A Flexible Analytic Model for a Dynamic Task-Scheduling Unit for Heterogeneous MPSoCs

Oliver Arnold, Benedikt Noethen, and Gerhard Fettweis
Vodafone Chair Mobile Communications Systems
Technische Universität Dresden
Dresden, Germany
{oliver.arnold, benedikt.noethen, fettweis}@tu-dresden.de

Abstract— **In this paper, a heterogeneous Multiprocessor System-on-Chip (MPSoC), controlled by a dedicated task scheduling unit, is presented. This unit, known as CoreManager, is responsible for dynamic data-dependency checking, task scheduling, processing element allocation and data-transfer management. Three different CoreManager approaches are analyzed and compared. An analytical model is derived for each CoreManager implementation. The configuration parameters for the models are determined through system analysis. For this purpose, a tool flow has been developed to build the MPSoC and generate data traces. For the benchmarks employed, the relative error of the analytical model was shown to be lower than 6.3 % on component and 6.9 % on system level compared to the measurements.**

Keywords-Heterogeneous MPSoC, Dynamic Task Scheduling, CoreManager, Analytical Model

I. INTRODUCTION

Multiprocessor System-on-Chips (MPSoCs) are composed of several types and numbers of processing elements (PEs) and allow increasing performance and energy efficiency. In order to cope with the stringent performance-efficiency requirements, architectures exploiting parallelism and data locality both at system and core level [1] are required. Even though data-level and instruction-level parallelism within the PEs is essential, the main focus of this work is in the functional, i.e., task-level parallelism, based on the data flow model [2].

Increasing the system complexity in terms of application parallelism and number and types of resources may lead to a dramatic increase of system management costs, thus causing performance degradation. For this reason, the efficient implementation of the management unit becomes a major issue in system design. Therefore, an analytical model is necessary to predict and analyze the runtime behavior of the management unit and the heterogeneous system.

This work compares the performance and capabilities of a dedicated task scheduling unit, called CoreManager. Three different implementation approaches are regarded: a RISC-based solution (CM-RISC), an approach with Very Long Instruction Words (CM-VLIW) and an implementation based on an extended instruction set architecture (CM-EIS). A flexible analytical model has been derived for each implementation approach. Furthermore, a tool flow has been developed to build a heterogeneous MPSoC and to generate data traces. The configuration parameters for the models have been analytically derived and the obtained results compared to the measurements.

Some examples of heterogeneous hardware platforms are the Cell Broadband Engine [3] and Sandbridge SB3011 SDR platform [4]. The Tomahawk MPSoC was developed to execute applications from the multimedia as well as the signal processing domain [5]. It includes a dedicated task scheduling unit. In [6], a comparison between a software and a hardware scheduling approach is presented. The programming model used in this work is similar to CellSs [7]. Further programming models are, e.g., Cilk [8], Sequoia [9], and Ct [10].

The extension of the instruction set of standard processors is available in many areas [11][12]. In this work, a RISC core is extended by several newly introduced instructions to improve task scheduling performance as well as energy consumption. A similar approach was presented in [13].

According to the taxonomy given in [14], the used dynamic task scheduling is centralized and applies complete information exchange to schedule aperiodic tasks. Complete information exchange refers to the collection of events from all processing elements. The platform used in this work can be understood as a distributed system due to the separate address spaces of the processing elements [15].

The remainder of the paper is organized as follows. In section II, the hardware system and the programming model are presented. In the following section, the tool flow is described. Section IV presents the components of the task scheduling unit, called CoreManager. It is analytically described in the next section. Section VI shows the results of the system. The parameters of the analytical models are presented. Furthermore, a comparison of the analytical model and the measurements is given.

II. SYSTEM MODEL

A. Hardware Model

A heterogeneous MPSoC is depicted in Fig. 1. It consists of several functional blocks, which are connected by a Network-on-Chip (NoC). A router is available for each system component, which is connected to its neighbors by point-to-point data links. The routers are responsible for

 ISBN: 978-1-61208-308-7

packet scheduling and arbitration. XY routing is applied. Further details about the integrated NoC can be found in [16].

The Application Processor (APP) is formed by a Tensilica 570t core and has 2-way set-associative instruction and data caches, each 16 Kbyte in size. It is placed next to an off-chip memory interface for fast data access. The data plane of the system is composed of several types and numbers of processing elements (PEs), which are controlled by the CoreManager. The CoreManager is responsible for task scheduling, PE allocation, and data transfer management. The CoreManager presents an interface which allows connecting to the application running on the APP.

Three off-chip global memories are included (MEM_0, MEM_1 and MEM_2), each one having 256 MB. Each PE has its own dedicated direct memory access controller (DMAC) to perform data transfers between the global memories and their local memories. Furthermore, data can be fetched from local memories of other PEs.

