CM_ISA++: An Instruction Set for Dynamic Task Scheduling Units for More Than 1000 Cores

Oliver Arnold, Benedikt Noethen, Gerhard Fettweis
Vodafone Chair Mobile Communications Dresden
Technische Universität Dresden
Dresden, Germany

Abstract—In this paper a dynamic task scheduling unit for many-core systems with over 1000 cores is introduced. It is called CoreManager. It dynamically schedules thousands of tasks of several applications, allocates processing elements, controls the prefetching of data transfers, and explicitly manages the on-chip memories. For many-core systems a high task throughput and a low latency are essential for its success. Therefore, the CoreManager integrates a newly developed application-specific instruction set, called CM_ISA+++, for a superior scheduling performance. The design of the instruction set of the CoreManager is presented, explained and the performance of each component is analyzed. Furthermore, the CoreManager is integrated and evaluated in a many-core system with 1008 processing elements. Our CoreManager implementation outperforms a RISC-based implementation by 193x in scheduling and 419x in processing element allocation performance. Consequently, scalability of the system as well as task throughput and latency is dramatically improved compared to RISC-based scheduling approaches.

II. APPLICATION SCENARIO

Fig. 1. Concurrently executed applications and their parallelism profiles.

A common approach for resource management employs static scheduling. For instance, a parallelism profile can be used to analyze the application and to obtain the required number of PEs as well as the space-time mapping for each task. In Fig. 1, an example is depicted. For each application a parallelism profile is generated. Nevertheless, due to fixed space-time mapping of the static task scheduling approach the concurrent execution with non-deterministic start times of several applications leads to poor system utilizations. In conclusion, the following application and task properties cannot be sufficiently handled by a static scheduling approach:

- Several types and numbers of applications concurrently running on the same hardware resources
- Non-predictable start times of applications
- Dynamic priorities on application level and task level
- Variable workload of applications due to unknown input parameters
- Non-deterministic task execution times

Dynamic data dependency checking eases the programming of the system, but introduces a major degradation in system performance [6]. Consequently, it cannot be applied for many-core systems. The task scheduling unit presented in [7] and [14] is extended with an application-specific instruction set. The chip implementation is presented in [8]. Nevertheless, it does not scale to over 32 cores. Furthermore, the number of applications and the number of tasks per application is limited. The hardware-based scheduling unit presented in [9] features a fixed scheduling approach and is only suitable for multi-core systems. In contrast, our work has a fully flexible scheduling approach and is capable to handle over 1000 processing elements due to the newly developed scheduling-specific instruction set. Furthermore, the scheduling algorithm is adaptable at runtime.

In [3] the OpenMP programming model is used to map tasks to the IBM Cyclops-64 architecture. 80 processors are available on a chip. In [4] a software-based runtime engine for many-core system is introduced. In none of the both approaches a dedicated task scheduling unit is used. Furthermore, performance metrics for the applied scheduling implementation are missing. In [5] a task scheduling unit with an application-specific instruction set is introduced. Nevertheless, the introduced overhead reduces the scalability of the implementation to 16 cores.
As a result of this observation, a dynamic task scheduling approach is applied in this work. It allows an efficient space-time mapping at runtime by considering the current system status.

III. HARDWARE SYSTEM MODEL

A many-core system with two hierarchy levels is used. The first level of hierarchy consists of 21 clusters (CLU). Each CLU is composed of 48 processing elements (PE). Hence, 1008 PEs are integrated. The top level is depicted in Fig. 2. All modules are connected by a two-dimensional 5x5 Network-on-Chip (NoC). XY-routing is applied between the routers (RT).

Further details about the NoC are presented in [11]. All NoC links are considered serial links, achieving a point-to-point connectivity with a throughput of 80 Gbits/s in each direction. Additional modules are an FPGA interface (GPIO), a debug module (DEBUG) and a general purpose IO-interface (GPIO). All PEs of all CLUs are controlled by a single instance of a dynamic task scheduling unit, called CoreManager (CM). It is described in detail in section V.

