Compiler-in-loop Architecture Exploration for Efficient Application Specific Embedded Processor Design

Manuel Hohenauer, Hanno Scharwaechter,
Kingshuk Karuri, Oliver Wahlen, Tim Kogel,
Rainer Leupers, Gerd Ascheid, Heinrich Meyr
Integrated Signal Processing Systems,
Aachen University of Technology,
Aachen, Germany
E-mail: lisa@iss.rwth-aachen.de

Gunnar Braun
CoWare Inc.
Aachen, Germany
E-mail: gunnar@coware.com

Abstract

Retargetable C compilers are key tools for efficient architecture exploration for embedded processors. In this paper we describe a novel approach to retargetable compilation based on LISA, an industrial processor modeling language for efficient application-specific instruction set processor (ASIP) design. In order to circumvent the well-known trade-off between flexibility and code quality in retargetable compilation, we propose a user-guided, semi-automatic methodology that in turn builds on a powerful existing C compiler design platform. Our approach allows to include generated C compilers into the ASIP architecture exploration loop at an early stage, thereby allowing for a more efficient design process and avoiding application/architecture mismatches. We present the corresponding methodology and tool suite and provide experimental data for two real-life embedded processors that prove the feasibility of the approach.

1. Introduction

An increasing number of embedded Systems-on-a-Chip (SoC) employ application-specific instruction set processors (ASIPs) as building blocks [1]. Such processors show highly optimized instruction sets and architectures, tailored for dedicated application domains such as image processing or network traffic management. ASIPs are getting more and more attractive due to their balance between computational efficiency and flexibility. Furthermore, ASIPs offer a high potential for IP reuse and product differentiation.

Due to the complexity introduced by such SoC designs and time-to-market constraints, the designer's productivity has become the vital factor for successful products. In the current technical environment, embedded processors and the necessary development tools are designed manually, with very little automation. This is because the design and implementation of an embedded processor, such as a DSP device embedded in a cellular phone, is a highly complex process composed of the following phases: architecture exploration, architecture implementation, application software design, and system integration and verification.

During the *architecture exploration phase* (fig. 1), software development tools (i.e. C compiler, assembler, linker, and cycle-accurate simulator) are required to profile and benchmark the target application on different architectural alternatives. 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. 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. *Architecture exploration* is usually an iterative process that is repeated until a best fit between selected architecture and target application is obtained. Every change to the architecture specification requires a complete new set of software development tools. These changes on the tools are carried out usually manually. This results in a long, tedious and extremely error-prone process.

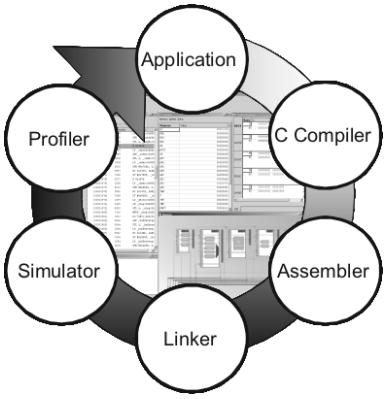

Figure 1. Tool based processor architecture exploration loop

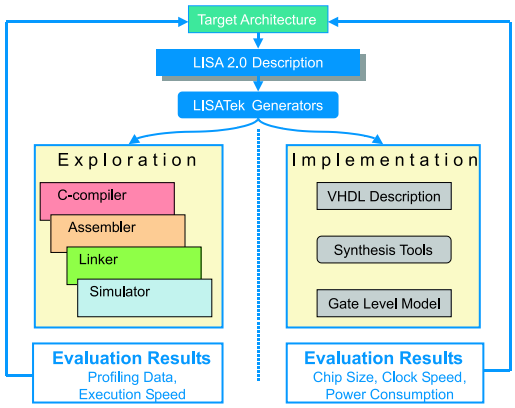

Figure 2. LISATek EDGE based ASIP design flow

In the *architecture implementation phase*, the specified processor is converted into a synthesizable HDL model. With this additional manual transformation, quite naturally, considerable consistency problems can arise between the architecture specification, the software development tools, and the hardware implementation.

During the *software application design phase*, software designers need a set of production-quality software development tools. Since the demands of the software application designer and the hardware processor designer place different requirements on software development tools, new tools are required. For example, the processor designer needs a cycle/phase-accurate simulator for hardware/software partitioning and profiling, which is very accurate but inevitably slow, whereas the application designer demands more simulation speed than accuracy. At this point, the complete software development tool-suite is usually re-implemented by hand – consistency problems are self-evident. In the *system integration and verification phase*, co-simulation interfaces must be developed to integrate the software simulator for the chosen architecture into a system simulation environment. Again, manual modification of the interfaces is required with each change of the architecture.

