Compiler Generation from Structural Architecture Descriptions *

Florian Brandner Dietmar Ebner Andreas Krall

Institut für Computersprachen
Technische Universität Wien
Argentinierstrasse 8/E185
1040 Vienna, Austria
{brandner,ebner,andi}@complang.tuwien.ac.at

ABSTRACT

With increasing complexity of modern embedded systems, the availability of highly optimizing compilers becomes more and more important. At the same time, application specific instruction-set processors (*ASIPs*) are used to fine-tune hardware platforms to the intended application, demanding the availability of retargetable components throughout the whole tool chain.

A very promising approach is to model the target architecture using a dedicated description language that is rich enough to generate hardware components and the required tool chain, *e.g.*, assembler, linker, simulator, and compiler.

In this work we present a new structural architecture description language (*ADL*) that is used to derive the architecture dependent components of a compiler backend — most notably an instruction selector based on tree pattern matching. We combine our backend with *gcc*, thereby opening up the way for a large number of readily available high level optimizations. Experimental results show that the automatically derived code generator is competitive in comparison to a handcrafted compiler backend.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors – Retargetable compilers; Code generation;

General Terms

Algorithms, Languages, Measurement, Performance

Keywords

ADL, Architecture Description, Retargetable Compiler

*This work is supported in part by ON DEMAND Microelectronics and the Christian Doppler Forschungsgesellschaft.

1. INTRODUCTION

In order to meet the requirements of modern embedded applications in terms of efficiency and performance, techniques such as hardware/software codesign, design space exploration, and application specific instruction-set processors (*ASIPs*) are more and more used to tailor hardware architectures to a particular application. At the same time, the complexity of those applications rises steadily, demanding the availability of highly optimizing compilers to exploit features of the target platform.

A very flexible and sustainable approach to accomplish retargetability of compiler backends is the use of architecture description languages (*ADLs*) in order to specify the target architecture. A single specification can be used to develop a variety of (semi-)automatically retargetable components such as compilers, assemblers, linker, simulators and hardware synthesis tools.

Such systems have gained much interest both in academia and industry in the recent past and can be roughly divided into two categories according to the way in which architectures are specified:

- *Structural Model* The target processor model is described by an (abstract) netlist. Datapaths and component structures are explicitly specified and transparent to the particular tools.
- *Behavioral Model* Processors are specified in terms of their instruction set from a programmer's point of view.

While behavioral models are in general well suited for retargetable compilers and simulators, automatic hardware synthesis becomes almost impossible. In addition, structural models reflect a component based approach, as it is used in hardware design, very well and have significant advantages in terms of reusability and compactness of the description.

We propose a new structural *ADL* based on *XML* that is suitable for both automatic tool chain retargeting and hardware synthesis. Our approach follows a component based paradigm that enables the reuse of existing modules and is both extendable and comprehensible. In this work we give an overview of our *ADL* and the design of an optimizing retargetable compiler backend for *DSP* architectures. In particular we describe how to derive the rule set for an instruction selector based on tree pattern matching, thereby bridging the gap between the structural model of our description and the behavioral model used in a compiler.

Our backend is loosely coupled with the architecture independent frontend of *gcc*, the *GNU* Compiler Collection [21], and allows us to reuse a large number of readily available high level optimizations. We present some characteristics of *ADL* descriptions for the *MIPS R2000* and the *CHILI*, a novel *DSP* architecture developed by ON DEMAND Microelectronics. Experimental results indicate that the code quality is competitive compared to a handcrafted instruction selector.

2. RELATED WORK

In [26], Pees et al. present *LISA* — an *ADL* that aims to simplify the specification of pipelined processors and is used to automatically generate interpreted and compiled instruction set simulators. In order to support the generation of compilers, Braun et al. added a semantic extension to the operations in *LISA* [4]. There, *micro-operations* are used to define each operator's meaning. The semantics of a micro-operation are kept in a separate library. In [5], Ceng et al. present a method to derive rules for a tree pattern matcher from micro-operations. The compiler uses *basic rules*, which are sets of machine-independent templates, that transform *IR* operators to micro-operations. For instructions that cannot be handled by these transformations, *e.g.*, *SIMD* instructions, compiler intrinsics are provided.

Expression, a language for design exploration of Systems-On-Chip is presented by Halambi et al. [15]. It supports the retargeting of a compiler and simulator, as well as the generation of *VHDL* models [23]. An architecture description consists of three major sections: the specification of the instruction set, a structural model, and tree patterns for the compiler. All major compiler components, such as the instruction selector, instruction scheduler and register allocator are (semi-)automatically retargeted.

The *AVIV* compiler [16] uses an *ISDL* machine description. In this compiler, the statements of a basic block are converted into split-node directed acyclic graphs. Within such a *DAG*, operation nodes are duplicated for each functional unit that is able to perform the operation. The compiler does operation grouping, functional assignment, a preliminary register allocation, and instruction scheduling at the same time based on several heuristics.

