Migration-Aware WCET Estimation for Heterogeneous Multi-Cores

Peter Munk[*][†]
[*]Corporate Sector Research and Advance Engineering
Robert Bosch GmbH
70442 Stuttgart, Germany
peter.munk@bosch.com

Jan Richling[†]
[†]Communications and Operating Systems Group
Technische Universität Berlin
10587 Berlin, Germany
jan.richling@tu-berlin.de

***Abstract*—Heterogeneous Multi-Processor Systems-on-Chip (MPSoCs) are increasingly used in the embedded safety-critical domain with real-time constraints. Fault tolerance, temperature distribution, and energy management in such systems can be improved by reconfiguration mechanisms that rely on task migration. In order to serve a migration request while meeting all deadlines, the remaining worst-case execution time (WCET) must be known as tightly bound as possible. We show that scaling the WCET by a linear factor to compensate for migration between heterogeneous cores can lead to dangerous underestimations. Our approach is to split the WCET of a task into parts and derive the WCET for all parts on all individual core architectures. We present a formal model to show how the remaining WCET can be calculated as the sum of unprocessed parts. We differentiate between two levels of heterogeneity, namely same instruction set architecture (ISA) with different performance characteristics and different ISAs. We propose three implementation concepts to partition a task, calculate the remaining WCET, and perform the migration.**

I. Introduction

An increasing number of today's Multi-Processor Systems-on-Chip (MPSoCs) contain heterogeneous cores with different performance characteristics and architectures in order to enable power-saving technologies and accelerate application-specific tasks, e. g. in the multimedia processing domain. Such processors invade embedded systems with real-time constraints due to an increasing computational demand.

Scheduling protocols for hard real-time systems require the a priori knowledge of the worst-case execution time (WCET). In this paper, we address the fact that the remaining WCET of a task that is migrated between heterogeneous cores is in general not proportional to the difference of the cores' performance characteristics. The behavior of most real applications varies over execution time, and most migration requests arrive sporadically. Thus, it is difficult to find a linear scaling factor that allows to determine a tight bound of the remaining WCET.

To show the inherent problem, we conducted the following motivating experiment: We wrote a program that contains three subroutines. The first routine, "Mem", copies 1 KB of memory, the second routine, "Calc", repeats some integer arithmetic functions one thousand times, and the third routine, "Combined", copies 1 KB of memory to a location calculated by some integer arithmetic function. The source code was compiled using GCC with debug information but without optimization or LibC. Bound-T [1] was used to derive the WCET of the sample program on an ARM7 TDMI and an AVR ATMega128. The ratio between the WCETs of the ARM and the AVR version of the entire program is 1.72. Figure 1 shows that scaling the WCETs of the subroutines with this ratio is not exact. When migrating a task between heterogeneous cores, the WCET also depends on the access latency of other resources such as memory or I/O. For the memory dependant subroutine, the real ratio between the ARM and the AVR version is 1.85. This results in a longer WCET than estimated by scaling. Thus, the scheduling policy cannot guarantee to meet all deadlines, and hazardous situations could arise.

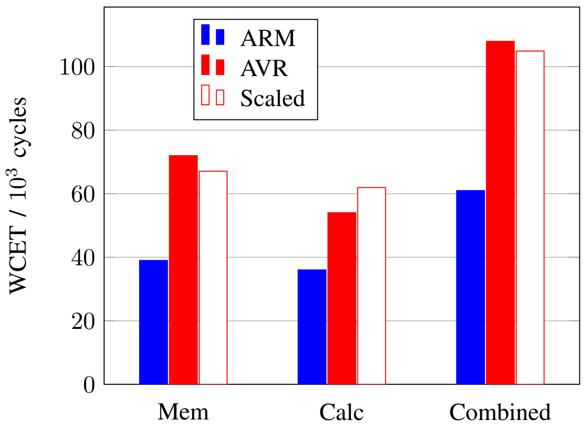

Figure 1. The WCETs of three different subroutines on an ARM7 TDMI and an AVR ATMega128 compared to the scaled values for the AVR processor.