This iterative exploration approach demands very flexible *retargetable* software development tools (C compiler, assembler, simulator/debugger etc.) that can be quickly adapted to varying target processor configurations. Such tools are usually driven by a processor model given in a dedicated specification language.

One of the major challenges in this context is *retargetable compilation* for high-level programming languages like C or C++. First of all, it is difficult to extract the compiler semantics from a processor model in a description language that might not be explicitly designed for supporting retargetable compilation. Moreover, there exists a trade-off between the compiler's flexibility and the quality of compiled code. Since embedded systems usually demand very high code quality, retargetable compiler design has to be performed carefully in order to ensure that these demands can be met.

Though many approaches to retargetable compilation exist (see section 2 or [4] for an overview), only few have made their way into industrial practice so far. Examples include Tensilica's configurable Xtensa processor approach [5], which exploits some retargeting capabilities of the GNU C compiler, or Target's CHESS compiler [6], which is largely based on in-house compiler technology. In order to ensure sufficient code quality, these approaches tend to sacrifice flexibility, e.g. the Xtensa is based on a largely predefined RISC processor core, while CHESS is primarily targeted towards DSP processors.

In this paper we present a new approach to retargetable C compiler generation that is based on the LISA 2.0 processor modeling language [3] and that is fully integrated into an existing industry-proven ASIP design tool platform. Our main intention is to preserve the highest flexibility possible while still achieving high code quality. The key concepts to achieve this goal are the reuse of a powerful C compiler design platform with many built-in code optimizations, dedicated new algorithms for automatically extracting certain compiler components, as well as a graphical user interface to support semi-automatic retargeting and user interaction whenever required.

The paper is organized as follows. Section 2 discusses further related work. Section 3 provides a brief system overview. The core part of this paper is section 4, describing the mixed automatic and semi-

automatic retargeting methodology in our framework. Experimental results for two real-life ASIP architectures are given in section 5, while section 6 concludes and mentions future work.

2. Related work

Well-known retargetable C compiler tools for general-purpose processors include the GNU C compiler [7] and LCC [8]. These tools have a preference for ”compiler-friendly” RISC-like target processors and thus are not well-suited for ASIP applications. Moreover, they rely on very dedicated, heterogeneous processor modeling languages. Such custom languages are well tailored to compiler retargeting but cannot serve any other purpose in ASIP design (e.g. instruction set simulator retargeting or generation of HDL models for implementation), and hence would imply problems of model inconsistencies in the ASIP design flow.

Compiler systems like FlexWare [9], SPAM [10], or LANCE [11] include dedicated code optimizations for embedded processors but use rather heterogeneous processor modeling formalisms comprising of multiple sections of different languages. Other systems like RECORD [12], AVIV [13], Trimaran [14], or Mescal [16] are focused on special families of target machines like DSPs, VLIWs, or NPUs and are hence not very flexible.

The approaches that come closest to ours are Expression [2], ASIP Meister [17], and CHESS [6]. Similar to our approach with the LISA language, Expression uses a dedicated, unified processor *architecture description language* (ADL) with applications beyond compiler retargeting (e.g. simulator generation), but the retargeting capabilities for complex real-life processor architectures have not yet been demonstrated. Like our approach, ASIP Meister builds on the CoSy compiler platform [18]. However, it has no uniform ADL (i.e. target machine modeling is completely based on GUI entry) and the range of target processors is restricted due to a predefined processor component library. CHESS, already mentioned in section 1, uses the nML ADL [19] for processor modeling and compiler retargeting. Unfortunately, only few details about retargeting CHESS have been published. Like LISA, nML is a hierarchical mixed structural/behavioral ADL that (besides capturing other machine features) annotates each instruction with a *behavior description*. While LISA permits arbitrary C code for behavior descriptions, such descriptions in nML are restricted to a predefined set of operators, which probably limits the flexibility of CHESS.

3. System overview

The compiler framework presented in this paper builds on the *LISATek EDGE Processor Designer* (fig. 2), a tool suite for embedded processor design available from CoWare Inc. [20], an earlier version of which has been described in detail in [3]. 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 ICORE1.

The *LISATek EDGE Processor Designer* 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 amongst others software development tools like instruction set simulator [21], debugger, profiler, assembler, and linker, and it provides capabilities for VHDL and SystemC generation for hardware synthesis. The retargetable C compiler is seamlessly integrated into this tool chain and uses the same single ”golden reference” LISA model to drive retargeting.

