C Compiler Retargeting Based on Instruction Semantics Models

Jianjiang Ceng, Manuel Hohenauer, Rainer Leupers, Gerd Ascheid, Heinrich Meyr, Gunnar Braun

▶ To cite this version:

Jianjiang Ceng, Manuel Hohenauer, Rainer Leupers, Gerd Ascheid, Heinrich Meyr, et al.. C Compiler Retargeting Based on Instruction Semantics Models. DATE'05, Mar 2005, Munich, Germany. pp.1150-1155. hal-00181287

HAL Id: hal-00181287
https://hal.science/hal-00181287v1

Submitted on 23 Oct 2007

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

C Compiler Retargeting Based on Instruction Semantics Models

Jianjiang Ceng, Manuel Hohenauer,
Rainer Leupers, Gerd Ascheid, Heinrich Meyr
Integrated Signal Processing Systems
Aachen University of Technology

Gunnar Braun
CoWare, Inc.
Dennewartstrassse 25-27
Aachen, Germany

Abstract

Efficient architecture exploration and design of application specific instruction-set processors (ASIPs) requires retargetable software development tools, in particular C compilers that can be quickly adapted to new architectures. A widespread approach is to model the target architecture in a dedicated architecture description language (ADL) and to generate the tools automatically from the ADL specification. For C compiler generation, however, most existing systems are limited either by the manual retargeting effort or by redundancies in the ADL models that lead to potential inconsistencies. We present a new approach to retargetable compilation, based on the LISA 2.0 ADL with instruction semantics, that minimizes redundancies while simultaneously achieving a high degree of automation. The key of our approach is to generate the mapping rules needed in the compiler's code selector from the instruction semantics information. We describe the required analysis and generation techniques, and present experimental results for several embedded processors.

1. Introduction

With the increasing industrial acceptance of ASIPs as SoC building blocks, *retargetable C compilers* have become important tools in the system-level design flow. For instance, the embedded processor design tool suites from CoWare, Target Compiler Technologies, and Tensilica incorporate retargetable compilers that can be quickly adapted to varying target architectures.

In contrast to traditional compilers that target only a single processor architecture, retargetable compilers can generate code for different targets based on an external, editable *processor model*. They serve two main purposes in the design flow. First, they are required in processor *architecture exploration*. In this phase, the initial ASIP architecture gets fine-tuned for the intended applications which are mostly given in the form of C code. Second, retargetable compilers (possibly enhanced with target-specific code optimization techniques) can be used to quickly generate *production compilers* needed by the ASIP end users for software development.

In the domain of compiler construction for general-purpose processors, retargetable compilation has been a subject of research for quite some time. Systems like GCC [1] or LCC [2] have been successfully used for various CISC and RISC processors. In the context of ASIP design, where a tight link to hardware and SoC design flow is required, two mainstream approaches to retargetable compilation can be identified. The first one builds on a predefined, yet *configurable processor core* (e.g. Xtensa, ARCtangent) that can be optimized by the user via addition of custom machine instructions. In this case, semi-custom compiler systems (such as a modified GCC) can be used, and retargeting is implemented by making new instructions available to the compiler in the form of *intrinsics*. While this approach offers the advantage of reusing well-proven compiler tools, it restricts the flexibility of ASIPs and requires manual (and non-portable) C source code modifications.

The second approach is based on the paradigm of *architecture description languages* (ADLs) that permit high-level (i.e. beyond RTL) modeling of processors for early design phases such as architecture exploration and system-level verification. The challenge in retargetable compilation based on ADL models, however, is the large variety of potential target architectures, which is essentially only limited by ADL's capabilities. The ADL approach permits very high flexibility in ASIP design, but demands advanced techniques for efficient compiler retargeting.

