Programming MPSoC Platforms: Road Works Ahead!

Rainer Leupers
RWTH Aachen University, leupers@iss.rwth-aachen.de

Soonhoi Ha
Seoul National University, sha@snu.ac.kr

Andras Vajda
Ericsson SW Research, andras.vajda@ericsson.com

Rainer Dömer
UC Irvine, doemer@uci.edu

Marco Bekooij
NXP, marco.bekooij@nxp.com

Achim Nohl
CoWare Inc, achim@coware.com

Abstract—**This paper summarizes a special session on multi-core/multi-processor system-on-chip (MPSoC) programming challenges. The current trend towards MPSoC platforms in most computing domains does not only mean a radical change in computer architecture. Even more important from a SW developer´s viewpoint, at the same time the classical sequential von Neumann programming model needs to be overcome. Efficient utilization of the MPSoC HW resources demands for radically new models and corresponding SW development tools, capable of exploiting the available parallelism and guaranteeing bug-free parallel SW. While several standards are established in the high-performance computing domain (e.g. OpenMP), it is clear that more innovations are required for successful deployment of heterogeneous embedded MPSoC. On the other hand, at least for coming years, the freedom for disruptive programming technologies is limited by the huge amount of certified sequential code that demands for a more pragmatic, gradual tool and code replacement strategy.**

I. Introduction

MPSoC programming practices are currently under rapid evolution and are far from maturity. To strike the right balance between use of legacy code and advanced parallel programming techniques will require more research. In particular, this holds for programming embedded MPSoC architectures. Heterogeneity of the processing elements and communication architectures, as well as real-time requirements and multi-application usage scenarios make the sole reuse of existing multicore programming technologies difficult. The following sections reflect a subset of the state-of-the-art by presenting various approaches for MPSoC SW development. Section II aims at a holistic problem view, focusing on future HW and SW challenges as we move from multicore to true "manycore" architectures. In section III, different models for scheduling real-time tasks are discussed. Section IV describes MAPS, a prototype tool for semi-automatic code parallelization and task-to-processor mapping. An key question is how to hide away the target platform complexity from the programmer, which is discussed in section V. Section VI addresses a framework for source code transformation to complement code partitioning tools. Finally, section VII addresses another important, still frequently neglected, question: how to efficiently debug MPSoC SW, and how to utilize modern Virtual Platform technologies for this purpose.

II. Real-Time Applications

In this section we consider the issues that need to be addressed on MPSoC platforms in order to support applications with real-time requirements. We build our position around the following principles:

- HW shall have homogeneous ISA, scalable, fast and low-latency chip interconnect and frequency variability per core
- Operating systems shall support time-shared and space-shared processing resource scheduling, core frequency adaptability as well as enforcement of strict core and process data locality
- Programming models shall be predictable, deterministic and shall rely on existing sequential semantics.

These items are detailed in the following subsections.

A. HW Issues

The requirement of (near) linear performance increase with the addition of new processing cores can only be achieved by being able to treat the *cores as uniform resources*. Introducing knowledge of any non-homogeneous characteristics of underlying HW into software obviously means a priori partitioning of the functionality to different types of HW, which – being a daunting task in itself – will inhibit scalability,

978-3-9810801-5-5/DATE09 © 2009 EDAA

Authorized licensed use limited to: UNIVERSITEIT TWENTE. Downloaded on January 27, 2010 at 09:20 from IEEE Xplore. Restrictions apply.

or at least keep it suboptimal. Hence, we argue that the only HW architecture that allows (near) linear performance boost of a generic application shall be a *homogeneous architecture, at least in terms of ISA* (instruction set architecture). Uniform ISA guarantees that any piece of software can be executed on any of the processor cores and there's no need to partition the software to be compiled for various ISAs.

