A holistic methodology for network processor design

Olaf Bonorden* Nikolaus Brüls† Uwe Kastens‡ Dinh Khoi Le‡
Friedhelm Meyer auf der Heide* Jörg-Christian Niemann§ Mario Porrmann§
Ulrich Rückert§ Adrian Slowik‡ Michael Thies‡

Abstract

The GigaNetIC project aims to develop high-speed components for networking applications based on massively parallel architectures. A central part of this project is the design, evaluation, and realization of a parameterizable network processing unit. In this paper we present a design methodology for network processors which encompasses the research areas from the application software down to the gate level of the chip. Key components of this holistic approach have been successfully applied to characteristic examples of architecture refinements.

1. Introduction

A particular attraction of the GigaNetIC project grounds on the interdisciplinary coupling of the different research groups that marked the project right from the beginning. Following a top-down approach, network applications are analyzed and partitioned into smaller tasks, e. g., according to a bulk-synchronous parallel programming model. The tasks are mapped to dedicated parts of the system, where a parallelizing compiler exploits inherent instruction level parallelism. The hardware has to be optimized for these programming models in several ways. Synchronization primitives for both programming hierarchies have to be provided and memory resources have to be managed carefully. Another important task is the development of hardware accelerators for special embedded applications to achieve a higher throughput and to reduce energy consumption. There are members of three research groups involved. The Research Group of Theoretical Computer Science develops and analyzes models for the architecture and the on-chip interconnection network. The Research Group of Programming Languages and Compilers develops the retargable and parallelizing compiler, and the Research Group of System and Circuit Technology is intrusted with the realization of a resource-efficient system design. As an industrial partner, Infineon Technologies provides state of the art chip production technologies needed for such a complex System-on-Chip (SoC) design.

1.1. System architecture

The proposed architecture is based on massively parallel processing, enabled by a multitude of processors, which form a homogeneous array of processing elements arranged in a hierarchical system topology with a powerful communication infrastructure. The proposed architecture is structured into the following three domains: *SoC level*, *Cluster level*, and *PE level*.

As a main target for our project the resulting Network Processing Unit (NPU) architecture should be easily parameterizable in terms of number of clusters, number of processing elements per cluster, and the available bandwidth of the cluster interconnection network. Each of these domains offers characteristic opportunities for optimizations where the different research groups cooperate closly. Offering opportunities for parameterization the architecture can be adapted to new requirements of the applications and of the network infrastructure as well. In the following, we will briefly present the discussed domains.

1.2. SoC level

The coarse-grained level of our architecture is the SoC level. For integrating a huge amount of processors on a single chip it has to be assured that these units are able to communicate efficiently over an on-chip interconnection

*Heinz Nixdorf Institute and Institute of Computer Science, University of Paderborn, Germany, {bono,fmadh}@upb.de

†Infineon Technologies, Munich, Germany, Nikolaus.Bruels@infineon.com

‡Institute of Computer Science, University of Paderborn, Germany, {uwe,le,adrian,mthies}@upb.de

§Heinz Nixdorf Institute and Institute of Electrical Engineering and Information Technology, University of Paderborn, Germany, {niemann,mario,rueckert}@hni.upb.de

network. Design, analysis, and evaluation of such networks and the appropriate communication protocols are an important research topic. To support the software developer, special programming models and libraries are developed (cf. Section 2). Following these programming paradigms our chip will speed up many network applications in a cost and power saving way.

1.3. Cluster level

The next domain is the cluster level where the compiler exploits instruction level parallelism and takes advantage of custom hardware accelerators. Design of the compiler is driven by the need for flexibility on both ends. At the application level, the software for a network processor has to be adapted to new requirements within a short period of time, e. g., to fulfill new security standards, quality criteria, or to refocus the network processor to a specific networking domain. Furthermore, this software has to be mapped to hardware variants that favour diverse system parameters: high throughput, low power consumption, or low manufacturing cost. To meet these requirements, key parts of the application software and the compiler are generated from concise high-level specifications, as well as a cycle accurate simulator. This allows to evaluate quickly combinations of application software and hardware it is mapped to. Simulation provides feedback that gives rise to modifications on all levels of the system.

1.4. PE level

The Processing Engine (PE) level is the most fine-grained level for system modifications. On the one hand, modifications and extensions of the processor's instruction set are implemented. On the other hand, hardware accelerators for various tasks are developed. Our architecture builds on the S-Core processor, a 32 bit RISC processor core that we have developed previously (cf. Section 4). Through using the USLI technology provided by Infineon, which currently allows feature sizes of 130 nm, it is possible to shrink the area required for one S-Core to less than 0.2 mm^2. Another essential task is the design of an efficient on-chip network that – together with the memory hierarchy and the necessary control units – is needed for the high on-chip data throughput. The development of embedded applications realized in hardware and software accompanied by a cost analysis for area, power consumption and performance is also in our scope of interest.

2. Modelling the architecture

There are different ways to process data, e. g., a stream of packets, in parallel. The simplest way is called parallel servers. Here the packets are distributed among all the servers in a round robin fashion and processed independently of each other. This is very efficient, nearly the optimal linear speedup can be achieved. But this approach is only feasible if there are many small packets, because it does not improve the processing time of a single packet. But the main problem is: Packets are handled independently, i. e., one cannot have any state information that belongs to a stream. Thus an interconnection network between the processing units is needed to meet the demands of modern networking processors.