Two types of PEs are integrated in the system: a digital signal processor (DSP) and a RISC processor. For each type, ten processors are instantiated. In the proposed approach, a PE can solely operate on its local on-chip memory. No cache misses can occur. Task execution time is consequently deterministic, which leads to a better predictability at system level. PEs' instruction and data memory size is 32 Kbyte each. Prefetching of data is possible for the next two tasks, but must be explicitly annotated by the CoreManager. Similar to the PEs, the CoreManager solely works on local on-chip memories. Its instruction and data memory size are 32 Kbyte each. Data transfers to the local memories of the PEs and task execution can be performed concurrently. A clock frequency of 333 MHz is applied for all components.

B. *Programming Model*

The used programming model, called taskC, is based on tasks as a main entity [15]. A task is a collection of instructions which are atomically executed. In Fig. 2, a source code example is shown. For each task input and output, data transfers are specified with IN, OUT and INOUT operators. For each transfer, a pointer and a size are specified at runtime. 2-dimensional data transfers are supported. For example, in software defined radio systems, the data locations of a task are specified after the header is processed. No static data analysis is possible for these kinds of applications.

The task execution is not done by the APP itself. The APP only sends the task description, which is composed of the task name and the data information, to the CoreManager. In Fig. 2, two task descriptions are transferred, either taskType1 and taskType2 or taskType1 and taskType3. The APP is additionally responsible for evaluating control-code dependencies, e.g., the if-else clause in Fig. 2. Data-dependencies between tasks are evaluated by the CoreManager at runtime. The taskSync command is a barrier and synchronizes the APP and the data plane execution. After the APP returns from this function it is assured that all tasks are finished and all output transfers have been completed.

III. TOOL FLOW

A newly developed tool flow is used to specify the system configuration and to generate the simulation environment. An overview of all components is shown in Fig. 3. The hardware architecture is specified in a configuration file containing two parts. The first part is responsible for the system level. The second part specifies the capabilities of the CoreManager. By using the Tensilica Xtensa Processor Generator (XPG) the CoreManager as well as the PEs are created [17]. RTL code and suitable Compilers are generated as well. The InstGenerator and the TaskCompiler are responsible for the compilation of tasks and their extraction into a separate data array. The application itself is compiled with the Tensilica 570t Compiler. Binaries for the PE and the CoreManager are linked into the APP binary. These binaries are loaded at runtime to the corresponding cores.

Three types of hardware designs are generated: A Tensilica-based cycle-accurate simulation environment (XTSC), a FPGA prototype, and an ASIC prototype. The TaskVisualizer allows visualization of results. In particular, it shows task execution and data transfers. More information on the TaskVisualizer can be found in [15]. The CoreManager Profiler and the DebugVisualizer allow an offline and online analysis of the CoreManager. More information on these tools can be found in [18].

