Tomahawk: Parallelism and Heterogeneity in Communications Signal Processing MPSoCs

OLIVER ARNOLD, EMIL MATUS, BENEDIKT NOETHEN, and MARKUS WINTER,

Technische Universität Dresden

TORSTEN LIMBERG, ZMDI Dresden

GERHARD FETTWEIS, Technische Universität Dresden

Heterogeneity and parallelism in MPSoCs for 4G (and beyond) communications signal processing are inevitable in order to meet stringent power constraints and performance requirements. The question arises on how to cope with the problem of system programmability and runtime management incurred by the statically or even dynamically varying number and type of processing elements. This work addresses this challenge by proposing the concept of a heterogeneous many-core platform called Tomahawk. Apart from the definition of the system architecture, in this approach a unified framework including a model of computation, a programming interface and a dedicated runtime management unit called CoreManager is proposed. The increase of system complexity in terms of application parallelism and number of resources may lead to a dramatic increase of the management costs, hence causing performance degradation. For this reason, the efficient implementation of the CoreManager becomes a major issue in system design. This work compares the performance and capabilities of various CoreManager HW/SW solutions, based on ASIC, RISC and ASIP paradigms. The results demonstrate that the proposed ASIP-based solution approaches the performance of the ASIC realization, while preserving the full flexibility of the software (RISC-based) implementation.

Categories and Subject Descriptors: C.3 [**Special-Purpose and Application-Based Systems**]: Signal Processing Systems

General Terms: Design, Experimentation, Measurement, Performance

Additional Key Words and Phrases: Tomahawk, CoreManager, taskC, software defined radio, network-on-chip, heterogeneous MPSoCS, task scheduling, runtime management, instruction set extensions, hardware scheduler, real-time, software scheduler, data dependency checking, out-of-order scheduling

ACM Reference Format:

Oliver Arnold, Emil Matus, Benedikt Noethen, Markus Winter, Torsten Limberg, and Gerhard Fettweis. 2014. Tomahawk: Parallelism and heterogeneity in communications signal processing MPSoCs. *ACM Trans. Embedd. Comput. Syst.* 13, 3s, Article 107 (March 2014), 24 pages.

DOI: <http://dx.doi.org/10.1145/2517087>

1. INTRODUCTION

4G wireless communications systems introduce advanced transmission techniques, for instance, orthogonal frequency division multiplexing (OFDM), multiple antennas, adaptive modulation and coding, and hybrid automatic repeat requests which

The major part of this research work has been done within the scope of the projects WIGWAM and CoolBase-Station, funded by the German Federal Ministry of Education and Research. A minor part was funded by the European Union within the scope of the IMData, E2R and EMUCO projects.

Author's address: O. Arnold; email: oliver.arnold@tu-dresden.de.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2014 ACM 1539-9087/2014/03-ART107 \$15.00

DOI: <http://dx.doi.org/10.1145/2517087>

enable realization of high-throughput and low latency wireless networks. This, however, simultaneously increases the receiver algorithmic complexity. Even though the absolute receiver complexity grows with data rate, receiver implementation complexity with respect to Moore's law decreases, as moving in modulation to OFDM has reduced the relative complexity of receiver signal processing from an exponential (GSM) and linear (UMTS) to a logarithmic (LTE), and therefore fewer transistors are required per received bit. This, in addition with customer's willingness to pay more for receiver flexibility, gives the opportunity for software defined radio (SDR) solutions. However, such an approach results in increased architecture heterogeneity and parallelism in order to meet stringent power constraints and performance requirements.

Numerous concepts of programmable single and multicore SDR architectures have been introduced recently, for instance, OnDSP [Kneip et al. 2002], EVP [Berkel et al. 2005], STA [Cichon et al. 2004], Icera, ADRES [Novo et al. 2005; Bougard et al. 2008], XiSystem, SODA [Lin et al. 2006], LeCore [Nilsson et al. 2009; Liu et al. 2009], MuSIC [Ramacher 2007], Sandblaster [Glossner et al. 2007], ConnX [Tensilica 2012]. A good overview of SDR solutions can be found in Anjum et al. [2011]. Recent studies have shown that even multicore architectures can be power efficient in scenarios with tight performance requirements [Horowitz and Dally 2004; Asanovic et al. 2006]. However, due to the additional costs and complexity of programmable systems, a considerable portion of energy losses may be incurred by ineffective parallel programming techniques. Several programming interfaces have been proposed for multicore processor systems, for instance, CUDA (NVidia), OpenMP, MPI, Cilk (MIT) [Frigo et al. 1998], CellSs (Barcelona Supercomputing Center) [Bellens et al. 2006], Sequoia (Stanford University) [Fatahalian et al. 2006], Ct (Intel) [Ghuloum et al. 2007] and OpenCl (Khronos Group), however, each of them is suitable for a specific set of applications. The question arises on how to cope with the problem of programming and system management associated with the statically or even dynamically varying number and type of processing elements. This work addresses this challenge by proposing the concept of a heterogeneous multicore platform called Tomahawk. Apart from the template definition of the Tomahawk system architecture, in this approach, a unified framework including a model of computation, a programming interface and a dedicated runtime management unit called CoreManager is proposed.

In order to cope with the stringent performance-efficiency requirements for 4G base-band signal processing, the Tomahawk architecture exploits parallelism and data locality both at system and core level. Even though data-level and instructionlevel parallelism in PEs is essential, the main focus of this work is on the functional, that is, task-level parallelism. The Tomahawk platform is C-programmable, based on embedded software written as modular tasks according to the data flow model [Lee and Messerschmitt 1987]. The increasing system complexity in terms of application parallelism and number of resources may lead to 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. This work compares the performance and capabilities of various CoreManager HW/SW solutions based on ASIC, RISC and ASIP paradigms.

In the remainder of the article the concept of this multicore platform is firstly introduced including the model of computation and the system architecture template. It follows the description of the runtime management unit (CoreManager) and the associated task-parallel programming interface. In order to determine the effect of the CoreManager on system flexibility and performance various HW/SW implementations of the CoreManager are analyzed and discussed.

Fig. 1. An example of DAG a) with tasks T2, T3, T4, input I1 and output O1. In b) is the same graph with additional outer loop O2-I2.

2. CONCEPT OF MULTICORE PLATFORM

This section presents the concept of a heterogeneous multicore platform called Tomahawk. The Tomahawk platform basically comprises a *model of computation*, a *programming interface*, a *runtime management*, and a *template of system architecture*.