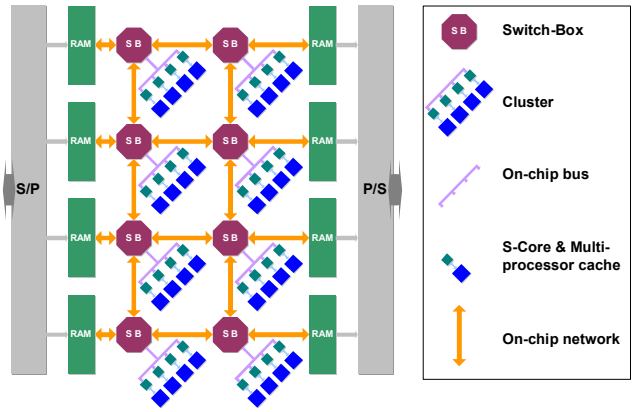

Figure 1. Architecture of the chip

In the next section we will discuss the topology of this on-chip network. In Section 2.2 we describe a model for parallel computing, the *bulk-synchronous parallel (BSP)* model and motivate its use for developing algorithms for our system.

2.1. Topology of the on-chip network

The main design goal for the on-chip network is efficiency, i. e., high throughput and small latency. But there are some restrictions: The degree of the nodes has to be constant and small due to implementation issues, and the links between nodes should have small length because of the latency. Moreover, the network should scale with the number of nodes.

We consider different topologies and simulate characteristic routing problems. Our first design contains 32 processing engines (PE). One commonly used technique to connect nodes is a bus. Busses reduce the maximal distance in a network (e. g., distance is one if all devices are connected by one bus), but the congestion is too high if there are more than a small number of nodes, i. e., busses do not scale with the number of nodes. Architectures that connect more than four processors to one shared memory have to use sophisticated crossbars instead of simple busses. We want to combine the advantages of a bus (small distance) avoiding the

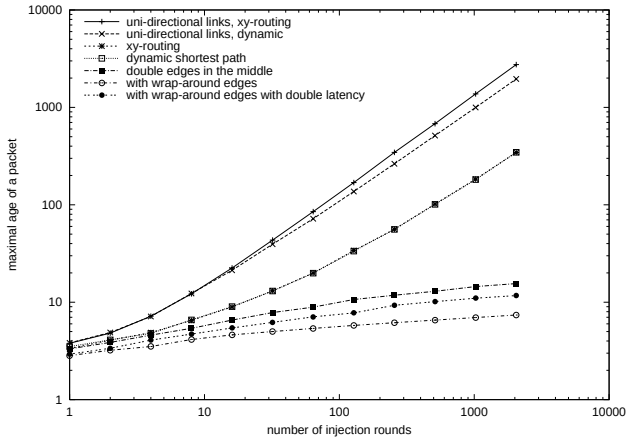

Figure 2. Routing on different kind of grids

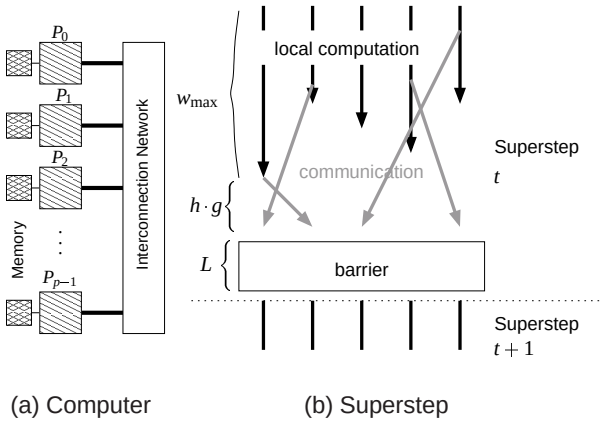

Figure 3. BSP model

loss of scalability by creating a hybrid network. A small constant number of nodes (e. g., four) is connected with a bus using shared memory. These clusters are connected by point to point links. See Figure 1 for an example.

In the following we will discuss the topology of the network between these clusters. Our simulations use a random routing problem. In each of the first n rounds, each processor injects a packet for a random destination. The simulation has finished if all packets have reached their destinations. Each link can handle one packet in a round (unidirectional in the first two tests) or one packet in each direction (bidirectional, all other tests). We consider the maximum time needed for a packet to reach its destination, this time is of the same order as the buffer size needed in each node and is a good indicator of the stability of the system under high load.

We start with a simple 2×4 mesh. We use a mesh because of its regular structure, all links have the same length and there are no crossings. Furthermore, a mesh can be easily scaled to fit the available chip area. We consider two different routing algorithms: xy-routing sends the packet first to the right column, then to the destination row, it is called an *oblivous* routing scheme because of its fixed path system, and second a dynamic routing scheme. Here the packets are sent along an arbitrary shortest path, so if the edge in one dimension is occupied, the other one is tried.

The results of our simulations are shown in Figure 2. The mesh cannot handle a huge number of packets. For unidirectional links there is a small difference for the two routing schemes, but they both offer inferior performance. If we use bidirectional links, both protocols take roughly the same time. There are almost always enough packets for each link and the links on the left column exhibit the same load as on the right links due to the random choices of the destinations. The problem of the 2×4 mesh is its small bisection. Congestion on the two edges in the middle degrades routing times. If we double the bandwidth of these two links, we get a much more stable and efficient network. Another approach is to introduce wrap around edges. This is even faster because it reduces the diameter of the network. But these edges have to be much longer than the other ones and the implementation may increase the latency of these two long edges. As shown in Figure 2, this is very efficient, even if the latency on the wrap-around edges is two, i. e., a packet needs two rounds to use these links, but one can send one packet each round, so the bandwidth is the same.