![Fig. 2. Many-core system: top hierarchy level](image)

Each CLU is composed of 48 PEs (see Fig. 3). They form the second hierarchy level. A star-mesh topology NoC is used to connect six PEs over a router (RC) to their neighbors. Altogether, nine routers are integrated. The central router allows a connection to three memory ports, to a cluster debug-unit and to the corresponding router RT on top level. The memory ports are connected to the three physical off-chip memory ports by serial on-chip data links.

All PEs are integrated using a PE integration framework. It consists of the following parts: a finite state machine (FSM) for PE control, a direct memory access controller (DMAC) for data transfers, and a local instruction memory as well as a data memory. 32 KB instruction memory and 64 KB data memory are integrated. In this work, a PE solely operates on its local memories. Consequently, no cache misses occur. Local memory access latency is one cycle. Additionally, up to four task descriptions are stored in a FIFO memory. Task descriptions are generated by the CoreManager and contain all necessary information for task execution. For instance, the configuration of the FSM and the DMAC as well as the instruction and data pointers to the local memories are included. Data prefetch is supported.

![Fig. 3. Processing cluster (CLU) with 48 PEs](image)

IV. SIMULATION ENVIRONMENT AND TOOL CHAIN

For performance evaluation the hardware system presented in the previous section is implemented in a cycle-accurate Tensilica XTSC environment [12]. It allows an integration of processors, the NoC, memories as well as further modules.

The CoreManager instruction set extensions are developed with the tool flow depicted in Fig. 4. In a first step, the CoreManager runtime behavior is profiled and analyzed. Hence, the most time consuming parts of the firmware are unveiled (hotspots). Based on these results a scheduling-specific instruction set is developed to improve the CoreManager performance. The instructions will be explained in detail in the next section. After the specification of a new instruction set, a processor generator is used to generate a cycle-accurate processor model and a corresponding compiler [12]. The potential performance improvements are made available by replacing parts of the CoreManager firmware by instruction macros. The verification can be performed, e.g., by applying unit tests, regression tests or equivalence checks. In our work a unit test is used for each newly introduced instruction. If their correctness is assured, a further profiling and instruction development iteration is performed. The hotspots are either reduced or shifted to other parts of the firmware. If the performance requirements are met, the HDL processor generator is used to generate an RTL description of the processor. It is synthesized to obtain area, power and timing figures of the core. Furthermore, the obtained core can be used for integration in a silicon prototype.
V. COREMANAGER

The CoreManager dynamically controls the data plane of the hardware system. It is responsible for: application to cluster mapping (CLU allocation), task scheduling, PE allocation and the management of the local memories. Furthermore, the CoreManager is used for explicit power management. As the PEs, the CoreManager solely works on its local instruction and data memory. Memory access latency is one clock cycle. States are additional memories in the CoreManager. They are explicitly managed by the newly introduced instructions. In contrast, registers are managed by the compiler. In the following, the designs of the most important instructions are explained.

CLU Allocation: For each application a primary and a secondary CLU are allocated as soon as the application is started. Both CLUs are next to each other. The primary CLU with the lowest utilization is chosen. Each CLU can be allocated up to 64 applications. The newly developed instructions for the CLU allocation are depicted in Fig. 5. Two instructions, and thus two execution cycles are necessary: a pre-selection and a CLU allocation instruction. Up to 32 CLUs are supported. The load of each CLU is stored in an 8-bit internal processor state. Consequently, no access to the local memory is required. In the CLU pre-selection stage eight CLUs are selected for further evaluation. In the next instruction (CLU allocation), these CLUs are compared with each other to find the lowest load. The topology information is used to find a suitable second CLU. Furthermore, the CLU allocation instruction updates the CLU load states. Therefore, two values are added to the load states of the primary and the secondary CLU. If necessary, further CLUs can be allocated during the execution of an application. Only neighboring CLUs are considered. As before, the CLU with the lowest utilization is chosen. The same instructions as before are used, non-neighboring CLUs are suppressed by a CLU mask.

![CLU Allocation Instructions](image)

Task Scheduling: The task scheduling determines the next task of an application. A dedicated value for each task is used. For instance, priorities, deadlines or sequence orders in the case of inter task dependencies are evaluated. In the latter case, tasks with the same values can be executed at the same time. This fully flexible task scheduling approach easily allows adaptions towards further scheduling algorithms. Applications with any number of tasks are supported. Tasks in a certain task window are evaluated. The size of the task window is determined by the application itself. It is shifted as soon as a task is finished.