Since not all software will be completely parallelized, i.e., there will be parts that will remain sequential and hence, in virtue of Amdahl's law, will represent a bottleneck in the scalability of the application. Therefore, while maintaining the homogeneous nature of the ISA, there is a need to boost the performance of individual cores in order to achieve higher execution speed for sequential code. Such approach shall help mitigate the problem of legacy single-threaded applications as well. Hence, we argue *that the frequency at which each core executes shall be modifiable at a fine-grain level during program execution* and according to the needs of the executing application(s).

In general, looking at the HW architecture on the chip level, in order to achieve easy scalability of HW, while maintaining the same fundamental structure from the SW point of view*, the design shall avoid any centralized constructs* and rely instead on a *fully distributed, homogeneous* approach, including L1 and L2 cache / local memory – i.e., L2 cache / local memory shall be bound to cores.

B. *Operating system issues*

There are two factors that will have a radical impact on how operating systems are constructed:

- Applications *will be inherently parallel*, with an optional serial component
- HW will evolve to a direction where *cores will be abundant*, yet usually simpler resources

In practice, this shift will result in applications requiring two types of computing resources:

- Computing resources for running *sequential, single-threaded code*; this need shall be met with a time-slice of a time-shared core, similarly as today
- Computing resources for executing *parallel software*; this need shall be met with the allocation of multiple space-shared cores *completely dedicated to executing a single application*

Hence, operating systems will have to make the shift to a more *space-sharing* approach, while retaining some of the characteristics of time-sharing systems. In fact, there is a need for scheduling algorithms that can in a reactive way mitigate multiple requests for parallel computing resources as well sequential computing resources that shall be met using a HW base that is basically homogeneous, but can be adjusted by e.g. modifying the frequency at which each core is running. In addition, especially for the purpose of real-time systems, a predictable approach shall be designed, that can meet application dead-line requirements. To the best of our knowledge, no such algorithm has been published yet.

When it comes to memory management, we believe a key characteristic shall be the strict *enforcement of locality*, at least for on-chip memory. This has a number of important implications:

- Protection of each core's *resource integrity* and guaranteeing a framework for *locally sequential execution*
- De-coupling of execution on each core and enforcing a *messaging based programming model*, at least on the OS level

We believe this approach guarantees the removal of all barriers to enforcing *run-to-completion, sequential semantics* on each parallel resource core, critical for any real-time application. In addition, it *enforces memory locality* and *isolation*, with obvious implications for programming models, which we will address in the following.

C. *Programming models*

The HW and OS framework introduced in the previous chapters defines the key underlying execution environment characteristics for an effective parallel programming model:

- *Homogeneity* of execution environment in terms of ISA
- Two type of ISA-compatible computing resources: *time-shared* and *space-shared resources*
- Strict *data locality enforcement*, with an option for small-scale shared memory usage
- *Single-threaded* execution on space-shared cores

We believe that for existing applications, support for frequency boosting of cores enhanced with pre-fetching support from space-shared cores as well as shared memory semantics with no need for locks is the best short term strategy. Automatic parallelization for general problem domains is a hard problem with very limited and usually non-scalable solutions so far.

For new applications, the key issue is to partition the problem into *parallel*, *individually sequential*, *de-coupled threads of execution*, communicating using *asynchronous messages* under the following assumptions:

- High speed processor resources are *scarce* and usually limited to a very few instances
- Low-speed (space shared) processor resources are *abundant*; however the application shall be fully functional – albeit with varying, yet predictable performance – starting from a minimal set of processing resources and scaling upwards to a virtually unlimited amount of low-speed (space-shared) processor resources

As a conclusion, we believe that the same principle shall apply for any parallel programming model as for the HW: it shall provide an *architecture as flat as possible, i.e. horizontally distributed rather than vertically distributed*, with as little hierarchy, as little shared resources and as little

Authorized licensed use limited to: UNIVERSITEIT TWENTE. Downloaded on January 27, 2010 at 09:20 from IEEE Xplore. Restrictions apply.

functional partitioning as possible in order to improve the scalability potential of the software with the increase in the amount of parallel computing resources.