The LISA instruction set simulator can be easily integrated into a co-simulation environment using a set of well-defined interfaces and verified quickly. In [25] the integration of several LISA models into the SystemC [26] 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 transition from instruction accurate abstraction to cycle accurate abstraction. 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. The builtin profiler, if enabled, detects and counts loops, sums up read and write accesses on registers and collects pipeline execution statistics like the number of stalls and flushes.

On the compiler side, our approach relies on the CoSy system from ACE [18]. CoSy is a modular C/C++ compiler generation system that offers numerous configuration possibilities both at the level of the intermediate representation and the backend for machine code generation (fig. 5). For our purpose, the latter is of primary interest. Similar to the GNU C compiler, CoSy comes with a heterogeneous, compiler-oriented processor modeling formalism (CGD) without direct connections to multi-purpose ADLs like LISA or nML. A CGD model consists mainly of three components:

- a specification of available *target processor resources* like registers or functional units
- a description of *mapping rules*, specifying how C/C++ language constructs map to (potentially blocks of) assembly instructions
- a *scheduler table* that captures instruction latencies as well as instruction resource occupation on a cycle-by-cycle basis

Apart from that, CoSy requires some more information like function calling conventions or the C data type sizes and memory alignment. From this information and the CGD model a C/C++ compiler can be automatically generated. We selected CoSy as a platform mainly due to its robustness, flexibility, and its large suite of already built-in code optimization engines.

3.1 Structure of a LISA Description

The following section outlines the structure of a LISA description briefly. A LISA processor description consists of two parts: The LISA operation tree (see fig. 4) and a resource specification. The operation tree is a hierarchical specification of instruction coding, syntax, and behavior. The resource specification describes memories, caches, processor registers, signals, and pipelines. An example for a single LISA operation is given in figure 3. The name of this operation is `reg_alu_instr` and it is located in the `ID` stage (instruction decode) of the pipeline `pipe`. The `DECLARE` section lists the sons of `reg_alu_instr` in the operation tree. `ADD` and `SUB` are names of other LISA operations that have their own binary coding, syntax, and behavior. As one can see the respective sections are referenced by the group declarator `Opcode` in the `CODING` and `SYNTAX` section. The `BEHAVIOR` section indicates that elements of the `GP_Registers` array resource are read, and the contents are written into pipeline registers. This means that the general purpose register file is read in the instruction decode stage.

The `ACTIVATION` section describes the subsequent control flow of the instruction "through" the processor instruction pipeline. The LISA operation referenced by the group `Opcode` (i.e. either `ADD` or `SUB`) is eventually located in a subsequent pipeline stage, which means that it will be activated in a subsequent cycle.

4. Compiler backend retargeting

Our LISA based C compiler generator can be coarsely viewed as a *LISA-to-CGD translator*. It extracts compiler-relevant information from a given LISA target processor model and emits CGD. Finally, CoSy is invoked to generate a C/C++ compiler using the generated CGD. However, this translation is far from trivial due to a number of reasons: while some information is explicit in the LISA model (e.g. via resource declarations), other relevant information (e.g. concerning instruction scheduling) is only implicit and needs to be extracted by special algorithms. Some further, heavily compiler-specific, information is not at all present in the LISA model, e.g. C type bit widths.

Compiler retargeting is further complicated by the *semantic gap* between the compiler's high-level model of the target machine and the detailed ADL model that, in particular, must capture cycle and bit-true behavior of machine operations. In order to ensure highest flexibility, LISA permits specifying machine operation behavior in the form of arbitrary C/C++ code. On the other hand, this feature makes it difficult

```
OPERATION reg_alu_instr IN pipe.ID
{
  DECLARE {
    GROUP Opcode = { ADD || SUB };
    GROUP Rs1, Rs2, Rd = { register };
  }

  CODING { Opcode Rs2 Rs1 Rd 0b0[10] }
  SYNTAX { Opcode ~" " Rd ~" " Rs1 ~" " Rs2 }

  BEHAVIOR {
    PIPELINE_REGISTER(pipe,ID/EX).src1 = GP_Registers[Rs1];
    PIPELINE_REGISTER(pipe,ID/EX).src2 = GP_Registers[Rs2];
    PIPELINE_REGISTER(pipe,ID/EX).dst = Rd;
  }

  ACTIVATION { Opcode }
}

OPERATION ADD IN pipe.EX
{
  REFERENCE Rs1, Rs2, Rd;
  SYNTAX { "ADD" }

  BEHAVIOR {
    unsigned int src1,src2;
    if (bypass_reg == Rs1)  src1 = bypass_content;
    else  src1 = PIPELINE_REGISTER(pipe,ID/EX).src1;
    if (bypass_reg == Rs2)  src2 = bypass_content;
    else  src2 = PIPELINE_REGISTER(pipe,ID/EX).src2;

    unsigned int result = src1 + src2;

    PIPELINE_REGISTER(pipe,EX/WB).dst = result;
    bypass_reg = Rd;
    bypass_content = result
  } ...
}
```