2.1. Model of Computation

2.1.1. Task Graph. Signal processing applications are usually modeled by directed graphs comprising i) nodes representing a computation and ii) edges associated with data transfers. In general, this kind of abstraction is called *data flow graph* (DFG). DFGs naturally enable to express the inherent parallelism and data locality of the application and hence are suitable for concurrent execution on parallel processors. Examples employing DFG are Simulink, LabView, Synchronous Data Flow [Lee and Messerschmitt 1987] or Kahn process networks. Depending on the model definition, the differences may appear in i) communication, that is, on how the data is produced and consumed by a node and ii) synchronization mechanism, that is, what is the firing rule initiating node computation. Special case of DFG is a *directed acyclic graph* (DAG), that is, a single pass DFG where each node (task) is executed exactly once per iteration. As the DAG execution terminates when the leaf nodes are reached, a retriggering is necessary to start the next iteration. This in turn is controlled usually by some master process. DAG is a general concept for representing signal/data flow based algorithms and for that reason it has been selected as a feasible model in this work.

In Figure 1(a) an example of a DAG with tasks T2, T3, T4, input I1 and output O1 is illustrated. Edges are associated with data transfers and hence represent implicitly task data dependencies. This in turn defines uniquely task partial ordering, that is, arrangement for DAG processing. For instance, in Figure 1(a) the task T4 can be fired after the results of tasks T2 and T3 are available. The following conditions must hold for deadlock free DAG computation before the task is assigned for execution.

- (1) The task completion time is finite.
- (2) The task data dependencies are uniquely determined.
- (3) Tokens, that is, chunks of data to be consumed by a task are available and are limited in number and size.

It is evident, that all three constraints can be simply met in practical systems. In addition, rule 2 implies that task linking, that is, data-to-task assignment can be realized dynamically during the runtime.

Yet another positive aspect of using DAGs lies in the fact that graph hierarchy can be introduced, that is, DAGs of any size can be treated as single tasks. Figure 1(a) illustrates an example defining a “super” task TG2 which abstracts the DAG with T2, T3 and T4.

Fig. 2. (a) Task model; (b) task execution procedure from the system point of view.

Loops are a general problem in data flow models since they may induce deadlocks. In the proposed model, internal feedback loops within a single DAG iteration are not allowed. As a result, deadlock-free execution of the DAG is guaranteed. Nevertheless, the outer loops between DAG iterations are tolerable, that is, the DAG output of one iteration can be fed back as input for the next iteration (e.g., loop O2-I2 in Figure 1(b)). It should be noticed that in order to keep the DAG processing deadlock-free, inter-iteration data-flow has to be decoupled by inserting a buffer element properly initialized with the respective number and type of tokens.

2.1.2. Atomic Tasks. The basic element of the DAG is an *atomic task* (Figure 2(a)). The task consumes and produces data while accomplishing some computation. During the task execution, temporary data are produced which characterize the *task state*. The task operation is uniquely determined by the input data comprising the input arguments and the task state. Once the input data are known, the task computation can be performed without any interaction with the rest of the system. Hence, it follows that:

- the task execution can be interrupted, in which case the task state and the input arguments have to be stored to allow the task execution to be continued;
- a task can be dispatched to an adequate *processing element* (PE), which in turn enables exploiting functional parallelism.

The principle of the task dispatching procedure is depicted in Figure 2(b). It is assumed that all input arguments as well as the task initial state are stored in the system global memory. The following operations are carried out once the task is assigned for execution.

- Fetch*. The task input arguments and states are transferred to the local memory of the target PE.
- Execute*. The task execution is started in the PE.
- Put*. After the task processing is completed (or interrupted) the output arguments and the task state are transferred back to the global memory.

Unlike standard threads and processes used in current operating systems, intertask communication during task execution is not allowed in the Tomahawk concept. This reasonably simplifies the synchronization mechanisms and makes the system response more predictable.

In general, task dispatching is accomplished during runtime and for this purpose, some kind of task description (*task activation record*) has to be defined. When the task is requested for execution, its description is processed by the runtime system in order to manage the data transfers and the task execution. A list of task description attributes

Table I. List of Task Description Attributes

Attribute	Description
Task ID	String for unambiguous task identification
Task Type	Specifies type of target PE (mandatory in case of heterogeneous PEs)
Task dependency	ID's of task predecessors (optional). When supported, it can reasonably reduce runtime data dependency check
Task location	Memory range specification where the task executable is located
WCET	Task worst case execution time (optional)
IN/OUT Data	Specification of memory region of a block of task input/output data. Note that multiple blocks can be specified
Data direction	Specification if the data is input or output argument
Data pattern	Specifies the arrangement of data elements in the memory. Supported arrangements are: linear, 2D, stride (optional)
# IN/OUT data	The number of IN and OUT data blocks. The maximum number depends on the particular implementation of runtime management.

Fig. 3. Example illustrating the model of computation in the Tomahawk concept, comprising a) a FSM and b) the corresponding DAGs ; c) depicts a program's flow graph based on sequential programming language.

is depicted in Table I. Note that the set of supported attributes may vary depending on the requirements and on the particular system configuration.

2.1.3. Control Flow. Signal processing applications become more complex and irregular in terms of control flow. Therefore, pure *data-flow machines* associated with data-flow models are not advisable from the flexibility and efficiency point of view. Instead, *finite state machines* (FSM) complement the data-flow for the purpose of control-flow modeling.

FSM consist of i) nodes associated with a system state (i.e., a mode of operation) and ii) edges representing transitions between the states. The transition rule is defined as a logical condition on some state variables. The functionality, that is, the computation associated with a FSM node, is determined by a specific DAG. Once a state of the FSM is activated, the corresponding DAG is triggered for execution. An example of a FSM representing an application's control-flow is illustrated in Figure 3(a). In this case, the FSM consists of five states S1, S2, S3, Start, and End, where Start and End are termination nodes. In Figure 3(b), the FSM states are expanded by the corresponding specific DAGs whereby the edges in this case represent the task data dependencies.

Fig. 4. Content of queue a) generated by example in Figure 3 and state sequence S1, S2, S2, S3. In b) is the task rearrangement according to data dependency.

The question arises, how to realize the FSM-DAG model using sequential programming languages like, for instance, C/C++. The reason for using sequential C/C++ is purely pragmatic, as this language is de facto standard in embedded signal processing systems. C/C++ can well express the FSMs of control flow using constructs if-else/while/for. However, there is no support for describing parallel DAGs. A language extension or a special API is hence necessary for this purpose. In this work the approach of a specific API with preprocessor directives is followed. Functions being subject of acceleration are substituted in-place with *task call primitives*. Consequently, the resulting code remains sequential. In Figure 3(c) is a flowchart of a sequential program implementing the FSM-DAG model of Figures 3(a) and 3(b). In order to exploit the inherent parallelism of the application, the sequentially coded FSM-DAG has to be transformed back to parallel task graphs in runtime. When the task execution is approaching during the sequential code execution, the task description is sent to the task queue of the runtime system instead of the task computation.

Based on this, the runtime system recomposes the DAG and dispatches the tasks to PEs according to the task dependencies and the resource availability. By these means the *out-of-order task execution* is thus feasible (task-level super-scalar principle). Figure 4(a) depicts an example with task queue for the task graphs in Figure 3 for the state sequence S1, S2, S2, S3. Some of the tasks without data dependencies can be processed concurrently. For this reason, the tasks are rearranged in order to be scheduled according to their dependencies (Figure 4(b)). In this case, 3 PEs can be used simultaneously.