Qin et al. introduce the Mescal Architecture Description Language [27] (*MADL*). It supports the generation of instruction set simulators and can be used for register allocation and instruction scheduling modules of a compiler.

In *MIMOLA* [22], a processor is described by a netlist of hardware modules. Leupers et al. present in [19] a *BDD*-based technique to extract an instruction set from a *MIMOLA* processor model. The RECORD compiler [20] uses these instructions in its code selection algorithm by generating a tree grammar from the extracted register transfer (*RT*) templates. For hardware entities that can store values (e.g. registers), non-terminals are created, while processor ports, hardware operators, and hardwired constants are mapped to terminals of the grammar. Each *RT* template is converted into a rule using the corresponding terminals and nonterminals. They use `iburg` [10] to generate an instruction selector from the grammar.

The Trimaran compiler uses the *MDes* [14] machine description language to automatically customize the register allocator and instruction scheduler. In particular the resource tables used during scheduling are derived from *MDes* specifications. A simulator may be derived for variants of the `HPL-PD` architecture only, retargetability of the simulator is thus limited.

Fauth et al. developed the architecture description language *nML*; see [8]. Two compilers have been built around *nML*. The *CHESS* code generation environment [17] targets mainly fixed-point digital signal processors. The *nML* model is translated into an instruction set graph that is processed using a special *bundling* algorithm for code selection.

The *CBC* compiler [7] also retargets itself using a *nML* description. A phase called *macro expansion* transforms the *IR* into operations of the target processor. The authors apply a heuristic node duplication technique to extend their method to graphs and use the `iburg` matcher generator.

Seng et al. propose *PD-XML* [29], a *XML* based hardware description language. They have separate descriptions for the instruction set architecture (*ISA*) and the microarchitecture and demonstrate how their language might interface with existing hardware description languages.

Azevedo et al. [2] developed *ArchC*, an architecture description language for piplined processors based on *System C*. It is possible to derive *System C* models, simulators and assemblers from *ArchC* specifications. Descriptions consist of an architecture model, defining the instruction set, registers, memories, and pipeline. The behavior of instructions is described separately using *System C* functions.

Recently, Farfeleder et al. [6] proposed an *ADL* that is used to derive an instruction selector for the *xDSPcore* — a five-way variable-length *VLIW* architecture. Contrary to our approach, their descriptions contain tool specific specifications, *e.g.*, rules for the tree pattern matcher generator, that are — to some extent — recombined automatically.

The Xtensa system from Tensilica simplifies the addition of extensions to Tensilicas core processor [31, 12]. Extensions are described using a specialized language, and the processor and tool chain are automatically generated. However, it is not possible to modify the core architecture limiting the possibilities for design exploration.

3. CONTRIBUTION AND STRUCTURE

In this paper, we present a fully retargetable compiler backend based on a newly developed structural *ADL*. The language is designed with a strong focus on extensibility, compactness, and comprehensibility. In contrast to previous approaches at our group [6], specifications for the particular components are derived automatically from a simple and compact semantic description, thereby shifting complexity from the architecture description to the generator tools.

The language is expressive enough to describe practical architectures while remaining conceptually simple in order to be processable by automatic tools, thereby opening the door for the generation of further parts of the tool chain, hardware synthesis, and hardware verification.

We describe in detail how to derive the architecture dependent parts of a compiler backend — most notably an optimizing instruction selector — and present experimental results, comparing them to two handcrafted compiler backends.

The rest of this paper is organized as follows. Section 4 gives a brief overview of our compiler backend and its architecture dependent components. In Section 5 we present the main concepts of our new *ADL* and give an overview of an example specification. We show in Section 6 how to

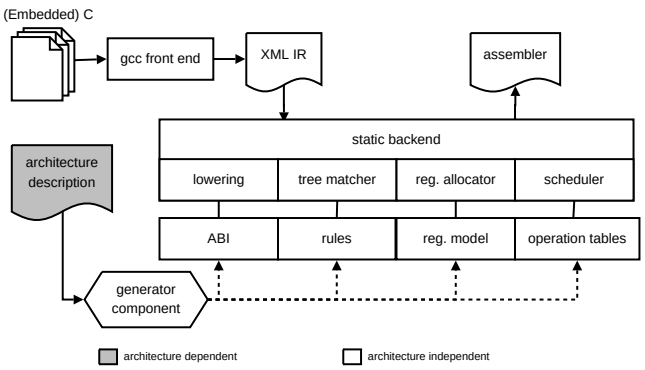

Figure 1: Overview of the main components of our backend. Dashed lines represent compiler compile time input that is an inherent component of the final executable.

```
<!-- define register type R_t -->
<RegisterType name="R_t" width="32"
              repeatcount="8">
 <Port name="Rs" writeable="false" />
 <Port name="Rt" writeable="false" />
 <Port name="Rd" readable="false" />
 <Port name="Rd_hi" offset="16"
       width="16" />

 <Constant index="0" value="0" />
</RegisterType>
\<!-- define a concrete register file-->
<Register name="R" type="R_t"
          category="integer␣base␣index" />
```

Listing 1: Sample declaration of a register file consisting of eight 32 bit registers. Port *Rd_hi* accesses only the upper halfword and register 0 is a hardwired constant.

derive a tree pattern based instruction selector from a given specification and present experimental results in Section 7. Concluding remarks can be found in Section 8.