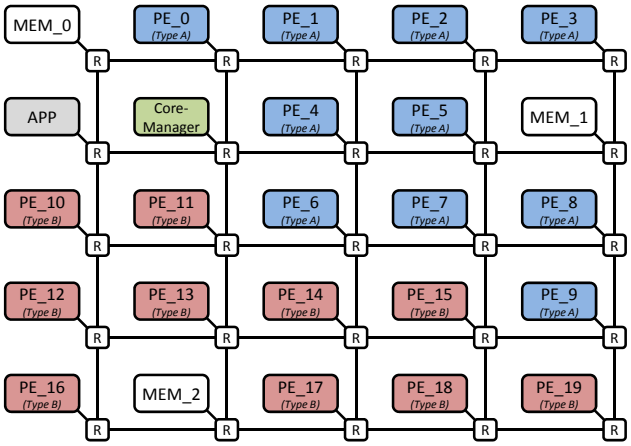


Figure 1. System Model: heterogeneous MPSoC

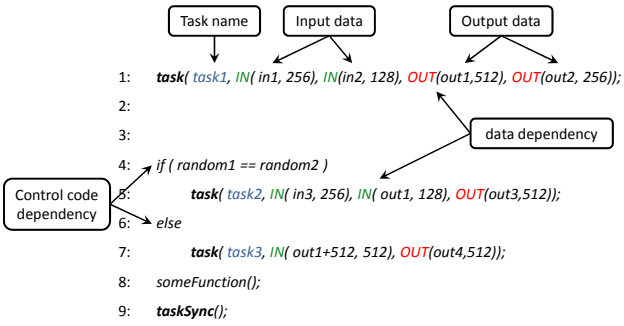


Figure 2. Programming model example

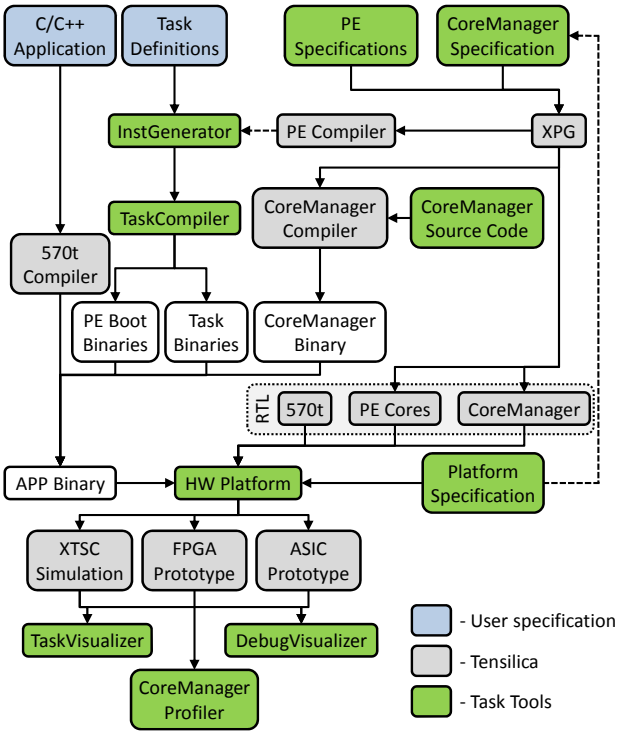


Figure 3. Tool flow

IV. CoreManager Structure and Behavior

The major components and the internal data flow of the CoreManager are depicted in Fig. 4. The operational sequence is as follows. Firstly, the APP retrieves the ID of an empty task slot by reading the CM_2_APP first-in first-out (FIFO) memory (step 1). Afterwards, the APP writes the task description (e.g., the task name and the input and output data) to the task buffer in the corresponding task slot (step 2). As soon as writing the task buffer is finished, the same task slot ID is written to the APP_2_CM FIFO (step 3). The CoreManager reads this FIFO. It firstly performs a data-dependency checking among all tasks which are currently in the system. For this purpose, (1) must be evaluated for each transfer for all tasks. The array formed by pointer p_1 and size s_1 of task 1 is compared with the array formed by p_2 and s_2 of task 2. p_1, p_2, s_1 and s_2 are assumed to be greater or equal to zero. The equation is valid if a dependency is found.

$$dep = (unsigned)(p_1 - p_2) < s_2 \;\|\; (unsigned)(p_2 - p_1) < s_1 \quad (1)$$

In the particular case of the CM-EIS processor, the operations shown in (1) are merged into one instruction, which is thus executed in a single clock cycle. Furthermore, the application of 4-SIMD vectorization enables the execution of four parallel dependency checks. A more detailed explanation of the dynamic data-dependency checking of the CM-EIS processor can be found in [18].

If no data-dependency is found, the task is included in the *ready* task list. Otherwise, the task is annotated at the corresponding preceding tasks descriptions (step 4-6).

In the next step, the task-scheduling module selects the most suitable task from the *ready* task list (step 7). Two scheduling approaches are currently available. An *as-soon-as-possible* scheduling approach prioritizes the tasks according to their time of arrival in the CoreManager. The second possibility is an *earliest-deadline-first* approach, which favors tasks with the closest deadline. The scheduling is only performed if a suitable PE is available for the task.

After the scheduling process, a PE is allocated and local memory for the necessary data is reserved (steps 8-9). The implemented PE allocation approach is depicted in Fig. 5. The PE allocation is based on two bit masks: One corresponding to the PEs currently available and one corresponding to the PEs annotated as suitable for a task. The number of PEs determines the number of necessary bits (a dedicated bit is reserved for each PE). A value of one represents an available or suitable PE. An *AND* operation is performed on the bit masks representing the currently available and the suitable PEs. The PE associated to the first bit with a value of one (i.e., the first available and suitable PE) is subsequently allocated. In addition to this, for each task type the preferred and suitable PEs can be specified. The implemented PE allocation approach prioritizes preferred PEs accordingly. In order to increase data locality, a task can be scheduled on the same PE as its predecessor task, thus allowing the reuse of its output data. The number of memory transfers is hence reduced and the performance is improved.

A StartupUp Code is subsequently generated (step 10) by the CoreManager. It contains all necessary information to configure the PE (e.g., pointers to the instruction code) and all task data. It is transferred by two additional DMACs situated next to the CoreManager (step 11-12).

As soon as the task is finished, a packet is sent over the NoC and stored in the PE Finished FIFO (step 13). The CoreManager can evaluate this information (14-16). All successors of the executed tasks are put in the *ready* task list if no further dependencies are annotated (step 17). Finally, the corresponding task slot is made available for the APP by writing the task slot ID in the CM_2_APP FIFO (step 18).

