Application Specific Compiler/Architecture Codesign: A Case Study

Oliver Wahlen, Tilman Glökler, Achim Nohl, Andreas Hoffmann,
Rainer Leupers, and Heinrich Meyr

Integrated Signal Processing Systems
Aachen University of Technology
Aachen, Germany
lisa@iss.rwth-aachen.de

ABSTRACT

This paper proposes an architecture exploration methodology for application specific instruction set processors (ASIPs) including a C compiler and a VHDL model in the exploration loop. For a given application the target architecture is an instance of the scalable ALICE VLIW architecture which will be presented in this paper. In a case study it will be explained how the LISA processor design platform in conjunction with the CoSy compiler environment significantly reduces the time for exploration cycles. Using a typical telecommunications application, the quality of the resulting architecture and its performance are compared to the ICORE2 processor - a manually designed ASIP for efficient processing of computation intensive kernels.

Categories and Subject Descriptors

C.1.1 [**Processor Architectures**]: Single Data Stream Architectures—*VLIW architectures*; C.3 [**Computer Systems Organization**]: Special-purpose and Application-based Systems—*Signal processing Systems*; D.3.4 [**Programming Languages**]: Processors—*Code Generation*

General Terms

Design, Experimentation

Keywords

ASIP, architecture exploration, retargetable compiler

1. INTRODUCTION

In the implementation phase of an embedded system, the developer is confronted with decisions in a design space of at least five major axes: performance, power consumption, flexibility, design time, and silicon area. One of the first questions which is usually answered by the designer's experience is the target architecture class or combination of architectures. ASICs, FPGAs, application specific instruction set processors (ASIPs), DSPs, μCs or general purpose processors (GPPs) form the bounds in the design space mentioned above.
In high volume application domains where a certain degree of flexibility is indispensable and where power and performance matter — which is the case for many telecommunication applications — ASIPs are an important compromise between hardwired ASICs/FPGAs and power hungry and slower architectures of GPP vendors.

To find the optimal design space parameter tradeoff for a given application domain, the evaluation of several architectural alternatives is essential. Unfortunately, the development and verification effort of ASIP based designs significantly increases the time required for this so called architecture exploration. Within a given development time this can easily lead to suboptimal and less competitive solutions or even complete project failures. The limited reusability of the ASIP's HDL specification and its corresponding software development and verification tools like code generation utilities and simulator additionally increase the development costs. Furthermore, shorter and shorter product cycles put very high requirements on the design methodology and the design tool support.

In this paper we illustrate a typical compiler/architecture codesign methodology and evaluate its efficiency in the form of a case study: For a given mobile telecommunication application we designed an ASIP with the corresponding code generation-, profiling, and debugging software. To achieve meaningful results all components of the system were generated by sophisticated design tools consisting of the CoSy®[1] [5] compiler development system, the LISA [3] processor design platform, and Synopsys' Design Compiler [24]. We analyzed the complete design flow from the application specified as a C program down to the hardware/software implementation in the form of a netlist and machine code: On the one hand the quality of the

LCTES'02–SCOPES'02, June 19-21, 2002, Berlin, Germany.

[1]CoSy is an international trademark of ACE Associated Computer Experts bv.

generated compiler was evaluated by comparing a modified version of the CoSy compiler to GCC [21]. On the other hand the hardware efficiency was analyzed by comparing our design results to an ASIP called ICORE2 which was designed exclusively for the telecommunication kernel with an assembly programmer's model in mind.

Our ASIP development is based on a scalable processor architecture called ALICE. ALICE was designed to be easily targetable by a C compiler. Nevertheless it should be extensible and flexible enough to meet the performance requirements of given applications with the smallest possible architectural overhead.
The key element of our case study is the C compiler in the exploration loop: We demonstrate that the CoSy compiler can easily be retargeted to take advantage of new architectural features and that it significantly speeds up the software creation and verification. In conjunction with the profiling capabilities of the LISA platform these environments significantly reduce the time for the exploration loop depicted in figure 1.

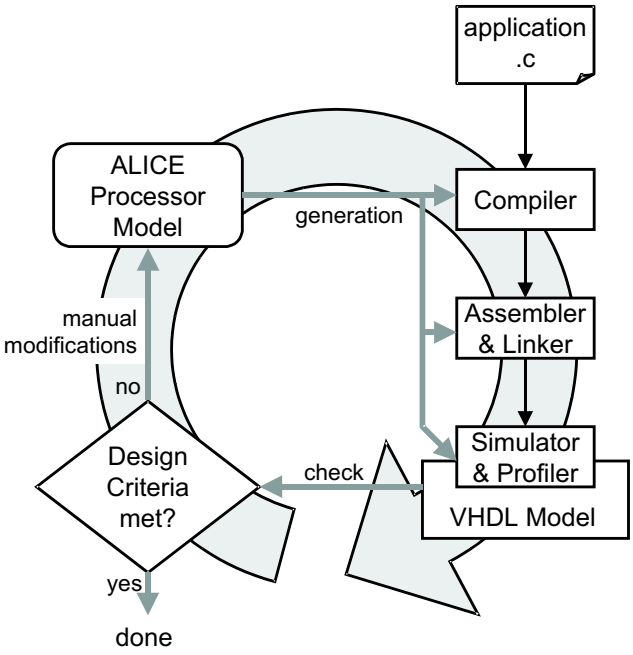

Figure 1: ALICE architecture exploration loop

An overview of related work is given in section 2. The LISA and CoSy environments are explained in sections 3 and 4, respectively. Section 5 introduces the ALICE processor template and points out the reconfigurable parameters of ALICE. The telecommunication application used for our case study and the architecture exploration methodology are illustrated in section 6. The results of the case study are presented in section 7 followed by conclusions in section 8.