D. Conclusions

In this section we exposed a HW, OS, run-time system and programming model framework that can support real-time applications deployed on chip multi-processors with several tens and hundreds of cores. The key underlying principle – which we believe shall apply to other areas of parallel computing as well – is that of *complete and uniform distribution,* with *no or very few central, shared resources* and a *flat, de-coupled software architecture* made up of *asynchronously communicating, internally sequential components*. For real-time applications with strict requirements on deterministic and time-constrained behavior, this is the most promising approach that can provide scalability as well as easy portability across platforms.

III. Time-triggered Versus Data-driven Real-time Systems

The Hijdra project at NXP-research addresses the design and programming of predictable multiprocessor systems for real-time stream-processing application in car-radios and mobile phones. The developed system [4] is data-driven to overcome some limitations of pure time-tiggered systems.

In time-triggered systems [3], timers periodically trigger the start of the task executions. In our data-driven system, the start of the execution of the tasks is triggered by the arrival of data, except for the source and sink tasks which are periodically triggered by a timer.

In time-triggered systems, the tasks are triggered according to a periodic schedule computed at design-time. For our data-driven system it is sufficient to show at design time that a valid schedule exists such that the periodic source and sink task can execute wait-free [5]. Since we only need to show existence of a schedule, we can reason in terms of a worst-case schedule that bounds the schedules, i.e. arrival times of data that can occur in the implementation. As a consequence, data-driven systems can execute tasks aperiodically, while satisfying timing constraints.

Tasks in a data-driven system execute aperiodically as a result of varying execution times and data dependent consumption and production behavior of the tasks. In a data-driven system, all tasks, except sink and source start on the arrival of data. Therefore, in such a data-driven system, data is not necessarily corrupted in case the execution time of a task exceeds an unreliable worst-case execution time estimate. This implies that even when the schedule that was derived at design time does not pessimistically bound all data arrival times, there is not necessarily corruption of data in the implementation. In a time-driven system, the data is corrupted in this situation because data would be overwritten in a buffer or the same data would be read again. Typically, the functionality of the applications is not robust to corruption of data inside the application, while often the functionality is robust to corruption of data at the sink and source tasks.

From these observations we conclude that a data-driven approach puts less constraints on the application software than a time-triggered approach.

IV. The MAPS Programming Framework

The MAPS project is part of RWTH Aachen´s Ultra high speed Mobile Information and Communication (UMIC) research cluster [2]. It targets efficient code generation for multiple applications at a time and predefined heterogeneous MPSoC platforms. MAPS is thus inspired by a typical problem setting of SW development for wireless multimedia terminals, where multiple applications and radio standards can be activated simultaneously and partially compete for the same resources. The overview of MAPS work-flow is shown in Figure 1. Applications can be specified either as sequential C code or in the form of pre-parallelized processes. In addition, using some lightweight C extensions, real-time properties such as latency and period as well as preferred PE types can be optionally annotated. On top of that, a concurrency graph is used to capture potential parallelism between applications, in order to derive the worst case computational loads. MAPS uses advanced dataflow analysis to extract the available parallelism from the sequential codes (see [1] for a more detailed description of code partitioning) and to form a set of fine-grained task graphs based on a coarse model of the target