4. COMPILER OVERVIEW

The goal of our backend is to serve as a testbed for the development of new optimization techniques and designs for embedded systems. The abstraction of the target machine provided by the architecture description allows us to easily compare the effect of different optimizations and modifications for a large number of architectures and variants.

Figure 1 shows the main components of our backend and their interaction. A modified version of *gcc* 4.1 is used to export the internal intermediate representation (*IR*) of the source program to a *XML* file, which is imported into our own high level intermediate language. High level constructs, such as array references, function calls and variable declarations, are converted into an architecture independent low level form. In addition, an *ABI* interface allows to rewrite function calls and global symbols. Code generation is done using a modified version of *lburg* [11, 10], a tree pattern matcher generator. Virtual registers are replaced with hard registers using an extended Briggs style global *graph coloring register allocator*; see [28]. Both before and after register allocation, a *list scheduler* is invoked that reorders machine instructions to heuristically minimize the number of stalls. In order to detect structural and data hazards, we make use of the operation table approach [30, 24].

A separate generator tool, that is invoked solely during compiler compile time, generates a register file model, operation tables for instruction scheduling, and a set of rules for our instruction selector. All these architecture dependent compiler components are derived from a single architecture specification; see Section 6.

5. ARCHITECTURE DESCRIPTION

Most *ADL*s found in literature were designed with a specific application in mind and later extended or slightly modified in order to enrich them with additional information.

Languages initially intended for simulator generation typically provide a detailed model of the architecture using a subset of C/C++. It is hard to extract a behavioral model from these descriptions, that is suitable for other task, *e.g.*, compiler generation. Usually this problem is solved by including a specification for both the simulator and the compiler. However, it is difficult to guarantee consistency between these two models and reusing them for different applications is almost impossible. LISA [26, 5] and previous work at our group [6] are examples of this approach.

We propose a new *ADL* that facilitates the reuse of architecture specifications for various purposes and easy integration with other software systems, *e.g.*, compiler or simulator frameworks. Based on a structural description of the architecture, we are able to derive a behavioral model of the instruction set; both models may be used for further processing.

In our *ADL* architecture specifications consist of a set of *components* interconnected by *data links*. Components may either correspond to hardware, *e.g.*, registers, caches, memories and functional units, or may represent abstractions, such as immediates (values embedded into the instruction word) and constants. Hardware components are associated with *ports* that allow data links to be connected, while immediates and constants are directly connected to the netlist. All components are instantiations of a particular *type* – a kind of template used to create multiple identical components. Types facilitate the reuse and exchange of components across different architecture descriptions.

In addition to the hardware model, our *ADL* allows to express assembler syntax, binary encoding, and parts of the application binary interface (ABI).

5.1 Register Files and Memories

A *register type* defines the number and width of registers, as well as ports of a register file. Ports might be read-only or write-only respectively and can be restricted to a contiguous range of bits within a particular register, *e.g.*, a particular halfword or a single bit. Some of the registers within a register type can be defined to represent hardwired constants. Listing 1 shows the declaration of a register type and subsequent instantiation of a register.

A *memory type* defines basic properties such as size, latency and ports. In addition to the data width, the ports of memories and caches also define the particular address width. *Cache types* specify the particular cache organiza-

```
<UnitType name="EX_t">
 <Input name="RS" width="32" />
 <Input name="RT" width="32" />
 <Output name="RD" width="32" />

 <Operation name="addi" syntax="op3_s" >
  <Syntax syntax="op3_s" token="mnemonic"
          value="addi" />
  <Predicate name="addr" />
  <Body>
    <add a="RS" b="RT" d="RD" />
  </Body>
 </Operation>
 ...
</UnitType>
```

Listing 2: An excerpt of the specification of the execute unit type.

```
<Unit name="EX" type="EX_t" >
  <Input input="RS" select="DE.RS_o"
         stageboundary="true" />
  <Input input="RT" select="DE.RT_o"
         stageboundary="true" />

  <Output output="RD" select="Memory.@read" />
</Unit>
```

Listing 3: Instantiation of the execute unit type, and connections to other components.

tion which has to be known to the consuming tools. Each cache is connected to a particular memory at the point of instantiation.

In order to support input values that are encoded within the instruction word, one can define *immediates*. They consist of a length specification and can be connected freely to input ports of other components.

5.2 Units

The central part in an architecture specification are *unit types* and instances thereof. Unit types may contain a complete network of sub-components (including units) or represent semantic information using *operations*.

