The Universal Configurable Block/Machine System – An Approach for a Homogeneous Configurable SoC-Architecture

Christian Siemers\(^1\) and Sascha Wennekers\(^2\)

\(^1\) University of Applied Sciences Nordhausen, Weinberghof 4, D-99734 Nordhausen, Germany
cristian.siemers@computer.org

\(^2\) Technical University Clausthal, Julius-Albert-Str. 4, 38678 Clausthal-Zellerfeld, Germany
sascha.wennekers@tu-clausthal.de

Abstract

This approach uses a homogeneous architecture with two configuration layers: The lower layer may configure for the execution of blocks of instruction on a cycle-by-cycle base and plays the role of programmability, while the upper part enables the architecture to adapt to heterogeneous architectures. This UCB/UCM-system is expected to adapt to certain situations and to be optimisable for performance- and real-time-dominated applications. The paper introduces the system including a classification system and some application examples.

1 Motivation of the Approach

The migration from ASIC design towards SoC design has several advantages as well as challenges [1]. In fact being more than a silent, smooth, evolutionary way, SoCs request for new architectures and design methodologies.

This approach presents a homogeneous system of reconfigurable blocks with additional reconfigurable structures for the machine [2]. The basic idea can be described by the following items:

- Deeply embedded systems, especially in the area of mechatronics, often request for performance and/or real-time behaviour optimisations. In the case of performance optimisations, the application may be partitioned into parallel threads with varying complexities. Examples for this case are motion estimations inside combustion engine control, knock detection, parameter field interpolation etc.
- In the case of hard real-time constraints, the actual responsivity of the system may vary during runtime in orders of magnitude.

To meet all constraints with one hardware architecture, a block-based approach was introduced defining a block layer (‘sea-of blocks’) and a higher machine with additional equipment for configuration. The block layer consists of programmable/reconfigurable blocks. These UCBs, Universal Configurable Blocks, are programmable in the sense that they execute an instruction-based program (sequential program mode). On the other side, once an appropriate set of instructions is stored inside the UCB for execution, it works like a configuration. In this case execution must be explicitly stopped by replacing the program part, so UCBs may work in a virtually fixed mode.

The higher machine is called UCM, Universal Configurable Machine, and consists of at least one block with additional hardware. The UCM may be configurable too, especially in the case that more than 1 UCB is integrated. While the machine appears to be homogeneous, a heterogeneous, application-specific system may be configured inside the UCM during compile-time. This two-level system of configurability and the partitioning into executing and managing parts are responsible for the flexibility and the adaptivity of the approach.

The corresponding development system is (in principle) able to build heterogeneous parts for execution (e.g. exclusively used parts for application-specific co-processors) and impacts therefore the performance of the system within a wide range. The runtime scheduling (inside the operating system) on the other side may do the same by dynamic profiling, but for some applications, especially within the class of responsive embedded systems, the scheduling mechanism for the later discussed event density \( E \) is even more important.

The remaining paper starts with an introduction into the basics of the UCB/UCM system. Recently published work [3] has shown the effectiveness of this concept concerning performance. One example is given for a compile-time extraction of a configurable co-processor.

For understanding the effects of runtime scheduling inside the UCB/UCM-system, the event density \( E \) is defined [4] and some values are given.

2 Universal Configurable Blocks

A Universal Configurable Block (UCB), comparable with a very coarse-grained PLD, consists of a register file, a configurable asynchronous network with some arithmetical and logical capacity and optionally some auxiliary circuits (see Figure 1). Optionally a load/store pipeline as well as private memory (random access or stack organised) may be added to the UCB. The width of data busses and registers inside is related to the actual implementation and may vary between different instantiations.

This paper has been published in: Proceedings of the Workshop Heterogeneous reconfigurable Systems on Chip (SoC), Hamburg, Germany, April 2002.
The sub-units of a UCB are designed to create a configurable block for computations using a partial or complete network, and the following types are included:

- **Arithmetical Unit (AU, type A):** The computations inside the UCB are mostly performed by the AUs. Differing from the well-known Arithmetical-Logical-Unit, the AU includes just one or few (in this case configurable) computational combinations, for example, adding two integer values. Connecting the inputs to the sources, the result is available at the output of this subunit for use inside the whole network.