2.2. The BSP model

The *bulk-synchronous parallel (BSP)* model was proposed by Leslie Valiant in [18] as a bridging model for parallel computation, i. e., on the one hand BSP is a machine model for the hardware architecture and on the other hand a programming model for algorithm designers. The main goal is to find a common standard for both parts, like the von-Neumann model for sequential computers, that leads to efficient algorithms and implementations on various hardware. The BSP model is widely used for developing and analysing algorithms.

The model consists of three parts: A view of the computer (cf. Figure 3, (a)), i. e., an abstraction of the hardware, a programming model (cf. Figure 3, (b)) and a cost model for predicting the running time of BSP algorithms. A BSP computer consists of p processors with local memory connected by an arbitrary interconnection network that supports point-to-point messages and barrier synchronization. The model does not assume a specific topology or exploit locality. The program is divided into several parts, called *Superstep*s. In each superstep a processor can do local computations and send messages to other processors. At the end of each superstep a synchronization function is called. When all processors have reached this barrier, all messages sent in the previous superstep are received and can be pro-

cessed in the next superstep. Designing algorithms this way has two advantages: first, you do not have to consider receiving of messages, the communication is one-sided. This prevents deadlocks caused by missing or wrongly ordered receive operations. Second, the behaviour of the algorithm is always deterministic and does not depend on the order of messages or network timing. Moreover, BSP provides a cost model. The cost of a superstep is the sum of the maximum local work $w_{\max}$, the maximum amount of data that is sent or received by one processor h multiplied with the *gap* g of the network and the time L (*latency*) for one barrier synchronization.

There exists several libraries for efficient implementation of BSP algorithms on monolithic parallel computers as well as on workstation clusters, e. g., the PUB library [4]. Designing the whole system from the processors to the network allows us to consider the needs of a fast implementation of the BSP functionality, e. g., a new PE instruction for synchronization. This extension and the low latencies on the chip leads to very small BSP parameters for the gap g and the synchronization time L.

Formally, the BSP model has been used mainly for solving large problems with a coarse grained communication structure, we use this model for developing and implementing efficient algorithms for problems in the area of network processors because the small time needed for synchronizations allows fine grained algorithms.

3. Software Level

Within this section we describe design fundamentals of the parallelizing ANSI-C compiler, a corresponding simulator which incorporates a performance visualizer, and a generator for packet-classification engines. In the domain of NPU design exploration, this tool-set in concert bridges the gap between high-level programs encoded according to BSP, and the underlying massively parallel hardware. Important parts of the compiler, simulator, and classification engine will turn out to be generated from compact specifications for reasons described below.

3.1. Application Software

Even in the very specific domain of network processing there still exists a broad range of applications to be executed on dedicated network processors, the extreme examples being layer 2 switching on the very low end and layer 7 application inspection, e. g., for the sake of intrusion detection, at the very high end. Following a top down approach, it is the packet-classification task that drives the NPU and triggers hardware accelerators for compression, encryption and other computationally intensive tasks. Thus the software for

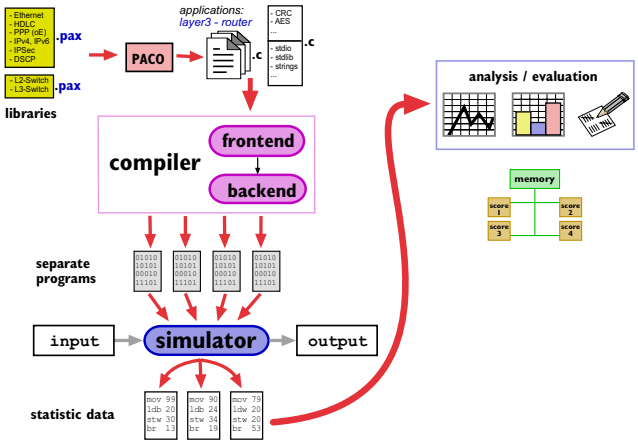

Figure 4. Software-Level Tool Chain

packet-classification provides the top-level software skeleton of an S-Core cluster. Functionally simpler pieces of software with clear interfaces are plugged into that skeleton and arrange for encryption, compression, framing and the like. Since our main goal is to establish a flexible design environment for NPUs, we decided to use compact specifications to describe the packet-classification task and to generate the software skeleton from these specifications. During the generation process, the compiler's optimizer takes care of the particular combination of protocols to be supported and important parameters of the hardware, e. g., number of PEs within a cluster, amount of memory per PE and the like, to synthesize an efficient packet-classifier that is optimized for the particular context to be deployed in.

The packet-classification task can be further decomposed into a task that repeatedly classifies a slice of a packet, and then triggers one or more actions depending on the result of this partial classification, until finally the entire packet has been processed. During this packet traversal the classifier processes fields that are related to different layers of the OSI reference model and that serve a broad range of application protocols for voice and data.

To allow for rapid customization of the protocol set the classifier to be generated should be aware of, we designed a modular and flexible classification language that allows for a high level abstraction, yet, efficient implementation of the packet-classification task. Using this specification language, a protocol is broken down into rules that describe *structure* and *properties* of objects, and rules that describe *processing* of objects. Thus the abstract classification machine uses an unbounded object pool and classification rules are attached to that pool. They identify objects, trigger actions, and finally deposit resulting objects in that pool again. Thus a rule maps n input objects to m output objects, where an object may be a protocol data unit or some internal object, e. g., a route-table entry, interface-counter or whatsoever.