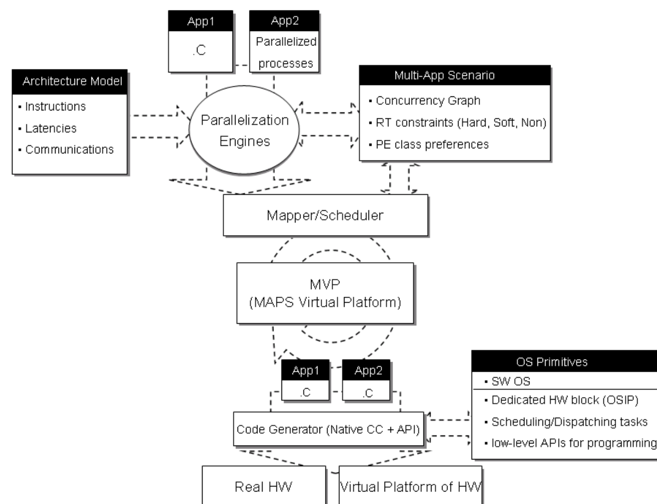

Figure 1: MPSoC Software Design-flow by MAPS

architecture. Initial case studies on partitioning applications like JPEG encoder indicate promising speedup results with considerably reduced manual parallelization efforts. Using optimization algorithms, the task graphs are mapped to the target architecture, taking into account real-time requirements and preferred PE classes. Hard real-time applications are scheduled statically, while soft and non-real-time applications are scheduled dynamically according to their priority in best effort manner. The resulting mapping can be exercised and refined with a fast, high-level SystemC based simulation environment (MAPS Virtual Platform, MVP), which has been designed to evaluate different software settings specifically in a multi-application scenario. After further refinement, a code generation phase translates the task graphs into C codes for compilation onto the respective PEs with their native compilers

Authorized licensed use limited to: UNIVERSITEIT TWENTE. Downloaded on January 27, 2010 at 09:20 from IEEE Xplore. Restrictions apply.

and OS primitives. The SW can then be executed, depending on availability, either on the real HW or on a cycle-approximate virtual platform incorporating instruction-set simulators. While targeting only SW OS at the moment, in the future MAPS will also support a dedicated task dispatching ASIP (OSIP, operating system ASIP) in order to enable higher PE utilization via more fine-grained tasks and low context switching overhead. Early evaluation case studies exhibited great potential of the OSIP approach in lowering the task-switching overhead, compared to an additional RISC performing scheduling in a typical MPSoC environment.

V. RETARGETABLE EMBEDDED SOFTWARE DESIGN METHODOLOGY

Embedded software design for MPSoC means parallel programming for non-trivial heterogeneous multi-processors with diverse communication architectures and design constraints such as hardware cost, power, and timeliness. Two major models for general purpose parallel programming are MPI and OpenMP: MPI is designed for distributed memory systems with explicit message passing paradigm of programming while OpenMP is designed for symmetric multiprocessors with a shared memory. While an MPI or OpenMP program is regarded as retargetable with respect to the number of processors and processor kinds, we consider it *not* retargetable with respect to task partition and architecture change since the programmer should manually optimize the parallel code considering the specific target architecture and design constraints. Another difficulty of programming with MPI and OpenMP is that it is the programmer's responsibility to confirm satisfaction of the design constraints, such as memory requirements and real-time constraints. The current practice of embedded software design is multithreaded programming with lock-based synchronization, considering all target specific features. Thus, the same application should be re-written if the target is changed. Moreover it is well-known that debugging and testing a multithreaded program is extremely difficult.

In order to increase the design productivity of embedded software, we propose a parallel programming model called common intermediate code (CIC), based on which the HOPES design flow is defined as shown in Figure 2 [6]. In a CIC, the *potential* functional and data parallelism of application tasks are specified independently of the target architecture and design constraints. CIC tasks are concurrent tasks communicating with each other through channels. CIC tasks can be automatically generated from other front-end program specifications or manually written by a programmer.

Information on the target architecture and the design constraints is separately described in an xml-style file, called the *architecture information file*. Based on this information, the programmer maps tasks to processing components, either manually or automatically. Then, the CIC translator automatically translates the task codes in the CIC model into the final parallel code, following the partitioning decision. The CIC translation involves synthesizing the interface code between tasks and a run-time system that schedules the mapped tasks, extracting the necessary information from the architecture information file needed for each translation step. Based on the task-dependency information that tells how to connect the tasks, the translator determines the number of inter-task communication channels. Based on the period and deadline information of tasks, the run-time system is synthesized.

