A WCET-Aware Parallel Programming Model for Predictability Enhanced Multi-core Architectures

Simon Reder*, Leonard Masing*, Harald Bucher*, Timon ter Braak†, Timo Stripf‡, Jürgen Becker*
*Karlsruhe Institute of Technology, †Recore Systems B.V., ‡emmtrix Technologies GmbH

***Abstract*—Increasing performance requirements for cyber-physical systems in real-time applications raise the necessity to migrate to multi-core processor systems. However, commercial of the shelf multi-core systems are often inappropriate for the real-time domain and real-time capable multi-core programming models are rare. In this paper, we present a solution developed within the EU research project ARGO. By means of a predictability-enhanced NoC-based multi-/many-core architecture, we investigate hardware properties that can help to improve the predictability of the platform and the programming model. Both platform and programming model are complemented by a WCET-aware Architecture Description Language (ADL). This enables a certain degree of hardware abstraction while preserving the relevant details for accurate multi-core WCET analysis algorithms. Target platform and programming model are designed to be statically analyzable by multi-core WCET computation tools, that are part of the automated WCET-aware software parallelization tool flow developed in the ARGO project.**

I. Introduction

Today's trend towards smarter and increasingly complex cyber-physical systems raises the need for high computation performance in embedded architectures. This trend also applies to systems that operate in safety-critical environments, e.g. in automotive, avionic or industrial applications. This entails additional challenges, since these components are often required to meet hard real-time constraints. To provide the necessary computational performance for complex applications, modern multi-core architectures become more and more important in the real-time sector. To ensure that real-time constraints are met, it is necessary to compute at compile time a safe upper bound for the Worst Case Execution Time (WCET) of all relevant software routines on the given hardware [1]. This problem can only be solved with sufficient accuracy, if the timing behavior of both hardware and software is predictable. In the multi-core case, it is significantly more challenging to meet these requirements, such that predictable multi-core processor architectures, WCET-aware parallel programming models as well as efficient compiler tools are still a field of intense research [2][3].

The main difficulty in predicting the timing behavior of multi-core processors compared to the single-core case is, that various hardware resources (e.g. buses/interconnects, memories and shared caches) are shared between the processor cores [4]. For non real-time applications, resource sharing can help to increase the utilization of individual resources and overall reduce the required amount of hardware units per core. However, for real-time applications it is necessary to consider the worst case execution times which are correlated to the worst case resource usage. For shared resources in most multi-core architectures, it is very challenging to tightly predict the worst case resource usage at compile time. A fundamental prerequisite is to be able to determine which shared resource accesses may occur in parallel and how they interfere with each other in terms of the access times. Predicting and minimizing this interference is one of the key challenges when designing real-time capable multi-core software and hardware.

Typical commercial of the shelf (COTS) multi-core hardware is optimized for average case performance rather than for predictability[2]. When computing upper bounds for the access times in such systems, it is often required to make pessimistic assumptions (e.g. that shared caches are cold or that the bus is frequently blocked by other processor cores). This can lead to very high or even unbounded WCET estimates for the parallel software which negates the advantages over the single-core case. Therefore, it is inevitable to optimize both hardware and software for WCET awareness and predictability in order to significantly improve the worst case performance of multi-core systems.

Developing, optimizing and testing parallel software for multi- or many-core processors is already time-consuming and costly for non real-time applications. In case of time critical applications, the problem of interference and the lack of extensive hardware abstraction lead to even more complexity which must be handled during software development. This is obviously a great drawback for the productivity which could be significantly eased by improved parallelization and code generation tools for time critical applications. The EU Horizon 2020 research project ARGO[1] addresses this shortcoming by providing novel methods and tools to automatically generate code with WCET guarantees for high-performance embedded multi-core systems.

A. The ARGO WCET-Aware Parallelization Flow

In the following, we provide a brief overview of the general ARGO approach before presenting a predictability-enhanced NoC-based target architecture and a WCET-aware programming model which are the main focus of this paper. The proposed programming model and target architecture are important prerequisites for the multi-core WCET analyzer that will be developed in ARGO.