 ISBN: 978-1-61208-308-7

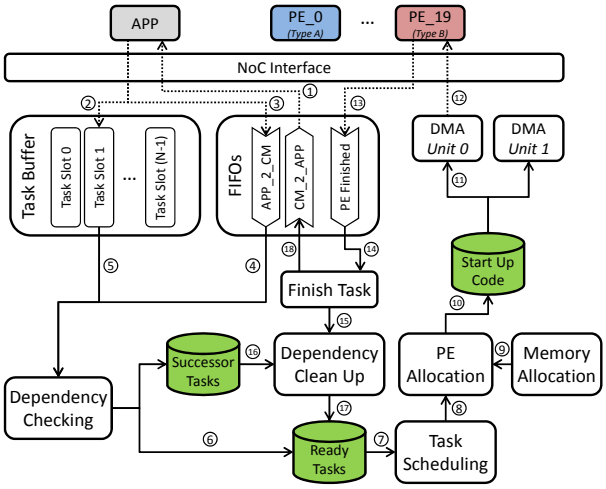


Figure 4. CoreManager structure and data flow

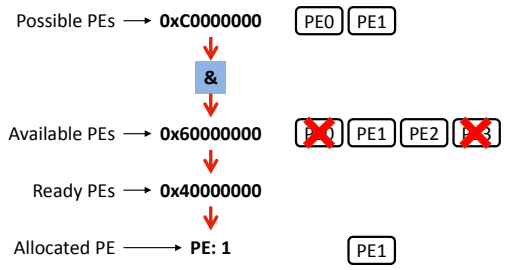


Figure 5. PE allocation

In the following, three versions of the CoreManager are compared and analyzed. The first one, called CM-LX4, is based on a Tensilica LX4 RISC core. The second solution integrates a Very Long Instruction Word approach in the CoreManager (CM-VLIW). The third version extends this processor with an improved instruction-set architecture especially suitable for the needs of a task scheduling unit (CM-EIS).

In Table I, the newly introduced instructions for the task scheduling are shown and shortly described. 16 task slots are always concurrently processed and can be hence evaluated in a single clock cycle. For each task slot, a validity bit is present. If it is set to a value of one the corresponding task slot is valid and can be used for the evaluation. The evaluation of the valid bit is included in the processing time of one clock cycle. In Fig. 6, the new instruction for finding the smallest values out of the *ready* task list is additionally shown. In this example, a minimum operator is applied. Nevertheless, it can be adapted at runtime to determine the maximum value. For each task slot, a 16-bit value must be defined. It can be flexibly used to specify, e. g., deadlines and priorities. The evaluation of all task slots is done in parallel.

TABLE I. TASK SCHEDULING INSTRUCTIONS

Instruction	Explanation
SCHED_SET(slot, val)	Value of a specified slot is set.
SCHED_SET_ALL(val)	All tasks slots are set to a specific value.
SCHED_MIN(slot, val)	Retrieves the smallest valid task slot. The task slot ID and its value are returned.
SCHED_MAX(slot, val)	As above, but the highest value is returned.
SCHED_INC(val)	Adds a value to all task slots. Task slot values saturates at 65535.
SCHED_DEC(val)	Subtracts a value from all task slots. Task slot values saturates at 0.

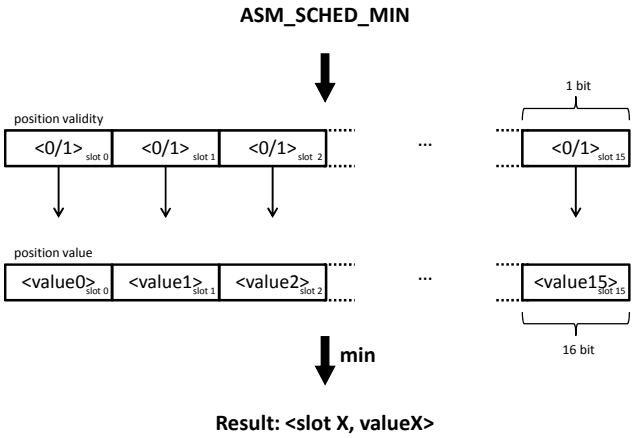


Figure 6. Instruction set architecture extension for task scheduling

V. ANALYTICAL MODEL

The developed analytical model depends on the following input parameters:

- n_{IN} : Number of input transfers
- n_{OUT} : Number of output transfers
- $n_{Transfers}$: $n_{IN} + n_{OUT}$
- n_{TIS} : Tasks currently available in the system
- n_{PE-ID} : Allocated PE number (starting with 0)
- $n_{Successor}$: Number of successor tasks

In the following, all components of the CoreManager will be analyzed and described using analytical methods. The timing is described in clock cycles. The CoreManager solely works on its local memory. Consequently, no external memory accesses are required and its processing time is hence independent of the clock frequency of the remaining system.

A. Dynamic Data-Dependency Checking

Equation (2) describes the necessary time for the dynamic data-dependency checking stage on the CM-LX4 and CM-VLIW processors. A quadratic dependence on the

number of output transfers is present. IN-IN transfer comparisons are not performed. In the case of independent tasks, no data dependencies have to be checked. Thus, $t_{Dep,\mathrm{var}}$ can be directly set to 0.

$$t_{DepCheck} = t_{Dep,init} + t_{Dep,TIS} * n_{TiS} + t_{Dep,Transfer} * n_{TiS} * (2 * n_{IN} + n_{OUT}) * n_{OUT} \quad (2)$$

In the case of the CM-EIS processor, the processing time of the dynamic data-dependency checking can be described by (3). IN and OUT transfers are not distinguished. Nevertheless, IN-IN transfers are not considered as a dependency. A quadratic dependence on the number of transfers is present.

$$t_{DepCheck,CM-EIS} = t_{Dep,init} + t_{Dep,TIS} * n_{TIS} + t_{Dep,Transfer} * n_{TIS} * \left(n_{Transfers}\right)^2 \quad (3)$$