The CIC translator is the key ingredient that makes the CIC tasks truly retargetable with respect to architecture change and partitioning decision. As a preliminary experiment, we have designed a CIC translator for the Cell processor with an H.264 encoding algorithm as an example [7]. From the same CIC specification, we also generated a parallel program for an MPCore processor that is a symmetric multi-processor, which confirms the retargetability of the CIC model.

There are many issues to be researched further in the future, which include optimal mapping of CIC tasks to a given target architecture, exploration of optimal target architecture, and optimizing the CIC translator for specific target architectures. In addition, we have to extend the CIC to improve the expression capability of the model.

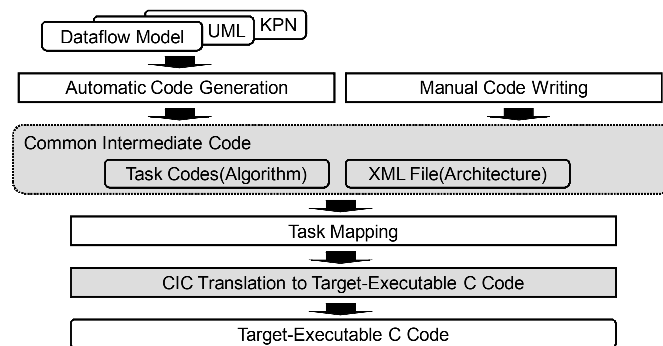

Figure 2: HOPES design flow for retargetable embedded software design

VI. DESIGNER-CONTROLLED RECODING FOR MULTI-CORE PARALLELIZATION

To overcome the complexity in MPSoC design, researchers have developed sophisticated design flows that significantly reduce the development time through automation. While much work has focused on synthesis and exploration tools, little has been done to support the designer in the critical design specification phase. In fact, our studies on industrial size examples have shown that about 90% of the system design time is spent on coding and re-coding of MPSoC models even in the presence of algorithms available as C code. Moreover, for programming heterogeneous multi-core systems, existing automatic parallelization techniques are insufficient. Most embedded applications need to be restructured and partitioned manually by the designer, a tedious, error-prone, and lengthy coding process.

Authorized licensed use limited to: UNIVERSITEIT TWENTE. Downloaded on January 27, 2010 at 09:20 from IEEE Xplore. Restrictions apply.

To overcome this modeling and parallelization bottleneck, we have developed a novel designer-controlled approach [8] to *recode* applications written in a C-based SLDL [9]. Our *Source Recoder* specifically addresses the automation gap in creating structured, parallel, flexible and analyzable SoC models starting from C reference models. We aim at resulting models that support explicit parallelism (both data and functional parallelism), clean structural and behavioral hierarchy, clear separation of computation and communication, and static analyzability without ambiguities resulting from pointers and irregular code structure.

Our recoder is designer-controlled and based on interactive transformations. Unlike traditional automatic compilers, we use interactive source-level transformations which can be chained together by the designer to expose desired properties in the model through code restructuring.

For example, to expose explicit data parallelism in the model, the designer uses her/his application knowledge and invokes re-coding transformations to split loops into code partitions, analyze shared data accesses, split vectors of shared data, localize variable accesses, and finally synchronize accesses to shared data by inserting communication channels. Further, similar code partitioning and data structure re-structuring transformations can be used to expose pipelined parallelism in the model. Additionally, code restructuring to prune the control structure of the code and pointer recoding to replace pointer expressions can be used to enhance the analyzability and synthesizability of the models.