Figure 3. LISA operation description example

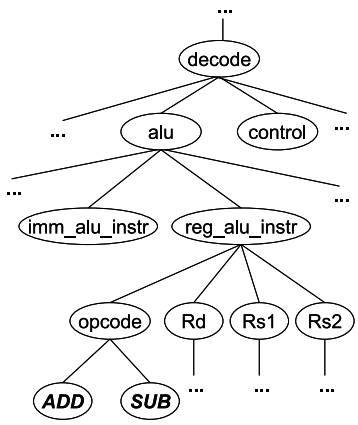

Figure 4. LISA operation tree

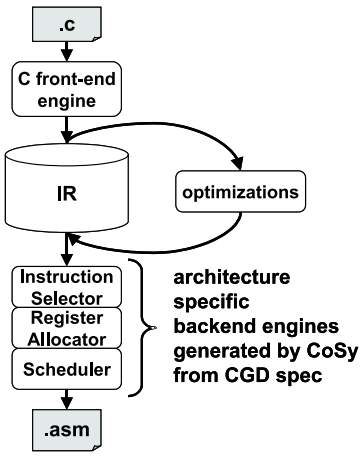

Figure 5. Generated CoSy compiler structure

to extract compiler semantics from such ”informal” models of instructions. To further complicate matters, the instructions behavior is usually distributed over several pipeline stages.

Since we intentionally do not want to sacrifice flexibility nor code quality to address these challenges, we employ a pragmatic *semi-automatic approach* to retargetable compilation. Compiler information is automatically extracted from LISA whenever possible, while GUI-based user interaction is employed for other compiler components. The GUI reads a given LISA model and presents all relevant machine features (e.g. resources and machine operations) to the user who can then refine/modify/add to the presented information. This approach is further detailed in the following sub-sections, with emphasis on code selector retargeting.

4.1. Machine parameters, stack layout, and calling conventions

Purely *numerical parameters* not present in the LISA model are directly entered via GUI tables. Such parameters include information like C type bit widths, type alignments, minimum addressable memory unit size etc. which are only useful for the compiler.

The compiler generator currently supports a single, generic *stack organization*. Support for more irregular layouts like a roving frame pointer as used in some DSPs is planned for future work. The generic version assumes the architecture provides stack and frame pointer registers as well as register-offset addressing. Based on this generic stack model, the user provides abstract operations needed for code generation for function prologues and epilogues. These abstract operations are later automatically mapped to real machine instructions by means of the set of specified code selection rules, using the mechanism described

in section 4.4.

Finally, a *calling conventions* GUI dialog allows the user to define the preferred passing of function parameters or return values for each C data type (either in registers or on stack).

4.2. Register allocator

Register allocation is already fully provided by CoSy. Therefore, retargeting the register allocator in our framework is reduced to the selection of allocatable registers out of the set of all available registers in the LISA model. For instance, registers selected as frame or stack pointer need to be excluded from allocation. Some processor architectures allow to combine several regular data registers to ”long” registers of larger bit width. The composition of long registers is also performed via the GUI.

4.3. Instruction scheduler

For a given set of instructions, a scheduler decides which instructions are issued on the processor in which cycle. For instruction level parallelism (ILP) architectures, this does not only mean that the scheduler decides on the sequence in which instructions are executed, but it also arranges instructions to be executed in parallel.

As described in [15], the freedom of scheduling is limited by two major constraints: structural hazards and data hazards. Structural hazards result from instructions that utilize exclusive processor resources. If two instructions require the same resource, these two instructions are mutually exclusive. A typical example of structural hazards is the number of issue slots available on a processor architecture: It is never possible to issue more instructions in a cycle than the number of available slots. So called *reservation tables* are used to describe this kind of hazards.

Data hazards result from the temporal I/O behavior of instructions. They can be subdivided into Read After Write (RAW), Write After Write (WAW), and Write After Read (WAR) hazards. An example for a RAW dependency would be a multiplication that takes two cycles to finish computation on a processor without interlocking hardware followed by a second instruction that has to consume the result of the multiplication. In this case the second instruction has a RAW dependence of two cycles onto the multiplication, which means that the second instruction must be issued two or more cycles after the multiplication. The data hazards are captured by so called *latency tables*.