2.1.4. Concept of System Architecture. In this section we introduce a concept for the Tomahawk system architecture. Based on the proposed FSM-DAG model of computation, the Tomahawk concept integrates both a *control-plane* and *data-plane* subsystem (Figure 5). The control-plane comprises a CPU, a global memory and peripherals. It is dedicated to the execution of the application's control flow, hence representing a master unit in the system hierarchy. On the other side, the data-plane consists of a cluster of heterogeneous PEs, each equipped with local program and data memories. The data-plane machine is intended for the application's signal processing, and hence represents a slave unit in the system hierarchy.

The data-plane is interfacing the control-plane through a controller called *CoreManager* (CM). The CoreManager is appearing as a CPU's standard peripheral and it decouples the control- and data-plane subsystems both physically and logically. The proposed system arrangement presents several advantages.

(i) The Control-Plane Subsystem

- Operating system.* The processor architecture with unified address space enables the usage of common embedded operating systems including RTOS.
- Programming languages and compilers.* Due to the standard computer environment of the control-plane, standard programming languages and compilers can

Fig. 5. Tomahawk computing architecture.

be used for programming applications. Note that in the Tomahawk project the C++ Language has been used for this purpose.

- Code reusing.* Porting legacy or reference code is straightforward.
- Multicore CPU.* As the data-plane subsystem appears as a standard peripheral from the point of view of the control-plane, the shared-memory multicore architecture can be employed as a CPU without affecting the data-plane subsystem.
- Memory hierarchy.* A unified address space enables the optimization of memory arrangement and hierarchy in terms of memory size, bandwidth and latency by combining cache, SDRAM and scratchpad components.
- Software development.* The CoreManager virtualizes the PEs, that is, it hides to the programmer the details of the data-plane configuration in terms of number and type of installed or active PEs. As a result, the code execution on systems with different PE configurations is feasible.

(ii) The Data-Plane Subsystem

- Data locality.* Data locality is exploited by using scratch-pad local data memories. PEs cannot access the global memory directly. Instead, the data exchange between global and local memories is accomplished by the CoreManager's DMA resources. The PEs' computations are carried out using the local memory.
- Software development.* The PE's address space is isolated from that of the control-plane subsystem and therefore, the PE's software may be developed without any knowledge of the control-plane subsystem.
- Heterogeneity.* In general, any number and type of PEs can be used in the data-plane subsystem resulting in a heterogeneous many-core architecture. In addition, the PEs can be general purpose, application specific or nonprogrammable cores.
- Scalability.* In order to provide the computing resources suitable for a given application, the number and type of PEs can be configured statically and/or dynamically, without necessity of adapting to the application software.

2.2. CoreManager

The CoreManager is responsible for the task scheduling, the PE allocation and the data transfer management. In this work, the concept of centralized runtime control is addressed. The CoreManager is hence designed as a dedicated unit, allowing a global view on the system states (e.g., information from all PEs is collected). Moreover,

Fig. 6. Internal structure of the CoreManager.

interrupts associated with task, data and PE management are completely handled by the CoreManager. As a consequence, the overhead associated with context switching and interrupt handling is reduced at the application processor and the overall system performance is thus improved. *Direct memory access controllers* (DMACs) are used for burst data transfers between the global and the PEs' local memories.

The CoreManager can be implemented as an application-specific integrated circuit (ASIC), as an application-specific instruction-set processor (ASIP), or as a compiled software binary running on a general purpose processor. Each implementation style favors different kinds of applications. An ASIC can be used, for instance, in latency-critical environments, whereas an implementation on a general purpose core allows greater flexibility. For the latter case even a complete reconfiguration at runtime is possible. The CoreManager is thus highly adaptable to the application demands.

The *Data Management* is additionally performed by the CoreManager. The local memories are used as explicit intermediate data buffers, where data reside as long as they are needed. By this means, data locality is increased and performance is improved.

2.2.1. CoreManager: Internal Structure. A composition of the main components of the CoreManager is shown in Figure 6. The task description is sent by the application processor to the internal memory of the CoreManager. Only a limited number of tasks can be simultaneously present at the CoreManager. This number is determined by the *task window size*. As soon as the maximum number of tasks is reached, the application processor is stalled until a present task in the CoreManager is finished. The task description contains the necessary information to execute the task on a PE. The minimum information required consists of the task type and the task's input and output data region. The latter is specified by a pointer and the size of the region. As soon as all dependencies are resolved a task is placed in the task-ready list. After task scheduling and PE allocation, the DMACs are configured in order to transfer the necessary data to the PEs before the task execution.

The CoreManager is able to evaluate the input and output region of each task to build a *task dependency map*. Assume task T1 with output data 1D-array represented by the pointer and size tuple (p_1, s_1) , and task T2 with input data 1D-array associated with tuple (p_2, s_2) (Figure 7 (left)). The parameters p_1, p_2, s_1 and s_2 are assumed to be greater or equal zero. For the *dependency check*, the following equation must be evaluated:

$$d = (p_1, s_1) \cap (p_2, s_2), \quad (1)$$

Fig. 7. The principle of 1D (left) and 2D (right) data dependency checking.

where $\cap$ is an intersection logical operator producing *True* if the intersection of the two sets is not empty. If $d = \text{True}$ then the data dependency between T1 and T2 is present. If no dependency is found and at least one PE of the required type is available, the task is immediately scheduled. If a dependency exists, the predecessor and successor tasks are annotated. As soon as all predecessor tasks are finished the current task can be executed. It should be noticed that PE-specific functionality must be available in the CoreManager, since some PE types must be booted prior to the execution of a task, while other require initial allocation of the local memory.

In order to enable a more effective implementation of MIMO and multimedia algorithms, the CoreManager supports in addition 2D-array data transfers. The descriptions of input and output data are thus extended by the parameters *line count* and *stride*. The *stride* specifies the offset in bytes between two lines of the data block. The size determines the length of each line. An example of data dependency between two 2D transfers is shown in Figure 7 (right).

3. PROGRAMMING INTERFACE: TASKC

A programming interface called *taskC* was developed in this work. TaskC is similar to CellSs [Bellens et al. 2006]. It applies the concept of atomic kernels with input and output data. In contrast to CellSs heterogeneous PEs are supported. Furthermore, it is portable between systems containing different numbers of PEs of each PE type. The programming model is composed of a task call and a task definition. The latter must be compiled for each PE type. For some PE types, for instance, a nonprogrammable hardware block, no task definition is necessary. In Listing 1 an example of a task definition is shown. It is enclosed between pragma statements. These statements are evaluated by a special-purpose compiler, which splits the control and the task definition code in separate parts. Each task definition is compiled for all specified target PE types. Subsequently, a binary file containing all the compiled task definitions is created. Each task definition can be debugged, validated and verified without any change in the main program. This decoupling speeds up the application development process. In this regard, even an incremental development of the applications is possible, for instance,

LISTING 1. Task definition

```
# pragma TASK_BEGIN some_task // start task definition
# pragma TASK_TYPE some_dsp // specify target PE
void some_task (void *i1, void *i2, void * out) {
    // function implementation goes here
    // calls to sub - functions are allowed
}
# pragma TASK_END // finish task definition
```

by using nonoptimized but verified task definitions which can be replaced in the future by their performance-optimized versions.