2. RELATED WORK

All ASIP design environments that comprise an integrated development of both software tools and architectural specification can be classified into two categories: On the one hand environments are bound to a single processor template whose tools and architecture can be modified to a certain degree. On the other hand there are approaches that permit a free specification of the processor at the expense of restrictions on the quality and/or availability of the tools.
Examples for processor template based approaches are Xtensa and Jazz which are commercially available from Tensilica and Improv, respectively. **Xtensa** [27] is a scalable RISC processor core. Configuration options include the width of the register set, memory, caches etc. New instructions and functional units (FUs) can be added using the Tensilica Instruction Language (TIE). All software tools including (GNU based) C-compiler, assembler, linker, simulator, debugger, and a synthesizable hardware model (HDL-code) can be generated.
Improv's **Jazz** [13] Processor is a VLIW processor and part of the so called *Programmable System Architecture* (PSA) which permits the modeling and simulation of a system consisting of multiple processors, memories, and peripherals. The data width of the Jazz processor, the number of registers, the depth of the hardware task queue are configurable. It is also possible to add FUs (e.g. ALUs, MACs, Shifters) and to define custom functionality (in Verilog). Code generation and simulation tools can be automatically generated.
Both approaches have limitations when implementing hardware concepts that are not provided by the environment or when modifications to the provided software are necessary.
A hardware/compiler co-development methodology for the **Vector IRAM media processor** is described in [9]. By utilizing the Cray compiler environment for supercomputers (PDGCS) this scalable vector processor targets the domain of multimedia applications, too. In contrast to our approach this proposal does not focus on the instruction level parallelism exploitable by a VLIW architecture but on data parallelism that must be contained in the target application for the vector processor to be effectively utilized.
PICO [22] (Program In Chip Out) and **Trimaran** [29] are both part of the Compiler and Architecture Research Program (CAR) of HP Labs. PICO is an environment that automatically designs parallel computing systems for applications written in C. VHDL-descriptions for non-programmable processors (systolic-array processors) as well as custom EPIC or VLIW processors can be generated. Trimaran uses the *MDes* processor description language to retarget compiler, assembler and cycle-level simulator to a range of VLIW architectures called *HPL-PD*. Since the PICO environment utilizes Trimaran's Elcor compiler backend its architectural scope for programmable architectures is limited to HPL-PD.
PEAS [18] is an ongoing hardware/software codesign project at Osaka University. A GUI based architecture description drives generators for HDL code, compiler, assembler, linker, and simulator. There is support for several architecture types (e.g. VLIW) and a library of configurable resources. Instruction set and micro-operations are separately described. Unfortunately, no detailed results about the target architecture range and code quality have been published yet.
BUILDABONG [14] from the University of Paderborn is an architecture exploration framework aiming at ASIP optimization by architecture/compiler codesign. The target processor model is an Abstract State Machine (ASM) which is derived from the XASM description language or from a

schematic entry tool. Besides the RTL model a simulator and a compiler can be generated. However there is few information available concerning the simulation speed, the architectural scope, and the code quality.

Approaches that permit a free definition of the processor in the form of a dedicated language can further be divided into environments that focus on the instruction set or on the processor's architectural structure. Typical instruction set (IS) based languages are nML and ISDL. The **nML** [1] language describes the processor's instruction set as an attributed grammar with extensions reflecting the set of legal instructions. The Chess/Checkers tool-suite [26] which is commercially available from Target Compiler Technologies is based on nML with extensions for describing architectural aspects like pipelining. The architectural scope is limited to DSPs and ASIPs. Compiler, instruction set simulator and HDL generation are provided.

Using the **ISDL** [11] environment compiler, assembler, and simulator can be generated from an ISDL instruction set description. The architectural focus are "orthogonal" VLIW architectures: since constraints on the coding of instructions have to be explicitly specified it is very hard to describe architectures that have complex decoders.

An architecture description language that describes the target machine by a hierarchical RTL netlist is **MIMOLA** [20]. There are two MIMOLA based compilers: The MSSQ compiler is very flexible but the quality of the generated code is sometimes insufficient. The RECORD [16] compiler produces better code but it is restricted to DSPs.

The **UPFAST** [23] system uses a microarchitecture description written in the Architecture Description Language (ADL). A cycle level simulator, an assembler, and a disassembler can be generated automatically. The generated simulator is less than two times slower than a handwritten simulator.

FlexWare2 [19] from STMicroelectronics provides support for generating compiler, assembler, linker, simulator, and profiler using the Instruction Description Language (IDL) in a database oriented approach. The compiler is based on the CoSy [5] compiler development system described in section 4. Its code quality is comparable to hand written assembly although the verification effort for the different code generation tools is quite high due to separate descriptions. The system is intended for in-house use only and is not publicly available.

An approach for addressing both the processor's behavior and its structure is also introduced by the **EXPRESSION** [2] environment from UC Irvine. The behavior is described in the form of operations that are contained in the slots of ILP instructions and that are mapped to generic compiler operations. Besides a specification of the architectural components the structural description characterizes the pipeline, the data transfer paths and the memory subsystem. Using this information a cycle-accurate structural simulator and an optimizing ILP compiler can be generated. It is currently not possible to generate HDL-models. Architectures described so far include TI C6x and Motorola 56k DSPs. However experimental results on code quality for these targets have not yet been published.

Instead of generating a processor HDL-model and corresponding code generation tools from a processor description or a processor template it is also possible to generate an ASIC hardware model directly from the system specification. Adelante's **AR|T designer** [7] (C based) and Synopsys' **Behavioral Compiler** [24] (behavioral VHDL based) are examples for this approach. Disadvantages are the reduced flexibility of the hardware and a limited hardware efficiency.