The ARGO tool-flow[5] starts with a high-level application written with the Scilab/Xcos[2] software for numerical computa-

[1]http://www.argo-project.eu/
[2]http://www.scilab.io/

978-3-9819263-0-9/DATE18/©2018 EDAA

tion and model-based application development. The goal is to automatically generate parallel C code optimized for worst case execution time on a given multi-core target platform. The basic structure of the ARGO tool-flow is depicted in Figure 1. All steps are embedded into an iterative cross-layer optimization loop, that can optionally be interactively controlled by the end-user using a graphical user interface.

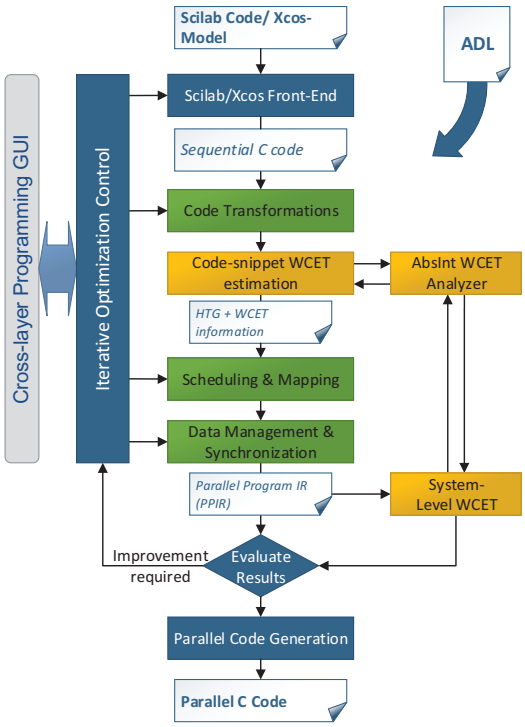

Fig. 1. Structure of the ARGO tool-flow.

The first step in the tool-flow is the conversion of the input application, which may consist of a combination of Scilab and Xcos artifacts, to the target language C99 (ISO/IEC 9899:1999). The resulting single-core C code utilizes a well predictable subset of C99 and is platform independent. The parallelization process starts with applying loop transformation techniques in order to improve data locality and the tightness of the WCET analysis.

Subsequently, a parallel program schedule and mapping is generated for the transformed sequential program. The corresponding "Scheduling and Mapping" step is based on the Hierarchical Task Graph (HTG) representation introduced in [6] and uses the results of a sequential WCET analysis with the AbsInt aiT WCET analysis tool[7] as cost metric. Given the HTG representation enriched with WCET information, approaches like [8] can be used to solve the scheduling and mapping problem.

After the scheduling step, the generated solution has to be applied to the program representation. The scheduler uses a simplified synchronization model that is limited to the granularity of the HTG representation. In the "Data Management & Synchronization" step, fine-grained refinements are carried out by optimizing the placement of synchronization & communication operations as well as the mapping of data fields to memories. This results in the ARGO Parallel Program Intermediate Representation (PPIR). The PPIR is used for further platform specific optimizations and finally to generate parallel C99 code. To verify real-time constraints, the PPIR and the parallel output code are passed to the ARGO System-Level WCET estimator. This tool generates a safe upper bound for the WCET of the multi-core application with respect to the target hardware. The WCET estimation and the other tool flow steps rely on a description of the target platform using an Architecture Description Language (ADL) to get information about the hardware.

To optimize the program for WCET, the ARGO tool flow exploits the WCET properties of the target hardware, which are exposed by the programming model and the WCET-aware ADL. In the following, we present the progress of ARGO in elaborating those properties by means of an exemplary NoC-based target platform and introducing a corresponding programming model complemented by an WCET-aware ADL. We identify a set of enhancements for NoC-based multi-core and many-core platforms which help to improve the timing predictability of memories and interconnects in the processor system. The corresponding parallel programming model is designed to explicitly preserve information about the underlying hardware resources in order to enable accurate WCET-analysis of the corresponding programs at compile time. The proposed ADL includes dedicated worst case timing attributes and is used to represent platform information in a defined way to support the hardware abstraction concept of the programming model.

II. A Predictable NoC-based Multi-Core Architecture

The InvasIC[3] platform is a configurable, tile-based multi-core architecture which is investigated and demonstrated on large FPGA-based prototyping systems. Each tile is either a regular computing tile consisting of processors and a local memory or represents a special tile like an I/O tile or a memory tile containing a large shared memory. The tiles are interfaced to each other in a meshed structure via an on-chip network. As the InvasIC hardware architecture was specifically designed for run-time adaptivity and performance/power/area concerns, a typical configuration is not well suited for real-time execution. In the ARGO context, a real-time capable subset was selected and incompatible components were adapted or replaced to be WCET-analyzable as described in the following.

