cluster contains 17 identical VLIW cores: 16 *compute cores* that are dedicated to general-purpose computations and one *Resource Manager* (RM) core whose responsibility is to manage the processor resources on the behalf of the entire cluster (maps and schedules threads on compute cores) and organize the communication between its cluster and the other clusters, as well as with the main off-chip memory. Every

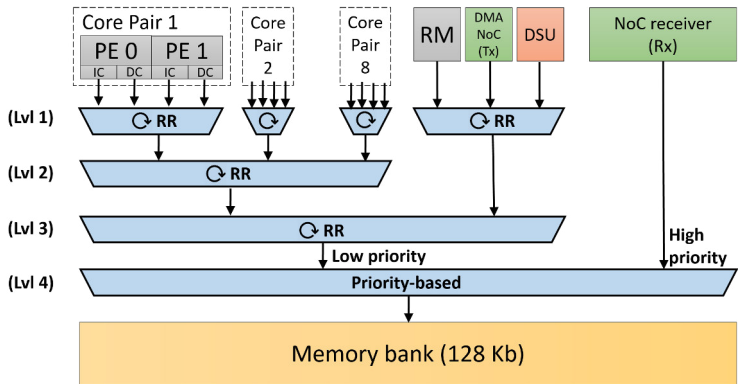

Figure 2 Memory request arbitration of the Kalray MPPA-256.

compute core has a private instruction and data cache. All the cores are fully timing compositional [6] in the sense that they do not exhibit timing anomalies. Additionally each cluster also contains a Debug Support Unit, a network interface for receiving data requests from the Data-NoC and a DMA engine used for data transmission over the Data-NOC.

Regarding the organization of the memory subsystem, each cluster has a local shared memory comprising 16 banks, each with a capacity of 128 KB, for a total memory capacity of 2 MB per cluster. The memory banks are organized into two groups of equal sizes, referred to as the banks on the *left* and *right* sides of the memory, respectively. Similarly, the 16 compute cores are grouped into 8 *core-pairs*[1]. Although the cluster address space can be divided among banks in an interleaved fashion (useful for high-performance and parallel applications), this work uses the blocked memory mode where the address space is divided in a sequential manner. This results in more predictable system behavior, which is a desired characteristic in systems subject to timing requirements.

The arbitration of memory requests is performed in four levels (stages), as depicted in Fig. 2. The first three levels use the Round-Robin arbitration scheme. The first level arbitrates memory requests issued from the two data caches and instruction caches of each core-pair. At the second level, the requests issued from each core-pair compete against the requests coming from the other core-pairs. At the third level, requests from all core-pairs compete against requests from the RM, the DSU, and the DMA. Finally, at the fourth level, the scheduled requests compete with those coming from the D-NOC (Rx) under static-priority arbitration, where requests from the NoC always have a higher priority. Note: in order to minimize contention, the second, third, and fourth levels of arbitration are replicated for each memory bank. The first level is only duplicated for memory banks located on the left and the right side of the memory, respectively, so that paired cores can access banks on different sides in parallel without interfering with each other. From the organization of the memory and its four-stages arbitration mechanism, it is easy to discern that there can be substantial interferences arising from different sources, e.g., from the concurrent accesses to memory banks from different cores or the contention on the NoC for the access to off-chip memory.

Application test-cases: We measure the execution time of a set of programs taken

[1] This organization in core-pairs is specific to the "Andey". The second generation of Kalray's manycore processor family, the MPPA2 high-speed I/O processor (codename "Bostan") dropped this architectural choice.