It is the duty of the optimizer to synthesize efficient code according to the specification given. Therefore, the generator applies *static* optimizations to generate code for attribute extraction and identification. It applies *feedback driven optimization* to derive information on instance numbers and conditions that are evaluated in a certain setting for some object type. The result of the optimization phase is a partitioning of the virtual object pool into multiple distributed object pools. These object pools are assigned customized data structures that efficiently support search and update operations on them. Moreover, the optimizer attaches rule engines to pools, determines access paths, and finally maps the rule engines to BSP processes. Thus the optimizer refines the implementation of a BSP framework that coordinates the entire NPU, cf. Section 2.

Currently, the classification-generator has specifications for a couple of low level protocols like Ethernet, PPP, and IPv4, as well as specifications for more sophisticated protocols like TCP, UDP, IPSec, and SSL. The most complex specifications completed so far describe important aspects in terms of behavior of a switch and a router. The generator has successfully translated these specifications into ANSI-C code. Linked with a library that provides implementations for checksum computations, encryption (3DES, AES, SHA-1, MD5), route lookup and memory management, the generated code has been shown to correctly switch and route packet traces that have been collected with standard packet sniffers.

3.2. Retargable Compiler

In our scenario the compiler is primarily used to evaluate proposed variations of the base architecture. We want to make cycle-accurate performance predictions based on realistic application software, well before corresponding hardware designs are available. Hence, the compiler must be quick to adapt to the envisioned changes: addition of specialized super-instructions (cf. Section 3.3.1) and changes in instruction timing. Towards this goal, central parts of the compiler backend are described by high-level specifications. From these specifications the code selection and scheduling phases are automatically generated, as well as several machine-specific support modules of the compiler.

3.2.1 Code Selection Phase

The generated code selection phase converts an intermediate language tree, which represents one statement of the original ANSI-C program, and rewrites it into a code sequence suitable for the target processor. Our code selection is based on a variant of BURS technology [8] that uses a tree grammar to specify the capabilities of the target processor. As shown in Figure 5, each grammar rule maps a tree

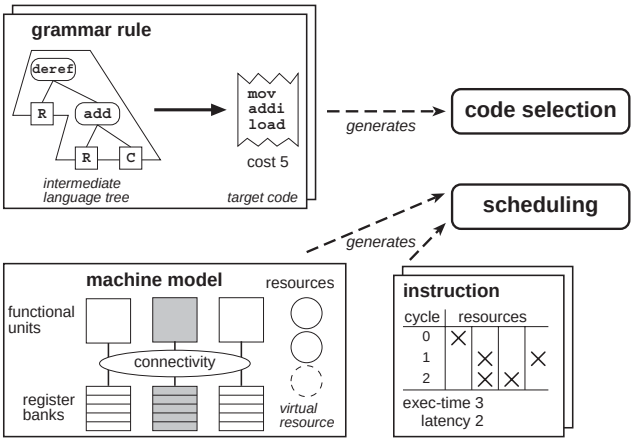

Figure 5. Specification of backend phases

fragment to a semantically equivalent code sequence and is annotated with costs, e. g., execution time of its target code. The non-terminals of the tree grammar determine, how tree fragments can be combined. They express, how information is passed between sequences of target code. Typically, each non-terminal of the grammar corresponds to an addressing mode of the target processor. At compile time each intermediate language tree is covered with tree fragments so that the sum of the associated costs over the whole tree is minimal.

Such an expressive and powerful approach to code selection offers several benefits in our scenario. Tree grammars with associated rule costs favor incremental development of the code selection phase. Additional super-instructions can be easily integrated as tree fragments that cover larger sections of an input tree. Generating the code selection phase is very quick (well below 1 second) and enables liberal exploration of different proposed instruction sets. The generation step also performs consistency checks that help to debug the tree grammar: the generator warns if the grammar rules cannot cover some intermediate language trees and discovers superfluous rules that will never be used because of cheaper alternatives.

Although the S-Core is a RISC architecture where code selection is supposedly straightforward, the power of cost optimal tree covers can still be used to advantage. Our code selection incorporates constant folding of address computations. This includes implicit computations like address offsets relative to a function's stack frame.

While the S-Core supports only very few addressing modes in load/store operations, our tree grammar for the S-Core features 39 different non-terminals, i. e., conceptual addressing modes. Many pseudo addressing modes stem from machine instructions that accept different ranges or classes of integer values as immediate operands. These give rise to a variety of specialized code sequences for some basic arithmetic operations.

3.2.2 Scheduling Phase

The compiler performs automatic fine-grained parallelization to exploit a cluster of four coupled S-Core processors. Previous research has shown parallelizing compilers to be very effective for a small number of processors [7]. Coarse-grained parallelism across different clusters of S-Cores is expressed explicitly by means of BSP (cf. Section 2.2).

Automatic parallelization in our compiler is built around a library of hand-written scheduling algorithms. An abstract model of the target machine adapts the algorithms to the processor in question. The library includes several algorithms based on list-scheduling and software-pipelining techniques. Especially the latter kind of techniques is well-suited for inner loops of processing intensive applications, e. g., data encryption and compression [16].

The machine model provides a high-level view of the target processor, reduced to just those aspects that are relevant for scheduling decisions to be made at compile time. The model describes the available functional units and register banks of the processor, as well as their capabilities and their connectivity (cf. Figure 5). For each machine instruction the model specifies execution time, latency, and resource requirements in each clock cycle. The underlying set of processor resources abstracts far from the actual hardware design. Hardware resources that contribute no discriminating scheduling constraints can be safely omitted. Each complex interaction between multiple interrelated hardware resources is condensed to a single *virtual resource* that captures just the resulting behavior on the instruction-level [17].