The implementation of the task scheduling is depicted in Fig. 6. Four tasks are concurrently scheduled. In the first step (LD), 16 task values are loaded by the load-store unit from the local memory. The next instruction (S0) does a pre-schedule. In this stage, four tasks with the smallest values are obtained from each half of the loaded tasks. Tmp_0 to Tmp_3 result from tasks T0 to T7 and Tmp_4 to Tmp_7 from task T8 to T15, respectively. Furthermore, the pointer for the load-store unit is updated. In the schedule instruction (S1) the eight obtained values from the previous stage are compared with the already available four minimum values (Min_0 to Min_3). Afterwards, the four results Res_0 to Res_3 are written to the minimum values. The last instruction (FIN) concludes the task scheduling. It outputs the four smallest tasks and their values. Furthermore, Min_0 to Min_3 are reset to maximum.

In Fig. 7, a pipeline snippet of the execution of the task scheduling is depicted. After two cycles the data is loaded from the local memory and processed by S0 and S1. These three instructions are iteratively executed. The number of iterations is determined by the task window size divided by 16. After the last iteration the FIN instruction returns the scheduled tasks. The invalidations of the already scheduled tasks are not shown. Therefore, the updateTask instruction is executed after the FIN instruction. It is realized by a dedicated bit mask – one bit for each task. A zero value marks an invalid task position.

![Task Scheduling Instructions](image)
**PE allocation:** A PE is dynamically allocated on a per CLU basis. If no PE is available further CLUs are used. The PE allocation is depicted in Fig. 8. In the first step, the current load value for each PE is loaded (LD). 64 PEs are concurrently evaluated. For each PE two bits are available. The bit representation is as follows: 0-empty PE, 1-one task, 2-two tasks and 3-three tasks. Hence, up to three tasks can be scheduled for one PE. The PE allocation instruction (PE_Alloc) determines the PEs with the lowest number of tasks. Just like the task scheduling, four PEs are concurrently allocated. Furthermore, the CLU is updated (CLU_Ptr++).

**Local Memory Management:** Data must be fetched to a PE as fast as possible. Therefore, the CoreManager determines if the input data of a task can be concurrently prefetched to the execution of the predecessor task. Consequently, data locality as well as performance is increased. A block based approach is used for the implementation. For each PE four blocks (4 bits) are used. Each bit represents a taken (bit=1) or empty block (bit=0). Block size is 12 KByte each. Blocks next to each other are considered for larger amounts of data. All four allocated PEs are concurrently evaluated. The design of the instruction is similar to the PE allocation shown in Fig. 8. Instead of the PE load values the occupied memory blocks are used. Two loads are necessary to fetch all data. If free blocks are available, the data is marked as valid for prefetch. Afterwards, the modified blocks for all four PEs are written back to the local memory.

**Application Scheduling:** A list-based scheduling approach is used. It includes dynamic priorities. For instance, the task scheduling is executed more frequently for higher prioritized applications. Furthermore, the application scheduling is adaptable at runtime to consider task granularity and CLU utilizations. Special instructions for the processing of linked list are used.

**DMAC Config:** A DMAC is integrated in the CoreManager to transfer the task descriptions from its local memory to the PEs. The DMAC is controlled by transferring the required data from internal states of the CoreManager to DMAC registers.

**Release PE:** As soon as a task is finished, the CoreManager is informed. Hence, the PE load of the corresponding CLU is adapted. The CoreManager releases PEs of the same CLU in parallel.

**VI. EXPERIMENTAL RESULTS**

In this section, performance, area and power consumption of three CoreManager versions are compared. In the first two versions, a plain SW-based firmware is executed. Therefore, a Tensilica Diamond 108MINI [13] and a Tensilica Xtensa LX5 RISC core (CM_RISC) [12] is used. Both cores have similar features. In comparison to the 108MINI, the instruction and data bus width of the CM_RISC is increased from 32 to 64 and from 32 to 128 bit, respectively. Two local memories are integrated: a 32 KByte instruction and a 128 KByte data memory. Memory latency is one cycle. The third CoreManager version includes the scheduling-specific instruction set (CM_ISA++). It uses the CM_RISC core as basis. All cores integrate a polling based approach for event handling. Additionally, an interrupt based approach is available (CM_ISA++IRQ). It uses the same instruction set as the CM_ISA++ core.