Name	Description	LoC	Origin
ammunition	C compiler arithmetic stress test	2508	misc
cjpeg_jpeg6b_transupp	JPEG image transcoding routines	1599	MediaBench
cjpeg_jpeg6b_wrbmp	JPEG image bitmap writing code	1296	MediaBench
dijkstra	All pairs shortest path	227	MiBench
epic	Efficient pyramid image coder	994	MediaBench
gsm_decode	GSM 06.10 provisional standard decoder	1368	MediaBench
gsm_encode	GSM 06.10 provisional standard encoder	1940	MediaBench
h264dec_ldecode_block	H.264 block decoding functions	1574	MediaBench
mpeg2	MPEG2 motion estimation	1533	MediaBench
ndes	Complex embedded code	407	MRTC
rijndael_decoder	Rijndael AES decryption	3043	MiBench
rijndael_encoder	Rijndael AES encryption	1024	MiBench
statemate	Statechart simulation of a car window lift control	1053	MRTC

Table 1 The 13 benchmark programs for which we measured the execution time. "LoC" is the total number of lines of code of all source code files belonging to a benchmark, empty lines and comments are not counted. This information as been taken as is from [1].

from the TACLeBench benchmark suite (from those labelled as "sequential benchmarks"). TACLeBench provides a freely available and comprehensive benchmark suite for timing analysis, featuring complex multi-core benchmarks [1]. We selected 13 programs out of the 102 programs available in the TACLeBench suite (see Table 1). Those programs are provided as ANSI-C 99 source codes that are 100% self-contained, i.e. no dependencies to system-specific header files via "#include" directives (eventually used functions from math libraries are also provided in C source code) [1].

3 Our approach to measuring the execution times

Our timing analysis methodology is based on the intuitive idea that the total execution time of any piece of code, e.g. a basic block, a software function, or an entire application, can be seen as composed of two main terms: the "intrinsic" time spent executing every instruction of the code and the time spent waiting for a shared software or hardware resource to become available. It is fundamental to clearly understand the difference between these two components.

The maximum intrinsic execution time (MIET): For a given set of input data, it is the time that the program takes to produce *the* corresponding output[2], *assuming that all software and hardware services provided by the execution environment and shared among different cores are always available* (the core running that program never stalls waiting for one of these resources to become available). That is, the intrinsic execution time of a program is its execution time when it runs in isolation, i.e. with no interference whatsoever with

[2] We assume that there is no functional random behaviours involved in the definition of the analysed program. That is, the outcome of evaluating a condition is never the result of an operation involving randomly-generated numbers. Under this assumption of not involving randomness in the control flow of the program, running it multiple times over a same set of input data always results in taking the same path throughout the program's code and thus execute the exact same sequence of instructions and eventually, it always produces the same output.

the rest of the system on the shared resources. Note that it does not mean that the code and data of the program are preloaded in the caches before execution, rather it means that wherever the information is stored, there will be no interference when fetching it.

On an "ideal" hardware architecture every instruction should take a constant number of cycles to execute (i.e. there is no time variation what-so-ever) and thus running the same program in isolation over the same set of input arguments always results in the exact same execution time. Although this may sound like a very strong assumption to make in practice, we will see that on a platform such as the Kalray MPPA-256 this assumption is reasonable. By running a preliminary set of tests with the same program an arbitrary number of times over the same inputs, we experienced a variation of its execution time of typically less than 0.1% of the maximum observed.

The maximum extrinsic execution time (MEET): For a given set of input data, it is the time that the program takes to produce its output *assuming a maximum interference on all the shared resources*. That is, the extrinsic execution time of a function is its execution time assuming that all the software and hardware services provided by the execution environment and shared among the cores are constantly saturated by requests from other system components. As we will see, contrary to the intrinsic execution time, the extrinsic execution time is generally subject to substantial variations due to the high number of processor resources shared amongst software functions.