A. Processing-Tiles

In the InvasIC architecture, the components within the computing tiles are mainly built from standard building blocks available as source code in the gaisler research library (GRLIB)[4]. An example setup can be seen in Figure 2. The LEON3 processing cores are connected via an AHB bus to the local memory, the network adapter, and any additional slave

[3]http://invasic.de/
[4]http://www.gaisler.com/index.php/products/ipcores/soclibrary

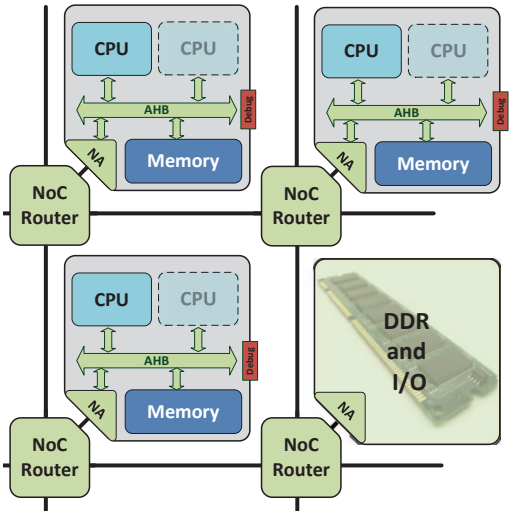

Fig. 2. A 4-tile example of the NoC-based InvasIC architecture.

component (used for e.g. debugging and software loading). The LEON3 cores themselves are WCET-analyzable by commercial tools like AbsInt aiT[7] which impose only minor restrictions on its configuration, e.g. the replacement strategy of the L1 caches. The AHB bus is also WCET-analyzable[9], yet it is trivially analyzable in a single master design which can also be configured for the InvasIC platform.

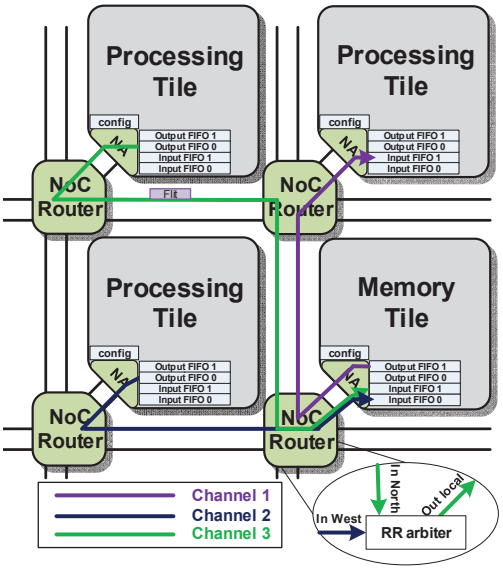

Fig. 3. Channels between tiles in the InvasIC architecture.

B. Predictable communication

Communication between tiles is realized through the concept of unidirectional channels as shown in Figure 3. The network consists of two major components that need to support these channels: The routers and the network adapter. The routers are the backbone of the network, organized to form a meshed topology. They offer a so called guaranteed service (GS) connection that can be used to establish end-to-end connections with a guaranteed bandwidth and latency. The unidirectional GS mechanism ensures at the same time, that livelocks and deadlocks are not possible at hardware level. In contrast to state-of-the-art predictable NoC architectures like [10], the GS connections use round robin (RR) arbitration instead of TDMA. RR offers better worst case latency and throughput, if the communication pattern (i.e. the maximum number of accesses that may happen concurrently) is known. In combination with the proposed programming model and ADL, detailed interference analysis algorithms are therefore able to exploit the RR arbitration and obtain less pessimistic WCET bounds compared to the TDMA case.

The GS connections are realized through reservation of virtual channel buffers in each hop along the way from source to destination. Because there are a limited number of virtual channels, the tool chain must take these constraints into account for the mapping of tasks to tiles. To establish such a GS connection, the network adapter was designed to offer a configuration interface in addition to the input and output buffers of the channels for the data transmission. The GS connections can be established in an initialization phase and afterwards simple write and read operations are used to transmit the data with predictable delay and throughput.