**A. CoreManager Performance**

The performance of the main parts of the CM_ISA++ is analyzed and compared with the RISC-based approaches. In the case of the 108MINI and the CM_RISC, the average number of cycles is used.

Table I illustrates the processing time of the CLU allocation. The number of CLUs is varied between four and 32. CM_ISA++ has a constant processing time of five clock cycles. In the case of 32 CLUs, the CM_ISA++ core is 83x and 54x faster compared with the 108MINI and CM_RISC, respectively.

In Fig. 9, the results of the dynamic task scheduling stage are depicted. The number of tasks is varied between 32 and 2048. A difference in the processing time of over two orders of magnitude can be observed between the CM_ISA++ and the RISC cores. The CM_ISA++ is 193x and 118x faster than the 108MINI and CM_RISC, respectively.

**TABLE I. CLU ALLOCATION PROCESSING TIME**

<table>
<thead>
<tr>
<th>#CLUs</th>
<th>Processing Time (Cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>56</td>
</tr>
<tr>
<td>8</td>
<td>108</td>
</tr>
<tr>
<td>16</td>
<td>212</td>
</tr>
<tr>
<td>32</td>
<td>417</td>
</tr>
</tbody>
</table>

In the case of 32 CLUs, the CM_ISA++ core is 83x and 54x faster compared with the 108MINI and CM_RISC, respectively.
The processing time of the dynamic PE allocation is depicted in Fig. 10. In this experiment, the number of PEs is varied between eight and 1024. In the case of 1024 PEs the CM_ISA++ core is 419x and 233x faster than the 108MINI and CM_RISC, respectively.

In Table II, the processing time for the data management is shown. The number of occupied blocks is set to different 4-bit values. The processing time of the CM_ISA++ core is always five cycles. This is a performance improvement of up to 6x compared to the RISC-based processors. The processing time of the RISC-based cores depends on the already occupied blocks.

The system utilization $\eta_S$ is used for a comparison on system level. $\eta_S$ is described by (1). The utilization $\eta_{PE}$ of all PEs ($\#PEs=1008$) are added. $\eta_{PE}$ of a PE is obtained by (2).

The results are depicted in Fig. 11. The system utilization $\eta_S$ over the average task execution time is shown. In comparison to the CM ISA++, the 180MINI as well as the CM_RISC need 29x and 18x longer task execution times to achieve the same system utilization of 900 PEs, respectively. The interrupt based version CM_ISA++IRQ does not scales as good as the CM_ISA++ core, which uses a polling-based approach. The static scheduling approach represents an upper boundary. The CM_ISA++ nearly achieves the scalability of the static scheduling approach.

**Table II.** Local Memory Management (Data Size: 16 KByte, Block Size: 12 KByte)

<table>
<thead>
<tr>
<th>#Allocated Blocks</th>
<th>108MINI</th>
<th>CM_RISC</th>
<th>CM_ISA++</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>16</td>
<td>18</td>
<td>7</td>
</tr>
<tr>
<td>0x5</td>
<td>36</td>
<td>36</td>
<td>7</td>
</tr>
<tr>
<td>0x7</td>
<td>36</td>
<td>32</td>
<td>7</td>
</tr>
<tr>
<td>0xA</td>
<td>44</td>
<td>36</td>
<td>7</td>
</tr>
<tr>
<td>0xD</td>
<td>36</td>
<td>32</td>
<td>7</td>
</tr>
</tbody>
</table>
C. Area, Timing and Power Consumption

All versions of the CoreManager have been synthesized with Synopsys Design Compiler for a 65 nm low-power TSMC process using typical case conditions (25°C, 1.25 V). The processor and the memories are regarded. For the latter, low power TSMC libraries are used. Power Consumption was simulated as follows: the CoreManager and the memories are synthesized as previously described. Results of this step are a netlist on gate level as well as a Standard Delay Format (SDF) file with timing information. Mentor Questa is used for simulation of representative test cases. Herby, a dump file is created containing all switching activity of the entity. In a last step, Synopsys PrimeTime with the previously created files as input is used to obtain the corresponding power consumption. The synthesis results reflect the accurate area and timings that would be used in the final processor after the tape-out.