The native CoSy *instruction scheduler* is instruction block based and hence not directly suitable for our framework, which requires scheduling at the granularity of single instructions. Hence, we have replaced the default scheduler by a custom scheduler (an improved version of a list scheduler, capable of efficiently filling delay slots) that is generated fully automatically from the LISA model. These techniques are described in a separate paper [22] and are guaranteed to result in a correct (yet sometimes too conservative) instruction scheduler. Therefore, the extracted scheduler characteristics (instruction latencies and reservation tables) are additionally displayed in the GUI to the user who may decide to manually override certain instruction latencies in case of too conservative latency estimations.

4.3.1 Extracting instruction latencies

Based on the LISA activation chains (see fig. 6), the access to processor resources for each instruction can be easily found out. The starting point of all activation chains is a special operation named `main`. It is executed in every control step of the simulator and activates the operation(s) in the first pipeline stage (fetch) which in turn activate(s) operations in subsequent pipeline stages. This means that based on the activation chain, the execution clock cycle for each LISA operation can be determined. Furthermore it can be analyzed whether the C code in the `BEHAVIOR` section of the operations reads or writes processor resources of the LISA model. The access direction and the resource names are organized in an instruction specific vector. Starting from cycle 0, each vector component represents a cycle that is required to execute the instruction. With this information possible data hazards between instructions can be detected and the latency can be calculated. The vectors of two example assembly instructions are depicted in figure 7. In this example the multiplication writes to the register `R14` in cycle one which is read by a subsequent add instruction in cycle zero which results in a RAW latency of two.

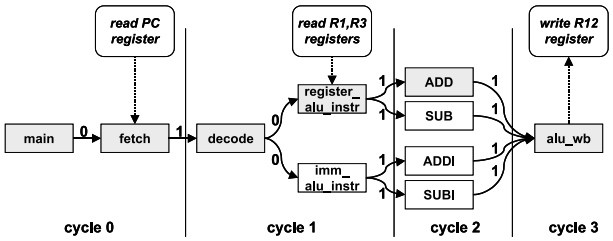

Figure 6. Activation chains of LISA operations

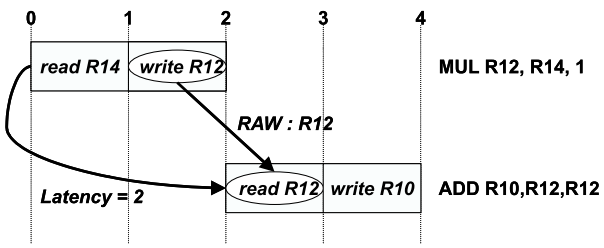

Figure 7. Latency analysis of two activation chains

4.3.2 Extracting reservation tables

The scheduler generator analyzes the instruction coding format of a LISA processor model and emits a reservation table. A resource vector describing the resource allocation is assigned to every instruction. To decide if an instruction can be scheduled into a cycle the instruction's resource vector is checked for conflicts with already allocated resources from other instructions in the current cycle. This mechanism allows or disallows the parallel scheduling of instructions.

4.4. Code selector

Retargeting the code selector is the most challenging task due to the *semantic gap* between the compiler's model and the detailed ADL processor model that was already mentioned at the beginning of section 4. This gap causes the following key problems w.r.t. code selector generation:

- Even though the mapping of C operations, or the compiler's intermediate representation (IR) operations, respectively, to machine instructions is intuitively clear, there is usually *no one-to-one correspondence* between IR operations to instructions. On the one hand, a single IR operation might need to be implemented by multiple instructions, e.g. for compiling a multiply operation onto a processor without a multiply instruction. On the other hand, a single machine instruction (like MAC on a DSP processor) might cover multiple IR operations.

- The behavior of machine instructions or operations in LISA is given in the form of arbitrary C/C++ code (possibly containing pipeline register transfers or side effects not relevant for code generation). It is nearly impossible to extract semantics of an instruction from such a general description. As an example, consider the snippet of a LISA operation description in fig. 3. It describes the execute stage of an ADD instruction and contains a partial description of the forwarding mechanism in the instruction pipeline. Since the actual operation performed (`result = src1 + src2`) is deeply embedded in the behavior description, it is obviously very difficult to analyze that the compiler semantics of this operation is nothing but an ADD instruction.