3.1 Extraction of the MIET: the isolation mode

In order to extract the MIET of an application, the platform is configured in what we call the "isolation mode": the entire application is assigned to a single thread that is pinned to a core and all the other cores are shut down and kept idle. This is done to minimize the interference with the rest of the system. To enforce this mode of execution, we have implemented a platform-specific API for the Kalray MPPA-256. This API provides a set of functions and global parameters to perform the following tasks:

- Enforce that the thread running the analyzed program is executed, uninterruptedly, on a single core,
- Synchronise the IO cores and the cluster cores so that it is guaranteed that nothing runs in the background that could interfere with the execution of the analysed program, and
- Perform additional operations on the demand of the user to [re]configure the cluster before processing. Specifically, any of the following actions, or a combination of them can be performed: (a) Activate/deactivate the data cache; (b) Invalidate the data cache; (c) Flush the data cache; (d) Change the operating mode of the data cache (i.e. make the cores stall on access or not); (e) Invalidate the Data-TLB; (f) Activate the instruction cache; (g) Invalidate the instruction cache; (h) Set the address mapping scheme of the 2MB shared memory in each cluster (i.e., set the address mapping scheme to "interleaved" or "sequential" mode).

With these configuration options, we define two different configurations of the platform in order to give a hint of the type of results that can be expected from applications with very different memory access profiles:

- **The High-Performance (HP) configuration.** In this configuration, the data cache is enabled; The content of the data cache is neither invalidated nor flushed before each execution; The "stall-on-access" mode is disabled (that is, the core does not stall while waiting for a data to be fetched); The content of the data TLB is not invalidated before each execution; The instruction cache is enabled; The content of the instruction cache is

not invalidated before each execution; and the 2MB shared memory of the cluster is set in "interleaved" mode, to allow data to span several banks.

- **The Low-performance (LP) Configuration.** In this configuration, the data cache is disabled; The "stall-on-access" mode is enabled; The instruction cache is disabled; and the 2MB shared memory of the cluster is set in "sequential" mode, to allow data to span multiple adjacent banks if and only if it does not fit in the bank currently in use.

Clearly, using the LP or HP configuration has a substantial impact on the execution time of the application. The reason for defining these two platform configurations is not to assess the level of performance that is achievable in general on the Kalray-MPPA 256. Rather, we want to give a hint at the type of results that can be expected from applications with very different memory access profiles. To understand this relation between the platform settings and the memory access pattern of the application, it is important to understand that if the memory footprint of the analyzed program (instruction + data) is small enough, it will fit entirely in the private cache of the core on which the application is run. Therefore, when executing the program using the HP configuration, the instructions and data will be loaded once in the private cache and the program will not need to communicate further with the shared memory. That is, running a program with a small memory footprint using the HP configuration is equivalent to running a program with very limited communication with the shared memory. As it will be seen, when using the HP configuration the execution time of some of the benchmark programs used in our experiments is not, or almost not, affected by the execution of other programs running concurrently on other cores. On the contrary, when the LP configuration is used, since the caches are disabled, the analyzed program must frequently communicate with the shared memory during its execution. As a result, it is way more subject to interference with other programs running concurrently on the other cores and accessing the shared memory as well.

3.2 Extraction of the MEET: the contention mode

In order to extract the MEET of each application, the platform is configured in what we call the "contention mode". In this mode, we start each application and try to interfere as much as possible with its execution while it is running. The objective of the contention mode is to create the "worst" execution conditions for the application so that its execution is constantly suspended due to interference with other programs. This gives us an estimation of the maximum execution time of the application when it suffers maximum interference from other programs on the shared resources.

The contention mode is similar to the isolation mode in that the analyzed program is assigned to a single thread that is pinned to a core (here, core 0 of cluster 0). However, on the contrary to the isolation mode that shuts down all the other cores of the cluster (thereby nullifying all possible interference within that cluster), we deploy onto all these other cores small programs that we call "Interference Generators" (or IG for short). Those programs are essentially tiny pieces of code that have for sole purpose to saturate all the resources (e.g., interconnection, memory banks) that are shared with the application under analysis running on core 0. Remember that the objective of the contention mode is to create the worst execution conditions for the execution of the analyzed program, i.e. conditions in which its execution is slowed down as much as possible due to contention for shared resources. The IGs are deployed and started *before* the analyzed program starts executing, and they are stopped *after* it has run for a pre-defined number of times.