B. Task Scheduling

The task scheduling finds the most suitable task from the *ready* task list. In the case of the CM-LX4 and CM-VLIW processor, the *ready* task list is sequentially searched for the smallest or largest value. For each task slot, the valid bit is evaluated. Equation (4) can be used to describe the processing time of this stage.

$$t_{Scheduling} = t_{Scheduling,const} + t_{Scheduling,\mathrm{var}} * n_{TIS} \quad (4)$$

In the case of the CM-EIS processor, the task scheduling time for up to 16 task slots is constant. Hence, (4) can be transformed to (5).

$$t_{Scheduling,CM-EIS} = t_{Scheduling,const} + t_{Scheduling,\mathrm{var}} * \left\lceil \frac{n_{TIS}}{16} \right\rceil \quad (5)$$

C. PE Allocation

Equation (6) determines the necessary time for the PE allocation.

$$t_{PE-Alloc} = t_{PE-Alloc,const} + t_{PE-Alloc,\mathrm{var}} * n_{PE-Id} \quad (6)$$

For the CM-EIS core up to 32 PEs can be evaluated in a single cycle. Hence, (6) can be modified as:

$$t_{PE-Alloc,CM-TIS} = t_{PE-Alloc,const} + t_{PE-Alloc,\mathrm{var}} * \left(1 + \left\lfloor \frac{n_{PE-Id}}{32} \right\rfloor\right) \quad (7)$$

D. Local Memory Allocation

Three different allocation approaches for the local memories are available. The *single-space* allocation occupies the whole memory for one task. The *top-down* allocation allows two tasks to use the same local memory. The most sophisticated mode of operation is the *block-based* allocation. The whole local memory is divided in equally sized blocks. In this case, eight blocks are used. The necessary processing time for the allocation of local memory is determined by (8).

$$t_{Mem-Alloc} = t_{Mem-Alloc,Init} + t_{Mem-Alloc,Transfer} * n_{Transfers} \quad (8)$$

E. DMAC configuration

The configuration time of the DMACs for transferring the Start Up Code is always the same. It can be described with (9).

$$t_{DMAC-Config} = t_{DMAC-Config,const} \quad (9)$$

F. Clean Up

The processing time after a task is finished depends on the number of successors per tasks. Additionally, the task slot ID must be written to the CM_2_APP FIFO. The processing time of the Clean Up stage can be expressed with (10).

$$t_{Clean-Up} = t_{Clean-Up,const} + t_{Clean-up,Successor} * n_{Successor} \quad (10)$$

G. System Level

A combination of the processing times of the components of the CoreManager leads to a system-level latency point of view. The processing time of the CoreManager for each part can be separated in a processing time before and after task execution. Equation (11) describes this behavior.

$$t_{Task-\mathrm{Proc}} = t_{Task-Start} + t_{Task-End} \quad (11)$$

Both terms on the right side of (11) can be individually expressed with (12) and (13), respectively. By using these equations it is possible to predict the performance of the CoreManager and determine its influence on the system.

$$t_{Task-Start} = t_{DepCheck} + t_{Scheduling} + t_{PE-Alloc} + t_{Mem-Alloc} + t_{DMAC-Config} \quad (12)$$

$$t_{Task-End} = t_{Clean-Up} \quad (13)$$

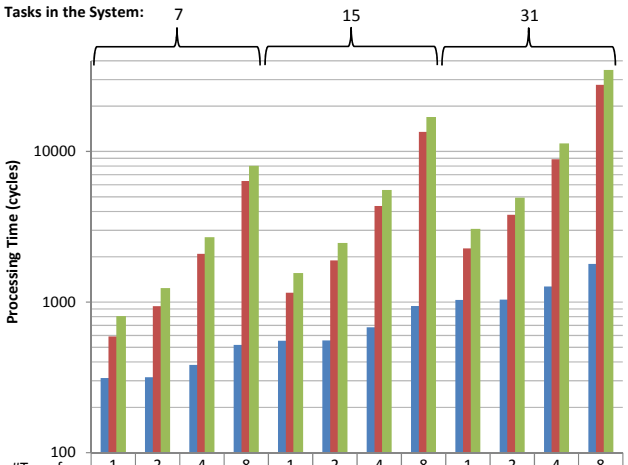
VI. RESULTS

In the first part of this section, the measured results of the CoreManager components are presented. Configurable task descriptions are used to measure the processing time. Especially corner cases are regarded. The FPGA prototype is used for all measurements. The integrated DebugUnit is responsible for generating traces at runtime. The DebugUnit is a dedicated component placed next to the CoreManager. It is used to observe the dynamic decisions of the CoreManager. The analysis of the traces is done with the DebugVisualizer. The processing time of the CoreManager components is deterministic due to the instruction and data fetch solely from its local memories. The same input leads to the same result and the same processing time. Due to this deterministic behavior, the presented results are valid for RTL and Netlist simulation as well as the ASIC prototype.