In this paper, we describe a novel technique of compiler retargeting for LISA 2.0, a C/C++ based industrial ADL [3]. The proposed technique allows for a high degree of automation by extracting the core part of the compiler backend (the *code selector* that maps intermediate code to assembly) from the ADL model largely without user interaction. In this way, a high speedup in compiler generation is achieved, that eventually contributes to a more efficient ASIP design flow. Our technique is based on the *instruction semantics* information in LISA 2.0 models which provides a higher abstraction view of instruction behaviors than C/C++ descriptions.

The remainder of this paper is structured as follows. In

IEEE COMPUTER SOCIETY

section 2, we provide a survey of related works. Section 3 gives a brief system overview, while section 4 summarizes LISA's instruction semantics modelling concept. The focus of this paper is the automatic generation of code selectors. Section 5 this described in detail. The results of two case studies are given in section 6, and section 7 gives concluding remarks.

2. Related work

Early retargetable compilers for ASIPs include FlexWare [4], SPAM [5], and RECORD [6]. The last one builds on the MIMOLA modeling language and extracts code selector descriptions that can be further processed by iburg [7] to generate a code selector. The range of possible target architectures is limited, though, mainly due to the use of specialized processor modeling formalisms.

ASIP design systems using dedicated ADLs include AVIV [8], CHESS [9], Mescal [10], Expression [11], ASIPMeister [12], and ArchC [13], some of which include retargetable C compilers. Several of these systems focus on a specific class of ASIP architectures, e.g. DSP or VLIW, while others limit flexibility through a predefined library of processor components. Such kind of constraints is introduced by the fact that different processor development tools can hardly share the same description of instructions. For example, the generation of instruction set simulators requires a very detailed behavior description, while C compiler generation needs a clear description of instruction semantics without any structural or internal behavioral detail. So far, such contradictory demands are solved by introducing specific ADL constructs which provide information for the generation of different tools. For example, in the Expression ADL, there exist dedicated *operation mapping* sections that guide the code selector generation, while simulator generation is driven by other language constructs.

nML [9] and AXYS LISA [14] both have a hierarchical structure of descriptions, similar to the LISA 2.0. However, few information about their compiler generation is available. We have described an earlier version of the LISA 2.0 based retargetable compiler framework in [15]. Though most of the compiler components can be generated automatically, the code selector generation still requires some manual efforts. The technique proposed in this paper aims to further automate the code selector description generation.

3. System overview

C compiler generation is part of the LISA ASIP design flow sketched in fig. 1. A retargetable C compiler is integrated with CoWare's LISATek Processor Designer tool suite [3] that supports ASIP design by generating software development tools, synthesizable HDL models, and co-simulation interfaces from the LISA 2.0 processor models. With this approach, ASIP design is simplified due to a high degree of automation and the fact that just one processor model is needed for the entire design flow.

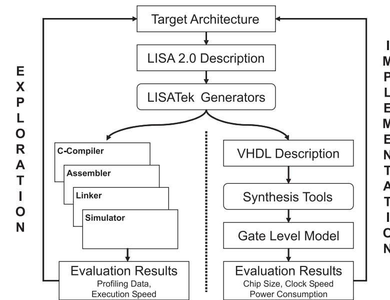

Figure 1. LISA 2.0 Based Design Flow

The main challenge of the compiler retargeting in this design flow is adaptation of the compiler backend (or *code generator*). A backend generally has a *code selector*, a *register allocator*, and an *instruction scheduler*. We use the CoSy system from ACE [16] as compiler development framework. The CoSy system contains a complete C/C++ frontend, a set of processor independent optimization engines and a retargetable compiler backend. The register allocator provided by the CoSy needs the names of allocatable registers to generate a target specific register allocator. In [15], works have been done to extract these information from LISA 2.0 models. We implemented our own instruction scheduler. Techniques, which are used to automatically generate scheduler descriptions, are described in [17]. The code selector generation in our previous work relied on a semi-automatic GUI based approach. Based on a LISA 2.0 model, compiler designers can use a GUI to specify code selector's mapping rules. According to our experience, this part of work is the most time consuming part in the whole retargeting process. Moreover, for a designer without enough compiler knowledge, it is difficult to determine if the created rules are complete for a working C compiler or not. Hence, further automation of the code selector generation is desired.