While in most previous approaches these problems are solved by reducing the compiler's flexibility or the expressiveness of the underlying ADL, in our approach we employ a user-guided code selector specification. The compiler generation GUI comprises a so-called *mapping dialog* (see fig. 8). This dialog presents to the user the set of IR operations to be covered in order to implement a "minimal" operational compiler (top left window in fig. 8) as well as the hierarchically organized set of machine operations in the given LISA model (right window).

By means of a convenient drag-and-drop mechanism, the user can compose *tree patterns* or *mapping rules* (top center window) from the IR operations. Like in most compilers, these mapping rules are the basis for the tree pattern matching based code selector in CoSy. Likewise, the link between mapping rules and their arguments on the one hand and machine operations and their operands on the other hand is made via drag-and-drop in the GUI. Naturally, multi-instruction rules as well as complex instructions like MAC can also be entered this way. The example from fig. 8 shows the mapping defined for a 32-bit multiply operation, which is implemented by a sequence of two 16-bit multiply instructions and an ADD instruction.

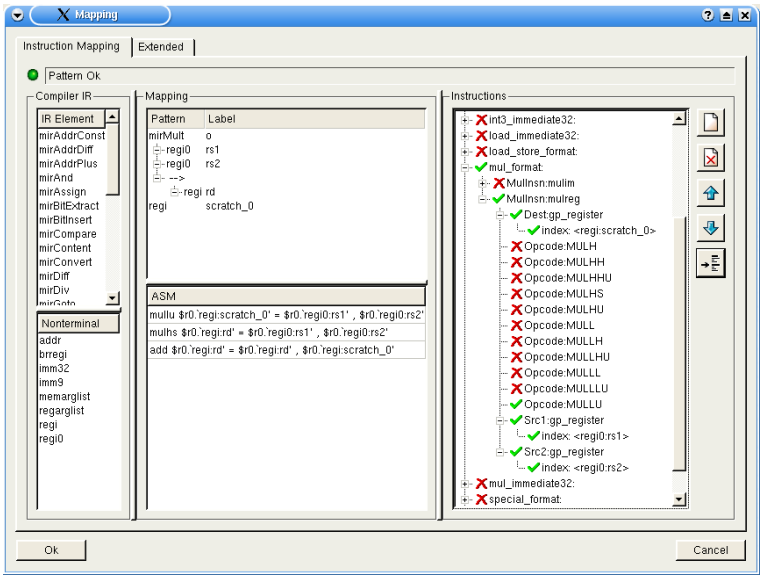

Figure 8. GUI Mapping dialog

Based on this manually established mapping, the compiler generator looks up the required assembly syntax of involved instructions (bottom center window) in the LISA model and can therefore automatically generate the code emitter for the respective mapping rule. The output of the code emitter is symbolic assembly code, which will be further processed by the register allocator and the instruction scheduler during code generation.

The mapping dialog also provides additional capabilities, e.g. for capturing rule attributes (e.g. type-dependent conditions for rule matching) or for reserving scratch registers for use in complex multi-instruction rules, such as in the above 32-bit multiply example.

Providing mapping rules for all IR operations enables the generation of a ”minimal” compiler suitable for early architecture exploration. At any time, the user may refine the code selector by adding more dedicated mapping rules that efficiently cover special cases leading to higher code quality.

The final output of the GUI is a compiler specification file in CoSy’s CGD format, from which in turn a C/C++ compiler is generated fully automatically. During compiler retargeting, the session status of the GUI can be saved in XML format and can be resumed at any time. Changes in the underlying LISA processor model are detected and all automatic retargeting phases are repeated if necessary. Due to the largely manual specification of the code selector, the GUI status may obviously be inconsistent after a change in the LISA model (e.g. removal of a certain instruction). In this case, the GUI restores its status as far as possible and prompts the user in case of inconsistencies. Only in case of significant changes, e.g. a complete rearrangement of the instruction set hierarchy, the code selector specification must be revised entirely. The user, however, is responsible for maintaining the correctness of the mapping rules, since pure changes in the instruction behavior description, without changing the hierarchy or the assembly coding, are not yet detected automatically in the current version.

5. Experimental results

In order to validate our approach, we have applied it to two complex, real-life embedded processor models. One is the PP32 NPU from Infineon Technologies (an earlier version of which is described in [23]), basically a RISC architecture customized for efficient protocol processing. The other one is the ST200 a high-performance VLIW machine from STMicroelectronics [24]. Both target processors have been modeled in LISA and C compilers have been generated with the tools mentioned above.