 ISBN: 978-1-61208-308-7

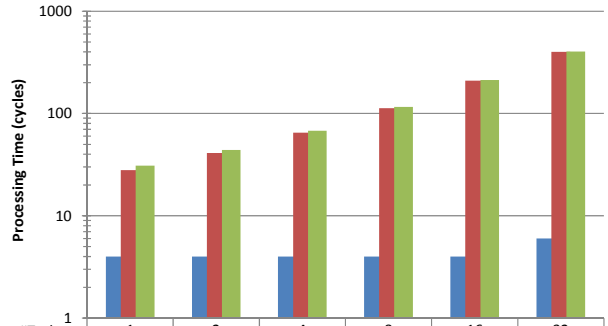
In the second part of this section, the previous results are analyzed to obtain the parameters of the analytical model. The last part of this section presents a comparison of the analytical model with the measurements of real applications.

In Fig. 7, the results of the dynamic data-dependency checking stage are depicted. All transfers are divided in 50 % input and 50 % output transfers. In the case of one transfer, an INOUT type is used. In the analytical model an INOUT transfer is regarded as an OUT transfer. n_{TIS} is varied between 7, 15, and 31. The number of transfers is set to 1, 2, 4, and 8. A difference in the processing time of over one order of magnitude can be observed between the CM-LX4 and the CM-EIS CoreManager. In Fig. 8, the processing time of the task scheduling is shown. The number of tasks in the *ready* task list is varied between 1 and 32. In Fig. 9, the results for the PE allocation are depicted. It is distinguished between the annotation of possible and possible/preferred PEs per task. The results for the local memory allocation are shown in Table II. The processing time depends on the already allocated blocks and on the number of transfers. The configuration of the DMA controller of the CoreManager needs a constant processing time of 12 cycles per task. In Table III, the processing time of the Clean Up stage is shown. For each successor task the necessary time is increased.



#Transfers	1	2	4	8	1	2	4	8	1	2	4	8
CM-EIS	313	317	383	519	553	557	679	943	1033	1037	1271	1791
CM-VLIW	592	939	2093	6373	1152	1891	4349	13493	2272	3795	8861	27733
CM-LX4	806	1239	2693	8036	1558	2471	5557	16948	3062	4935	11285	34772

Figure 7. Dynamic data-dependency checking results. The available tasks in the system and the number of transfers are varied.



#Tasks	1	2	4	8	16	32
CM-EIS	4	4	4	4	4	6
CM-VLIW	28	41	65	113	209	401
CM-LX4	31	44	68	116	212	404

Figure 8. Task Scheduling results. The number of tasks in the ready list is varied.

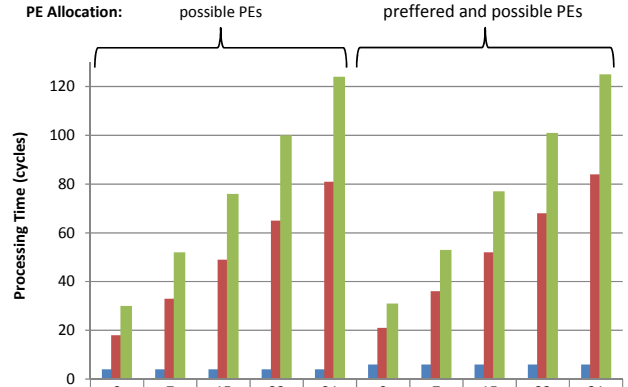
TABLE II. LOCAL MEMORY ALLOCATION: PROCESSING TIME (IN CYCLES)

Avail. Blocks	#Transfers	CM-EIS	CM-VLIW	CM-LX4
0x0	2	10	51	66
	4	20	56	78
	8	34	64	92
0x1	2	10	51	65
	4	20	56	77
	8	34	64	91
0x3	2	10	52	67
	4	20	57	79
	8	34	65	93
0x7	2	10	56	72
	4	20	61	84
	8	34	69	98
0x9	2	10	55	62
	4	20	55	74
	8	34	63	88
0x12	2	10	53	60
	4	20	53	72
	8	34	61	86

TABLE III. CLEAN UP: PROCESSING TIME (IN CYCLES)

#Successor Tasks	CM-EIS	CM-VLIW	CM-LX4
1	44	124	190
2	54	150	228
4	72	186	276
8	108	306	400

 ISBN: 978-1-61208-308-7



PE-ID	0	7	15	23	31	0	7	15	23	31
CM-EIS	4	4	4	4	4	6	6	6	6	6
CM-VLIW	18	33	49	65	81	21	36	52	68	84
CM-LX4	30	52	76	100	124	31	53	77	101	125

Figure 9. PE allocation results. PE-ID is varied.

In order to obtain the parameters described in section V the minimum mean square error is used. The resulting values for all parameters are presented in Table IV. The superior performance of the CM-EIS core, which was already observed in the first part of this section, is also noticed here. Especially $t_{Dep,Transfer}$, $t_{Scheduling,const}$, $t_{Scheduling,\mathrm{var}}$, $t_{PE-Alloc,const}$ and $t_{Mem-Alloc,Init}$ are significantly lower compared to those of the CM-LX4 and CM-VLIW cores. In the case of the local memory allocation, the parameter $t_{Mem-Alloc,Transfer}$ of the CM-VLIW core is smaller in comparison to the CM-EIS core due to constant processing time of the CM-EIS core. The data dependent processing time of the CM-VLIW core leads in average to a smaller value for parameter $t_{Mem-Alloc,Transfer}$. Nevertheless, the overall processing time of the CM-VLIW core is still 2 to 5 times higher (see Table II).