4. Instruction semantics

The LISA 2.0 processor models comprise *resource specifications* (e.g. registers, memories, and pipelines) and *operations*. Operations contain machine descriptions from different views such as assembly syntax, pipeline timing, binary encoding, and behavior. In the standard LISA 2.0 *behavior view* of instructions, plain C/C++ code is used for sake of highest flexibility and simulation speed. Since the same behavior may be described with numerous syntactic variances in C/C++, it is generally not possible to accurately extract the high-level semantic meaning of the instructions automatically from it. Therefore, we added a *semantics view* of operations that captures behavior at a higher abstraction level [18].

IEEE COMPUTER SOCIETY

Our approach is to capture instruction semantics explicitly by using a limited and precisely defined set of *micro-operations*, similar to MIMOLA. Each micro-operation describes a computation on the processor resources that can typically be executed by a data path component. For example, the _ADD operator in fig. 2 represents an addition operation. The side effects of instructions are captured by annotation besides the micro-operations. The _C and _Z in the example indicate that carry flag and zero flag are affected by the calculation. Rs1 and Rs2 are input operands. The right arrow denotes Rd as the destination of the result. As mentioned before, LISA 2.0 processor models have a hierarchical structure. Rd, Rs1 and Rs2 in this example all refer to the reg_32 operation. Then, the semantics of the reg_32 operation defines those of the operands. In LISA, a concise semantics description is achieved through using such a structure.

```
OPERATION ADD IN pipe.EX{
    DECLARE{ GROUP Rs1, Rs2, Rd = { reg_32 }; }
    ...
    SEMANTICS{ _ADD|_C, _Z|(Rs1, Rs2)->Rd; }
}
```

Figure 2. LISA Operation with Semantics

The semantics view of LISA 2.0 ignores all structural details like pipelining, and provides a clean and unambiguous instruction behavior description. This can be used for several purposes. In [19], we have presented a technique for *synthesizing* instruction-set simulators from the instruction semantics. Since the semantics description is much simpler than the C/C++ description, this helps accelerating the modelling process in early architecture exploration when the concrete micro-architecture is not fully determined. Another important use of the semantics is compiler generation, especially code selector generation.

5. Code-selector description generation

The code selector generator in CoSy uses the dynamic programming tree matching algorithm [20]. It requires a *tree grammar* description of the target instruction set. The same approach is used in other code selector generators, e.g. iburg [7]. A tree grammar $G = (N, T, P, S)$ consists of finite sets N and T of *nonterminal* and *terminal* symbols, respectively, as well as a set P of *mapping rules* and a *start symbol* $S \in N$ [7]. The terminals T essentially describe the operations of the source language (e.g. C) and thus are target machine independent. Likewise, the start symbol S requires no special retargeting effort. Only the nonterminals, N, and the rules, P, need to be adapted to the target machine. N basically reflects available registers, memories, and addressing modes, while P defines how source language operations are implemented by the target instructions. Each mapping rule in P has the form of a *tree pattern* that may serve to cover a data-flow graph fragment during code selection. In the following sections, we describe how N and P are automatically generated from instruction semantics information.

5.1. Nonterminal generation

In tree grammar descriptions, nonterminals act as temporary variables connecting different grammar rules. For code selectors, they represent temporary storage locations like registers and memories that the compiler can use. Depending on the type of locations, nonterminals can be divided into four categories: *register nonterminals* representing the compiler usable registers, *immediate nonterminals* carrying the constant values that can be directly encoded in instruction codings, *addressing mode nonterminals* encapsulating the addressing modes, and *condition nonterminals* which are normally flag registers that are affected by different instructions.