The methodology we present in this paper combines the advantages of the processor template based environments and the free processor specifications: Our architecture exploration phase is based on a scalable ASIP. Like in other processor template based approaches this restriction of the design space permits a highly optimizing C compiler and fast analysis of hardware performance and costs. Thus design space iterations are quick and produce precise results. If the exploration shows that the architecture range of the processor template is too restrictive for an implementation most template based environments do not provide further solutions. In our approach the designer can take the LISA and CoSy models as a starting point for further separate optimizations: On its own, LISA is capable of describing a much wider range of processors and CoSy provides an extensible library of engines that permit the targeting of many architectural features.

3. LISA PROCESSOR DESIGN PLATFORM

The LISA processor design platform LPDP [3] is an environment that provides a consistent design flow for system level-, processor architecture-, and software design. A commercial version is available from LISATek Inc. [17]. The key component of the environment is the *Language for Instruction Set Architectures (LISA)* that describes the behavior, the structure, and the I/O interfaces of a processor architecture. The environment has been used to describe a wide variety of architectures including ARM7, C62x, C54x, MIPS32 4K, and to develop ASIPs like ICORE2 (see section 7.2).

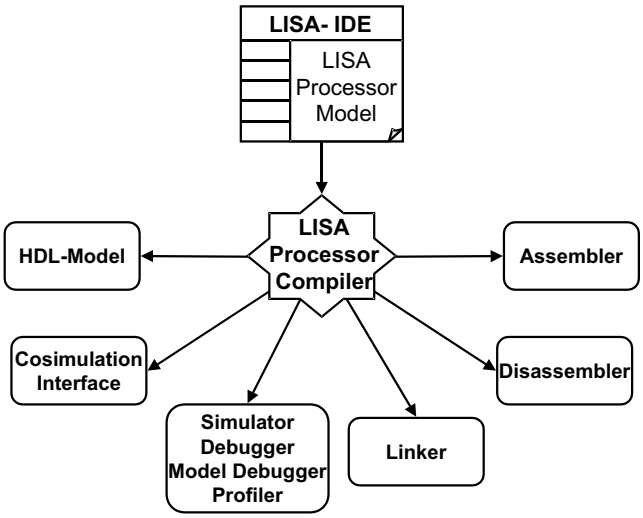

Figure 2: The LISA Processor Design Platform

The LPDP provides an Integrated Design Environment (IDE) to support the manual creation and configuration of the LISA model. From the IDE the so called *LISA processor compiler* is invoked. It parses the description and generates

the tools and models necessary for software design and architecture implementation:

- assembler
- disassembler
- linker
- simulator GUI including model debugger and profiler
- simulator core with API for HW/SW cosimulation
- HDL model of the processor's control path

To provide a seamless integration and verification of the LISA processor model into a system context the simulator comprises interfaces permitting cosimulation of the LISA model within a system simulation environment. In [4] the integration of several LISA models into the SystemC [28] environment is described. SystemC was used to model the processor's interconnection, external peripherals, memories, and buses on a cycle-accurate level.
The key functionality of the LISA processor design platform is its support for *architecture exploration*: In the phase of tailoring an architecture to an application domain LISA permits a graceful degradation of the model's abstraction level by supporting *instruction set models* and *cycle based models*. Beside the instruction's functionality the latter involves modeling of pipelines (stalls and flushes), registers, and latencies. Resources can also be modeled on several levels of abstraction. For example memories can be defined as a C type array on the simulation host or they can be modeled as the HDL model of a complex cache hierarchy that is interfaced over an address bus. The consequences of design decisions can seamlessly be monitored in the exploration process by utilizing the profiling capabilities of the LISA simulator. If profiling is enabled the simulator automatically observes the cumulative execution of instructions and LISA operations. It detects and counts loops, it sums up read and write accesses on registers and it collects pipeline execution statistics like the number of stalls and flushes.

4. COSY COMPILER ENVIRONMENT

Using the CoSy [5] Compiler Environment a designer can retarget HLL-compilers for a broad range of architectures in a flexible and modular manner. Programming languages supported by CoSy are C, C++, an extension to ANSI-C called DSP-C that introduces fixed point data types and arithmetic, Java, Fortran 95, and HPF. Beside the sources for standard libraries a regression test suite called SuperTest is included in the environment. CoSy provides a rich set of optimization and restructuring engines that include typical high level optimizations like copy/constant propagation, code motion, loop unrolling, fusion etc.
Figure 3 illustrates the modular concept of CoSy. The CoSy IR data structures are easily extensible and are generated in conjunction with corresponding interfacing functions from the so called *Structure Description Language (SDL)*. The dynamic calling sequence of the built-in or handwritten CoSy engines is described in the *Engine Description Language (EDL)*. However, for retargeting a compiler the most important component of the CoSy environment is the Backend Generator *BEG*. It takes *Code Generator Description*

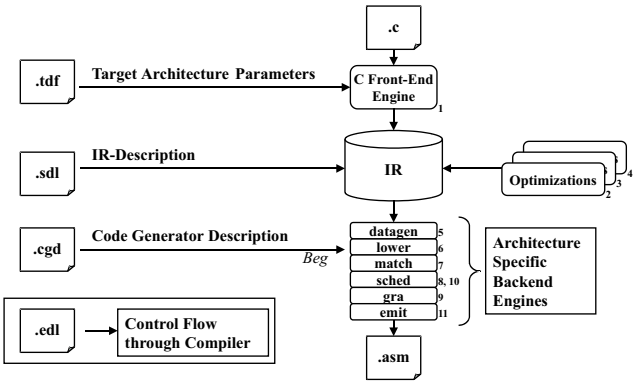

Figure 3: The CoSy Environment

(CGD) files as input and generates most parts of the compiler's backend. Generated components are a tree pattern matching based code selector, a scheduler that can also be used as a code compactor after register allocation, a global register allocator, and a code emitter. CGD permits specifying lowering rules that modify IR patterns before the pattern matcher starts its work. For complex restructuring it is possible to write lowering engines manually. The compiler environment is additionally parameterized by a so called *Target Description File (TDF)* that specifies sizes and alignments of data types.