C. Memory hierarchy

The memory model for the WCET-enhanced InvasIC architecture is based on a hierarchy: The first level is represented by a core specific private cache or scratchpad memory. The next level is a tile local memory that is accessible via the AHB. This memory is shared among all processors within a tile, yet is not visible between tiles. This is an adaptation from the original InvasIC architecture which follows the PGAS (partitioned global address space) approach and thus offers direct access for load and store operations onto the remote memory of any tile via global addresses. The final level of the memory hierarchy is the large DDR memory that resides on a separate tile. This memory is shared among all tiles, yet GS connections to it need to be established first before access to it is possible. Fair arbitration is guaranteed by a special interface between the memory controller and the NoC. This interface can accept incoming connections and arbitrates access to the memory controller among requests in a round robin fashion. In combination with a predictable memory controller, we can guarantee a worst-case latency and throughput for all end-to-end memory accesses.

III. WCET-aware Programming Model

The predictability enhanced multi-core hardware introduced in the previous section is one example, that shows how real-time specialized multi-core architectures could be implemented in the future. In order to efficiently predict multi-core hardware with non-constant interconnection delays depending on the number of interfering accesses, we introduce an extended

Architecture Description Language (ADL) and a WCET-aware programming model. Both are optimized to be statically analyzable concerning the parts of the program that may happen in parallel. The "may happen in parallel" (MHP) information can then be used to provide an upper bound for the number of concurrent accesses and in turn the impact of interference. Compared to abstract programming models (e.g. the Actor model), the proposed model does not entirely abstract from the underlying hardware but is abstract enough to be applicable to a wider range of predictable multi-core architectures. This enables WCET computation algorithms to e.g. take advantage of arbitration schemes like round robin (RR), which (in contrast to TDMA) provide better latency and throughput when the number of concurrent accesses decreases.

A. Advanced ADL for predictable Multi-Core Architectures

The ARGO tool flow and the proposed programming model are not preassigned to a specific architecture or platform configuration. However, several steps of the ARGO tool-flow rely on relatively low level hardware details to optimize the program and compute worst-case times. In order to provide the necessary information about the targeted platform, we introduce an advanced WCET-aware ADL based on the *Software-Hardware Interface for Multi-Many-Core 1.0* (SHIM) ADL developed by the Multicore Association [11].

The original SHIM ADL supports modeling the performance of hardware component for average case, best case and worst case. However, the performance model has not been designed to provide safe and at the same time tight upper bounds for the latencies of hardware operations. The SHIM extensions developed in ARGO[12] therefore complement the SHIM ADL by additional attributes to model e.g. the interference behavior of hardware resources.

The proposed worst case specific extensions focus on I) adding means to describe the behavior of caches in detail and II) extensions to model the effect of multiple concurrent accesses on the execution times with respect to the involved hardware units. In area I) SHIM provides attributes for size specific information like the line size and the cache associativity. We extend the attributes of caches by adding the cache replacement strategy, which is an important aspect for the worst case performance analysis. Without knowing the replacement strategy, a static analysis tool would not be able to predict which data is present in the cache at a given position in the program. Static WCET analysis tools can for instance take advantage of caches with "least recently used" (LRU) replacement strategy[13]. Besides the replacement strategy, predictable multi-core architectures like the architecture proposed in Section II typically do not provide cache coherency for shared memories. The underlying coherency protocol would be hardly predictable for the WCET analysis, such that caches are usually disabled for shared memory address ranges. To reflect this in the ADL, we add a corresponding attribute to specify if an address range is cached or not.

To avoid overly pessimistic worst case times for operations that are prone to interference, we extend the ADL in area II) in order to enable detailed interference modeling. This involves all hardware components that are shared between multiple cores and especially also the communication buses and interconnects. In SHIM, different worst case performance values can only be specified per access type, but not with respect to different numbers of concurrent accesses. In the proposed extension, we add an additional attribute, which is able to specify for which number of concurrent accesses the performance values are valid. This way, multiple performance values can be specified for different interference scenarios. This can be generalized in future, where we plan to add support for executable script snippets, such that algorithms for calculation of interference costs can be directly embedded into the ADL.