For task instantiation a task function call must be used. An example is shown in Listing 2. The memory regions for each task call are to be specified by the application designer by using the IN, OUT, and INOUT macros. A unified address space of the control-plane enables the data dependency analysis performed by the CoreManager. In the case that dependencies are present, the task is delayed until its predecessor task has finished its execution. Hence, the synchronization between tasks is ensured. The control code dependencies are evaluated by the application processor. An example of a control code dependency is the if-else clause in Listing 2.

LISTING 2. Task example: main function, including task calls

```
# include "task.h"
/* task data declarations */
int * input1, * input2, * output1, * output2 ;
int main () {
    /* initialize memories and data here */
    /* instantiate tasks */
    if (this_is_true)
        task (some_task, IN (input1, 256), OUT (output1, 16), OUT (output2, 64));
    else
        task (other_task, IN (input1, 64), IN (input2, 32), OUT (output1, 1024));
}
```

In order to synchronize control and data planes the taskSync command is employed, implementing a barrier synchronization mechanism. It is called on the application processor and it forces the CoreManager to inform the application processor about the completion of all previously started tasks. A task is completed as soon as all output data are written back to the global memory. Thus, memory consistency is assured.

4. TOMAHAWK CASE STUDIES

As it has been mentioned in the previous sections, the CoreManager is an essential element of the Tomahawk architecture, influencing the performance and the capabilities of the MPSoC system significantly. One of the most representative metrics characterizing the CoreManager's performance is the scheduling time, that is, the time difference between the task arrival and the task dispatch to a PE. The smaller the scheduling time relative to the task execution time is, the lower the degradation of the system performance will be. However, to reduce the scheduling time additional HW resources are required, which in turn increases the CoreManager costs. Therefore, a trade-off between the CoreManager performance and costs is necessary for system optimization. In addition to this, the flexibility of the CoreManager is a very desirable feature which enables reconfiguring the scheduling algorithms. The aim of this work is to provide a comparative analysis of several CoreManager implementations, based on different design paradigms. In this section three MPSoCs case-studies are presented and analyzed, particularly a CoreManager solution based on a reduced instruction set computer (RISC), an application-specific integrated circuit (ASIC) and an application-specific instruction-set processor (ASIP).

4.1. Software Architecture (RISC)

We start our investigation with a pure software solution of the CoreManager which provides the highest flexibility but also the lowest performance. In this approach, a

heterogeneous MPSoC composed of several types of ARM cores is employed, being one of the cores exclusively dedicated to the CoreManager [Arnold et al. 2010, 2011b]. The software approach enables high flexibility, for instance, scheduling and allocation algorithms can be changed even dynamically at runtime. The task window size determines the maximum number of simultaneous tasks in the CoreManager and is only limited by the available local memory of the CoreManager. For each task up to twelve data transfers can be specified. IN, OUT, and INOUT transfer types are available. Each of them can be used in 1D and 2D mode.

Three types of task dependency checking modes can be distinguished. Independent tasks are assumed in the first mode of operation. Thus, no dependency checking will be performed and all tasks can be executed as soon as the task description is transferred to the CoreManager and a suitable PE is available. The second mode performs an explicit specification of the task dependencies. All predecessor and successor tasks will be annotated with unique task identifiers (*taskId*). The data dependency checking at runtime is the third and most sophisticated mode of operation. In this case, the CoreManager checks a task precedence constraint with respect to other present tasks using equation (1). As soon as an overlap of memory regions of two transfers is found, a dependency is annotated. Read-Read transfers are omitted. The execution of the current task is delayed until all dependencies are resolved.

The information obtained by the data dependency checking state is reused for explicit memory management. For this purpose, the data is kept on a PE for a successor task. Bypassing data from one PE to another is also possible. By these means, data locality is increased and the number of necessary data transfers is decreased, thus improving system performance.

The interface between the application processor and the CoreManager is a potential bottleneck as soon as the task arrival rate reaches a point of saturation. For instance, for each task a task type must be transferred. In addition to this, for each task's transfer a pointer, a type, and a size must be transferred. For some applications it is possible to combine several tasks to one *taskSet*. This enables the specification of several independent tasks by means of only one task description. In the case of 2D transfers, the data offset is additionally specified. This value will be added to the pointer on each task iteration. The number of iterations is specified after the task type definition. A *taskSet* example is shown in Listing 3.

LISTING 3. TaskSet specification

```
taskSet(      taskType,
              <number of tasks>
              IN_SET(ptr_in, size_in, offset_in),
              OUT_SET(ptr_out, size_out, offset_out)
            );
```

For system analysis a heterogeneous MPSoC (Figure 8) is used to evaluate the software CoreManager approach. An ARM11MPCore, including four ARM1176 cores, is used as an application processor. Typically, an operating system is running on these cores. The CoreManager is implemented in software, running on an ARM926 processor. 64 Kbyte of local memory is available for instructions and data. The data plane consists of ten processing elements (PEs): eight ARM926 and two ARM1176 cores. These PEs are used for task execution. Each of them has 64 Kbyte of local memory, evenly divided in data and instruction memory. All PEs have sole access to their corresponding local memory. Thus, no cache misses occur. The memory access latency of all PEs is one clock cycle. Flash memory (connected by an ARM PL354 static memory controller), SDRAM

Fig. 8. System Model of the software CoreManager.

(connected by an ARM PL340 dynamic memory controller), and a 256 Kbyte on-chip scratch pad memory are integrated. These memories compose the global memories. Four direct memory access controllers (DMACs) ARM PL080 are responsible for the data transfers. The Platform Baseboard for the ARM11MPCore is used for reference timing, for instance, for the latencies of the local and global memories as well as the interconnection.

A priority can be annotated for each application, thus enabling soft real-time capabilities. For each priority a dedicated task queue is introduced in the system. The next task is always taken from the queue with highest priority. Data transfers are prioritized as well, hence allowing applications with higher priority to gather all necessary DMA channels. This leads to an increased throughput and a decreased latency on the application level.

Figure 9 shows the structure and internal states of the CoreManager. A task description is sent to the CoreManager. As soon as the task description is transferred, the data dependencies are dynamically checked with previous tasks. As previously mentioned, in the presence of dependencies the successor tasks are annotated. If no dependencies are found the task is placed in the task-ready list. As soon as a suitable PE is available the task is scheduled, local memory is allocated and the necessary transfers are included in the transfer list. In the case of a powered-off PE, the PE is booted prior to the task execution. Additional boot-code transfers are therefore scheduled. The PE frequency is set according to the task priority and the system load. As soon as the task is finished the OUT transfers can be executed.