Implementing the IG that generates the worst possible interference that an application could ever suffer is a very challenging task, if not impossible. This is because the exact

behaviour of the application to be interfered with (i.e. its utilization pattern of every shared resources and the exact time-instants of accessing it) should be known, as well as all the detailed specifications of the platform. Besides, even if those information were known, the execution scenario causing the maximum interference may be impossible to reproduce. Rather than concentrating our efforts on creating such a worst IG, we opted for the implementation of an IG that is "bad enough" and used it as a proof of concept to demonstrate how large can be the time-overhead incurred by the application under analysis due to the interference.

Our implementation of the IG consists of a single function "IG_main()" that is executed by a thread dispatched to every core on which the analyzed program is not assigned (recall that the application under analysis is executed sequentially on core 0). That is, every core that is not running the analyzed program runs a thread that executes IG_main(). Essentially, IG_main() executes three functions, namely: IG_init_interference_process(), IG_generate_interference(), and IG_exit_interference_process(). The first one is called upon deployment, at the beginning of execution of IG_main(), before the analyzed program start to execute and be timed. The second one is the main function. It creates interference on the shared resources. The call to that function is encapsulated in a loop that terminates only when the IG is explicitly told to stop. Finally, the third function is called when the analyzed application has been timed and the analysis process is about to end.

Let us now briefly describe our implementation of these three functions on the Kalray MPPA-256. We use a global array of integer called "my_array" and declare the three main functions described above as follows.

```
int* my_array;
inline void IG_init_interference_process() __attribute__((always_inline));
inline void IG_generate_interference() __attribute__((always_inline));
inline void IG_exit_interference_process() __attribute__((always_inline));
```

The first function "IG_init_interference_process()" simply allocates memory space to "my_array" (the size of 1024 integers) and fills that array with arbitrary values. Note that on the Kalray MPPA-256, a thousand integers occupy roughly half of the private data cache of a VLIW core in a compute cluster. The third function "IG_exit_interference_process()" simply frees the memory space held by "my_array". The second function, "IG_generate_interference()", is the main one and a snippet of its code is presented below.

```
inline void IG_generate_interference() {
   __builtin_k1_dinval();
   __builtin_k1_iinval();
   register int *p = my_array;
   volatile register int var_read;
   var_read = __builtin_k1_lwu(p[0]);
   var_read = __builtin_k1_lwu(p[8]);
   var_read = __builtin_k1_lwu(p[16]);
   // ...
   var_read = __builtin_k1_lwu(p[1015]);
   var_read = __builtin_k1_lwu(p[1023]);
}
```

The function starts by invalidating the content of the data and instruction caches. Then, it reads every element of "my_array", starting from the element K=0 and moving on iteratively from element K to element ((K+8) modulo 1024), until K reaches 1023. This way, every element of the array is read exactly once and every two consecutive readings access data that

	ISO			CON			
Name	**min**	**max**	**var (%)**	**min**	**max**	**var (%)**	**factor**
ammunition	1148.3	1148.3	0	8675.44	8676.04	0.007	7.56
cjpeg_jpeg6b _wrbmp	1.09	1.09	0	8.04	8.05	0.14	7.37
cjpeg_transupp	39.84	39.84	0	279.97	280	0.012	7.03
dijkstra	472.61	472.61	< 0.001	1552.71	1557.35	0.3	3.3
epic	64.15	64.15	0	510.87	511.03	0.031	7.97
gsm_decode	19.69	19.7	< 0.022	151.69	151.84	0.099	7.71
gsm_encode	47.19	47.19	0	340.17	340.22	0.013	7.21
h264_dec	0.46	0.46	0	3.83	3.83	0.126	8.39
mpeg2	2890.55	2890.55	0	20237.81	20238.08	0.002	7
ndes	0.63	0.63	< 0.548	4.46	4.5	0.753	7.09
rijndael_decoder	31.06	31.06	< 0.001	228.46	228.48	0.011	7.36
rijndael_enc	35.16	35.16	< 0.001	256.62	256.73	0.046	7.3
statemate	0.39	0.39	0	3.15	3.15	0.187	8.02