5. ALICE ARCHITECTURE TEMPLATE

Our methodology for compiler/architecture codesign is based on a scalable architecture called ALICE which is depicted in figure 4.

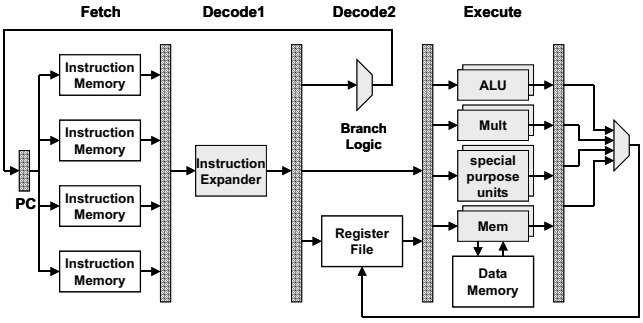

Figure 4: The ALICE Architecture

The main design goal of ALICE was to construct an architecture that could efficiently be targeted by a compiler and that could easily be modified to provide a processor with low overhead and high performance for a specific application.
Today many off-the-shelf DSPs solve the performance/cost/power tradeoff by introducing special purpose functional units (FUs) with dedicated register files, buses and a rich set of addressing modes. For these non-orthogonal architectures the recent trend of moving the entry point of the design flow from assembly to C level causes several problems in C compiler design. Some of the most important are:

- The C code has to be rewritten into a form that the code selector can map code to the special FUs.
- The phases of the compiler backend (code selection, scheduling, register allocation) become dependent on each other (phase coupling problem): decisions in one phase prevent possibilities in other phases.

To overcome these problems we have designed ALICE as a load/store RISC architecture with a singe general purpose register file.
Many high level languages like C make use of stack frames to implement function calls, local variables etc. To provide an efficient access relative to a stack pointer we chose register/offset memory access as the only addressing mode.

Using these design decisions the resulting architecture is very similar to the MIPS32 [10] architecture which ALICE is based on. Obviously, MIPS32 is a general purpose processor and is not suited for signal processing applications. In order to achieve the necessary computational performance ALICE parallelizes the execution of instructions. To reduce energy consumption and die size the decision of which instructions are to be executed in parallel are transfered into the compiler (in contrast to superscalar designs).
The resulting Instruction Level Parallelism (ILP) is depicted in figure 4 in the form of the parallel fetch slots in the first pipeline stage. To eliminate the typical code size problem of pure VLIW architectures that directly fetch the VLIW word from memory we introduced a two stage decoder that in the first phase decompresses a packet of instructions fetched from memory. A detailed explanation of the technique is given in [25]. The basic design properties of ALICE are reflected by its pipeline structure:

fetch: loads the next instruction packet from the memory banks.

decode 1: the instruction packet is expanded to an internal VLIW word.

decode 2: decodes the VLIW word for each unit, reads the register file, and calculates potential branch destinations.

execute: comprises the functional units for arithmetic/logic calculations or read/write memory.

writeback: writes results back into the register file.

The easily scalable design parameters of ALICE include:

- the number of equivalent functional units (FUs) (e.g. number of multipliers)
- the introduction of special purpose FUs (e.g. a FFT butterfly or a CORDIC)
- latency of pipelined/non-pipelined functional units
- number and connectivity of forwarding paths
- number of registers
- number and sharing of register file or memory ports
- word lengths

A detailed explanation how these parameters are utilized for tailoring ALICE to a given application will be given in the following section.

6. ARCHITECTURE EXPLORATION

6.1 The Application

In a case study we tailored the ALICE architecture to a typical mobile telecommunication kernel. The selected algorithm is an eigenvalue decomposition (EVD) of a complex matrix. The EVD is needed by estimation algorithms like the multiple signal classification (MUSIC) algorithm or direction-of-arrival (DoA) [12] algorithms. It is given in the form of a C procedure that calls two additional C functions which implement a COrdinate Rotation DIgital Computer (CORDIC) based calculation of sine, cosine, and arc tangent. All functions have undergone a float to fixed conversion which means that they contain only integer data types with scaling shifts. The CORDIC is a shift-add algorithm consisting of a single loop which makes it very suitable for hardware implementations. Its sources consist of 89 lines of C code. The length of the complete application is 474 lines. The EVD comprises several nested loops. The control flow through the loops depends on the result of the CORDIC calculation. This control flow orientation and the fact that the matrix dimension needs to be configurable make the EVD an algorithm predestinated for a programmable architecture.

6.2 Profiling

The first step of tailoring ALICE is to compile the algorithm with a CoSy based C compiler and to profile it on a highly parallelized version of the architecture.
Within the LISA model of the ALICE architecture the LISA Simulator was configured to obtain extensive execution statistics. As one can see in figure 5 the number of execution cycles is accumulated in virtual registers and the number of activations of all functional units is counted. Additional counters can easily be added to the model. In the disassembly window some of the simulator's profiling capabilities are depicted. Besides analyzing and counting loops they are visualized graphically. The execution of each instruction is counted and set into relation with the total amount of control steps. The graphical representation provides an intuitive way to find the *hot spots* in the assembly sources. The profiling results of the EVD kernel running on an ALICE architecture comprising 4 parallel ALUs, load/store units, and multipliers are depicted in figure 6. Using this configuration of ALICE it takes 107.895 cycles to compute the result of the EVD. The figure illustrates the percentage of cycles the different units were activated.