B. Model of Computation

The proposed programming model is based on a Process Network (PN) model of computation (MoC). The PN model was first introduced 1974 by Kahn [14] and consist of a set of processes which can communicate using a set of unidirectional channels with First-in-First-out (FIFO) characteristics. If a FIFO channel is empty when reading, the read operation will blocks until enough data is available to complete the operation. The original model by Khan assumes that all FIFOs are of infinite size, such that write operations can be non-blocking. Since infinite FIFO sizes are not feasible in real systems, the proposed programming model implements finite FIFO sizes with blocking write operations when channels are full. Additionally, we assume that all FIFO sizes are constant and known at compile time.

It is known, that cyclic dependencies and/or insufficient FIFO sizes can cause a Process Network to deadlock[15]. We exclude such deadlocks by means of the methodology to construct the parallel program in the tool flow. The parallel program representation is built by deserializing a sequential program, which implies that the result is *serializable* by construction. For serializable parallel programs in turn it can be proven, that they are free of deadlocks[16].

C. Process Interaction

Based on the Process Network MoC, we propose a hybrid message-based/shared memory communication model. This enables the efficient use of shared memory segments while private memory segments can be used for local data to avoid costly interference. The proposed programming model consequently does not provide Uniform-Memory-Access (UMA), but preserves a view of several non uniform memory segments depending on the hardware platform and it's ADL description. The programming model introduces these memory segments as logical memories, which can be either private to a single process or shared between a defined set of processes. Similar to the FIFO channels, we require that the set of processes which use a logical memory is known at compile time. The decision, which data is stored in which logical memory is under application control. The mapping of logical memories to the target platform memories is determined at compile time, while taking memory performance metrics into account. Figure 4

shows an example for two processes and the sets of assigned channels and memories.

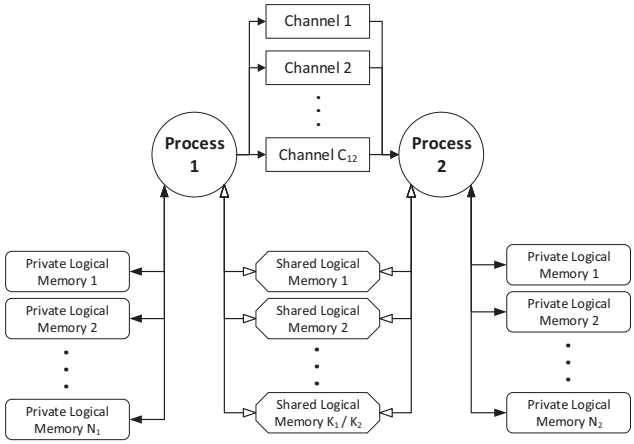

Fig. 4. Communication channels and logical memories for two processes

To enable synchronization between processes, we introduce the operations *signal* and *wait* which operate on one of the FIFO channels. The operations enforce a "happens after" relation, where *wait* blocks the calling process until another process calls a *signal* operation on the respective channel. This mechanism is achieved by sending a token on the FIFO in the *signal* operation which is received by a blocking read in *wait*.

In addition to pure synchronization operations, a possibility to transfer data between processes is required. The proposed programming model supports both message based communication and shared memory. For the former, we introduce *send* and *receive* operations on the FIFO channels, which transfer payload data in addition to the "happens after" synchronization of *signal*/*wait*.

When using shared memory, the programmer must ensure proper synchronization using *signal*/*wait* in order to avoid race conditions. To make sure that memory writes become visible on other cores in a defined way, the programming model needs to specify a memory consistency model[17]. In the proposed programming model, there are with *signal* and *wait* only two types of synchronization operations (*send*/*receive* can be considered as special cases of *signal*/*wait*). We enhance those operations with memory barrier semantics to allow for putting restrictions on the memory order. For accesses to shared memory, this results in a relaxed consistency model with weak ordering (see [17]). The memory barrier attached to the synchronization operations are selective, which means that they can apply to only a subset of the available shared memory segments. We distinguish between *ACQUIRE*, *RELEASE* and *RELEASE_ACQUIRE* memory barriers, where the latter is a combination of the previous ones. The semantics of the barrier types are borrowed from the memory order types introduced by the atomic libraries of the C11 standard (ISO/IEC 9899:2011). The *RELEASE* type is associated with *signal* operations and constrains the memory order such that all preceding writes to the respective memory segments become visible to other cores. To *wait* operations, we assign the *ACQUIRE* type, which makes sure that all subsequent reads on the memory segments see the latest data posted by other cores (with the most recent *RELEASE* barrier).