In LISA 2.0 processor models, accesses to these storage location or processor resources are normally described in a wrapper operation, like the operation reg_32 in fig. 3. In the instruction semantics description a set of micro-operators are provided to capture the semantics of the wrappers. The _REGI operator in the example stands for a register access. Its operand GPR is the name of the LISA resource that is used as register bank. The index of the accessed register is given by index, a LISA label whose value is determined by the instruction coding. Another important information, the bit-width of the registers, is specified with the notation <0,32> which means the register is 32 bit wide and the least significant bit is bit 0. Given this operation, a register nonterminal is generated, which can hold any type of data when its size fits.

```
OPERATION reg_32{
    DECLARE{ LABEL index; }
    CODING { value=0bx[4] }
    SEMANTICS { _REGI(GPR[index])<0,32>; }
    ...
}
```

Figure 3. Register Wrapper Operation

Similar to the register nonterminals, the generation of the immediate and the addressing mode nonterminals is based on two related micro-operators, _IMMI and _INDIR. Once these micro-operators are detected, the generator will create the corresponding nonterminals. Generally, the condition nonterminals represent the flag registers; their existence then depends on the use of four predefined flags, carry (_C), zero (_Z), overflow (_O), and negative (_N) flags. For example, the semantics statement in fig. 2 has the side effects of writing the carry and zero flag. Accordingly, a condition nonterminal is generated. These four kinds of nonterminals are processor specific elements in the mapping rules. They are created before the generation of the mapping rules.

5.2. Mapping Rule Generation

In general a mapping rule (a.k.a. tree rewriting rule) consists of three parts: a tree pattern, the result of the rule which is normally a nonterminal, and one or several associated

IEEE COMPUTER SOCIETY

machine instructions. The tree pattern represents a C level computation which can be performed by the processor. The input operand(s) of the computation is(are) also nonterminals. To generate mapping rules for a working code selector description, two questions need to be answered. The first one is, what tree patterns are needed to cover the complete set of C level operations for the target processor, and the second is, how the C level operations are mapped to the machine instructions.

5.2.1. Basic Rules A complete code selector description must cover all the operations that the C language might need. Since the C language does not change, the C level operations needed to be covered by a code selector are actually fixed. This makes it possible that a set of mapping rule templates can be prepared without the knowledge of a target processor. We created a set of such templates, and call them *basic rules*. Table 1 lists a basic rule and a CoSy mapping rule side by side. The `mirPlus` operator in both rules is a C level addition operation defined in CoSy. There are two major differences between basic rules and CoSy mapping rules. First, the operands in the tree patterns of the basic rules are placeholders, like `a`, `b`, and `c`, instead of nonterminals `nt_reg` used in CoSy rules. From a basic rule, it is very easy to create tree patterns fitting the target processor by replacing the placeholders with nonterminals generated in the previous step. In the generator, the basic rules needed by a complete coverage of C operations are put together in a so called *basic library*. It is used by the code selector generation for all target architectures.

Basic Rule	CoSy Mapping Rule

COSYIR mirPlus(a,b)→c;
PATTERN {_ADD(a,b)→c;}
	RULE o:mirPlus(a:nt_reg,b:nt_reg)→c:nt_reg;
	EMIT {
	fprintf(outfile, "add %s=%s, %s", c, a, b);
	}

Table 1. Basic Rule and CoSy Rule

The second major difference between basic rules and CoSy rules is that a CoSy rule has assembly instructions associated, e.g. the `fprintf` function prints an assembly instruction to the output file, while a basic rule has one or several semantics statements. In the generation procedure, when the suitable tree patterns are available, the next task is to find instructions that can perform the C operations in the tree pattern. This requires the semantics linked with the basic rules. The generator searches the instruction semantics described in the LISA model and selects the suitable instruction based on the result of the matching of the semantics.