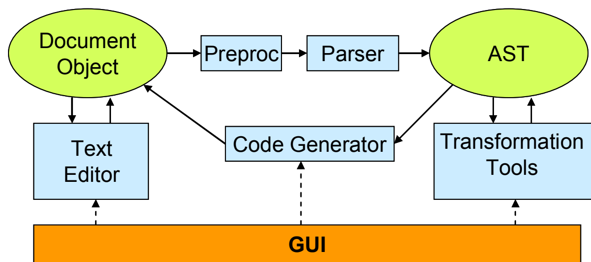

Figure 3: Designer-controlled Source Recoder [8]

Our Source Recoder is an intelligent union of editor, compiler, and transformation and analysis tools. The conceptual organization of the source recoder is shown in Figure 3. It consists of a Text Editor maintaining a Document Object and a set of Analysis and Transformation Tools working on an Abstract Syntax Tree (AST) of the design model. Preprocessor and Parser apply changes in the document to the AST, and a Code Generator synchronizes changes in the AST to the document object. The invoked analysis and transformation tasks are performed and presented to the designer *instantly*. The designer can also make changes to the code by typing and these changes are applied to the AST *on-the-fly*, keeping it updated all the time.

Unlike other program transformation tools, our approach provides complete control to the designer to generate and modify the design model suitable for the design flow. As such, we rely on the designer to concur, augment or overrule the analysis results of the tools, and use the combined intelligence of the recoder and the designer for the modeling task.

In summary, using automated source code transformations built into a model-aware editor, the designer can quickly recode the system model to create a parallel and flexible specification that can be easily mapped onto a multi-core platform. Our experimental results show a great reduction in modeling time and significant productivity gains up to two orders of magnitude over manual recoding.

VII. Debugging with Virtual Platforms

No matter what approach software engineers are taking to exploit the parallel processing power of homogeneous multi-core platforms, or to develop firmware for heterogeneous multimedia/wireless application subsystems, sooner or later they will be confronted with debugging. The defects that are being debugged may be the consequence of a problem in the hardware platform, a misinterpretation of a specification, or due to software design and implementation flaws. While writing efficient software for multi-core platforms/subsystems is getting more complex with the increasing parallelism, the same applies for the task of software debugging. In contrast to sequential software, concurrent software is characterized by a much larger number of failure modes. System deadlocks, race conditions and starvation are just a few to be mentioned here.

On top of this, debugging concurrent software does not take place just within the software layer. Since no sufficient and scalable multi-core hardware abstraction has yet matured, debugging parallel software often requires going far below the software layer, deep into the internals of platform hardware. Shared platform resources such as timers, interrupt controllers, DMAs, memory controllers, memories, semaphores may not be controlled anymore by single software stack. Removing defects in such an environment requires following a structured debugging process that is characterized by the following phases: (1) Triggering and recognizing the defect, (2) Reproducing the defect, (3) Locating the problem symptom, and (4) Locating and removing the root cause. However, some characteristics of traditionally used hardware prototypes prevent from conducting such a structured process, resulting in inefficient, expensive ad-hoc debugging. Debugging using real hardware is typically intrusive, which means that debugging impacts the system behaviour by itself. The resulting non-determinism is especially impacting the phases 2-4. The so-called "Heisenbug" is a prominent artefact of intrusive debugging. Those kinds of bugs disappear as soon as debugging is performed, since debugging can impact the sequence of operations within an MPSoC. This is because debuggers typically cannot halt the entire system. While the core under debug is stalled, other cores or timers continue to operate. Even worse, the lack of a consistent visibility into the core and peripheral registers and signals results in guessing about what is going on, rather than in a systematic analysis of the problem.

A *virtual* hardware platform overcomes those problems. A virtual platform is functionally accurate simulator of a SoC that executes exactly the same binary software that the real

Authorized licensed use limited to: UNIVERSITEIT TWENTE. Downloaded on January 27, 2010 at 09:20 from IEEE Xplore. Restrictions apply.

hardware executes. Using a virtual platform the entire system can be synchronously suspended from execution. This non-intrusive system suspension does not impact the system behaviour, as the system can resume the operation without recognizing that it has been halted. During a system suspend, a virtual platform provides a consistent view into the state of all cores and peripherals. Here, not only memory mapped registers can be inspected, but all peripheral registers and even signals. A watchpoint can be set on a signal, such as the interrupt line of a peripheral. The system execution will suspend when the signal is asserted. Afterwards, the execution of the interrupt handling routines can be inspected step by step on each core. Using real hardware, the peripheral interrupt may not be recognizable by the developer, as it may be wrongly masked. Peripheral access watchpoints allow suspending execution when a specific core or DMA is writing to a shared resource. Illegal access to memories or peripherals such as reported in [10], or race conditions on a shared memory access can be easily identified.