A simulation profiling the CoreManager on an ARM926 general-purpose RISC core at a frequency of 300 MHz was performed, using a GSM physical layer benchmark. On average, 7000 cycles were necessary to schedule a task. The relative distribution of processing time among the different CoreManager components is shown in Figure 10. As expected, a major part of the processing time is spent in the dynamic data dependency checking module, since all data transfers of a new task must be checked against

Fig. 9. Software CoreManager: structure and internal states

Fig. 10. Processing time of the CoreManager internal states for the pure software solution (normalized to PE allocation).

all data transfers of the already present tasks. The CoreManager software solution is hence not suitable for those applications where its long scheduling time (relative to the task execution time) may unacceptably degrade the system performance. This is in fact the case of the GSM physical layer processing: a scheduling time of $23\mu\text{s}$ (7000 cycles at 300MHz) would be unacceptable for baseband applications. Further details on system analysis and runtime management overhead can be found in Arnold et al. [2011b].

Fig. 11. Block schema of Tomahawk MPSoC.

Due to the flexibility of the software CoreManager an adaption to further system requirements and capabilities is possible. In Arnold et al. [2011a] an adaption towards a battery-aware task scheduling is shown. In this approach a collector, a dc-dc converter, and a battery module are introduced in the system. The CoreManager operates under different battery-aware modes to extend battery lifetime. The PE allocation, the task scheduling, and the data transfers are then executed considering the current battery status. In the case of unreliable systems, a failure-aware dynamic task scheduling approach can be applied [Arnold et al. 2011c]. It allows a detection and correction of sporadic errors. Furthermore, error-free PEs are favored while faulty ones are isolated. For this purpose, error states are introduced for each PE in the CoreManager in order to track the task execution failures.

4.2. Hardware Architecture (ASIC)

In Limberg et al. [2008] a dedicated hardware solution of the CoreManager was designed, which presented good performance scalability, for instance, for OFDM receiver algorithms. In this HW solution, the data dependency check is parallelized, which leads to a reduction of the dependency check latency by approximately two orders of magnitude (60–80 cycles). Furthermore, the task scheduling, the PE allocation and the transfer scheduling can be performed concurrently as well. For this purpose, separate units dedicated to each PE are used. As a case study, a heterogeneous Tomahawk MPSoC has been designed targeting baseband and multimedia applications. A block schema of the Tomahawk MPSoC is depicted in Figure 11. The control plane consists of two Tensilica DC212GP RISC processors, global memory and peripherals. The global memory is composed of three independent external DDR-SDRAMs, an external I2C EEPROM (for the boot code), and an on-chip low-latency 256 Kbyte SRAM memory. The global memories are mapped to a unified address space of the Tensilica CPUs and are accessible by all control-plane components. Tomahawk's control-plane is equipped with a rich set of peripherals (Table II) as well as additional processing cores, for instance, a filter ASIP, an LDPC decoder ASIP [Bimberg et al. 2007] and an entropy decoder. The on-chip-interconnect is realized using two low-latency, high-bandwidth, crossbar-like NoCs (Table III) [Winter and Fettweis 2006]. The data plane of the Tomahawk is composed of six fixed-point vector DSPs (VDSPs) and two scalar floating-point DSPs (SDSPs). All computing cores are based on the Synchronous Transfer Architecture (STA) [Cichon et al. 2004], except the Tensilica RISC's. STA cores employ local memories, direct data routing and explicit bypassing, which in turn reduces severely

Table II. List of Tomahawk Peripherals

Component	Description
FPGA bridge	Connecting additional external components
PCI Express interface	Single lane connection to host computer with bandwidth 2Gbit/s
VGA/Streaming interface	Connecting: i) AD and DA converters for SDR applications or ii) standard VGA controllers for multimedia applications
DMA controller	Freely programmable general purpose DMA controller
UART	Serial interface
GPIO	8-bit parallel general purpose I/O

Table III. Bandwidth of Data Transfer for Selected Tomahawk Components

Attribute	Bandwidth [Gbit/s @ fclk = 200Mhz]
Off-chip DDR-SDRAMs	3 parallel ports, 5.47/port
On-chip SRAM	2 parallel ports, 5.47/port
NoC control-plane	10 parallel p2p connections, 5.47/connection
NoC data-plane	3 parallel p2p connections, 5.47/connection

Fig. 12. Hardware CoreManger: building blocks.

the number of memory and register file accesses thereby reasonably lowering the power consumption.

4.2.1. Hardware CoreManager. Figure 12 shows the hardware CoreManager. The task description is transferred from the application processor DC212GP over the NoC to an input memory in the CoreManager. As a first step all transfer types within the memory identifier are analyzed. Subsequently, the dependency checker dynamically analyzes the intertask dependencies. In the Launch Reordering Table (LRT) all the dependencies are annotated at the corresponding predecessor tasks. As soon as all the dependencies of a task are resolved a core can be allocated by the *Core Selector*. For this, the heterogeneity of the platform (i.e., the available PE types) is taken into account. In the next step the local memory can be allocated. The input and output transfers

Fig. 13. Die photo of the Tomahawk prototype implemented in UMC 130nm.

Table IV. Implementation Results of Selected Tomahawk Components

Component	Power consumption [mW @ 175 MHz, 1.3 V]	Area [mm ²]	Memory Size of Area [%]
Tensilica DC212GP	30	2.50	15.8
Scalar DSP	27 ¹	3.33	91.1
Vector DSP	85 ²	3.80	79.5
Filter ASIP	86	4.54	85.0
LDPC Decoder ASIP	354	7.89	64.0
CoreManager	282 ³	5.95	24.3

¹FIR filter benchmark (measured)

²FFT benchmark (measured)

³Power consumption of fully loaded Core Manager without clock gating (estimated)

are configured in the Fetch (FRT) and Put Reordering Table (PRT), respectively. Data transfers are executed by three DMACs, each of them being responsible for one DDR memory bank. Lastly, PE status flags are utilized to update the PE's status, for instance, for indicating task completion.

4.2.2. Physical Realization. A chip prototype of the Tomahawk MPSoC was implemented in UMC's 130nm CMOS process with 8 metal layers (Figure 13). The total chip size including 480 I/O cells is $10 \times 10\text{mm}^2$, with a total transistor count of 57 million. A moderate target clock frequency of 175 MHz was selected for the purpose of reducing the power consumption. The typical case core supply voltage is 1.2V, the I/O voltage is 3.3V and 2.5V for the SSTL2 I/Os. Table IV summarizes the measured power consumption of the core components at 1.3V and 175 MHz. The total silicon area occupied and the relative memory size are additionally depicted.