6.3 Exploring the Number of equivalent Functional Units

As one can see the CoSy Compiler was able to utilize all functional units to a certain degree. The CoSy scheduler description that was required to obtain the results in figure 6 was automatically generated from the corresponding LISA model. This automation significantly reduces the verification effort between the two descriptions. An in depth explanation of this functionality is beyond the scope of this paper, though. The generated compiler does not only exploit parallelism inherently contained in the algorithm but it additionally increases parallel executions by heuristic loop unrolling. The scheduler's allocation strategy of mapping parallel instructions on functional units is to start with low unit indices and then rise to higher indices. For example, if two ALU instructions are scheduled to be executed in the

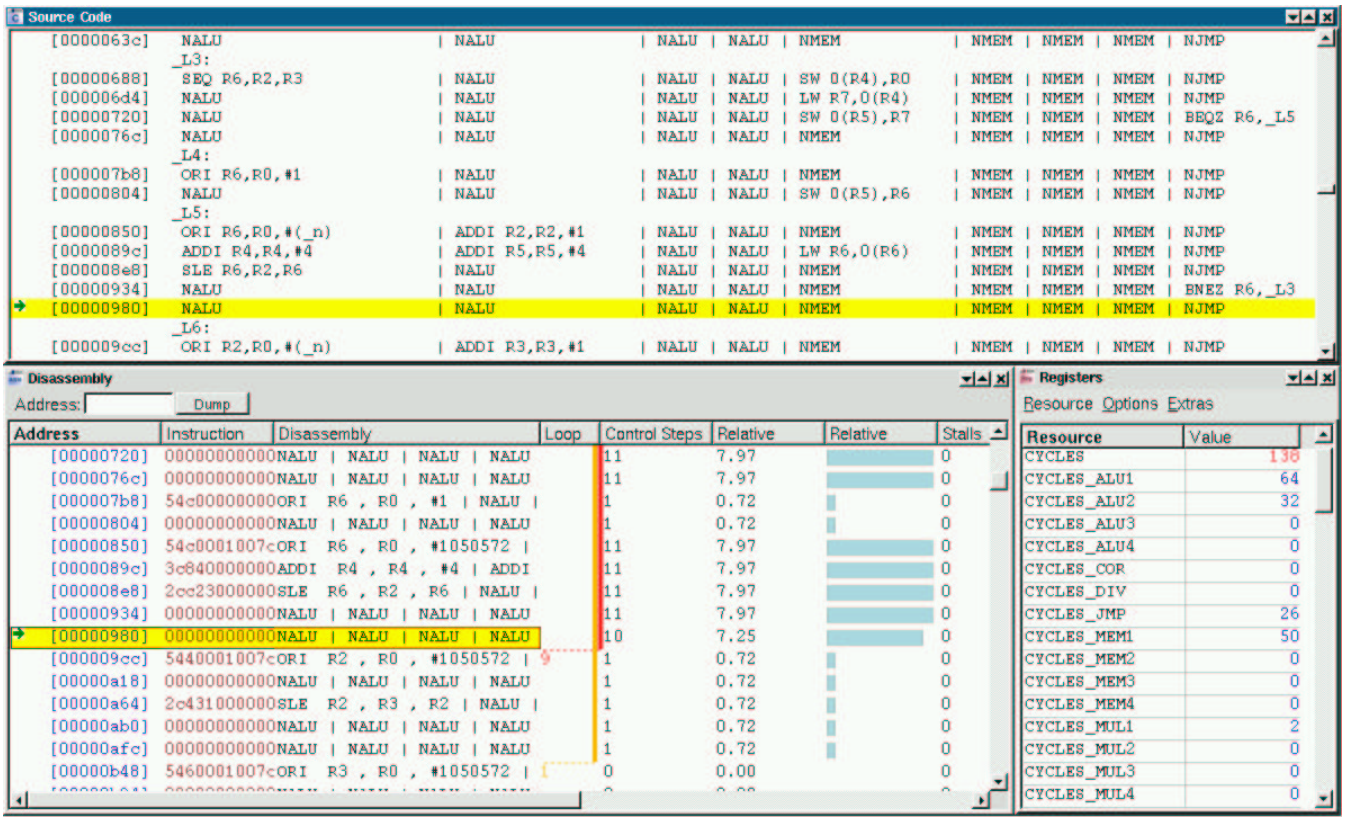

Figure 5: Profiling the ALICE architecture with LISA: simulation of parallelized assembly instructions, counting/visualization of loops, instruction executions, total execution cycles, activations of functional units

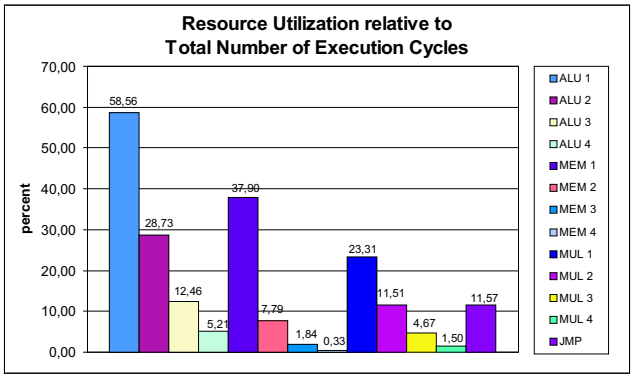

Figure 6: ALICE Resource Utilization using 4 ALUs, 4 MEMs, 4 MULs

same cycle they will be executed on ALU 1 and ALU 2. Consequently three parallel ALU instructions would occupy ALUs 1,2, and 3. Thus figure 6 also gives an impression of how much parallelism was exploitable by the compiler.
To analyze the effect of reducing the number of functional units a small manual change of the CoSy scheduler description is sufficient. The relevant excerpt from the CoSy scheduler description for the ALICE configuration in figure 6 looks like this:

```
RESOURCES
    alu1, alu2, alu3, alu4,
    mem1, mem2, mem3, mem4,
    mul1, mul2, mul3, mul4,
    jmp, div, cor;

TEMPLATES
    DEFTMPL := ();
    ALU_op  := alu1 | alu2 | alu3 | alu4;
    MEM_op  := mem1 | mem2 | mem3 | mem4;
    JMP_op  := jmp;
    MUL_op  := mul1 | mul2 | mul3 | mul4;
    DIV_op  := div;
    COR_op  := cor;
```