Operations contain a sequence of *micro-operations*, that define the effect of an operation on the units output ports and local registers. Micro-operations may read from local registers, temporaries or the units input ports, and may write to local registers, temporaries and output ports. Temporaries are volatile local storage, *i.e.*, the value is not preserved across different *invocations* of an operation. Table 2 lists some predefined micro-operations provided by the *ADL*. While it is possible to define additional micro-operations, the existing built-ins are sufficient for many embedded architectures.

Listing 2 shows a simplified unit type of the *MIPS R2000* execute unit. It specifies two 32 bit input ports, `RS` and `RT`, and a 32 bit output port `RD`. The values supplied by the two input ports are accumulated and written to the output port using the `add` micro-operation. Besides the behavioral model, also a reference to a *syntax template* and the definition of a *predicate* is shown. Syntax templates describe the assembler syntax of instructions using verbatim text intermixed with *syntax variables*. In this example the mnemonic

Move	move cmove
Conversion	sext zext trunc
Comparison	ceq cneq clt cltu cle cleu cgt cgtu cge cgeu
Logic	and or xor not
Shift	rol ror shl shr ashr
Arithmetic	abs add sub divrem divremu mult multu

Figure 2: Built-in *micro-operations* for integer arithmetic.

is modeled using a syntax variable that is bound to the value `addi` for this particular operation. The predicate `addr` indicates that the values produced by this operation may be used for address calculations by subsequent units/operations; a detailed description of predicates is given in the following section.

The instantiation of the execute unit from a unit type is shown in Listing 3. The two input ports defined by the execute unit type are connected to output ports `RS_o` and `RT_o` of the decode unit `DE`. The output port is connected to an instance of a data memory `Memory`.

5.3 Instruction Set

The instruction set is defined implicitly along paths through the netlist of the underlying architecture. Paths are automatically identified using a breadth first search starting at end points. An end point is a cache, memory or unit that either has no output ports, or has all output ports connected to registers only, *i.e.*, the execution of instructions is finished at the given component. Only data written to registers and/or memories during the instructions execution persists after this point.

The netlist is traversed in reverse order collecting components and data links until a register or immediate is encountered. The components found during this process are ordered according to the data dependencies implied by the data links. Note, that data link connections may be ambiguous, *i.e.*, several output ports may be connected to the same input port. In this case all possible combinations are enumerated and paths created accordingly. The behavioral model of instructions is created from operations defined along a given path.

Predicates and *conditions* restrict instructions to certain combinations of unit types, unit instances, and operations. A predicate is a symbol that is visible along a path, starting at its definition. Conditions verify the existence of a given predicate and discard invalid instructions.

The structure of the pipeline is automatically derived using data links marked as *stage boundaries*, which roughly corresponds to a pipeline register between the two ends of the link. Data forwarding between pipeline stages is described using special *forwarding links*, leading from a units output port to another units input port. The pipeline information is available to the instruction set representation, allowing detailed timing analysis.

Figure 3 shows an example netlist describing a simple pipelined architecture, loosely based on the *MIPS R2000* [25]. The example contains the register files `PC` and `R`, an immediate `Imm` and a data memory `Memory`. The register file `R` offers four ports, two read-only ports `Rs` and `Rt`, a write-only port `Rd`, and a port `Rd_hi`, that is both readable and

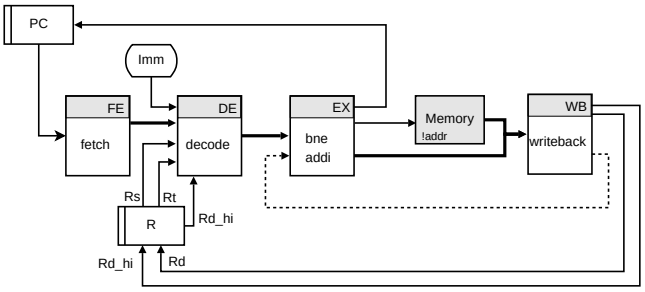

Figure 3: Example netlist for an architecture with four pipeline stages.

Micro-operations	Stage	Unit
`PC.p[0,0] = add(PC.p[0,0], 4)`	0	FE
`DE.Imm_o = sext(Imm)`	1	DE
`DE.Rs_o = move(R.Rs[0,31])`	1	DE
`EX.Rd_o = add(DE.Rs_o, DE.Imm_o)`	2	EX
`R.Rd[0,31] = move(EX.Rd_o)`	3	WB

Figure 4: Semantics of the `addi` instruction.

writeable. All but the last one access bits 0 through 31 of a given register, while `Rd_hi` accesses bits 16 through 31. The remaining components are functional units, each having at least one operation. Stage boundaries are represented using bold lines, forwarding links using dotted lines.