From the above discussions it should be obvious that thanks to the use of LISA and CoSy flexibility w.r.t. feasible target architecture classes is not a major concern in our approach. In retargetable compilation, high flexibility normally has to be paid with low code quality. In contrast, we aim at achieving acceptable code quality at the expense of a higher compiler retargeting effort than in a fully automatic retargeting approach.

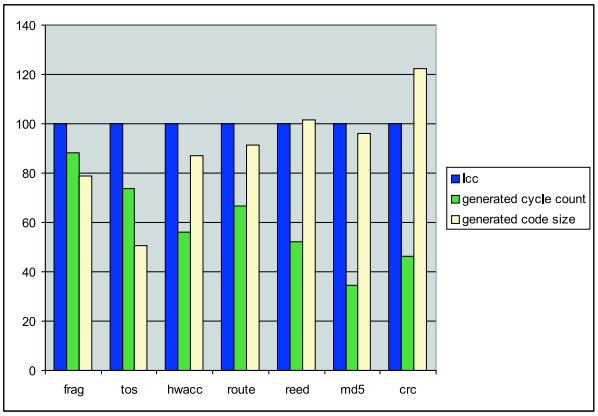

Figure 9. Rel. cycle count / code size PP32

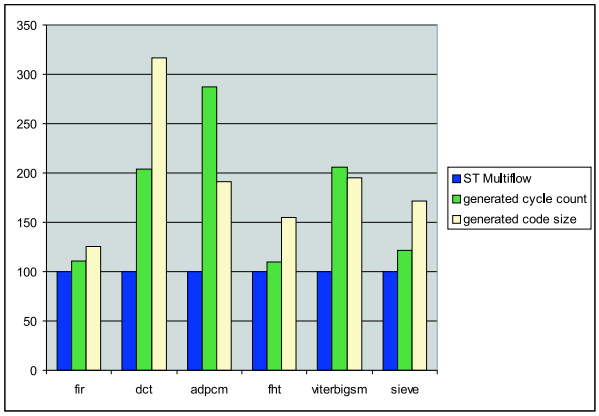

Figure 10. Rel. cycle count / code size ST200

This will be quantified in the following.

5.1. Infineon PP32

Since there is no vendor compiler available for the PP32 yet, we have manually retargeted the LCC compiler [8] to the PP32 as a baseline for comparisons. In addition, we have used our above semi-automatic approach to generate a C compiler with CoSy based on a LISA model of the PP32. The required retargeting time (starting from scratch with a given LISA model, not including the verification effort) was about one man-week in both cases. Thus, the GUI-based approach did not save time for developing the initial compiler. However, due to its tight embedding into the LISATek EDGE platform, with tools like assembler and debugger immediately at hand, it strongly facilitates architecture exploration.

In addition, our approach leads to higher code quality than for the LCC based compiler. Fig. 9 shows the relative cycle count and code size of code generated for seven benchmarks extracted from NPU applications, with the cycle count for the LCC based compiler set to 100%. Thanks to a richer set of built-in code optimization techniques, the generated CoSy based compiler on the average leads to an improvement of 40% in cycle count and 10% in code size.

5.2. ST200

For the ST200, we have compared our generated compiler to the ST Multiflow, a highly optimizing VLIW compiler provided by the processor vendor. The required initial retargeting time was approximately two man-weeks. The relative cycle count and code size is shown in fig. 10. In this case, the generated compiler shows an average overhead of 73% in cycle count and 90% in code size, partially due to extensive function inlining. These are acceptable values taking into account that the development time for the Multiflow compiler probably was orders of magnitude higher and we essentially compared it to an ”out-of-the-box” generated compiler without machine-specific optimizations.

Analysis of the generated code showed that by adding custom optimization engines, e.g. for exploiting predicated execution, significantly higher code quality could be easily achieved at the expense of higher manual effort. This is typically the stage in the design process where an initial generated compiler for architecture exploration would need to be refined to a highly optimizing production compiler. This is actually out of the scope of retargetable compilation. However, in future versions we plan to better differentiate compiler generation for different architecture classes (like VLIW, RISC, DSP) and to automatically invoke dedicated code optimization flows for these classes.

6. Conclusions

This paper has described a new and practical approach to retargetable C compilation for embedded processors that works for real-life target machines based on the LISA ADL. It is embedded into an industrial tool suite for ASIP design. Such an integrated approach, based on only a single ”golden” target

processor model, is key for an effective ASIP design environment. The C compiler can be incorporated into the processor architecture exploration loop right from the beginning, so as to avoid hardware/software mismatches. In the case of the PP32, for instance, performing architecture exploration including the C compiler has already led to instruction set modifications to make the processor more ”compiler-friendly”.