Table 2 Results in the LP configuration, in which the instruction and data caches are disabled. The columns "min" and "max" are expressed in millions of cycles; the columns "var" are expressed in % and correspond to $(\max - \min)/\min$; the column "factor" is the ratio between the maxima in contention and isolation, i.e. factor $= \max(\text{CON}) / \max(\text{ISO})$.

are located exactly $8 \times 4 = 32$ bytes apart in the memory (the size of an integer is standard on the Kalray, i.e. 4 bytes). This is done on purpose knowing that the private data cache line of every VLIW core in the compute clusters of the Kalray MPPA-256 is 32 bytes long. Consequently, every reading causes a cache miss and the value must then be fetched from the 2MB in-cluster shared memory, hence it creates traffic on the shared memory communication channels and potentially interfere with the application being analysed. At runtime, this function is called repeatedly in an infinite while-loop until the IG receives the command to stop (that command is sent at the end of the execution of the program under analysis).

By running the application concurrently with these IGs, every request that it sends to read or write a data in the shared memory is very likely to interfere with a read request from one of the IGs. As reported in Section 4, the variation in the execution time between the isolation and contention modes is substantial.

4 Experimental results

We ran each benchmark application one thousand times, each time over the same input data[3], in isolation and contention modes and both with the HP and LP configurations. Tables 2 and 3 expose the results in the LP and HP configurations, respectively. It is important to stress here that for each program, *we used the same input set for the thousand runs*! We did so in order to focus solely on the variation of execution time due to the interference between concurrently-running processes. The input data set that we used is the one provided "by default" that is available immediately on downloading the source code of the benchmark programs from the TACLe website [1]. From the results we made few interesting observations:

- With the LP configuration (Table 2), for 8 out of 13 tested programs, the execution takes

[3] Since the inputs are fixed, the remaining variability in the MIET should be caused by the initial hardware state (like contents of caches, state of the branch predictor, etc.).

	ISO			CON			
Name	**min**	**max**	**var (%)**	**min**	**max**	**var (%)**	**factor**
ammunition	247.24	247.25	< 0.002	248.07	248.09	< 0.008	1.003
cjpeg_jpeg6b _wrbmp	0.23	0.23	< 0.067	0.23	0.23	< 0.217	1.003
cjpeg_transupp	8.9	8.9	< 0.004	8.9	8.9	< 0.005	1.00003
dijkstra	101.92	101.92	< 0.001	101.39	101.4	< 0.009	0.995
epic	18.81	18.81	< 0.002	18.84	18.84	< 0.023	1.002
gsm_decode	4.44	4.44	< 0.017	4.44	4.45	< 0.042	1.002
gsm_encode	11.08	11.08	< 0.001	11.1	11.1	< 0.028	1.002
h264_dec	0.09	0.09	< 0.149	0.09	0.09	< 0.347	1.002
mpeg2	619.77	619.77	< 0.001	620.31	620.33	< 0.003	1.0009
ndes	0.13	0.13	< 0.679	0.13	0.13	< 0.937	1.003
rijndael_decoder	9.29	9.29	< 0.002	9.54	9.55	< 0.102	1.03
rijndael_enc	10.13	10.14	< 0.105	10.31	10.32	< 0.164	1.02
statemate	0.09	0.09	< 0.33	0.1	0.1	< 1.83	1.11

Table 3 Results in the HP configuration, in which the instruction and data caches are enabled. The columns "min" and "max" are expressed in millions of cycles; the columns "var" are expressed in % and correspond to $(\max - \min)/\min$; the column "factor" is the ratio between the maxima in contention and isolation, i.e. factor $= \max(\text{CON}) / \max(\text{ISO})$.

the *exact* same time, to the nearest CPU cycle, when running in isolation one thousand times over the same set of inputs. This is true even for the "mpeg2" program that executes in about 2890 millions of cycles. Its execution time remains constant throughout the 1000 runs. Although this constance is commonly assumed in theoretical works (same input → same execution path → same output and same duration), we did not expect it to be 100% true in practice.