After the declaration of all resources, alternative possibilities of resources allocations are listed in the *TEMPLATES* section. For example, the line `ALU_op := alu1 | alu2 | alu3 | alu4;` indicates that a utilization of an ALU involves the allocation of either alu1, alu2, alu3, or alu4. To restrict the compiler to use only two of the four ALUs this line must be changed into `ALU_op := alu1 | alu2;` If the other lines are changed accordingly and after re-generating the C compiler it is quickly possible to analyze the performance of an ALICE architecture that contains two ALUs, two multipliers and a single memory unit as depicted in figure 7. Note that the overall execution time rises from 107,895 to 129,102 cycles.

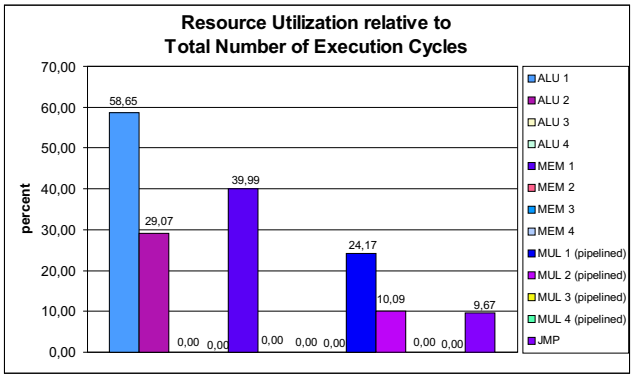

Figure 7: ALICE Resource Utilization using 2 ALUs, 1 MEMs, 2 pipelined MULs

6.4 Exploring Special Purpose Units

Since the profiling reveals that the EVD makes extensive use of the CORDIC functions the performance/power efficiency can be increased significantly by introducing a hardware implementation of the CORDIC algorithm into the architecture. Therefore an additional so called *LISA operation* is inserted into the ALICE model which increases the width of the architecture's internal VLIW word. For the LISA model and thus for the corresponding generated tools there is no verification problem because the C code of the CORDIC functions can be reused in the description of the new LISA operation. On the CoSy side the new unit can be addressed by an *intrinsic* which can be introduced to the compiler's parser by a single pragma and a pattern matcher rule that covers the intrinsic like a function call.

6.5 Exploring Latencies, Forwarding, and Register File Size

The modeling and exploration of latencies resulting from design alternatives (e.g. forwarding logic) can be done by changing the CoSy scheduler description. It is not necessary (though possible) to model latencies in the LISA model. In the ALICE LISA model all results are directly available after their calculation and latencies inside of functional units are not existent. The latencies and forwarding logic are modeled with a latency matrix which is part of the compiler's scheduler description. A reduced example looks like this:

```
TRUE     ALU_In  MUL_In:
ALU_Out  1       1,
MUL_Out  2       2;
```

This means that the ALU result is available in the next cycle whereas the multiplication result is available for both ALU and multiplier after two cycles. Together with the `MUL_op` template depicted above this CoSy description models the two stage pipelined multiplier which was used for figure 7. An architecture with two non-pipelined multipliers would have the following resource template: `MUL_op := (mul1 & mul1)|(mul2 & mul2)` Note that the ampersand indicates an allocation in a subsequent cycle.
In [15] it was pointed out that the register file size has a significant impact on energy consumption, code size, and execution time. Using ALICE the exploration of this important design parameter can easily be done by restricting the number of registers that are available for the compiler's register allocator. The corresponding statement in the CoSy description is depicted below. It makes registers R0 to R28 and R31 available for the compiler's register allocator.

```
AVAIL <R0..R28,R31>;
```

The sizes and alignments of C data types are modeled in CoSy by the *Target Description File (TDF)*. LISA provides an integer data type of arbitrary bit size for this purpose.

7. RESULTS

The time for a design cycle in the architecture exploration loop which has been presented in section 6 is dependent on the time required for rebuilding the CoSy- and LISA model and the simulation time. A complete rebuild of the CoSy compiler takes about thirty minutes on a Solaris Ultra 10. A compiler rebuild due to a change of the scheduler description takes about 5 minutes. This is the same amount of time required by LISA to generate its code generation tools and the simulator. The simulation time of the complete application in the LISA simulator is negligible, due to the use of high speed compiled simulation [6].

7.1 Execution Cycles

Figure 8 gives a good impression of the compiler's and the architecture's performance and the improvements that are achievable by tailoring ALICE to the EVD application.