All these major building blocks of the machine model map directly to language constructs in UPSLA*. This is our specification language for core parts of the compiler backend, as well as the associated simulator (cf. Section 3.3). UPSLA is a declarative, object-oriented language that allows one to factor out common items of information to achieve concise, maintainable specifications.

In comparison to an actual hardware design, UPSLA expresses a grossly simplified model of a processor. This allows new super-instructions (cf. Section 3.3.1) to be incorporated easily for evaluation. Also, initial exploration is unhindered by concerns about the complexity of a working hardware implementation.

Our scheduling phase treats the cluster of four synchronously clocked S-Core PEs similar to a VLIW processor with four general-purpose functional units [17, 16]. Two additional pseudo units of the assumed VLIW processor capture behavior that affects more than one physical S-Core engine: a branch unit and a synchronization unit. The branch unit manages flow of control for all four PEs in concert. Similarly, explicit barrier synchronization of multiple PEs is conducted by the synchronization pseudo unit. When the compiler backend ultimately emits a set of four distinct programs for the four PEs, instructions of the pseudo units are integrated into all four code streams. However, the concept of centralized pseudo units guarantees structurally correct code after scheduling, without any artificial scheduling constraints. Furthermore, the pseudo units simplify code transformations during peephole optimization.

Although all processors of a cluster receive synchronous clock, barrier synchronizations (cf. Figure 9) are needed to lockstep processors after instructions with unpredictable timing. For example, latency of memory accesses depends on cache state and multiplication has data dependent timing on the S-Core. Cheap synchronization among processors inside a cluster has been one our first and most fundamental extension to the base S-Core architecture.

3.3. Cycle Accurate Simulator

Architectural designs, modifications thereof and draft decisions affecting compiler construction must be evaluated fast, simply, and repeatedly. Our approach is to generate an appropriate compiler and corresponding simulator for a specific target architecture. Then typical applications and suitable benchmarks are compiled and evaluated on that architecture. Therefore, the simulator picks up runtime information such as elapsed clock cycles, utilization of individual functional units, and frequency of instructions or instruction pairs, just to mention a few categories.

Central components of the simulator are generated from the same specification of the target architecture that is used for compiler generation. Since the simulator processes multiple memory images, one for each PU of an S-Core cluster, the simulator-generator needs information about the binary representation of individual instructions and their operational semantics. Given this input, it constructs a decoder that is tailored to the particular instruction set.

Our generator produces a balanced decision tree to allow for an efficient decoding of instructions. It also checks the specification for ambiguities and conflicts of intructions. Operational semantics of the instructions are expressed in terms of ANSI-C fragments, which the generator integrates into a general hand-written interpreter framework.

The simulator for a cluster of n S-Cores can be integrated as a monolithic but parametric component into a SystemC simulation of the entire NPU. It supplies sufficiently precise information with respect to clock cycles in order to achieve a cycle accurate yet efficient simulation of the entire NPU.

To allow for fast and simple evaluation of the S-Core architecture in question, we have implemented the visualization tool *JScore* shown in Figure 6. It takes data collected by the simulator and visualizes important characteristics of the

*Unified Processor Specification LAnguage

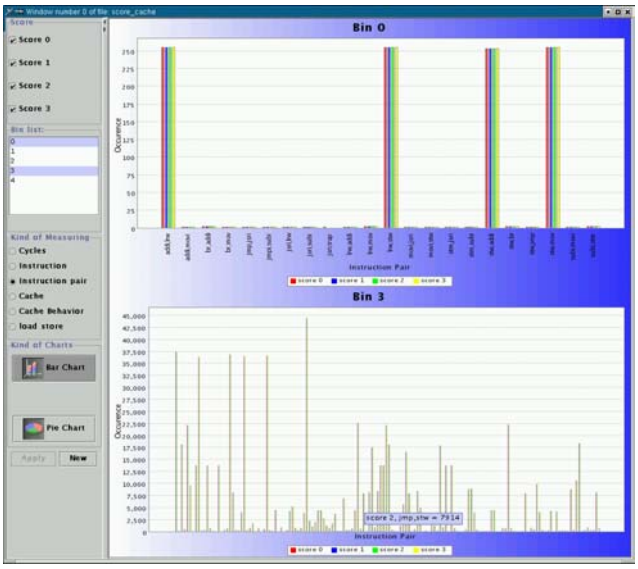

Figure 6. Performance visualization - JScore

executed program, e. g., elapsed clock cycles, the frequency of simple instructions and instruction pairs, load/store bandwidth utilization, cache behavior and the like. To detect hotspots, the tool allows to visualize several interesting regions of code separately. Thus the designer can dissect the impact of arbitrary code fragments on the overall performance.

3.3.1 Evaluation of super-instructions

The data recorded by the simulator allows to tell apart different architectural variants by means of performance characteristics. The values measured guide further refinements of the architecture and/or the compiler. With respect to compiler construction, it is of special interest to adapt design parameters that relate to the instruction set and common instruction pairs found in the compiled code. Instruction pairs do not simply consist of adjacent instructions, but expose direct data-flow dependence, i. e., they exchange data using registers. This information is also collected within the simulator and can be visualized by *JScore*. The instruction pairs emphasized by the visualizer are typical candidates for super-instructions to be implemented in hardware for the sake of speed-up. During our initial evaluations it only took us a couple of minutes to introduce new instruction-pairs using the design cycle sketched above, and to generate a compiler and corresponding simulator that made use of these new instructions. Applying the particular modifications shown in Figure 7, we achieved a speed-up of 1.4 for a simple checksum function.