- With the HP configuration (Table 3), still in the isolation mode, all the programs have experienced a different execution time, which is thus due to the non-determinism of the cache. Sometimes this variation is small, still it is always there.
- Comparing the results of the contention mode between the LP and HP configurations, we see that using the caches has somewhat isolated the programs from each other. Under the HP configuration, The IGs are able to increase the execution time of the analyzed program only by a small factor (1.11 being the worst-case observed). This is because once the program is loaded into the cache (both instructions and data), the program does not need to further communicate with the 2MB of shared in-cluster memory. Therefore the IGs do not have a mean to interfere substantially with its execution. In the LP configuration however, the execution time is increased by a factor up to 8, which means that the analyzed program is 8 times slower due to interference with concurrently-running processes! This slow-down factor clearly advocates the use of specialized techniques to prevent processes from interfering with each other at runtime (or at least mechanisms should be set to mitigate the effect of this interference). We believe that a slow-down factor of similar magnitude could be observed even with the caches enabled (i.e. in the HP configuration) if the analyzed program had to communicate frequently with the shared memory, hence giving the opportunity to the other processes running on the other cores to interfere with its execution.

Important note: All the benchmarks that we have analysed seem to have data sets that mostly fit into the data cache of the core on which they are deployed. It would be useful and

highly interesting to conduct further experiments on benchmarks with larger data set sizes. Intuitively, it seems that the results of the "contention" mode for benchmarks with larger data set sizes would be somewhere in between what we see in this paper for the LP and HP configurations. Due to time and space constraints, we were not able to include such results in this paper.

5 Conclusion

The paper aimed at quantifying the effect of the inter-process interference on the processor shared resources. We showed that on the Kalray MPPA-256 (Andey) manycore platform the interference can slow down the execution of a program by a factor of 8, and this slow down factor is obtained in conditions that may not even be the worst (the IGs certainly do not generate the maximum interference). Of course, many questions remain open: What is the maximum slow-down factor that we could experience at runtime? What is the relation between the slow down factor and the memory access pattern of the analyzed program? And of course, how to totally isolate the processes from each other without degrading too much the performance? We plan to make more elaborated experiments in the near future to answer those questions, or at least to provide insights that would enable us to answer them.

Acknowledgements

This work was partially supported by National Funds through FCT/MEC (Portuguese Foundation for Science and Technology) and co-financed by ERDF (European Regional Development Fund) under the PT2020 Partnership, within the CISTER Research Unit (CEC/04234); also by the European Union under the Seventh Framework Programme (FP7/2007-2013), grant agreement n° 611016 (P-SOCRATES).

References

1 Timing analysis on code-level. `http://http://www.tacle.eu/index.php/activities/taclebench`. Accessed: 2016-05-06.

2 B. Dupont de Dinechin, Y. Durand, D. van Amstel, and A. Ghiti. Guaranteed services of the NoC of a manycore processor. In *Int. Workshop on Network on Chip Architectures*, pages 11–16, Cambridge, United Kingdom, 2014.

3 B. Dupont de Dinechin, D. van Amstel, M. Poulhiès, and G. Lager. Time-critical computing on a single-chip massively parallel processor. In *Design, Automation & Test in Europe Conference & Exhibition*, pages 1–6, Dresden, Germany, 2014.

4 B. Lesage, D. Griffin, S. Altmeyer, and R. I. Davis. Static probabilistic timing analysis for multi-path programs. In *Real-Time Systems Symposium, 2015 IEEE*, pages 361–372, 2015.

5 V. Nélis, P. M. Yomsi, and L. M. Pinho. Methodologies for the wcet analysis of parallel applications on many-core architectures. In *Digital System Design (DSD), 2015 Euromicro Conference on*, pages 748–755, 2015.

6 R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(7):966–978, July 2009.

7 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem; overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, 2008.