Typically, synchronization operations are the only valid criterion to determine if parts of the program may happen in parallel or not. By combining synchronization with memory barriers, we ensure that all memory operations are completed on synchronization, such that they do not interfere with program parts that are supposed to happen after the current one. Furthermore, this kind of memory barriers enable hardware optimizations like the use of core-local caches without a coherency protocol. A *RELEASE* operation would then for instance flush the local cache, such that writes to shared memories become visible to other cores. *ACQUIRE* operations would in turn invalidate local cache contents, such that the latest data from the memory is read by subsequent accesses.

D. Intermediate Representation

In the ARGO parallelization flow, the Parallel Program Intermediate Representation (PPIR) is used to represent a parallel program based on the proposed programming model within the compiler tools. The representation is suitable for directly generating parallel C code without further transformations. The PPIR can be regarded as special case of a Parallel Program Graph (PPG) as introduced by Sarkar et al. in [16]. The PPG is an extension to the well known Control Flow Graph (CFG) representation, which adds concurrency and introduces synchronization edges to express synchronization within a parallel program. To fit the proposed programming model, the PPG of the PPIR is required to consist of a fixed number of CFGs (one CFG for each processor core) connected by a start node, which initially forks the control flow into parallel flows (see MGOTO nodes in [16]). A PPG $G(N, E_{cont}, E_{sync}, TYPE)$ has besides two different sets of edges for control flow E_{cont} and synchronization E_{sync} also a node type mapping $TYPE$, which assigns a type to each node in N. In the PPIR we add two special node types $SIGNAL$ and $WAIT$, which correspond to the *signal*/*wait* operations of the programming model.

E. Analyzability

As stated before, it is beneficiary to precisely predict which program parts may happen in parallel in order to obtain tight bounds for the interference costs. To achieve this, we impose a fundamental restriction on the allowed synchronization. In particular we require, that there is always exactly one synchronization edge going from a node of type $SIGNAL$ to a node of type $WAIT$ in the PPIR representation. This results in a one-to-one correspondence of *signal* and *wait* operations, which needs to be known at compile time. The restriction can help to significantly reduce the number of possible program executions, since it eliminates the necessity to consider all possible combinations of *signal*/*wait* operations. Consequently, the number of possibilities for program parts that may happen in parallel is reduced, which in turn enables a less pessimistic interference calculation.

The restricted synchronization obviously has some implications on the overall structure of the program. To ensure that a token sent by a *signal* operation is always received by the corresponding *wait*, the number of FIFO accesses on both the sender and the receiver side must be equal. The *signal* and it's associated *wait* operation have then in both processes the same position in a given sequence of synchronization operations. It is however not necessary to know all possible sequences at compile time, as long as dynamic changes apply in the same way to both sides of the FIFO. A program that includes synchronization inside of conditional branches and/or loops must hence synchronize the corresponding iteration counts and branch conditions in both involved processes. If the iteration count is not known at compile time, this can be achieved by communicating loop/branch condition results between processes. For loops with fixed iteration count, no additional communication is required.

F. Realization on the proposed Hardware Architecture

In order to increase the predictability of the parallel programs, the programming model entities (Processes, Channels and Logical Memories as depicted in Figure 4) are statically mapped to corresponding hardware resources at compile time. For processes we require a one to one mapping to the processor cores, to avoid the necessity of a run-time scheduler.

Channels are realized using the guaranteed service channels provided by the NoC architecture described in Section II. Given the mapping of the processes to the cores, a guaranteed service channel is allocated at program initialization whenever two corresponding processes are connected by a FIFO. The FIFO sizes in that case depend on the buffer sizes of the NoC routers and other interconnection components. Since the sizes cannot be configured in software, the WCET analysis needs to take into account, that FIFO writes may block if no space is left in the buffers. Given the known platform configuration and the topology of the utilized guaranteed service channels, the required size information can be computed at compile time using the proposed ADL.