The heterogeneous data-plane subsystem of the Tomahawk MPSoC is composed of several vector and scalar DSPs whose features are summarized in Table V. Six vector DSPs (VDSPs) are implemented providing a total maximum performance of

Table V. Specification of Tomahawk's PEs

Component	Vector DSP	Scalar DSP
Data format	16 Bit fix-point	32 Bit floating/fix-point
Architecture	5-issue VLIW 4-way SIMD	3-issue VLIW Conditional instruction execution
Arithmetic	Fractional/Integer, Overflow/Saturation	Single precision FP (1FLOP/2cycles) Integer
Program memory	32 kB	32 kB
Data memory	16 kB (dual port)	32 kB (dual port)
Max. Performance	20 MOPS/MHz	2 MOPS/MHz

120 MOPS/MHz (21 GOPS at 175 MHz clock frequency). The measured average power consumption is 85 mW/core using a FFT benchmark. The VDSP supports concurrent computation and external data transfer, that is, data prefetching, for a pending task.

The scalar DSP (SDSP) is primarily dedicated to algorithms with high dynamic range requirements, for instance, channel estimation or matrix inversion for Multiple-Input Multiple-Output (MIMO) channels processing. Moreover, conditional execution of all instructions for low overhead control structures, for instance, bit-stream processing, is possible. The maximum computing power of both SDSPs is 0.7 GOPS at 175MHz. The measured average power consumption of the FIR filter benchmark is 27 mW/core. The CoreManager occupies 5.95mm² of silicon (including 1.7mm² for the three DMA controllers and 1mm² for a debugging unit). Fully loaded, it consumes 282 mW.

The reason for the relatively high power-area costs resides in the fact that a high degree of parallelism is exploited in the CoreManager in order to reduce the scheduling overhead. As previously mentioned, the software solution overhead spoils the performance scalability due to the relatively long scheduling time in comparison to the task execution times. The dedicated parallel HW solution of the CoreManager was designed to solve this problem, achieving a scheduling rate of ~ 70 cycles/task. Presenting an energy dissipation of about 100 nJ/task, the hardware solution outperforms in addition the energy efficiency of the software solution (500 nJ/task, host Tensilica DC212GP) by a factor of five.

In the Tomahawk concept, the number of PEs is chosen dynamically by means of clock and/or power gating, subject to the performance demands of the underlying application. For this purpose, the clock of the idle PEs can be switched off explicitly, in addition to clock gating at register-level. Power gating of PEs has been omitted in this work due to the negligible effects of the leakage power in 130 nm technology. The CoreManager itself is not clock-gated, hence leaving room for further power reduction. Both NoCs are based on a point-to-point master-slave architecture with 32-bit data-width, occupying an area of 0.4mm². In burst mode, the NoC supports transfers of up to 63 data words with static priority arbitration per slave. The crossbar-like architecture allows multiple parallel master-slave connections. The NoC operates at the full system clock frequency, achieving a sustained throughput of 5.47 GBit/s per connection. The dynamic power consumption of the fully loaded system considering both DC212GP cores, all vector and scalar DSPs, the CoreManager and the LDPC ASIP is 1260 mW.

For the purpose of performance scalability analysis, a communications signal processing benchmark corresponding to a simplified OFDM receiver has been developed. The receiver comprises an OFDM demodulator, a frequency domain equalizer and a QAM demapper (16 QAM), with IFFT size of 256. The analysis results correspond to real measurements on the Tomahawk. A single iteration, that is, demodulation and detection of one OFDM symbol, requires about 49000 cycles on a single VDSP. During the experiment, the number of active VDSPs has been dynamically selected in runtime.

Fig. 14. Results of the Tomahawk prototype regarding the throughput (left) and the core utilization (right), using an OFDM benchmark.

Fig. 15. HW/SW System Model, including NoC, Router (R), memories (MEM), application processor (APP), CoreManager (CM) and several types of processing elements (PE).

The throughput in terms of the number of receiver iterations per 1ms is depicted in Figure 14 (left) as a function of the number of active PEs. The maximum speedup achieved is four in this case. The performance loss for system configurations with more than three PEs is due to the limited bandwidth (only three ports) of the SDRAMs and the corresponding DMA channels. This has been identified as the main bottleneck limiting the performance scalability for the current system arrangement. The utilization ratio of the VDSPs for the same benchmark is depicted in Figure 14 (right).

4.3. Hardware/Software Architecture (ASIP)

In the previous sections, two extremes of the CoreManager's flexibility-performance design space have been discussed. While the ASIC solution provides the best performance (~ 70 cycles/task) without any flexibility, the RISC-based software solution features full reprogrammability with poor scheduling performance (~ 7000 cycles/task). The motivation of this part of the work is to study a CoreManager implementation presenting the performance of the ASIC realization while conserving the flexibility of the software approach. The general procedure we follow is the acceleration of critical parts of the reference software RISC solution by implementing additional dedicated instructions. For this purpose, a Tensilica framework has been adopted.

A heterogeneous Multiprocessor System-on-Chip (MPSoC) is shown in Figure 15. All blocks are connected by a Network-on-Chip (NoC). Routers (R) are integrated for data packet scheduling and arbitration. Each router is connected to its neighbors by point-to-point data links. Dimension-order routing (XY) is used. Further details about the NoC

Fig. 16. Selected platform components: a) PE subsystem, b) Application processor subsystem, c) CoreManager subsystem.

and the routers can be found in Winter and Fettweis [2006, 2011]. Global memory ports (MEM0, MEM1 and MEM2) are used to connect the system to off-chip SDRAMs. The application processor (APP) executes the sequential part of an application and hosts the operating system. The data plane of the MPSoC consists of eleven PEs. Particularly four DSPs, five general purpose (GP) cores and two ASIPs. One of the ASIPs is responsible for forward-error correction while the other performs multi-antenna (MIMO) symbol detection and demapping. The data plane is controlled by the CoreManager (CM). It is responsible for the dynamic data dependency checking, the task scheduling, the PE allocation, the data transfer management and the power management. In the latter case it determines the on-off power status and the frequency of each PE.

A more detailed view of each block is shown in Figure 16. In Figure 16(a) a FIFO is used to gather the tasks. As soon as a task is available the CoreManager Spin-Off (CM.SO) is responsible for the PE configuration and the data transfers. Up to four tasks can be scheduled to one PE. The task description in the FIFO contains all the necessary information for the data transfers and the task execution. The data transfers and the task execution are concurrently executed if a dedicated bit is set in the task description. Thus, explicit prefetching of data is enabled. Nevertheless, the CoreManager is responsible for the configuration of the CM.SO and the local memory allocation for the data transfers of each task. As in the previous cases, each PE can solely operate on its own local memory (no cache misses can occur). Consequently, the task execution time is deterministic, leading to a better predictability on system level. In Figure 16(b) the application processor is shown. In this case a Tensilica 570t is employed. It has 2-way set associative instruction and data caches, each 16Kbyte in size. In Figure 16(c) the CoreManager and its associated components are shown. The CoreManager works exclusively on local on-chip memories. 32Kbyte instruction and 32 Kbyte data memory is available. The CoreManager Transfer Unit (CM.TU) is responsible for the data transfers between the CoreManager's local memories and any other memory in the system. Timers and FIFO memories are integrated as well. The *DebugUnit* can be used for online and offline debugging, for instance, tracing the internal states. The dynamic decisions of the CoreManager can be thus visualized and analyzed.