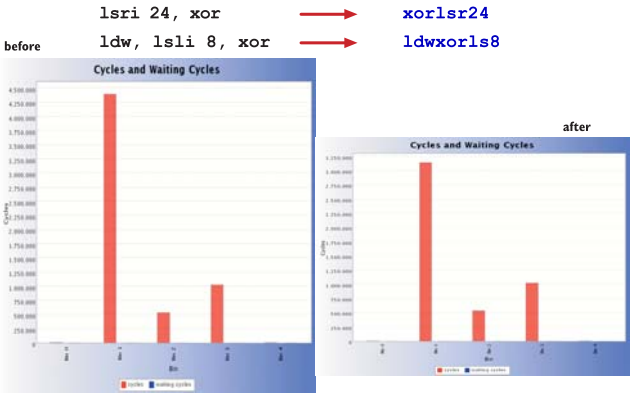

Figure 7. Impact of super-instructions

4. System architecture design

In order to handle the upcoming data traffic of the future, special high-integrated circuits with an optimized system architecture are needed for the network nodes. Furthermore, with decreasing feature sizes resulting in continuously growing SoC designs, our architecture has to scale in an appropriate way (cf. Section 2.1). This means that an architecture consisting of 32 processing engines, which is going to be realized in the first tapeout, should be easily expandable to an architecture comprising hundreds of processing engines while leaving the programming model unaffected (cf. Section 2.2).

4.1. Basic blocks for the SoC design

As mentioned in Section 1.1 our architecture is structured into three domains: SoC level, Cluster level, and PE level, with each domain possessing special core components. From a bottom-up view of the system, the first building blocks are the processing engines used in every cluster.

4.1.1 S-Core as a core component of the PE level

For the use in System-on-Chip (SoC) designs, we have developed the S-Core, a RISC processor core that is binary-compatible with Motorola's M-Core M200 architecture [11, 12]. The processor has been developed as a soft core using the hardware description language VHDL. This gives us the opportunity to reuse the core for different target technologies. The S-Core is a 32-bit RISC two-address machine with a straightforward load/store architecture. It has two banks of sixteen 32-bit registers, which can be alternatively used in user mode. Each instruction has a fixed length of 16 bits. This results in a high code density and therefore reduces memory demands for the application code. Beyond that, each instruction, except for the load and store instructions, works only on the registers. The execution of

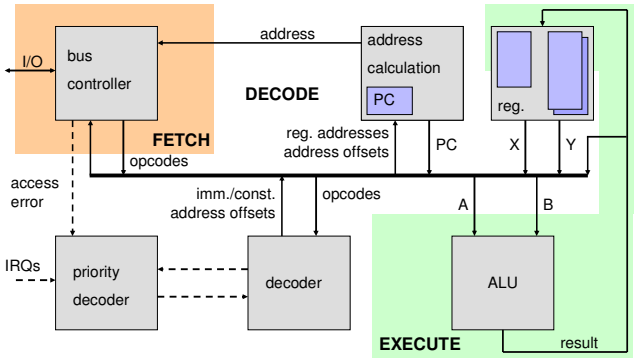

Figure 8. Architecture of the S-Core

most instructions takes one clock cycle. The S-Core features a short three-stage pipeline and an addressing interface that supports byte granular access. Therefore, it is ideally suited for the implementation of network protocols including medium access and routing or packet classification. Furthermore, the architecture can easily be expanded by adding application-specific instructions or coprocessors to the core (cf. Section 4.2). Our implementation of the CPU is resource-efficient and delivers reasonable performance for embedded systems. For an overview of the implemented architecture, cf. Figure 8. In the 0.13 µm 1.2 V Infineon standard cell process the S-Core needs less than 0.25 mm^2, and clock frequencies of more than 200 MHz can be achieved in this technology.

4.1.2 Switch Boxes – core components at the cluster level

The main components of the cluster level are switch boxes (cf. Figure 1)[5], which connect the different clusters of the SoC. Currently, a cluster consists of 4 S-Cores, which are connected to the switch box via a high-performance on-chip bus. The switch boxes provide the necessary interconnections for the on-chip network and resemble traditional switches in local area networks. We use a packet switching model instead of a circuit switching model [6]. This enables shared use of connections inside the network and simplifies the programming model, which can rely on the message passing paradigm. Therefore, well-known techniques from the world of parallel computers can be applied to massively parallel SoCs. The switch box itself consists of the on-chip network interfaces, local packet buffers, and local memory that is dynamically assigned by a local control unit that has to manage the data traffic of the cluster as well.

The interconnection between switch boxes can form arbitrary topologies, like meshes, tori, or butterfly networks. For the first implementation a mesh has been chosen (cf. Section 2.1) due to efficient hardware integration.

The parallel structure of the switch box architecture offers a high potential for fault tolerance. In case of a failure of one or more components other parts of the SoC can take over the respective operations. This can cause a strong increase of production yield. Furthermore, the topology is easy to implement on SoC architectures and provides a high scalability to make use of the increasing transistor count enabled by the enormous progress in ULSI technology.