- **Compare Unit (CoU, type B):** This sub-unit type generates conditional bits inside the network used for the conditional execution of computations or moves. Characteristics for the CoU are two input busses, one conditional bit as an output and the possibility for configurable comparisons.

- **Multiplexer (Mul_C, type C):** Type C Multiplexer are used to route sources to the inputs of AUs and CoUs with two input busses (of full bus width). These multiplexers connect the different sources with the inputs of the adjacent sub-unit.

- **Multiplexer (Mul_D, type D):** The routing of a number of possible sources to a certain register can be done by a multiplexer with only one output bus. Type D multiplexer connect the destination registers to the outputs of the AUs.

- **Demultiplexer (Demul, type E):** The results from the compare units (CoUs) have to be routed to the corresponding AUs, therefore a pair of CoU and AU establishes a conditional execution of an instruction.

- **Compare Unit 2 (CoU2, type F):** This optional unit generates the Ready-signal for other control parts of more integrated devices of the reconfigurable architecture. The purpose of this unit is to integrate loops in hardware.

- **The additional hardware is integrated to synchronize clocks with external and internal control signals.**

The optional private RAM shown in Figure 1 was originally not included in the preceding work [4] [5] [8]. The analysis of potential algorithms e.g. from network applications or signal processing has shown that they often demand for data tables exceeding the size of the register file. The integration of a private memory enables the explicit coding of such algorithms inside one UCB without extensive data accesses.

The focus of UCBs is set on fitting execution to a complete hyperblock of instructions. The configurations are processed directly during compile- or runtime, and it was shown that this is possible even in hardware with a latency of 1 cycle [2] [3]. This algorithm is called PDSP, Procedural-Driven Structural Programming.

### 3 Universal Configurable Machines

UCB implementations in connection with loading/sequencing hardware will be called Universal Configurable Machines (UCM). To receive a full flexibility of the complete concept, additional, partial configurable hardware beside the loader and sequencer will be included. This may lead to some confusion, but while a hyperinstruction may fit to any UCB on a block-by-block base, the overall configuration of the UCM will be at least quasi statically.

The partitioning of the executing part for hyperblocks and the control-flow managing part enables the quite simple doubling of the UCBs. Additionally a UCB can execute its configuration as long as this is not explicitly stopped by the UCM with a new configuration. This features makes execution inside UCBs quite autonomous with the advantage that the performance may scale with the number of UCBs. In summary this system follows the way of Instruction-Level Distributed Processing [6].

The additional hardware will contain configurable connections between several UCBs (for communication and/or connecting UCBs to perform larger hyperin-
structions), load/store pipelines for memory data access and configurable input/output system for external connections similar to well-known controller technology. The UCM will differ from known microcontroller implementations by the internal microarchitecture.

To describe the flexibility of UCM implementations, a classification scheme was introduced in [2]. This classification consists of a triple notation of the unsigned integers $b$ and the boolean variables $h$ and $s$. The variable $s$ describes the ability of the UCM to sequence hyperinstructions during runtime ($s = 1$) or not ($s = 0$). A non-sequencing UCM seems to be no computing machine but makes sense for small applications as shown below.

The boolean $h$ determines the ability of the UCM to execute more than one hyperinstruction in parallel ($h = 1$) or not ($h = 0$). For a multi-hyperinstruction UCM the presence of an at least rudimentary scheduler and more than one UCB inside is essential. Due to the fact that small threads which will be called microthreads will be able to fit into one UCB, the terms microthreads and hyperinstruction describe the same object but from different point of views.

Last not least the unsigned integer variable $b$ counts the number of UCBs inside the UCM. This number is deterministic, and there will be 1- and $b$-UCM ($b > 1$) implementations for different application classes. In summary, every UCM will be classified by a $bhs$-triple ($b$, $h$, $s$). The >S<puter [4] (fig. 2) and the rRISC-machine [3] as derivation with limited resources obtain both a classification as $(1,0,1)$-UCM.