IV. Conclusion

In this paper, we presented a WCET-aware programming model, an advanced ADL and a predictable hardware platform developed in the context of the EU project ARGO. Together, they shape a hardware/software environment for parallel code generation targeting real-time systems and high-accuracy multi-core WCET estimators. The programming model has been implemented on the presented hardware platform and the prototype version of the ARGO tool-chain is able to generate corresponding parallel code for an Xcos application from the avionics domain as well as a Scilab application from the industrial image processing domain.

ADL and programming model are used as basis for the multi-core WCET analysis tool developed within ARGO and help to keep parallel code and analysis tools largely platform independent. The approach can help to improve the tightness of interference analysis algorithms by exploiting the properties of arbitration schemes like round robin in combination with the may-happen-in-parallel analyzability of the programming model. With the proposed hardware architecture, we introduce a predictable and scalable NoC-based multi-/many-core processor platform compatible with the proposed ADL and programming model. As future work, the ARGO WCET analysis will be implemented on top of the proposed infrastructure and the approach will be evaluated using the resulting WCET and the above-mentioned use case applications.

Acknowledgment

This work has been co-funded by the European Unions Horizon 2020 research and innovation programme under grant agreement No 688131 – ARGO.

References

[1] R. Wilhelm *et al.*, "The worst-case execution-time problem – overview of methods and survey of tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 3, pp. 36:1–36:53, May 2008.

[2] M. Schoeberl *et al.*, "T-crest: Time-predictable multi-core architecture for embedded systems," *Journal of Systems Architecture*, vol. 61, no. 9, pp. 449 – 471, 2015.

[3] T. Ungerer *et al.*, "parmerasa – multi-core execution of parallelised hard real-time applications supporting analysability," in *Proc. Euromicro Conf. Digital System Design*, Sep. 2013, pp. 363–370.

[4] D. Kästner *et al.*, *Meeting Real-Time Requirements with Multi-core Processors*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 117–131.

[5] S. Derrien *et al.*, "Wcet-aware parallelization of model-based applications for multi-cores: The argo approach," in *Proc. Design, Automation Test in Europe Conf. Exhibition (DATE)*, Mar. 2017, pp. 286–289.

[6] Milind Girkar and Constantine D. Polychronopoulos, "The hierarchical task graph as a universal intermediate representation," *International Journal of Parallel Programming*, vol. 22, no. 5, pp. 519–551, oct 1994.

[7] C. Ferdinand and R. Heckmann, "ait: Worst-case execution time prediction by static program analysis," *Building the Information Society*, pp. 377–383, 2004.

[8] A. Emeretlis *et al.*, "A hybrid approach for mapping and scheduling on heterogeneous multicore systems," in *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, July 2016, pp. 360–365.

[9] J. Jalle *et al.*, "Ahrb: A high-performance time-composable AMba ahb bus," in *Proc. IEEE 19th Real-Time and Embedded Technology and Applications Symp. (RTAS)*, Apr. 2014, pp. 225–236.

[10] M. Schoeberl *et al.*, "A Time-Predictable Memory Network-on-Chip," in *14th International Workshop on Worst-Case Execution Time Analysis*, ser. OpenAccess Series in Informatics (OASIcs), H. Falk, Ed., vol. 39. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2014, pp. 53–62.

[11] The Multicore Association, "Software-hardware interface for multi-many-core (shim) specification v1.00 final," 2015, Accessed: 2017-08-23. [Online]. Available: http://www.multicore-association.org/workgroup/shim.php

[12] S. Reder *et al.*, "ARGO Deliverable 4.1: Specification of a WCET-aware abstract architecture description," The ARGO project consortium, EU Project Deliverable, 2016.

[13] C. Ferdinand and R. Wilhelm, "Efficient and precise cache behavior prediction for real-time systems," *Real-Time Systems*, vol. 17, no. 2, pp. 131–181, Nov 1999.

[14] G. Kahn, "The semantics of a simple language for parallel programming," *In Information Processing*, vol. 74, pp. 471–475, 1974.

[15] M. Geilen and T. Basten, "Requirements on the execution of kahn process networks," in *Programming Languages and Systems*. Springer Berlin Heidelberg, 2003, pp. 319–334.

[16] V. Sarkar and B. Simons, "Parallel program graphs and their classification," in *Languages and Compilers for Parallel Computing*. Springer Berlin Heidelberg, 1994, pp. 633–655.

[17] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: a tutorial," *Computer*, vol. 29, no. 12, pp. 66–76, Dec 1996.