Another building block at the cluster level is the on-chip bus. We are evaluating two versions of the bus structure. The first variant is a wishbone bus realization [15] with an arbiter using a round robin scheme. The second variant is a more complex realization of an ARM AMBA Multi-Layer-AHB-Interconnect-Matrix [1, 2] which is superior concerning performance aspects but requires more area. A detailed system analysis will determine the most efficient system configuration (cf. Section 4.3). To reduce the cost of memory accesses a multiprocessor cache is connected to each processor. A Harvard cache architecture is used, which allows prefetching and has special features needed for internal data exchange of the cluster.

4.1.3 On-chip network interconnections at the SoC level

The on-chip network forms the communication backbone of our architecture. Currently, we are evaluating synchronous and asynchronous communication structures that support dynamic bandwidth aggregation. Depending on the priority of data transmission and the power-saving mode, e. g., in case of a mobile product, more or less transmission lines can be combined to achieve the required bandwidth. The communication infrastructure is transparent to the programmer, all necessary decisions concerning the datatransmission via the on-chip network are handled by the switch boxes.

4.2. System enhancements

In this section we present system enhancements at the PE level and at the cluster level. Extensions on the SoC level will be discussed in the progress of the project. Instruction set extensions can be classified as enhancements on the PE level whereas application-specific hardware accelerators are located at the cluster level.

4.2.1 Instruction set extensions

The Motorola M-Core architecture reserves 11% of its opcode space for the implementation of new instructions such as super-instructions (cf. Section 3.3.1) or control operations like a synchronization barrier. The *barrier* operation (cf. Figure 9) is a special instruction which is inserted by the parallelizing compiler. With this synchronization method – which ideally takes only one clock cycle – it is possible to synchronize the involved PEs. This barrier is implemented

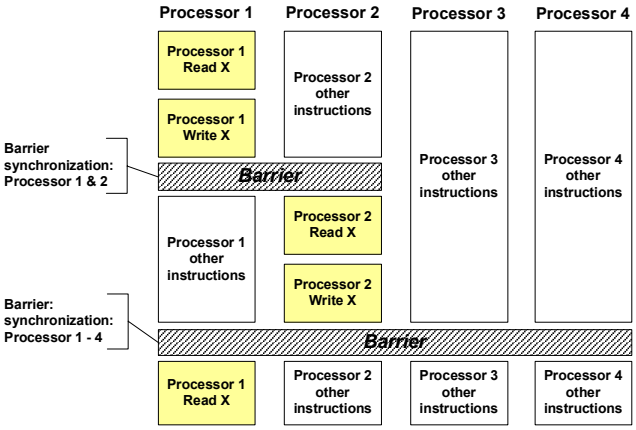

Figure 9. Cluster synchronization method

for each combination of synchronizations whether just two or even all of the CPUs need to synchronize. This simple and fast mechanism assures data consistency and minimum idle time of the CPUs.

Another potential for system enhancements is the implementation of super-instructions (cf. Section 3.3.1). By inserting these – mostly application-dependent – opcodes into the instruction decoder of the S-Core, a remarkable performance increase can be reached.

4.2.2 Embedded applications and application specific hardware accelerators

The efficient design of synthesizable processing engines for embedded real-time applications such as networking requires to be aware of resource requirements like execution time, energy and utilization of area and memory. The simplest processing engine is a system consisting of a general-purpose processor as well as memory and I/O units to interact with the environment. However, the energy consumption and execution time of such a system does usually not satisfy the design constraints for a single-purpose embedded application. By adding hardware accelerators or using an application-specific instruction processor (ASIP), the flexibility can be balanced off with the performance of the system.

In [9] we have presented a method to characterize and optimize embedded applications consisting of software and hardware parts. Our characterization environment delivers information about stack usage, memory accesses, execution time, and energy consumption for arbitrary code fragments of an embedded application. Our environment supports the addition of hardware units (coprocessors) to the processor-based engine to enhance the performance of the application. We use our performance monitoring toolchain *PERFMON* to determine characteristic parameters of our hardware and software macros to establish a corresponding SystemC library. This library facilitates a faster simulation of the hardware macros and an accurate characterization of energy consumption for both hardware and software blocks. Main targets of our analysis are typical functionalities needed by network processors like error detection, compression, and encryption algorithms.

4.3. System Simulation / System Emulation

The first tapeout of our architecture will consist of 32 S-Core processing engines and eight switch boxes.

Each PE is connected via a multiprocessor cache to the on-chip bus of a switch box (cf. Figure 1). To test the software in an early design phase, and to optimize different parameters of the architecture accompanied with a much faster simulation speed, a simulation model of the architecture will be developed in SystemC. A cycle accurate simulation model of the S-Core is generated in conjunction with the compiler backend (cf. Section 3.3). In this typical top down design flow, global system parameters are specified on a high-level of abstraction. Coming closer to the hardware implementation, these parameters are iteratively refined. For the SystemC simulation, we use the Synopsys design environment CoCentric System Studio.

For the verification of the basic hardware blocks on the Register Transfer Level (RTL) we use the Cadence NC-Sim VHDL simulation environment. Unfortunately, simulation at this detailed level is very slow. However, since the environment is synthesizable, it can be mapped to a hardware emulation system. We have developed the rapid-prototyping system *RAPTOR2000* (cf. Figure 10), which integrates all the important components to realize circuit and system designs with a complexity of up to 100 million transistors by means of FPGA-based application-specific modules [10, 14]. The RAPTOR2000 system consists of a PCI card to which daughterboards can be attached. These modules carry, for example FPGAs, which can communicate with the host system. By using this rapid prototyping system, detailed hardware analyses can be done in less time. The design can be embedded in a system environment, e. g., attached to an Ethernet, Fast Ethernet, or even Gigabit Ethernet module and can be tested in the whole system environment. When using our RAPTOR2000 rapid prototyping system, we achieve a speed-up of about 10,000 compared to the VHDL simulation on a 2.5 GHz Pentium 4.