In principle, most of the C operations can be covered by single instructions, which is the simplest case. We call this *one-to-one* mapping: one C operation maps to one machine instruction. However, ASIP designers always try to make the architecture as simple as possible, only if the application can be executed efficiently. Rarely used instructions might be left out of the design, but they are needed by a complete C compiler. *One-to-many* mapping is then used, which implements a C-level operation with a sequence of instructions. Moreover, the ASIP designers not only simplify the instruction set but also add instructions for program hot spots. These instructions accelerate the program execution by performing many C-level operations at once. To utilize them in a compiler, *many-to-one* mapping rules are needed. The following sections describe how the instructions are selected for these three kinds of mapping rules with the help of instruction semantics information.

5.2.2. One-to-one mapping As the simplest mapping method of all, one-to-one mapping is first used by the generator to find suitable instructions. The semantics statements of the basic rules are compared with the instruction semantics in the LISA model. Since some side effects in a real instruction might not be interesting to a C compiler, a successful one-to-one mapping does not require two totally identical semantics patterns. For example, suppose the generator tries to find a suitable instruction for the basic rule in table 1, and the semantics of the LISA operation depicted in fig. 2 is examined. Because the writing to carry and zero flag does not influence the result of an arithmetic addition in C level, the side effects in the instruction semantics are then ignored by the generator, and the instruction is chosen for the tree pattern generated from the basic rule. Eventually, a complete one-to-one mapping rule, in the form of the CoSy rule in table 1, is created.

Such adaptation in the one-to-one mapping can only compromise those effects not affecting the calculation results. Micro-operators still must be exactly the same for both compared semantics. If, due to the simplification made by the designers, a processor does not have the instruction to provide a one-to-one mapping, one-to-many mapping is then used.

5.2.3. One-to-many mapping In one-to-one mapping, the semantics statements in the basic rules are used by the generator to find a suitable instruction. However, if a processor does not have one instruction which can complete the calculation or operation, the generator needs to find a sequence of instructions, i.e. one-to-many mapping. For this purpose, it is important for the generator to know how the semantics statement in a basic rule can be implemented through a set of other semantics statements without affecting the result of computation. This is achieved by using the *semantics transformations*.

A *semantics transformation* specifies a sequence of semantics statements which together performs the same computation carried out by the original statement. An example can be found in fig. 4. The `_NEG` micro-operator represents a two's complement negation. The specified transformation provides a mathematically equivalent solution to do the negation operation. `_NOT` is the one's complement

IEEE COMPUTER SOCIETY

micro-operator. A two's complement can be done by doing a one's complement and plus one. With this transformation available, if the generator fails to find an instruction doing negation, it will then look for two instructions to do the complement and the addition operations sequentially.

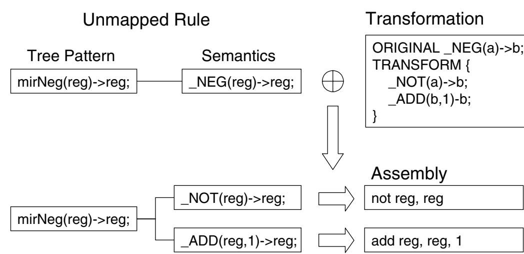

Figure 4. One-to-many Mapping

In principle this approach can be used to lower any operation, only if an equivalent transformation exists and can be expressed in the form of semantics statements. In the LISA compiler generator, a set of commonly used transformations are provided by default in a so called *transformation library*. Because of the variance of different instructions implemented in various architectures, it is not possible to specify a transformation library that can fit every ASIP under design. We provide the transformation library in form of a text file, so that if a C operation needs to be implemented with a very specific sequence of instructions the designer can create a custom transformation himself.

5.2.4. Many-to-one mapping Many-to-one mapping is especially important for ASIP design, because ASIPs heavily employ instructions that perform composite operations to accelerate application execution. However, since the designers can implement arbitrary combinations of operations in one instruction, it is difficult for the generator to prepare the tree patterns without knowing what the instructions do. Our approach is analyzing the instruction semantics in the LISA model and try to create a tree pattern with several C level operations out of it.