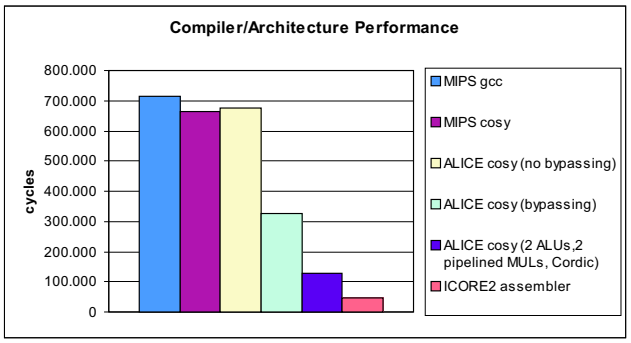

Figure 8: Performance of Compiler and Architecture

To evaluate the compiler's performance we modified the ALICE compiler to generate code for the MIPS32 [10] architecture and compared the assembly code of the EVD to the one generated by the freely available GNU compiler for MIPS32 [8]. The modifications took about one week and involved the elimination of all parallel units by reducing the fetch stage to a single instruction slot. Since MIPS only supports loading of 16 bit immediates and potentially stalls the pipeline dependent on the control flow some other modifications were necessary as well. However the high level optimizations for the MIPS32 CoSy compiler were inherited from the ALICE compiler. The CoSy compiler was called with the highest optimization level. GCC was called with -O3 so that both compilers make use of all available optimizations except for function inlining [2]. Both compilers generated more than

[2]In contrast to other GCC backends the MIPS32 compiler performs inlining only with option -O4

1500 lines of assembly code. As one can see in figure 8 the EVD compiled with the MIPS CoSy compiler requires about 50000 cycles less than EVD compiled with GCC.
The third bar from the left depicts the number of cycles required by an instance of the ALICE architecture to execute the EVD codec compiled with the corresponding CoSy compiler. This architecture comprises all functionality of the MIPS32 processor but parallelizes the utilization of a single ALU, a jump unit, a memory unit, and a multiplier. In contrast to MIPS32 there is no forwarding logic used in this architecture which causes a latency of two or three cycles for all operations. As one can see in the fourth bar only the introduction of forwarding logic lets the compiler efficiently use the parallel units: the total cycle count is reduced by 52 percent.
After further design space iterations the resulting ALICE architecture contains two ALUs, two pipelined multipliers with a latency of two cycles, a single memory unit, and a special purpose unit for the CORDIC algorithm. Independent of the compiler used for MIPS32 code generation this architecture evaluates the EVD codec in 18 percent of the cycles required on a MIPS32 processor.

7.2 Hardware Efficiency

To estimate the efficiency of the tailored ALICE architecture we compared it to another ASIP called ICORE2 using Synopsys' Design Compiler for logic synthesis. The design goal of ICORE2 was to provide an efficient and configurable processor for matrix and other linear algebra kernels that occur in mobile telecommunication applications. The important properties of ICORE2 are:

Complex data types: The registers are split into two parts to comprise real and imaginary part of a complex value. The mathematical operations of ICORE2 work on these complex values.

Vector and matrix instructions: There is extensive Single Instruction Multiple Data (SIMD) support. Beside vector-vector operations it is also possible to execute matrix-matrix and matrix-vector operations with a single instruction.

Addressing modes: There are dedicated addressing modes for fetching vector- or matrix elements.

CORDIC unit: The tailored ALICE processor and ICORE2 comprise the same CORDIC unit.

Hardware loops: There is support for zero overhead loops.

Hidden pipeline: The processor's pipeline is not visible to the programmer.

Since most of these features are quite difficult to address from an ANSI-C compiler the architecture was designed to be programmed in assembly. The rightmost column in figure 8 depicts the number of cycles a complete calculation of the 10×10 matrix EVD requires on the ICORE2 architecture.
The absolute execution time of the EVD which is calculated by the quotient of the number of execution cycles and the clock frequency is depicted in table 1. The table also lists the maximum clock frequency and the die size of the MIPS32, the tailored ALICE, and the ICORE2 architecture. All numbers have been obtained for a typical 0.18μm CMOS technology using defined worst case conditions for temperature, voltage, and fabrication. To support assembly programming, memory access and arithmetic/logic functionality are located in the same pipeline stage of ICORE2 which is also the reason for its reduced clock frequency compared to ALICE. This way the result of an instruction is available in the next cycle without the overhead of implementing a pipeline hazard detection.

Architecture:	**MIPS32**	**ALICE**	**ICORE2**
Frequency (MHz)**:**	170-200	190	140
Die Size (mm^2)**:**	≤ 1.0	≤ 1.5	≤ 0.4
Time for EVD (ms)**:**	4.19-3.57	0.43	0.32

Table 1: Architecture Comparison

8. CONCLUSIONS AND OUTLOOK

In this paper we demonstrated an architecture exploration methodology with a C compiler in the iteration loop. In a case study it was pointed out how the LISA processor design platform and the CoSy compiler environment can effectively be used to tailor the scalable ALICE architecture to a typical telecommunication kernel. The methodology and results were compared to the ICORE2 ASIP which was designed for the same application domain but comprises architectural features that require an assembly entry into the design flow.
Using the C compiler the software- and architecture designers can study the application's performance requirements immediately after the algorithm designer has finished his work. Afterwards the time required for the exploration steps presented in this paper is in the range of minutes or hours. In contrast, for ICORE2, even the writing and verification of the assembly code only took one week. Even worse this time is in the exploration loop each time the architecture is changed. An additional benefit of the compiler in the exploration loop is the fact that a compiler/architecture combination which is tailored for a certain application domain can easily be adapted to further applications of the same domain without the need to rewrite hundreds of assembly lines.
Of course the reusability and the restriction of architectural alternatives do not come for free: The hardware synthesis results show that the performance of ALICE stays about 30 percent behind ICORE2 and the die size is larger. The reason for this are ICORE2's zero-overhead loops, dedicated addressing modes, and special purpose instructions on the one hand and a less complex decoder and the absence of forwarding logic on the other. If these parameters are unacceptable the architecture used for exploration can nevertheless be used as a starting point for more complex refinements: In our example the designer could manually extend the C compiler to utilize the hardware loop support that comes with the CoSy environment or he could implement further intrinsics and optimization engines to target ICORE2's hardware features from the C compiler. Alternatively, he could use the LISA profiling capabilities to identify the *hot spots* of the algorithm and program them in assembly.
Future research will focus on a closer integration of the LISA and the CoSy environments to minimize the verification effort between the corresponding model descriptions. Beside

the generation of compiler components from LISA we want to extend the architectural scope that can be addressed by our compiler/architecture codesign methodology. The implementation of C level source code debugging is planned as well.