5. Conclusion

Meanwhile, the GigaNetIC project described in parts within this paper has grown to maturity such that the entire design cycle can be completed successfully. A design cycle covers the highest level of abstraction expressed in BSP, comprises generators and compilers for commonly

Figure 10. RAPTOR2000 Rapid Prototyping System

used network protocols as well as ANSI-C, and extends down to the lowest-level of hardware abstractions making use of IP cores and custom hardware.

Therefore, we are not only able to judge the performance of the proposed NPU for a certain set of protocols, but are also able to evaluate variations of that NPU with respect to every level of abstraction within minutes. Future work continues on all these levels and heads towards refinements of the BSP algorithms, extensions of the set of protocol specifications to special purpose applications, and improvements of the hardware synthesis and performance estimation toolchain. We expect to arrive at an integrated development environment that allows to customize important characteristics of the NPU in question, like supported networking protocols, resource demand and the like, yet allows for high efficiency in terms of manufacturing and deployment.

A couple of research projects that aim at the rapid development of NPUs have been described in the literature recently. Some of these projects also follow the approach to exploit coarse-grain and fine-grain parallelism [13]. Since the GigaNetIC project is not yet in a state to propose final architectures for specific networking domains, we do not compare particular results. Instead, we would like to mention that in terms of scope the GigaNetIC project exhibits notable similarity to the Mescal Project [3], such that it will be interesting to compare particular results with respect to the programming model, compilation techniques, and hardware optimization soon.

Acknowledgments

The project outlined in this report was funded by the Federal Ministry of Education and Research (Bundesministerium für Bildung und Forschung), registered there under 01M3062A. The authors of this publication are fully responsible for its contents. This work was supported in part by Infineon Technologies AG, especially the department CPR ST, Prof. Ramacher.

References

[1] ARM Ltd. *AMBA Specification (Rev. 2.0)*, 1999.
[2] ARM Ltd. *Multi-layer AHB*, 2001.
[3] D. I. August, K. Keutzer, S. Malik, and A. R. Newton. A disciplined approach to the development of platform architectures. In *Proc. of the 10th Workshop on Synthesis and System Integration of Mixed Technologies (SASIMI)*, 2001.
[4] O. Bonorden, B. H. H. Juurlink, I. von Otte, and I. Rieping. The Paderborn University BSP (PUB) library. *Parallel Computing*, 29(2):187–207, Feb. 2003.
[5] A. Brinkmann, J.-C. Niemann, I. Hehemann, D. Langen, M. Porrmann, and U. Rückert. On-chip interconnects for next generation system-on-chips. In *Proc. of the 15th Annual IEEE International ASIC/SOC Conference*, pages 211–215, Rochester, N.Y., USA, Sept. 2002.
[6] W. Dally and B. Towles. Route packets, not wires: On-chip interconnection networks. In *Proc. of DAC 2001*, pages 684–689, 2001.
[7] D. Fischer, J. Teich, M. Thies, and R. Weper. Efficient architecture/compiler co-exploration for ASIPs. In *ACM SIG Proc. of CASES 2002*, pages 27–34, Grenoble, France 2002, Oct. 2002.
[8] C. W. Fraser, D. R. Hanson, and T. A. Proebsting. Engineering a simple, efficient code-generator generator. *ACM Letters on Programming Languages and Systems*, 1(3):213–226, Sept. 1992.
[9] M. Grünewald, J.-C. Niemann, and U. Rückert. A performance evaluation method for optimizing embedded applications. In *Proc. of the 3rd IEEE International Workshop on System-On-Chip for Real-Time Applications*, Calgary, Alberta, Canada, 30 June - 2 July 2003, to appear.
[10] H. Kalte, M. Porrmann, and U. Rückert. Rapid Prototyping System für dynamisch rekonfigurierbare Hardwarestrukturen. In *AES2000*, pages 150–157, Germany, 2000.
[11] Motorola. *M-Core Reference Manual*, 1998.
[12] Motorola. *MMC2001 Reference Manual*, 1998.
[13] K. K. Niraj Shah. Network processors: Origin of species. In *Proc. of ISCIS XVII, The 17th International Symposium on Computer and Information Sciences*, October 2002.
[14] M. Porrmann, H. Kalte, U. Witkowski, J.-C. Niemann, and U. Rückert. A dynamically reconfigurable hardware accelerator for self-organizing feature maps. In *Proc. of SCI 2001*, volume 3, pages 242–247, Orlando, Florida, USA, 2001.
[15] Silicore Corporation. *WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores Revision: B.2*, 2001.
[16] A. Slowik, G. Piepenbrock, and P. Pfahler. Compiling nested loops for limited connectivity VLIWs. In P. A. Fritzson, editor, *Proc. International Workshop on Compiler Construction CC'94*, number 786 in Lecture Notes in Computer Science. Springer Verlag, Apr. 1994.
[17] E. Stümpel, M. Thies, and U. Kastens. VLIW compilation techniques for superscalar architectures. In K. Koskimies, editor, *Proc. 7th International Conference on Compiler Construction CC'98*, number 1383 in Lecture Notes in Computer Science. Springer Verlag, Mar. 1998.
[18] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, Aug. 1990.