CoWare Virtual Platforms provide a scriptable debug framework. Using a TCL based scripting language, the control and inspection of hardware and software can be automated. This scripting capability allows implementing system level software assertions, without changing the software code. System level software assertions enable the assertion of system level fault conditions. Those assertions can take the state of the entire system into account, which is defined by multiple cores, their software tasks, memories and peripheral registers. Correctness and performance of complex shared-memory communication, task scheduling and control can be asserted. The hardware and software tracing capabilities address another major problem of multi core software development – the ability to keep the overview during debugging. A history of function execution within the different processes, and their access to memories and peripherals, is of great help to understand and identify the cause of a defect. Summarizing, virtual platforms allow following a structured and systematic debugging process, resulting in a significant quality and productivity gain for the software engineer.

VIII. Conclusions

From the above discussions, several important research problems in MPSoC programming become obvious. First, will the future show more homogenous platforms (for sake of simplified SW development) or heterogeneous architectures (for highest energy efficiency)? Second, the importance of considering realtime constraints has been emphasized, which is a key differentiator between embedded and traditional high-performance computing. Furthermore, sample projects from academic research such as MAPS, HOPES, and Source Recoder, indicate that systematic progress is being made in tackling some key MPSoC programming challenges. Last but not least, practical programming always goes hand in hand with debugging, and this topic may deserve more attention as advanced parallelizing compilation tools become more and more widespread.

[1] J. Ceng, J. Castrillon, W. Sheng, H. Scharwaechter, R. Leupers, G. Ascheid, H. Meyr, T. Isshiki, H. Kunieda: MAPS: An Integrated Framework for MPSoC Application Parallelization, 45th Design Automation Conference (DAC), Anaheim (USA), Jun 2008

[2] UMIC: www.umic.rwth-aachen.de

[3] H. Kopetz: Event-Triggered versus Time-Triggered Real-Time systems, *Proc. Int. Workshop on Operating Systems of the 90s and Beyond*, Berlin(Germany), 1991

[4] A. Hansson, K. Goossens, M. Bekooij, J. Huisken: CoMPSoC: A Composable and Predictable Multi-Processor System on Chip Template, *ACM Transactions on Design Automation of Electronic Systems*, to appear

[5] M. Wiggers, M. Bekooij, P. Jansen, and G. Smit. Efficient Computation of Buffer Capacities for Cyclo-Static Real-Time Systems with Back-Pressure. In *Proc. IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2007.

[6] S. Kwon, Y. Kim, W. Jeun, S. Ha, and Y. Paek, "A Retargetable Parallel-Programming Framework for MPSoC", ACM Transactions on Design Automation of Electronic Systems (TODAES) Vol. 13 No. 3, Article 39, July 2008

[7] K. Kim, J. Lee, H. Park, and S. Ha, "Automatic H.264 Encoder Synthesis for the Cell Processor from a Target Independent Specification", in Proc. 6th IEEE Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia) Oct. 2008

[8] P. Chandraiah, R. Dömer: *"Code and Data Structure Partitioning for Parallel and Flexible MPSoC Specification Using Designer-Controlled Recoding"*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 27, no. 6, pp. 1078-1090, June 2008.

[9] D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, S. Zhao, *"SpecC: Specification Language and Methodology"*, Kluwer Academic Publishers, Boston, March 2000

[10] Jens Braunes, pls Development Tools: Multi-Core - A New Challenge for Debugging, EDA DesignLine, 14th October 2008

Authorized licensed use limited to: UNIVERSITEIT TWENTE. Downloaded on January 27, 2010 at 09:20 from IEEE Xplore. Restrictions apply.