Other hazardous situations can origin from imminent failures of cores in an MPSoC. In case these failures can be detected in time, job-level migration, i. e. during the execution of a task instance, allows to continue the execution on another core while keeping interim results and thus enhances the system's adaptivity. Core failures can arise from thermal hot spots on an MPSoC [2]. In order to manage the temperature distribution on an MPSoC, task migration has shown to be a useful method [2]. Furthermore, power-aware scheduling algorithms need to move tasks between heterogeneous cores in order to maximize energy efficiency [3]. Job-level migration also allows to continue the execution of a job on a higher performance processing element and thus enables the job to meet its deadline which would not have been possible otherwise. Altogether, these examples motivate research on migration-aware WCET estimation for heterogeneous MPSoCs.

We propose a migration framework that is split into a monitoring and decision unit (MDU) and a migration execution unit (MEU). The MDU requires the remaining WCET to derive its decision of when and where to migrate a job—the decision itself is based on criteria as sketched above. While the MDU is out of scope of this paper, we address the question how to derive the remaining WCET and how the MEU is able to migrate between heterogeneous cores in general. We assume all involved components to work without failures.

The contributions of this work are as follows:

- We propose a mechanism to increase the accuracy of the estimation of the remaining WCET after a job-level migration between heterogeneous cores by splitting the overall WCET of a task into multiple parts.
- We discuss implementation concepts of our partitioning idea at three levels, namely on (a) WCET analysis level, (b) compiler level, and (c) library level.

The remainder of this paper is structured as follows: Section II distinguishes between different levels of heterogeneity and evaluates migration concepts for heterogeneous architectures in general. Section III presents our general approach, which can be implemented at different levels as we show in Section IV. In Section V, we provide an overview of the related work. Our conclusions are drawn in Section VI.

II. Heterogeneous Migration

In this section, we discuss two different levels of heterogeneity in an MPSoC and address their implications on job-level migration.

A. Levels of Heterogeneity

We distinguish between cores with a common subset of the ISA but a different architecture or operating frequency, and entirely different ISAs.

An example for heterogeneous cores with identical ISA is Nvidia's Variable SMP platform, which contains five ARM Cortex A9 cores. The fifth core is built in a special low power silicon process and operates with reduced frequency to safe power.

An example of an MPSoC containing cores with different ISAs is Qualcomm's Snapdragon S4. Apart from two general purpose cores, it contains a digital signal processor (DSP), a multimedia co-processor, and a graphics processing unit (GPU).

B. Job-Level Migration

In general, job-level migration includes pausing the execution of the job, saving its current state, transferring the state to the destination node, restoring the state, and resuming the execution [4]. Note that we assume the hardware architecture to provide features to transfer the state from one core to another, e. g. a shared memory with global address space or a communication infrastructure in hardware.

At the first level of heterogeneity, the OS can obtain the state of the job in a similar way it does for context switches. The state consists of the content of the register file, the job's stack, and the job's dynamically allocated memory spaces.

On the second level of heterogeneity, migration is exacerbated by the need to save the job's state in an ISA-independent representation. There are different possibilities to migrate tasks between cores with different ISAs.

First, the problem can be solved by using a common intermediate language (CIL) and an interpreter [4]. This approach has the advantage that migration can be done at any time, since the state is always present in an ISA-independent representation. We assume that the language's run-time system or the interpreter is able to extract the state. An interpreted program has the disadvantage that more memory space and a longer execution time are usually required.

Second, the program can be compiled multiple times for different architectures, which is also known as "fat binary". Migration is possible if a mapping between states from different ISAs is known. By limiting migration to a specific set of points in execution time, a mapping function can be derived at compile-time.

III. Approach

In this section, we present a formal model to derive a guaranteed remaining WCET. The complexity of the model is increased stepwise.

Our key idea is to split a task's overall WCET into multiple parts $p_i \in P$. Ignoring heterogeneous architectures and hardware effects, the $WCET(T)$ for task T is $\sum_{i=1}^{n} p_i$, where each part p_i is element of the worst-case execution path and $n = |P|$ is the number of parts. We call the transition between two succeeding parts a *checkpoint* $c_i \in C$, so $c_{i+1} - c_i = p_i$. Note that the checkpoints can be extended to those commonly known from checkpointing and rollback mechanisms that are used to enhance fault tolerance.