Fig. 17. Evolution of the dynamic data dependency checking instruction.

4.3.1. CoreManager Instruction Set Extensions. The instruction set of the CoreManager is extended to increase the performance and the energy efficiency while keeping the flexibility of the software implementation. Therefore, very large instruction words (VLIW) as well as single-instruction multiple-data (SIMD) instructions are introduced in the design. For comparison a basic Tensilica LX4 core is used for reference. The basic LX4 core is configured with several additional functional units, for instance, a full-adder and a multiplier. A plain-C version of the CoreManager software is running on it. A Verilog-like language is used to integrate new instructions. In the first step the VLIW approach is used to group instruction for parallel execution. The compiler assists this step, for instance, it helps with register allocation. The SIMD instructions enable the concurrent execution of one instruction on multiple data words. More information of the newly introduced instruction set, including an overview of all available instructions, can be found in [Arnold et al. 2012].

As previously mentioned, the most time consuming part of the CoreManager is the dynamic data dependency checking. The evolution of this part and the transformation of the instruction set are shown in Figure 17. In the first line the C-Version is presented. Two memory regions of two transfers are compared. Each of them is specified by a pointer ($p0$ or $p1$) and a size ($s0$ and $s1$). Altogether, two compare, two subtract and one OR instruction are necessary. In a first step these instructions can be merged in one *asm_depCheck* instruction (2). Furthermore, false read-read dependencies are taken into account. In (3) 64-bit registers and a 64-bit data bus are introduced. In addition, SIMD can be applied (4). In a last step explicit load instructions can be used to increase data locality and thus decrease the number of memory transfers.

Results on component level are shown in Figure 18. A GSM physical layer application is executed as benchmark. Odd bars represent the Plain-C version, while even bars belong to the VLIW+SIMD execution. Minimum, average and maximum processing time are shown. The VLIW+SIMD version outperforms the Plain-C version. Especially the processing times of the dynamic data dependency checking (−94%), PE allocation (−85%) and task scheduling (−47%) are reduced. The PE allocation and the start code transfer have a fixed processing time even in the presence of a heterogeneous system. The processing time of the dynamic data dependency checking and the modification of the start codes vary according to the number of existing tasks in the system and the number of transfers per task. In corner cases an improvement of up to 97% can be achieved [Arnold et al. 2012].

Fig. 18. Processing time comparison on component level between Plain-C and SIMD+VLIW versions (logarithmic scale, application: GSM physical layer).

Fig. 19. Processing time of the CoreManager for dynamic data dependency checking and task initialization by varying the number of present tasks (logarithmic scale, 4 data transfers per task).

4.4. CoreManager Performance and Area Comparison

In this section the most time consuming part of the CoreManager, the dynamic data dependency checking, will be analyzed and compared for the three previously described CoreManager approaches. The same number of input/output data transfers is assumed. In terms of processing time (Figure 19), the ASIC approach (constant processing time of 70 cycles/task) clearly outperforms the flexible CoreManager implementations (RISC and ASIP). The current ASIC implementation can handle up to 16 tasks. Thus, a new task must be checked for intertask data dependencies with up to 15 tasks. It should be noticed that such a limitation does not exist in the flexible approaches. The ASIP solution presents a processing time approximately one order of magnitude lower than the RISC (ARM926) realization.

Table VI. Area and Performance Comparison of Different CoreManager Implementations

	ARM926	ASIC	ASIP
Area ^a [mm ²]	0.25 ^b	1.13 ^c	0.26
Frequency [MHz]	240 ^b	175	465
Time ^d [μ s]	20.35	0.4	0.9
AT Product ^e [mm ² * μ s]	4.59	0.45	0.23

^aScaled to 65 nm low power process, typical case (25°C, 1.25 V);

^bKeating et al. 2007; ^cCoreManager area of ASIC (excluding memory), scaled to 65nm by a factor of 4; ^ddynamic data dependency processing time for 8 tasks; ^earea-timing (AT) product.

In Table VI an area and performance comparison of all CoreManager implementations is shown. The processing time of the dynamic data dependency and the task initialization stage is depicted as well, assuming 8 already available tasks in the system. In order to compare the true silicon complexity, the area-timing (AT) product is additionally presented. The ASIP core has been synthesized with Synopsys Design Compiler for 65 nm low power process from TSMC under typical case conditions (25°C, 1.25 V). Only the logic area is considered, disregarding the area of the local memories. For timing correctness, the interfaces to the local memories have been nevertheless included. The reported maximum achievable frequency is 465 MHz and the occupied area is 0.26 mm². The AT product of the ASIP approach shows an improvement factor of nearly 2 in comparison to the ASIC and a factor of approximately 20 in comparison to the ARM926. In the case of 15 available tasks the AT product of the ASIP is 0.39 (still slightly better than the ASIC), while the ARM926 rises up to 8.6.

5. CONCLUSIONS

This work presented the Tomahawk framework comprising a model of computation, a programming interface, a CoreManager runtime management unit and an MPSoC architectural template. The Tomahawk MPSoC architecture targets primarily low power communications signal processing. The major innovations of this work are i) a dedicated runtime scheduler called CoreManager capable of scheduling tasks on processing element clusters with dynamically varying number and type of processing elements and ii) a programming interface supporting the heterogeneity of processing elements and specific 1D and 2D data arrangements for dependency check and data transfer. In addition to this, the platform is programmable in high level C/C++ language.

We studied and compared three different CoreManager approaches regarding scheduling performance and flexibility: i) a software RISC, ii) a full-hardware ASIC and iii) an ASIP-based solution. As demonstrated by the obtained performance figures, the ASIP-based CoreManager improves reasonably the scheduling performance with regard to the RISC approach by introducing some instruction-set architecture extensions as well as by exploiting instruction (VLIW) and data (SIMD) level parallelism. In addition to this, as the ASIP solution is based on a general purpose core, it remains highly flexible in terms of programmability. It allows a faster processing of critical parts, that is, the dynamic data dependency checking, task scheduling, PE allocation and data transfer management. The obtained results show an improvement for the dynamic data dependency checking stage greater than one order of magnitude in processing time compared to the ARM926-based approach. Future research will focus on the extension of the CoreManager's capabilities, for instance, hard real-time support, improving local data reuse, and introducing multicluster PE hierarchy.