3.1 Applications for $(1,0,0)$-UCM

The smallest possible implementation of a UCM is rather similar to the UCB and consists of one block executing just one hyperinstruction (or in terms of the application viewpoint: one microthread). This could be a well-designed architecture for small applications as members of a fieldbus system or even independent of others. The difference between a simple UCB/PDSP and the $(1,0,0)$-UCM is located inside the additional control unit, which contains no sequencer but a hyperblock loader to switch between several operational conditions or to run a sequence of configurations. Figure 3 shows the implementation of the $(1,0,0)$-UCM.

The control unit serves several input lines from the outside world, e.g. any process, or from inside computation. The task of this unit is to identify the operational condition from the process and to switch to the next hyperinstruction. The sequence of hyperinstructions is therefore equivalent to a sequential finite state machine (SFSM).

One example for a $(1,0,0)$-UCM as an application-specific co-processor may be given from the area of combustion engine control. These algorithms mostly include a look-up table and a mechanism for receiving interpolated values from this table for controlling the engine. The access and interpolation of table elements is frequently invoked and may be integrated into one UCB. Estimations show that 24 reconfigurations for a rRISC-like block [3] or only 3 for a UCB with ‘unlimited’ resources are needed to fully implement this access.

If this application is completely coded into one $(2,0,1)$-UCM using 2 threads, the main and the look-up table algorithm might execute in parallel.

3.2 A generalized $(b,h,s)$-UCM

In general, a UCM may contain more than one block serving more than one thread. This makes an extended control unit necessary, which now integrates not only the sequencing but also the scheduling subtask inside the UCM. Figure 4 shows this fact: The sequencing/scheduling unit sequences for every UCB inside the flow of
hyperinstructions on thread base, and the \((b,1,1)\)-UCM will in general act as a multithreaded computing machine.

Multithreading on the other side does not necessarily impact an extended interface to program memory. In comparison with the \((1,0,0)\)-UCM, a thread could consist only of one hyperinstruction, which is uninterrupted executed inside one dedicated UCB. Threads of this kind were introduced as microthreads and will produce high reactivity inside the UCM-application.

The microthread concept will work in connection with embedded system applications. Small control threads like boundary checks of digital values, data transmissions to/from memory or the interpolation for parameter fields (see example above) will fit into one UCB or a small amount of reconfigurations for one UCB, and for more flexibility UCBs will be connectable to integrate a greater hyperinstruction (or microthread).

Independent (non-micro-)threads on the other side will have no communication between UCBs and will execute independent from each other. The memory interfaces have to be designed for the extended access, this is subject to further simulation.

4 Introducing the Event Density \(E\)

A complete other aspect of this homogeneous system is the capability of space/time scheduling. The UCB/UCM-system is capable to execute program parts in space or time (sequence), and this may frequently change during runtime. To quantify this effect, the event density \(E\) was introduced \([7]\) with respect to the functional density \(F\) \([8]\).

As a first approach, \(E\) was defined as the average availability of the event-handler (e.g. the interrupt service routine) per area and time, measured in \(N_{\text{event}}/\lambda s\).

<table>
<thead>
<tr>
<th>Device Type</th>
<th>(E) ([N_{\text{event}}/\lambda s])</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microprocessor</td>
<td>0.01 .. 0.1</td>
</tr>
<tr>
<td>UCB (dedicated)</td>
<td>0.5 .. 5</td>
</tr>
<tr>
<td>PLD</td>
<td>350 .. 500</td>
</tr>
</tbody>
</table>

Table 1. Preliminary results for event density \(E\)

The problem of \(E\) is the influence of the operating system inside a microprocessor-based system. The reported values for \(E\) are measured without this dependence but will decrease in real systems. A standard event is defined as an interrupt request from any device outside to be serviced by the microprocessor or PLD. The service includes just the interrupt response time and the time to leave the service routine and return to normal operation (within a microprocessor based system). Table 1 summarizes the results for microprocessor and PLDs.

This shows that scheduling any event handler from sequential execution to exclusive execution inside one block will shift the event density by 1 or 2 orders of magnitude. This is the main effect for hard real-time systems.

5 Impacts on Development System, Compiler and Operating System

It has been one of the most important objectives of this approach to introduce a reconfigurable architecture with minimal impact on the development methodology, but there will be some – and there should be also some. For explaining details Figure 5 presents the different views to an application using a Y-diagram.