Branches, arithmetic computations, and address calculations are carried out by the execute unit `EX`. However, using the execute unit for different tasks may lead to undesirable effects. For example using the `bne` (branch on not equal) operation of the execute unit for address calculations is illegal, although certainly possible considering the hardware connections. This restriction is described using a condition `!addr` at the data memory. Only instructions that have the `addr` predicate defined are considered legal, all others are rejected. In this example only the `addi` operation may be used for address calculations, thus a corresponding predicate is added to the operations definition; see Listing 2.

Applied to the example netlist, the path finding algorithm yields three paths: `FE-DE-EX-Memory-WB`, `FE-DE-EX-Memory` and `FE-DE-EX-WB`. Accounting for conditions and predicates (`!addr`), four instructions are identified along these paths: `addi`, `load`, `store` and `bne`. Figure 4 shows the sequence of micro-operations of the `addi` instruction.

6. CODE SELECTION

Code Selection is one of the major phases of a compiler backend and aims to translate the internal *IR* of a compiler into machine instructions. One of the most popular approaches is *tree pattern matching* — a very fast and provably optimal (for single statements) method based on dynamic programming.

While we can derive most architecture dependent parts from our *ADL* without significant difficulty, the specification of an optimizing code selector requires some more effort. This section gives a very brief introduction into tree pattern matching and describes how to automatically derive a specification from a given architecture description.

6.1 Tree Pattern Matching

Tree pattern matching is a well known technique and goes back to Aho and Johnson [1], who were the first to propose a dynamic programming algorithm for the problem of code selection. Balachandra et al. present in [3] an important extension that reduces the algorithm to linear time by precomputing *itemsets*, *i.e.*, static lookup tables, at compiler compile time.

The same technique was applied by Fraser et al. in [11] in order to develop `burg` — a tool that converts a specification in form of a tree grammar into an optimized tree pattern matcher written in `C`. While `burg` computes costs at generator generation time and thus requires constant costs, `iburg` [10] can handle dynamic costs by shifting the dynamic programming algorithm to code selection time. This allows the use of dynamic properties for cost computations, *e.g.*, concrete values of immediates. However, the additional flexibility is traded for a small penalty in execution time.

Our code generator is based on a modified version of `lburg` – the tree pattern matcher generator available from the *LCC* ANSI C compiler [9].

6.2 Rule Generation

A tree grammar consists of finite sets N and T of *nonterminal* and *terminal symbols* and a set of *mapping rules* P. Each rule in P is a *tree pattern* with associated costs that covers a tree fragment of the *IR*. An optimal cover is a full cover of the expression tree with rules in P such that the sum of costs is minimized; see [10].

Terminal symbols T essentially describe the operations in the compiler's *IR*. Nonterminals usually reflect registers or particular modes and have to be generated from the architecture specification. Likewise, P specifies how a particular tree fragment is translated to machine instructions.

6.2.1 Deriving Nonterminals

Nonterminals are usually used as a kind of temporary "variable" in order to chain different rules together. We use nonterminals to represent immediates and registers defined by the architecture description.

The concrete data representation of immediates is not explicitly defined by the *ADL* but is given by the micro-operations using them. Thus immediates are mapped to a single nonterminal – `immediate` – that matches symbols and constants in the *IR*. Mapping rules verify that the involved data representations, *i.e.*, the bit width and signedness, of the *IR* and the rule pattern match.

Registers and their associated ports correspond to register classes during instruction selection. Instructions reading from a given register port must be supplied with virtual registers of the corresponding register class after instruction selection. For each register class a separate nonterminal is created, representing register class constraints on mapping rules.

The register file, depicted in the example netlist from Figure 3, offers four ports in total. Three of them access the same region of a register (bits 0 through 31), thus these three ports are summarized by one nonterminal called `RC_R`. The fourth port generates a second nonterminal, `RC_R_16_31`.

```
(1) RC_R_16_31:SHR(RC_R, 16)
(2) RC_R: BIT_INSERT(RC_R, RC_R_16_31, 16, 16)
```

Figure 5: Conversion rules derived from the example netlist.

6.2.2 Deriving Conversion Rules

Two register ports access possibly overlapping ranges of data bits of the same register. If the two ranges overlap, data written to one port may be read through the other. This is equivalent to a conversion between the corresponding register classes at no cost. Even if the two ranges do not overlap it is possible to convert between the two register classes, using a shift instruction to move data between the two ports.

Conversion rules for both cases are created automatically. However, the conversion between two register classes is only permitted if the data representation required by the *IR* can be preserved, *i.e.*, the bitwidth of the destination port is sufficient to hold the data type of the *IR*.

In our example netlist the `Rd_hi` port of the register file may be used to read the upperhalf of a register for free. In the *IR* this corresponds to a shift operation, as depicted by the first rule in Figure 5. Likewise, writing to the `Rd_hi` port matches a bit insertion operation of the *IR*, *i.e.*, 16 bits of the second operand are inserted into the first operand starting at position 16.

6.2.3 Deriving Rules

Deriving mapping rules from the instructions of an architecture is more complex, as instructions may have side effects. Currently, two classes of side effects are considered: memory accesses and control flow changes (*e.g.*, branches, exceptions).