Take the `MAC` (Multiply and Accumulate) instruction, which is a commonly used composite operation, as an example. Suppose the ASIP designer describes its semantics with the semantics statement in fig. 5. Two micro-operators are used, `_ADD` and `_MULII`. `_MULII` is a signed integer multiplication. The generator has the knowledge of the mapping between the semantics micro-operators and the CoSy tree pattern nodes. With this knowledge, it can then create a corresponding tree pattern from the instruction semantics without user interaction. In the example, `mirPlus` is the CoSy tree pattern node corresponding to the micro-operator `_ADD`, and `mirMult` maps to the `_MULII` operator. If the source code contains a concatenated multiply and addition operation, this many-to-one mapping rule will tell the code selector to use the `MAC` instruction instead of two separate multiply and addition instructions.

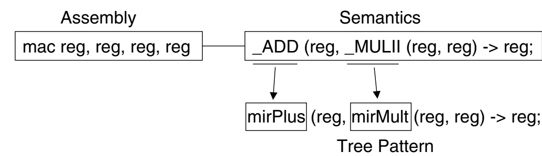

Figure 5. Many-to-one Mapping

5.2.5. Semantics Intrinsics Generally, the many-to-one mapping works for the arithmetic instructions whose semantics can be described with a chain of micro-operations like in the example 5. However, it is not applicable to those instructions performing multiple data assignments, e.g. SIMD (Single Instruction Multiple Data) instructions. Since the tree matching algorithm covers one sub-tree at a time, it is not possible to do SIMD instruction mapping without changing the internal intermediate representation of the source code. Another approach to utilize such instructions is using compiler known functions (a.k.a compiler intrinsics) which are directly translated into corresponding instructions by a compiler. For this, we use *semantics intrinsics* to describe those instructions who have complex semantics that cannot be used by tree matchers, like SIMD instructions. In LISA 2.0 models, the semantics intrinsics are used as customer micro-operators, e.g. the `"_DADD"` in the example 6. For the generator, the real semantics of the intrinsics is no more important, since creating corresponding function definition is a straightforward one-to-one translation.

Figure 6. Semantics Intrinsics

6. Experimental results

To examine the efficiency of our compiler generator, we have done case studies using the ST220 and PP32 processors as target architectures, the same as in our previous work [15]. Since the instruction semantics description of LISA 2.0 is quite straight forward, it is very easy for a LISA model developer to extend a model with semantics. For our two driver architectures, on average, it took 2 man weeks to develop the semantics models [18]. Based on them, most of the mapping rules needed by a C compiler are automatically generated. The CPU time used by the generator is negligible. However, both processors need a few custom transformations to cover some C operations. Table 2 provides the statistics of the generated rules for both processors and the number of custom transformations.

The custom transformations of both processors are mainly used to cover operations which cannot be executed with one machine instruction such as the signed/unsigned division and the modulo operation. Several standard library functions are invoked by the transformations to accomplish such operations. For ST220, a transforma-

IEEE COMPUTER SOCIETY

tion is used to perform the one's complement operation with a ST220 specific instruction that does bitwise *not* and *or* (NOR) at once. PP32 also has some very specific transformations. For example, the load of a 32 bit immediate value is done with two instructions. The first one loads the higher half of the value into the target register and left shift the result by 16 bits. The second one adds the lower 16 bits to the target register.

	one-to-one	one-to-many	many-to-one	user trans.
ST220	176	13	5	9
PP32	71	19	0	12

Table 2. Rule Statistics of ST220 & PP32

The generated compilers have been verified using the Supertest compiler validation test suite of the CoSy system. The code quality is examined against our previous work in [15]. The results are pretty close. The semantics generated compilers have an average of 5% cycle count overhead, and the code size is about 18% larger. These overheads are mainly due to the fact that the basic rules used for the mapping rule generation are designed to be conservative, so that they can be used by all architectures. The manually crafted code selector has a more aggressive instruction selection policy, e.g. the integral promotion for some of the C arithmetic operators was omitted with the assumption that the values in the registers are always correctly sign or zero extended. In general, the code quality produced by the generated compiler still cannot compete with the handwritten highly optimized compiler. However, the major focus of this work is to push the automation of compiler generation to its edge.