ACKNOWLEDGMENTS

We would like to acknowledge Prof. René Schüffny, head of the Parallel VLSI-Systems and Neural Circuits Chair at the TU Dresden, for providing his experienced team of researchers for the backend work. Furthermore, we would like to thank Hendrik Ahlendorf, Frank Siebler, Pablo Robelly, Markus Ullmann, Johannes Lange, Arne Lehmann, Boris Boesler, Patrick Herhold, Reimund Klemm, Marcos B. Tavares and Marcel Bimberg for their contribution to design and development of MPSoC prototype. We are also thankful to the ZMD AG Dresden for providing access to their wafer prober and to the Institute of Semiconductors and Microsystems at the TU Dresden for providing image of a die. Finally, we would like to thank Synopsys, ACE, Tensilica, Xilinx and Altera for sponsoring Software, IPs and prototyping FPGAs.

REFERENCES

- O. Anjum, T. Ahonen, F. Garzia, J. Nurmi, C. Brunelli, and H. Berg. 2011. State of the art baseband DSP platforms for Software Defined Radio: A survey. *EURASIP J. Wireless Comm. Networking*.
- O. Arnold and G. Fettweis. 2010. Power aware heterogeneous MPSoC with dynamic task scheduling and increased data locality for multiple applications. In *Proceedings of the X International Workshop on Systems, Architectures, MOdeling, and Simulation (SAMOS'10)*.
- O. Arnold and G. Fettweis. 2011a. Self-aware heterogeneous MPSoC with dynamic task scheduling for battery lifetime extension. In *Proceedings of the Workshop on Computing in Heterogeneous, Autonomous N' Goal-oriented Environments (CHANGE'11)*.
- O. Arnold and G. Fettweis. 2011b. On the Impact of dynamic task scheduling in heterogeneous MPSoCs. In *Proceedings of the International Conference on Embedded Computer Systems Architectures, Modeling and Simulation (SAMOS'11)*.
- O. Arnold and G. Fettweis. 2011c. Resilient dynamic task scheduling for unreliable heterogeneous MPSoCs. In *Proceedings of the SCD Semiconductor Conference Dresden (SCD'11)*.
- O. Arnold, B. Nöthen, and G. Fettweis. 2012. Instruction set architecture extensions for a dynamic task scheduling unit. In *Proceedings of the IEEE Annual Symposium on VLSI (ISVLSI'12)*.
- K. Asanovic, R. Bodik, B.C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D.A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. 2006. The landscape of parallel computing research: A view from Berkeley. Tech. Rep. UCB/EECS-2006-183, University of California, Berkeley.
- P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta. 2006. CellSs: A programming model for the Cell BE architecture. In *Proceedings of the ACM/IEEE Supercomputing Conference*.
- K. Berkel, F. Heinle, P. P. E. Meuwissen, K. Moerman, and M. Weiss. 2005. Vector processing as an enabler for software-defined radio in handheld devices. *EURASIP J. Applied Signal Processing*, 16, 2613–2625.
- M. Bimberg, M. B. S. Tavares, E. Matus, and G. Fettweis. 2007. A high-throughput programmable decoder for LDPC convolutional codes. In *Proceedings of the 18th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP'07)*.
- B. Bougard, B. De Sutter, S. Rabou, D. Novo, O. Allam, S. Dupont, and L. Van der Perre. 2008. A coarse-grained array based baseband processor for 100Mbps+ software defined radio. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*. 716–721.
- G. Cichon, P. Robelly, H. Seidel, E. Matus, M. Bronzel, and G. Fettweis. 2004. Synchronous transfer architecture (sta). In *Proceedings of the 4th International Workshop on Systems, Architectures, Modeling, and Simulation (SAMOS'04)*. 126–130.
- K. Fatahalian, T. J. Knight, M. Houston, et al. 2006. Sequoia: Programming the memory hierarchy. In *Proceedings of the IEEE Conference on Supercomputing*.
- M. Frigo, C. E. Leiserson, and K. H. Randall. 1998. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.
- A. Ghuloum, E. Sprangle, J. Fang, et al. 2007. Ct: A flexible parallel programming model for Tera-scale architectures. Tech Rep., Intel Corporation.
- J. Glossner, D. Iancu, M. Moudgill, G. Nacer, S. Jinturkar, S. Stanley, and M. Schulte. 2007. The sandbridge sb3011 platform. *EURASIP J. Embed. Syst.* 1, 16–16.
- M. Horowitz and W. Dally. 2004. How scaling will change processor architecture. In *Proceedings of the IEEE Solid-State Circuits Conference*. 132–133.
- M. Keating, D. Flynn, R. Aitken, and K. Shi. 2007. *Low Power Methodology Manual: For System-on-Chip Design*. Springer.
- J. Kneip, M. Weiss, W. Drescher, et al. 2002. Single chip programmable baseband ASSP for 5 GHz wireless LAN applications. *IEICE Trans. Electronics E85-C*, 2, 359–367.

- E. A. Lee and D. G. Messerschmitt. 1987. Synchronous data flow. *Proc. IEEE* 75, 9, 1235–1245.
- T. Limberg, M. Winter, M. Bimberg, R. Klemm, E. Matus, M. B. Tavares, G. Fettweis, H. Ahlendorf, and P. Robelly. 2008. A fully programmable 40 GOPS SDR single chip baseband for LTE/WiMAX terminals. In *Proceedings of the 34th European Solid-State Circuit Conference (ESSCIRC'08)*.
- Y. Lin, H. Lee, M. Who, et al. 2006. SODA: A low-power architecture for software radio. In *Proceedings of the 33rd International Symposium on Computer Architecture (ISCA'06)*. 89.
- D. Liu, A. Nilsson, E. Tell, D. Wu, and J. Eilert. 2009. Bridging dream and reality: Programmable baseband processors for software-defined radio. *IEEE Commun Mag.* 47, 134.
- A. Nilsson, E. Tell, and D. Liu. 2009. An 11 mm, 70 mW fully programmable baseband processor for mobile WiMAX and DVB-T/H in 0.12 m CMOS. *IEEE J. Solid-State Circuits* 44, 1, 90–97.
- D. Novo, W. Moffat, V. Derudder, and B. Bougard. 2005. Mapping a multiple antenna SDM-OFDM receiver on the ADRES coarse-grained reconfigurable processor. In *Proceedings of the IEEE Workshop on Signal Processing Systems Design and Implementation*. 473–478.
- U. Ramacher. 2007. Software-defined radio prospects for multistandard mobile phones. *Computer* 40, 10, 62–69.
- Tensilica. 2012. <http://www.tensilica.com>.
- M. Winter and G. Fettweis. 2006. Interconnection generation for system-on-chip design. In *Proceedings of International Symposium on System-on-Chip, Tampere*. 91–94.
- M. Winter and G. Fettweis. 2011. Guaranteed service virtual channel allocation in NoCs for run-time task scheduling. In *Proceedings of the Design Automation and Test in Europe (DATE'11)*.

Received December 2012; revised May 2013; accepted August 2013