In case a part p_i is repeated in a loop, the repetition variable r_i saves the number of unprocessed iterations. Thus, the WCET can be derived by $WCET(T) = \sum_{i=1}^{n} r_i p_i$. The upper bound of repetitions is typically known in the real-time domain, since it is required by static timing analysis tools. In case of nested loops, the repetition variable represents the product of the repetitions of all nested loops.

We assume that the remaining worst-case execution path of each part is known and that the WCET of each part can be derived for different heterogeneity levels by applying established techniques. Thus, each part is a vector with values for all heterogeneous architectures $a \in A$, i. e. $p_i = \{p_i^1, \ldots, p_i^a, \ldots, p_i^m\}$, where $m = |A|$ is the number of different architectures. If a job is migrated to an architecture a when it reaches checkpoints c_i, the remaining WCET equals

$$WCET_{remain}(T, c_i) = \sum_{j=i}^{n} r_j p_j^a + WCMT, \qquad (1)$$

where p_j^a are the parts on the remaining worst-case path. $WCMT$ is the worst-case migration time, i. e. the time required to pause the job, save, transfer, translate, and restore its state, and to resume the job on the destination core.

If a job is migrated to an architecture a between checkpoints c_i and c_{i+1}, an upper bound of the remaining WCET can be derived by (1) with the last passed checkpoint c_i and the remaining number of unprocessed iterations r_i.

If caches and pipelining effects are taken into account, the WCET without checkpoints is less than or equal to the sum of the WCETs between the checkpoints, i. e. $WCET(T) \leq \sum_{i=1}^{n} r_j p_i^a$, since the static timing analysis tool has to assume that the caches and the pipeline are not loaded after each checkpoint. This overestimation could be mitigated by calculating the remaining WCET for the entire remaining worst-case execution path at once instead of summing up the remaining parts as in (1). However, this leads to an exponentially growing number of remaining WCETs.

IV. Migration-Aware WCET

In the following, we present different concepts how the current position of the execution, i. e. which parts have already been executed, can be determined, and how it can be applied for the different levels of heterogeneity.

A. Static Timing Analysis Extension

The input of most static timing analysis tools is the compiled executable of a task. After decoding the executable, its control flow graph (CFG) is reconstructed. Each node in the CFG equals a basic block (BB) that represents a sequence of instructions with only one entry point and only one exit point [5]. The remaining worst-case execution path and the WCET of each BB is derived by static timing analysis tools with additional information about the target processor. Thus, a part p_i as defined in Section III can simply be represented as a BB. The number of BBs and remaining worst-case paths can become rather large for real applications. To mitigate this effect, a part can be represented by a group of contiguous BBs, e. g. a small loop.

Since the static timing analysis tool is aware of the compiled executable, the currently active part can be determined by the value of the program counter (PC). If the currently active part is inside of a loop, the number of left repetitions r_i has to be known to determine the remaining WCET. When a migration request arrives, the number of unprocessed repetitions can be conservatively estimated by the time passed since entering the repeating section and the upper bound of repetitions.

The advantage of this method is that the source code does not have to be changed. Migration is always possible, independent of the time when the migration request arrives. In case migration takes place between two checkpoints, the remaining WCET can be determined by including the current part to the sum of unprocessed parts as mentioned in Section III.

However, the same property also restricts the application of this method to the first heterogeneity level, i. e. same ISA but different performance characteristics. In general, there is no possibility to find a mapping between all instructions of different ISAs, and a restriction to checkpoints requires some modification of the original source code.

B. Compiler Extension

The compiler is able to instrument intermediate code at all transitions of consecutive parts, thus adding checkpoints c_i. Again, a part can represent one or more contiguous BBs of intermediate code. Using multiple back-ends, this allows to generate binaries for different ISAs with a common set of checkpoints which can be analyzed by static timing analysis tools. The currently executing part and the number of left repetitions can be determined with the help of the instrumented code.