7. Conclusions

In this paper, we have presented a novel approach to automatically generate C compilers from the LISA instruction semantics models. Though using a semantics description introduces certain redundancies, they are kept minimal and local in the model. Three different mapping rule generation methods were developed. With them, compiler designers only need to take care of architecture specific features of their processors. The generator will do most of the work automatically. Moreover, since the semantics actually describes the instruction behavior, it is much easier for the ASIP designers to use them. Less compiler knowledge is required to build a compiler using our approach. A user extensible library is employed in our generator, so that flexibility is kept. Two contemporary processors were tested, and both present promising results. However, the code quality of both compilers can only be considered as results from out-of-box compilers. To improve it, further code optimization techniques are needed. In future, we will focus our work on providing ASIP designers a retargetable interface to the compiler, so that they can develop specific optimization for ASIP code generation.

References

[1] Free Software Foundation GCC. *http://gcc.gnu.org*.

[2] C. Fraser and D. Hanson. *A Retargetable C Compiler : Design and Implementation*. Benjamin/Cummings Publishing Co., 1994.

[3] Coware Inc. *http://www.coware.com*.

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

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

[6] R. Leupers and P. Marwedel. Retargetable Generation of Code Selectors from HDL Processor Models. *European Design & Test Conference (ED & TC)*, 1997.

[7] C. Fraser, D. Hanson, and T. Proebsting. Engineering a Simple, Efficient Code Generator Generator. *ACM Letters on Programming Languages and Systems*, 1(3), 1992.

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

[9] J. V. Praet, D. Lanneer, G. Goossens, W. Geurts, and H. D. Man. A Graph Based Processor Model for Retargetable Code Generation. *European Design and Test Conference (ED & TC)*, 1996.

[10] W. Qin and S. Malik. Flexible and Formal Modeling of Microprocessors with Application to Retargetable Simulation. *Design, Automation, and Test in Europe (DATE)*, 2003.

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

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

[13] P. Viana, E. Barros, S. Rigo, R. J. Azevedo, and G. Araujo. Exploring Memory Hierarchy with ArchC. *15th Symposium on Computer Architecture and High Performance Computing (SBAC)*, 2003.

[14] AXYS Design Automation, Inc. *http://www.axys.de*.

[15] M. Hohenauer, O. Wahlen, K. Karuri, H. Scharwaechter, T. Kogel, R. Leupers, G. Ascheid, H. Meyr, G. Braun, and H. van Someren. A Methodology and Tool Suite for C Compiler Generation from ADL Processor Models. *Design Automation & Test in Europe (DATE)*, 2004.

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

[17] O. Wahlen, M. Hohenauer, R. Leupers, and H. Meyr. Instruction Scheduler Generation for Retargetable Compilation. *IEEE Design & Test of Computers*, 2003.

[18] J. Ceng, W. Sheng, M. Hohenauer, R. Leupers, G. Ascheid, H. Meyr, and G. Braun. Modeling Instruction Semantics in ADL Processor Descriptions for C Compiler Retargeting. *Int. Workshop on Systems, Architectures, Modeling, and Simulation (SAMOS)*, 2004.

[19] G. Braun, A. Nohl, W. Sheng, J. Ceng, M. Hohenauer, H. Scharwaechter, R. Leupers, and H. Meyr. A Novel Approach for Flexible and Consistent ADL-driven ASIP Design. *Design Automation Conference (DAC)*, 2004.

[20] A. V. Aho, M. Ganapathi, and S. W. K. Tjiang. Code generation using tree matching and dynamic pogramming. *ACM Trans. Program. Lang. Syst.*, 11(4):491–516, 1989.