The novelty of the proposed approach is that high flexibility and acceptable code quality are achieved at the same time, yet at the expense of additional manual compiler description effort. However, our case studies indicate that this effort is very reasonable, given the significant advantage that (in contrast to today's common practice) an operational C compiler is available early in the ASIP design process. We have presented tools that, under the constraint of a given multi-purpose ADL as an input language, automate a significant part of C compiler retargeting. Compared to compiler generation with a pure ”stand-alone” system like CoSy, the compiler description effort is largely reduced. Moreover, our tools hide most compiler technology internals from the ASIP design engineer, who thus can better concentrate on architecture optimization.

In the future, we plan to further automate the C compiler generator, in particular w.r.t. code selector generation. Furthermore, we will evaluate the methodology for further representative embedded processor architectures, e.g. DSPs with more irregular architectures, in order to determine additional data points on the description effort vs. code quality trade-off.

References

[1] A. Oraioglu, A. Veidenbaum: *Application Specific Microprocessors (Guest Editors Introduction)*, IEEE Design & Test Magazine, Jan/Feb 2003

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

[3] A. Hoffmann, H. Meyr, R. Leupers: *Architecture Exploration for Embedded Processors with LISA*, Kluwer Academic Publishers, ISBN 1-4020-7338-0, Dec 2002

[4] R. Leupers, P. Marwedel: *Retargetable Compiler Technology for Embedded Systems - Tools and Applications*, Kluwer Academic Publishers, ISBN 0-7923-7578-5, Nov 2001

[5] Tensilica Inc.: http://www.tensilica.com

[6] Target Compiler Technologies: http://www.retarget.com

[7] Free Software Foundation/EGCS: http://gcc.gnu.org

[8] C. Fraser, D. Hanson: *A Retargetable C Compiler: Design And Implementation*, Benjamin/Cummings, 1995

[9] C. Liem, P. Paulin, M. Cornero, A. Jerraya: *Industrial Experience Using Rule-driven Retargetable Code Generation for Multimedia Applications*, 8th Int. Symp. on System Synthesis (ISSS), 1995

[10] G. Araujo: *Code Generation Algorithms for Digital Signal Processors*, Ph.D. thesis, Princeton University, Department of Electrical Engineering, 1997

[11] R. Leupers: *Code Optimization Techniques for Embedded Processors*, Kluwer Academic Publishers, 2000

[12] R. Leupers, P. Marwedel: *Time-Constrained Code Compaction for DSPs*, IEEE Transactions on VLSI Systems, vol. 5, no. 1, Mar 1997

[13] S. Hanono, S. Devadas: *Instruction Selection, Resource Allocation, and Scheduling in the AVIV Retargetable Code Generator*, 35th Design Automation Conference (DAC), 1998

[14] Trimaran home page: http://www.trimaran.org

[15] J. Hennessy and D. Patterson: *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers Inc., 1996

[16] Mescal home page: http://www.gigascale.org/mescal

[17] S. Kobayashi, Y. Takeuchi, A. Kitajima, M. Imai: *Compiler Generation in PEAS-III: an ASIP Development System*, Int. Workshop on Software and Compilers for Embedded Processors (SCOPES), 2001

[18] Associated Compiler Experts bv: http://www.ace.nl

[19] A. Fauth, J. Van Praet, M. Freericks: *Describing Instruction-Set Processors in nML*, European Design and Test Conference, 1995

[20] CoWare Inc.: *LISATek product family*, http://www.coware.com

[21] A. Nohl, G. Braun, A. Hoffmann, O. Schliebusch, H. Meyr, R. Leupers: *A Universal Technique for Fast and Flexible Instruction-Set Architecture Simulation*, Design Automation Conference (DAC), 2002

[22] O. Wahlen, M. Hohenauer, R. Leupers, H. Meyr: *Instruction Scheduler Generation for Retargetable Compilation*, IEEE Design & Test of Computers, Jan/Feb 2003

[23] X. Nie, L. Gazsi, F. Engel, and G. Fettweis: *A New Network Processor Architecture for High-Speed Communications*, IEEE Workshop on Signal Processing Systems (SiPS), 1999

[24] F. Homewood and P. Faraboschi: *ST200: A VLIW Architecture for Media-Oriented Applications*, Microprocessor Forum, 2000

[25] A. Hoffmann and T. Kogel and H. Meyr, *A Framework for Fast Hardware-Software Co-simulation*, European Design and Test Conference,2001

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