In general, it is not safe to use instructions accessing memory for computations other than the memory access. For example a memory store with a pre-increment addressing mode, can not be used to increment a register. Thus only patterns matching `LOAD` and `STORE` operations of the *IR* are generated from these instructions. This is a limitation of the tree pattern matching approach, that is only applicable to expression trees. Instructions with memory side effects can be modeled using a *DAG*, but not using a tree. Usually a postprocessing pass, *e.g.*, a peephole optimizer, recombines these instructions to circumvent this limitation.

Control flow is modeled by (conditionally) defining the program counter (pc). Regular instructions are assumed to increment the pc by a constant value, other instructions, *e.g.*, branches, are analyzed and classified as shown in Figure 6. In addition to this classification the analysis also identifies call instructions that store the program counter to a register or memory. If an instruction is identified to be a branch or call, specific mapping rules are generated matching `GOTO` and `CALL` operations of the *IR*.

Considering instructions without side effects only, mapping rules are created by processing the list of micro-operations of each instruction. A rule is created for each assignment to a register port. Micro-operations supplying values to the assignment are added to the pattern of the rule and are directly mapped to operations of the *IR* using a lookup table. Values produced by preceding micro-operations are translated to patterns recursively until a `move` micro-operation is encountered reading the value of a register or immediate. Register operands are replaced by the nonterminals of the particular register class and immediates by the `immediate` nonterminal respectively.

reg. instructions	pc = pc + const.
abs. branch	pc = imm.
abs. branch	pc = reg.
cond. abs. branch	pc = (cond.) ? imm. : pc + const.
cond. abs. branch	pc = (cond.) ? reg. : pc + const.
rel. branch	pc = pc + imm
rel. branch	pc = pc + reg
cond. rel. branch	pc = pc + (cond. ? imm. : const.)
cond. rel. branch	pc = pc + (cond. ? reg. : const.)

Figure 6: Control flow characteristics of instructions.

Note, that tree patterns may only produce a single result. This restriction does not apply to the micro-operations in our *ADL*. To overcome this shortcoming, we duplicate rules for each result produced. The same situation arises with instructions having multiple results. Here we generate multiple independent rules and try to recombine redundant instructions using a postprocessing procedure after instruction selection is completed.

Along with the construction of rule patterns, conditions are created that have to be satisfied for a mapping rule to be applicable during instruction selection. These conditions cover the representation of data, such as the bit width, sign information and type of the *IR*. Finally, for each rule, costs are calculated from the pipeline model and an emit function is created that is used to derive machine instructions once a cost minimal cover is obtained by the dynamic programming algorithm.

6.2.4 Specializations and Templates

In order to derive a complete code selector, each operation of the *IR* has to be implemented by at least one rule. However, in general the instruction set will not directly match all the required operations in the *IR*.

One such case occurs when a particular operation in the *IR* is "simulated" by a more general instruction by hardwiring some of the inputs, *e.g.*, a `MOVE` operation can be easily implemented by adding zero to the source operand. Likewise, certain useful operations can be obtained by specifying the same input to its operands. We call a rule to be a *specialization*, if it is derived from an instruction by predefining some of its inputs.

Similarly, many *DSP* architectures implement only a subset of the operations in the *IR*. This reduces both complexity of the hardware layout and required code size. The missing operations have to be emulated by a series of available instructions.

Moreover, the code selector will in general produce more efficient code if it is able to exploit algebraic laws, *e.g.*, commutativity or associativity of certain operations. Many of those simple optimizations can be easily expressed in terms of tree patterns that can be handled by the dynamic programming algorithm.

Our rule generator is able to handle those cases using a set of built-in *templates*. Templates consist of a set of tree patterns and a combine function. The tree patterns specify,

```
(1) RC_R: PLUS(RC_R, SEXT(imm))
(2) RC_R: PLUS(SEXT(imm), RC_R)
(3) RC_R: PLUS(RC_R, imm)
(4) RC_R: PLUS(imm, RC_R)
(5) RC_R: SEXT(imm)
(6) RC_R: imm
(7) RC_R: RC_R
```

(a)

```
(1) stmt:  COND(NEQ(RC_R, RC_R), GOTO(imm))
(2) stmt:  COND(RC_R, GOTO(imm))
```

(b)

Figure 7: Patterns derived from the `addi` and `bne` instructions.

```
(1) %TmpL: SHL(%Value, SHORT_SIZE)
(2) %Result:  ASHR(%TmpL, SHORT_SIZE)
```

(a) Required tree patterns.

```
%Result:  SEXT(%Value)
```

(b) Resulting pattern

Figure 8: Template for the sign-extend operation.

in an architecture independent way, which instructions need to be available to simulate the desired operation. The combine function specifies how these instructions have to be emitted during instruction selection.