Migration on the second heterogeneity level, i. e. different ISAs, is restricted to the checkpoints. So a job can only be migrated if it has reached a checkpoint. This way, the remaining worst-case execution path on the new architecture and the state are known at compile-time. Further, the compiler is able to create a mapping and an intermediate representation of the state to be transferred. This information can be saved externally, e. g. in the MEU, in order to limit the increase of the size of the executable. Restricting the migration to checkpoints also leads to a more precise calculation of the remaining WCET. Note that this method also allows and enhances migration on the first heterogeneity level by applying the same methods as in the previous approach and providing a more accurate value of unprocessed repetitions.

One possible implementation to serve a migration request is to set a flag when the request arrives at OS level. The flag is checked by the instrumented code and if set, the migration is performed in cooperation with the OS.

Another approach is to save the state whenever the execution reaches a checkpoint independent of a pending migration request. Once a request arrives, the OS immediately stops the task's execution and starts the migration based on the previously saved state. While this also increases fault tolerance if accompanied by established checkpoint and rollback mechanisms, it has to be carefully adjusted with non-idempotent functions. Such functions interact with their environment, e. g. by I/O or by inter-task communication and cannot be repeated without side-effects. To cope with such functions, the compiler adds a checkpoint after each non-idempotent command and the OS delays migration requests during these commands. Thus, non-idempotent commands are never interrupted by migration requests and the time a request is pending is minimized.

C. Manual Definition

Instead of relying on automatic mechanisms at compiler or static timing analysis level, the user can exactly specify at which point in time of the task's execution a migration is permitted. Under the assumption that the user has additional knowledge about which data has to be migrated and which data can safely be neglected, the overall amount of data to be transferred can be decreased.

To ease the specification of checkpoints in the source code, the user is provided with a library that facilitates the definition of migration points and the corresponding state. The library also keeps track of the current iteration in case a checkpoint is defined within a loop. Additionally, non-idempotent commands can be encapsulated by the library to ensure consistency with external states. Once the program is compiled and the static timing analysis tool has determined the WCETs of all parts on all architectures, this data is fed back into the library, e. g. by saving the values to a specific memory location.

The library offers an interface for the OS that allows to retrieve and to instantiate a job's state in a machine-independent format. Additionally, the library provides the remaining WCET. This does not restrict the first approach, so migration on the first heterogeneity level is always allowed.

V. Related Work

Process migration has been an active research topic since the 1980s and has mainly focused on the high-performance computing (HPC) domain [4]. The problem of process migration between heterogeneous machines has been addressed by several projects, e. g. [6], [7]. While such approaches typically rely on checkpointing mechanisms to transfer the process state in a machine-independent format, the embedded domain and the effects on real-time constraints are not addressed.

One of the first approaches to migrate tasks on an embedded processor was presented by Bertozzi et. al [8]. Other projects have proposed migration schemes for embedded MPSoCs without a memory management units (MMU) and with focus on streaming applications [9], [10] or without shared memory [11], [12]. Due to a special model of computation, this can be achieved without user-defined checkpoints [13]. However, all of them focus only on homogeneous MPSoCs.

Gantel et. al [14] migrate tasks between heterogeneous cores of an MPSoC on top of µC/OS-II. Migration requests are pending until the execution has reached a user-defined checkpoint. Similar to our work, the authors focus on real-time applications. However, we argue that the authors only rely on measurements, while our approach considers the WCET to guarantee hard real-time constraints.

Kumar et at. [15] present two MPSoCs of the first level of heterogeneity and propose a set of heuristics for dynamic job-to-core assignment. The heuristics are based on measurements derived from hardware performance counters during a sampling phase. In this phase, the scheduler permutes the job-to-core assignments. After the sampling phase, a new assignment is derived for the following steady phase. We argue that in contrast to [15], our approach is able to guarantee hard real-time constraints since we consider WCETs instead of measured runtimes.

VI. Conclusion

The remaining WCET after a job-level migration between heterogeneous cores has to be known in order to meet all deadlines of a real-time system even in case of migration. In this paper, we showed why linear scaling of the remaining WCET can lead to dangerous underestimation, even if the cores only differ in performance characteristics.