The corresponding relative errors are presented in Table V. The highest relative error corresponds to the dynamic data dependency checking stage. In the case of the CM-EIS core it is 6.3 %. These errors result from the data dependent execution time, .i.e., a dependency must be annotated at the predecessor of a task. Hence, an additional amount of processing time is needed. The DMAC configuration is for all cores perfectly predictable. Furthermore, the task scheduling and PE allocation models of the CM-EIS CoreManager have no error.

The relative errors for three real-world applications are depicted in Table VI. For each CoreManager approach the measured traces are compared with the prediction of the developed analytical models. The first two applications belong to the signal processing domain. In particular, the physical layer of a Global System for Mobile Communications (GSM) and Universal Mobile Telecommunications System (UMTS) are employed.

The third application is a JPEG decoding application. It decodes a picture with a resolution of 2560 by 1440 pixels. No data dependency checking is applied in the JPEG decoding application. Therefore, $t_{Dep,Transfer}$ is set to 0. Hence, no successor tasks are present in the Clean Up Stage. Each application is dynamically started several times.

All versions of the CoreManager have been synthesized with Synopsys Design Compiler for a 65 nm low power TSMC process using worst case conditions (125 °C, 1.08 V). For a target frequency of 333 MHz the occupied silicon area is 0.140 mm^2 (CM-LX4), 0.180 mm^2 (CM-VLIW) and 0.284 mm^2 (CM-EIS), respectively. Only logic area is evaluated, disregarding local memory area but including the memory interfaces (for timing correctness).

TABLE IV. MODEL PARAMETER

#Successor Tasks	CM-EIS	CM-VLIW	CM-LX4
$t_{Dep,init}$	118	107	110
$t_{Dep,TIS}$	32	68	92
$t_{Dep,Transfer}$	0.4	12.8	16.2
$t_{Scheduling,const}$	2.0	16.9	20
$t_{Scheduling,\mathrm{var}}$	2.0	12	12
$t_{PE-Alloc,const}$	2.0	22.0	32.0
$t_{PE-Alloc,\mathrm{var}}$	2.0	2.0	3.0
$t_{Mem-Alloc,Init}$	2	46.4	57.5
$t_{Mem-Alloc,Transfer}$	4	2.4	4.2
$t_{DMAC-Config,const}$	12.0	12.0	12.0
$t_{Clean-Up,const}$	34.9	99	161
$t_{Clean-up,Successor}$	9.2	25.5	29.7

TABLE V. MEAN RELATIVE ERROR COMPARED TO MEASURED VALUES FOR CONFIGURABLE TASKS FOR PARAMETER EXTRACTION

CoreManager Component	CM-EIS	CM-VLIW	CM-LX4
Data-Dependency Checking	6.3 %	3.6 %	2.5 %
Task Scheduling	0	0.8 %	0.9 %
PE Allocation	0	1.2 %	0.8 %
Local Memory Allocation	3.3 %	3.3 %	4.3 %
DMAC Configuration	0	0	0
Clean Up	0.6 %	2.4 %	1.3 %

TABLE VI. RELATIVE ERROR COMPARED TO MEASURED VALUES FOR REAL WORLD APPLICATIONS IN PERCENT (CM-EIS/CM-VLIW/CM-LX4)

CoreManager Component	Application		
	GSM	*UMTS*	*JPEG*
Data-Dependency Checking	3.6/3.2/3.9	6.9/4.3/2.6	0/0/0
Task Scheduling	0/0.2/0.4	0/0.3/0.6	0/0.3/0.3
PE Allocation	0/1.2/1.0	0/0.9/1.2	0/0.2/0.2
Local Memory Allocation	0/0/0	0/0/0	0/0/0
DMAC Configuration	0/0/0	0/0/0	0/0/0
Clean Up	0.4/0.2/0.4	0.3/0.9/0.4	0/0/0

 ISBN: 978-1-61208-308-7

VII. Conclusion and Future Work

In this paper, a central scheduling unit called CoreManager is analyzed. An analytical model has been derived from system analysis. A tool flow was introduced to generate the system and to obtain data traces. Parameters for all three CoreManager approaches have been derived from the analyzed data. It has been shown that the relative error on component level is less than 6.3 % compared to the measurements. On system-level with real application benchmarks, the relative error was shown to be lower than 6.9 %.

Future work aims at implementing a silicon prototype of the CoreManager in a heterogeneous MPSoC. Further optimizations of the architecture and the algorithms will be performed, especially regarding performance, area and power consumption.

Acknowledgment

The major part of this research work has been funded by the DFG through the cluster of excellence Center for Advancing Electronics Dresden and the European Union and the state of Saxony through the IMData project. A minor part was funded by the German Federal Ministry of Education and Research within the scope of the CoolBaseStations project.