The generator repeatedly checks if a particular template is applicable, *i.e.*, all required tree patterns used in the template body are available, and generates a specialized rule by replacing nonterminal variables in the template with concrete nonterminals of the tree patterns used to simulate the desired operation. A template might use temporary registers but must not clobber registers or memory addresses that would have been left untouched by the original operation.

We use templates mainly to provide default implementations for "complex" operations of the *IR* using basic instructions that are expected to be available on each architecture. Those templates are kept in a separate module and can be easily extended if necessary.

For example, the semantics of a sign-extend operation can be simulated by a combination of shift instructions. Figure 8 depicts the required, as well as the resulting tree patterns. `SHORT_SIZE` represents the size of a *short int*, other names preceded by % are nonterminal variables, that are bound to concrete nonterminals. Note, that names occurring multiple times always have to represent the same nonterminal, otherwise the template is not applicable.

Figure 7 shows rule patterns created from the `addi` and `bne` instructions of our example netlist, assuming that one register is bound to the constant value zero. The first patterns are initially derived from the instruction, while the others are generated using specialization and algebraic simplifications. The `SEXT` of the `addi` instruction is removed for patterns 3, 4 and 6. Instead, a check is introduced that restricts the immediate operands to signed integers during instruction selection.

	Lines	Paths	Instr.	gen. Rules	Rules
MIPS	791	3	46	158	163
CHILI	968	5	674	145	150

Figure 11: Architecture description characteristics.

7. EXPERIMENTAL RESULTS

We have used the *ADL* proposed in Section 5 in order to create hardware descriptions for the integer instruction set of a *MIPS R2000* core and the *CHILI*, a novel *DSP* processors developed by ON DEMAND Microelectronics.

CHILI is a 4-way *VLIW* (Very Long Instruction Word) architecture specifically aimed for efficient (mobile) video processing. Each slot has access to a general purpose register file offering 64 32-bit registers. Loads and stores can be issued on each of the four slots and are executed out of order in the data memory subsystem. Branches can only be issued in the first slot and expose a large delay of five cycles. To compensate for the large delay all instructions, except memory loads and stores, can be executed conditionally. All conditional variants occupy two slots and need to be issued on even slots.

While the *MIPS R2000* specification has been developed as a sample specification in sync with the *ADL*, we could come up with a *CHILI* specification written from scratch within a couple of days. The *CHILI* model consists of 968 lines of *ADL* code and defines 674 instructions (including all conditional variants). The *MIPS R2000* model is considerably smaller, and consists of 791 lines of *ADL* code.

In comparison, a model of a *MIPS R4000* based architecture described using the *Expression* [15] language consists of 4183 lines. About 1200 lines are needed for the specification of the integer and floating point instruction set, the structural model of the architecture is described using about 600 lines. The biggest part of the description is made up by pattern specifications for the compiler (about 2800 lines).

A description of the integer instruction set of the *MIPS R3000* using the *ArchC* [2] language consists of 2632 lines. 478 lines specify the instruction set, including the syntax and encoding of instructions. The behavioral model consists of additional 2154 lines of System-C code.

So far, the backend derived solely from those specifications is able to compile medium sized integer benchmarks. While this is the first `C` compiler available for the *CHILI*, we are able to make comparisons with existing compilers for the *MIPS R2000* architecture. All experiments were executed using a cycle accurate simulator [2]. Benchmarks were taken from the *MiBench* [13] and *Mediabench* [18] suites, omitting those with floating point operations. `cmac`, `dct32`, `dct8x8`, `serpent`, and `twofish` are additional benchmarks supplied by our research partner. These benchmarks are medium sized, ranging from 800 to 4400 lines of code.

To evaluate the automatically derived instruction selector we compared the runtime and codesize of our benchmarks to *LCC*, a retargetable ANSI C compiler developed by Fraser et al. [9]. It does not offer any optimizations apart from the code generator and a heuristic register allocator. The most frequently used local variables are allocated to registers, while the less frequently used are kept on the stack. The instruction selector is generated from a handcrafted set of

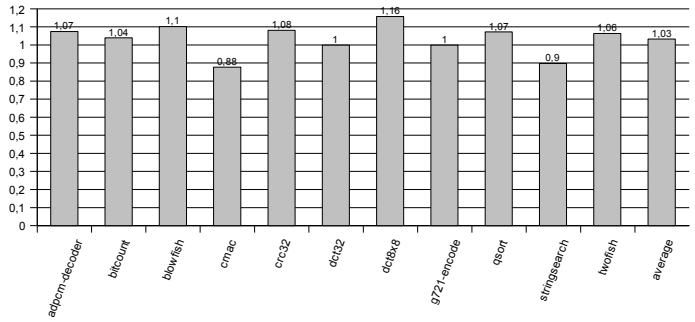

Figure 9: Performance improvement of the generated backend in comparison to *LCC*.