We differentiated between two levels of heterogeneity, namely a common ISA with different performance characteristics and entirely different ISAs. We presented general migration concepts for these levels. Our main idea to derive the remaining WCET is to split the WCET of a task into parts and derive the WCET for all parts on each heterogeneous core. Thus, the remaining WCET equals the sum of the parts on the remaining worst-case execution path. Based on the general migration concepts, we proposed three implementation concepts, namely on (a) static timing analysis level, (b) compiler level, and (c) library level.

The proposed mechanisms can also be used to estimate the energy consumption on the destination core and thus allow an optimization of the overall power efficiency. The further investigation of these topics as well as a practical implementation and evaluation of the proposed concepts is ongoing work.

References

[1] Tidorum Ltd., "Bound-T," 2013. [Online]. Available: www.bound-t.com

[2] P. Chaparro, J. Gonzalez, G. Magklis, C. Qiong, and A. Gonzalez, "Understanding the thermal implications of multi-core architectures," *IEEE Trans. Parallel Distrib. Syst.*, vol. 18, no. 8, pp. 1055–1065, 2007.

[3] O. Ozturk, M. Kandemir, S. W. Son, and M. Karakoy, "Selective code/data migration for reducing communication energy in embedded MpSoC architectures," in *Proc. of GLSVLSI*, 2006, pp. 386–391.

[4] D. S. Milojičić, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou, "Process migration," *ACM Comput. Surv.*, vol. 32, pp. 241–299, 2000.

[5] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem—overview of methods and survey of tools," *ACM Trans. Embedded Comput. Syst.*, vol. 7, no. 3, pp. 36:1–36:53, 2008.

[6] P. Smith and N. C. Hutchinson, "Heterogeneous process migration: The tui system," *Software: Practice and Experience*, vol. 28, no. 6, pp. 611–639, 1998.

[7] H. Jiang and V. Chaudhary, "Process/thread migration and checkpointing in heterogeneous distributed systems," in *Proc. of HICSS*, 2004, pp. 1–10.

[8] S. Bertozzi, A. Acquaviva, D. Bertozzi, and A. Poggiali, "Supporting task migration in multi-processor systems-on-chip: A feasibility study," in *Proc. of DATE*, 2006, pp. 15–20.

[9] M. Pittau, A. Alimonda, S. Carta, and A. Acquaviva, "Impact of task migration on streaming multimedia for embedded multiprocessors: A quantitative evaluation," in *Proc. of ESTIMedia*, 2007, pp. 59–64.

[10] A. Acquaviva, A. Alimonda, S. Carta, and M. Pittau, "Assessing task migration impact on embedded soft real-time streaming multimedia applications," *EURASIP J. Embedded Syst.*, vol. 2008, pp. 9:1–9:15, 2008.

[11] N. Saint-Jean, P. Benoit, G. Sassatelli, L. Torres, and M. Robert, "MPI-based adaptive task migration support on the HS-scale system," in *Proc. of ISVLSI*, 2008, pp. 105–110.

[12] G. M. Almeida, S. Varyani, R. Busseuil, G. Sassatelli, P. Benoit, L. Torres, E. A. Carara, and F. G. Moraes, "Evaluating the impact of task migration in multi-processor systems-on-chip," in *Proc. of SBCCI*, 2010, pp. 73–78.

[13] O. Derin, E. Cannella, G. Tuveri, P. Meloni, T. Stefanov, L. Fiorin, L. Raffo, and M. Sami, "A system-level approach to adaptivity and fault-tolerance in NoC-based MPSoCs: the MADNESS project," *Microprocessors and Microsystems – Embedded Hardware Design*, vol. 37, no. 6–7, pp. 515–529, 2013.

[14] L. Gantel, S. Layouni, M. E. A. Benkhelifa, F. Verdier, and S. Chauvet, "Multiprocessor task migration implementation in a reconfigurable platform," in *Proc. of ReConFig*, 2009, pp. 362–367.

[15] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas, "Single-ISA heterogeneous multi-core architectures for multithreaded workload performance," in *Proc. of ISCA*, 2004, pp. 64–76.