Table III provides the area, maximum achieved frequency and power consumption results. The logic area of the CM_ISA++ core is 3.2x larger than the CM_RISC core due to the newly introduced instructions. The maximum frequency of the CM_ISA++ core is slightly reduced from 555 MHz (CM_RISC) to 512 MHz. The power consumption of the processor variants are depicted on the third column of Table III. The basic core CM_RISC and standard instruction set consumes 68.2 mW. This is 2.5x more than the 108MINI processor – especially due to the integration of the local memories and the increased processor bus widths. By applying the newly developed instruction set the power consumption of the CM_ISA++ is increased by 2.2x to 148.5 mW. The energy consumption per scheduled task of the CM_ISA++ core is reduced by 41x and 50x compared to the 108MINI and CM_RISC, respectively.

Table IV illustrates the relative area consumption of the CM_ISA++ core. The basic core is nearly one third of the overall area. The task scheduling instructions has the highest relative area consumption of the newly introduced instructions.

### Table III. Synthesis Results and Comparison

<table>
<thead>
<tr>
<th>Core</th>
<th>(f_{max}) [MHz]</th>
<th>(A_{logic}) [mm²]</th>
<th>(A_{mem}) [mm²]</th>
<th>(P_{max}) [mW]</th>
<th>(E_{a}) at (f_{max}) [nJ]</th>
</tr>
</thead>
<tbody>
<tr>
<td>108MINI [13]</td>
<td>442</td>
<td>0.220</td>
<td>-</td>
<td>27.4</td>
<td>795.8</td>
</tr>
<tr>
<td>CM_RISC</td>
<td>555</td>
<td>0.164</td>
<td>1.566</td>
<td>68.2</td>
<td>966.4</td>
</tr>
<tr>
<td>CM_ISA++</td>
<td>512</td>
<td>0.527</td>
<td>1.566</td>
<td>148.5</td>
<td>19.3</td>
</tr>
</tbody>
</table>

\(E_{a}\) Energy per scheduled Task

Table IV. Relative Area Consumption (CM_ISA++)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Area [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic Core</td>
<td>31.1</td>
</tr>
<tr>
<td>CLU Allocation</td>
<td>21.1</td>
</tr>
<tr>
<td>Task Scheduling</td>
<td>25.9</td>
</tr>
<tr>
<td>PE Allocation</td>
<td>8.2</td>
</tr>
<tr>
<td>Memory Management</td>
<td>7.3</td>
</tr>
<tr>
<td>Other</td>
<td>6.4</td>
</tr>
<tr>
<td>SUM</td>
<td>100</td>
</tr>
</tbody>
</table>

VII. Conclusion and Future Work

A many-core system should provide a high task throughput and low latency. Consequently, one of the key tasks in the development of a many-core system is the performance of the scheduling unit. To achieve this, a dedicated task scheduling unit, called CoreManager, is developed. It dynamically allocates clusters and PEs. Furthermore, the flexible task scheduling and the management of the local memories are done at runtime. The CoreManager is optimized to dynamically control over 1000 cores. Therefore, each component is analyzed and accelerated with a newly developed application-specific instruction set, called CM_ISA++. We discuss the necessary design of each instruction. In our experiments, we show that the newly developed CM_ISA++ CoreManager is over two orders of magnitude faster compared with RISC-based processors. Consequently, a higher system throughput is achieved. Furthermore, energy efficiency is improved.

Future work aims at further optimizing the CoreManager architecture, especially regarding performance, area and power consumption. Furthermore, the CoreManager will be integrated in a silicon many-core prototype.

Acknowledgment

This research work has been funded by the DFG through the Center for Advancing Electronics Dresden and the European Union and the state of Saxony through the EuroServer and IMData project. Furthermore, we would like to thank Synopsys and Tensilica for software and IP.

References