Furthermore, we would like to thank Synopsys, Tensilica and Xilinx for sponsoring Software, IPs and prototyping FPGAs.

References

[1] K. Asanovic et al., “The landscape of parallel computing research: a view from Berkeley,” Technical Report UCB/EECS-2006-183, Electrical Engineering and Computer Sciences, University of California, Berkeley, Long Beach, CA, USA, Dec. 2006.

[2] E. A. Lee and D.G. Messerschmitt, “Synchronous data flow,” Proceedings of the IEEE, Vol.75, No.9, Sept. 1987, pp. 1235–1245, doi: 10.1109/PROC.1987.13876.

[3] C. R. Johns and D. A. Brokenshire, "Introduction to the Cell Broadband Engine Architecture," IBM Journal of Research and Development , Sept. 2007, vol.51, no.5, pp. 503-519.

[4] J. Glossner et al., "The sandbridge SB3011 SDR platform," Mobile Future, 2006 and the Symposium on Trends in Communications, SympoTIC '06, Joint IST Workshop, June 2006, pp. 2-5, doi: 10.1109/TIC.2006.1708006.

[5] T. Limberg et al., “A Fully Programmable 40 GOPS SDR Single Chip Baseband for LTE/WiMAX Terminals,” 34th European Solid-State Circuits Conference (ESSCIRC'08), Edinburgh, Great Britain, Sept. 2008, pp. 466-469, doi: 10.1109/ESSCIRC.2008.4681893.

[6] J. Lee, V. J. Mooney III, A. Daleby, K. Ingström, T. Klevin, and L. Lindh, “A comparison of the RTU hardware RTOS with a hardware/software RTOS,” ASP-DAC '03, Proceedings of the Asia and South Pacific Design Automation Conference, 2003, pp. 683-688, doi: 10.1109/ASPDAC.2003.1195108.

[7] P. Bellens, J.M. Perez, R.M. Badia, and J. Labarta, "CellSs: a Programming Model for the Cell BE Architecture," in SC’06, Proceedings of the Supercomputing conference, 2006, p. 86.

[8] M. Frigo, C. E. Leiserson, and K. H. Randall, “The implementation of the Cilk-5 multithreaded language,” Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, June 1998, pp. 212- 223, doi: 10.1145/277652.27772.

[9] K. Fatahalian et al., “Sequoia: Programming the memory hierarchy,” IEEE Conference on Supercomputing, 2006, p. 4, doi: 10.1109/SC.2006.55.

[10] A. Ghuloum, E. A. E. Sprangle, and J. Fang, “Flexible parallel programming for Terascale Architectures with Ct,” Intel Technology Journal, vol. 11, no. 3, Aug. 2007, pp. 185-196.

[11] A. Wang, E. Killian, D. Maydan, and C. Rowen, "Hardware/software instruction set configurability for system-on-chip processors," Design Automation Conference, 2001, pp. 184-188, doi: 10.1109/DAC.2001.156132.

[12] A. Chormoviti, N. Vassiliadis, G. Theodoridis, and S. Nikolaidis, "Enhancing Embedded Processors with Specific Instruction Set Extensions for Network Applications," Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications, 2005, IDAACS 2005, Sept. 2005, pp.199-203, doi: 10.1109/IDAACS.2005.282969.

[13] J. Castrillon, D. Zhang, T. Kempf, B. Vanthournout, R. Leupers, and G. Ascheid, “Task Management in MPSoCs: An ASIP Approach,” International Conference on Computer-Aided Design, San Jose, California, USA, 2009, pp. 587-594.

[14] H. G. Rotithor, "Taxonomy of dynamic task scheduling schemes in distributed computing systems," Computers and Digital Techniques, IEE Proceedings, Jan 1994, vol.141, no.1, pp.1-10, doi: 10.1049/ip-cdt.19949630.

[15] O. Arnold and G. Fettweis, "Power Aware Heterogeneous MPSoC with Dynamic Task Scheduling and Increased Data Locality for Multiple Applications," Embedded Computer Systems (SAMOS), 2010 International Conference on, July 2010, pp. 110-117, doi: 10.1109/ICSAMOS.2010.5642075.

[16] M. Winter and G. Fettweis, “Guaranteed Service Virtual Channel Allocation in NoCs for Run-Time Task Scheduling,” Proceedings of the Design Automation and Test in Europe (DATE'11), Grenoble, France, March 2011, pp. 1-6, doi: 10.1109/DATE.2011.5763073.

[17] Tensilica Inc., www.tensilica.com, since March 2013 Cadence (http://www.cadence.com) [retrieved August 2013]

[18] O. Arnold, B. Nöthen, and G. Fettweis, “Instruction Set Architecture Extensions for a Dynamic Task Scheduling Unit,” Proceedings of the IEEE Annual Symposium on VLSI (ISVLSI'12), Aug. 2012, pp. 249-254, doi: 10.1109/ISVLSI.2012.51.

 ISBN: 978-1-61208-308-7