The Y-diagram shows some layer with links between the several views. Complete tasks for example will be coded within one algorithm using classes etc. for homogeneous programmable systems. Inside the UCM concept a complete algorithm could be translated to an \(S^\text{puter}/R\text{ISC} \((1,0,1)\)-UCM) device or to a Xputer-like architecture \(((b,0,0)\)-UCM, \([2]\)) but the algorithm and the coding in high level language would remain (nearly) the same. These more self-contained applications may use \((b,1,1)\)-UCMs as microprocessor and/or as hardware accelerator, and it is subject of the compiler technology to translate the source code to well-performing executable code using the configurability of the UCM.

For application in the context of \textit{embedded systems}, the design flow may take another path. Identifying and
defining microthreads or threads in the problem domain will make it possible to describe the runtime behaviour of the system with more exactness. This is important for embedded and responsive systems with hard real-time capabilities, and the reconfigurable UCM architecture enables the developer to determine the system behaviour corresponding to the system description. This is performed by dedicated UCBs inside a \((b,1,1)\)-UCM and microthreads.

Fig. 5. Y-diagram for UCB/UCM development

Concluding the impacts on software, the following statements describe the situation with this approach for reconfigurable computing:

- The compiler technology for superscalar processors including the creation of hyperblocks is sufficient for the UCBs using the PDSP algorithm. The PDSP itself may be integrated in hardware or software (post-compiler run). Minor adaptations for generating UCB-optimized code may be integrated in the compiler. Besides specialized compilers e.g. for PDD-CCMs as well as fault tolerant applications could be developed generating the corresponding UCM configuration.
- The operating system has to support threading and microthreading, compiletime as well as runtime priorities and scheduling strategies to enable the complete system for deterministic behaviour under hard constraints. The interface between operating system, probably description language and UCM seems to be the most relevant part on the software side.
- The developer may define threads or microthreads, and he should do this especially for embedded systems to obtain hard real-time capabilities.

6 Classes of Applications

One of the remaining questions will be the discussion of those application classes, where this methodology for hardware/software co-design should be preferred. As the designer receives the task to explicitly define threads for using the advantages of the architecture, there should be some benefits, while automatic tools for co-design are promising at least more comfort during design phase.

The first class will be consist of small applications using the aspect of reconfiguration in the sense of re-use the hardware. A good example for this class are all kinds of build-in self tests (BIST) for controller and memory.

A memory build-in self test (MBIST) consists of some very small test algorithms trying to identify the usability and correctness of RAM cells. It could be shown during a small project that realising the MBIST in a \((1,0,0)\)-UCM using several reconfigurations would offer 20% space and 33% time reduction, compared with a dedicated hardware solution. This UCM was designed as a sequential finite state machine.
The unexpected time reduction results from more flexibility for the algorithm parts using reconfigurable hardware, not from increasing clock (the operating frequency of the studied MBIST is about 5 MHz). The additional benefit is of course re-usability of the UCB/UCM.

The second class of applications is inside the area of embedded systems with hard real-time constraints and a strongly event-driven environment. In this case the developer has to be sure to fulfill all demands, because the product has to work under all circumstances. This will lead to explicit multithreading with the use of (micro-) threads for interrupt services and dedicated UCBs for high priority interrupts.

In this case the use of one description language (e.g. C) for describing both sequential execution and dedicated hardware and the well-defined threading behaviour of the solution to guarantee real-time demands will count as the main benefits.

7 Conclusion and Outlook

It was the goal of this paper to discuss the approach for reconfigurable computing using a homogeneous System-on-Chip-architecture. This approach promises to adapt very well to certain application constraints, especially in the area of real-time. Future work on the hardware side will continue with simulation of several UCB architectures to achieve the ’ideal’ UCB and configurable UCM architectures to ensure the adaptivity.

The UCB/UCM architecture contains a more-dimensional scalability inside. The UCB block size is scalable, an optional connectivity between blocks makes them more flexible, and last not least the additional hardware units to form a UCM range from very small to large including operating system capabilities. This makes the UCM approach to be interesting for semiconductor industry.

The adaptation to heterogeneous problems results in runtime-specific heterogeneous configurations. It is definitely part of the (future) development system to define the heterogeneous parts or at least to enable the developer to do this. This problem of design space exploration is also part of future work.

8 References