9. REFERENCES

[1] A. Fauth and J. Van Praet and M. Freericks. Describing Instruction Set Processors Using nML. In *Proc. of the European Design and Test Conference (ED & TC)*, Mar. 1995.

[2] A. Halambi and P. Grun and V. Ganesh and A. Khare and N. Dutt and A. Nicolau. EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability. In *Proc. of the Conference on Design, Automation & Test in Europe (DATE)*, Mar. 1999.

[3] A. Hoffmann and T. Kogel and A. Nohl and G. Braun and O. Schliebusch and O. Wahlen and A. Wieferink and H. Meyr. A Novel Methodology for the Design of Application Specific Instruction Set Processors (ASIP) Using a Machine Description Language. *IEEE Transactions on Computer-Aided Design*, 20(11):1338–1354, Nov. 2001.

[4] A. Hoffmann and T. Kogel and H. Meyr. A Framework for Fast Hardware-Software Co-simulation. In *Proc. of the Conference on Design, Automation & Test in Europe (DATE)*, Mar. 2001.

[5] ACE – Associated Computer Experts bv. *The COSY Compiler Development System http://www.ace.nl.*

[6] Achim Nohl and Gunnar Braun and Oliver Schliebusch and Rainer Leupers and Heinrich Meyr. A Universal Technique for Fast and Flexible Instruction-Set Architecture Simulation. In *Proc. of the Design Automation Conference (DAC)*, Jun. 2002.

[7] Adelante Technologies. *AR|T Designer http://www.adelantetechnologies.com.*

[8] Algorithmics Ltd. *Tools and Services for MIPS Developers http://www.algor.co.uk.*

[9] C. Kozyrakis and D. Judd and J. Gebis and S. Williams and D. Patterson and K. Yelick. Hardware/Compiler Codevelopment for an Embedded Media Processor. *Proc. of the IEEE Microprocessor Architecture & Compiler Technology*, 89(11):1694–1709, Nov. 2001.

[10] D. Patterson and J. Hennessy. *Computer Organization & Design: The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., 1998. Second Edition.

[11] G. Hadjiyiannis and S. Hanono and S. Devadas. ISDL: An Instruction Set Description Language for Retargetability. In *Proc. of the Design Automation Conference (DAC)*, Jun. 1997.

[12] H. Krim and M. Viberg. Two decades of array signal processing research. *IEEE Signal Processing Magazine*, pages 67–95, Jul. 1996.

[13] Improv. *Jazz http://www.improvsys.com.*

[14] J. Teich and R. Weper. A Joined Architecture/Compiler Design Environment for ASIPs. In *Proc. of the Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, Nov. 2000.

[15] L. Wehmeyer and M. K. Jain and S. Steinke and P. Marwedel and M. Balakrishnan. Analysis of the Influence of Register File Size on Energy Consumption, Code Size, and Execution Time. *IEEE Transactions on Computer-Aided Design*, 20(11):1329–1337, Nov. 2001.

[16] R. Leupers. *Retargetable Code Generation for Digital Signal Processors*. Kluwer Academic Publishers, Boston, June 1997. ISBN 0-7923-9958-7.

[17] LISATek Inc. *http://www.lisatek.com.*

[18] M. Itoh and S. Higaki and J. Sato and A. Shiomi and Y. Takeuchi A. Kitajima and M. Imai. PEAS-III: An ASIP Design Environment. In *Proc. of the Int. Conf. on Computer Design (ICCD)*, Sep. 2000.

[19] P. Paulin. Towards Application-Specific Architecture Platforms: Embdedded Systems Design Automation Technologies. In *Proc. of the EuroMicro*, Apr. 2000.

[20] R. Leupers and P. Marwedel. Retargetable Code Compilation based on Structural Processor Descriptions. *Design Automation for Embedded Systems*, 3(1):1–36, Jan. 1998. Kluwer Academic Publishers.

[21] R.M. Stallman. *Using and Porting the GNU Compiler Collection*. Free Software Foundation, gcc-2.95 edition, Jul. 1999.

[22] S. Aditya and B. R. Rau and V. Kathail. Automatic Architectural Synthesis of VLIW and EPIC Processors. In *Proc. of the Int. Symposium on System Synthesis (ISSS)*, pages 107–113, Nov. 1999.

[23] S. Onder and R. Gupta. Automatic Generation of Microarchitecture Simulators. In *Proc. of the International Conference on Computer Languages (ICCL)*, pages 80–89, May 1998.

[24] Synopsys. *http://www.synopsys.com.*

[25] T. Conte and S. Banerjia and S. Larin and K. Menezes and S. Sathaye. Instruction Fetch Mechanisms for VLIW Architectures with Compressed Encodings. In *Proc. of the 29th Symposium on Microarchitecture*, Dec. 1996.

[26] Target Compiler Technologies. *CHESS/CHECKERS http://www.retarget.com.*

[27] Tensilica. *Xtensa http://www.tensilica.com.*

[28] The Open SystemC Initiative (OSCI). *Functional Specifcation for SystemC 2.0 http://www.systemc.org.*

[29] Trimaran. *An Infrastructure for Research in Instruction-Level Parallelism http://www.trimaran.com.*