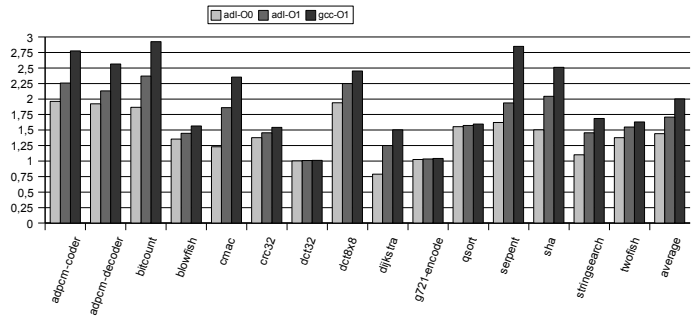

Figure 10: Speedup of the generated backend (`ADL`) and the traditional *gcc* compiler (`GCC`).

mapping rules using *lburg*. The rule set consists of 181 rules, with 1120 lines of rule specifications and `C` code.

All high level optimizations in the *gcc* frontend of our compiler were disabled for this comparison. In many cases the code generated by our instruction selector is faster than the code generated by *LCC*; see Figure 9. Only two benchmarks, `cmac`, and `stringsearch`, show a significantly decreased performance, due to useless jumps introduced by a bad ordering of basic blocks. The code size of the two compilers is almost identical, with less than 1% difference on all benchmarks. The results indicate that the rule set derived by our generator tool is competitive to the handcrafted rule set of *LCC*.

In addition, we compared the *ADL* based compiler to the original, highly optimized *gcc* backend. While we benefit from a number of high level loop transformations and optimizations, most of the traditional backend optimizations are implemented on *RTL* level and are thus not available in the `ADL` compiler.

Figure 10 shows the speedup of *gcc* and our automatically generated backend in comparison to the baseline compiler (`gcc -O0`) for two different optimization levels (`O0`, `O1`). The average improvement in comparison to the baseline compiler is about 39%. Even with optimizations enabled we still reach about 85% of *gcc*'s performance on average. Apart from the code generator, register allocator and instruction scheduler, currently no other optimizations are available in our backend. Thus we are lacking redundancy elimination and loop invariant code motion for address calculations, leading to a performance penalty. The high level optimizations available in *gcc* are not as effective as initially expected. Enabling additional optimizations often does not yield any improvements to the input of our backend, while optimizations at the *RTL* level of *gcc* show significant improvements to the code quality. On average the high level optimizations available to our compiler improve the runtime of the benchmarks by about 20%. The `cmac` and `stringsearch` benchmarks benefit the most, with an improvement of 55% and 37% respectively.

We expect to close the performance gap by adding some simple code transformations to our backend, *e.g.*, the elimination of redundant address computations. In addition, we expect improvements from future versions of *gcc*, as `RTL` based optimizations are gradually replaced with high level optimizations.

8. CONCLUSION AND FUTURE WORK

In this paper, we presented a fully retargetable compiler backend based on a structural *ADL* that is designed to be flexible enough for the generation of further parts of the tool chain, *i.e.*, assemblers, linker, optimized simulators, and also automatic tools for hardware synthesis and verification.

While it is straightforward to automatically derive the first group of tools, the latter group might be much more challenging. However, the prospective benefits for both hardware and software development are persuasive, which motivates us to continue work in those directions.

Computational results for a testbed description of a *MIPS R2000* core show that our automatically derived instruction selector is competitive in comparison to the handcrafted code generators of *LCC* and *gcc*.

9. REFERENCES

[1] Alfred V. Aho and Stephen C. Johnson. Optimal code generation for expression trees. *Journal of the ACM*, 23(3):488–501, July 1976.

[2] Rodolfo Azevedo, Sandro Rigo, Marcus Bartholomeu, Guido Araujo, Cristiano Araujo, and Edna Barros. The archc architecture description language and tools. *Int. J. Parallel Program.*, 33(5):453–484, 2005.

[3] A. Balachandran, D. M. Dhamdhere, and S. Biswas. Efficient retargetable code generation using bottom-up tree pattern matching. *Computer Languages*, 15(3):127–140, 1990.

[4] Gunnar Braun, Achim Nohl, Weihua Sheng, Jianjiang Ceng, Manuel Hohenauer, Hanno Scharwächter, Rainer Leupers, and Heinrich Meyr. A novel approach for flexible and consistent ADL-driven ASIP design. In *DAC '04: Proceedings of the 41st Design Automation Conference*, pages 717–722. ACM Press, June 2004.

[5] Jianjiang Ceng, Manuel Hohenauer, Rainer Leupers, Gerd Ascheid, Heinrich Meyr, and Gunnar Braun. C compiler retargeting based on instruction semantics models. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 1150–1155. IEEE Computer Society, March 2005.

[6] Stefan Farfeleder, Andreas Krall, Edwin Steiner, and Florian Brandner. Effective compiler generation by architecture description. In *LCTES '06: Proceedings of the 2006 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*. ACM Press, June 2006.

[7] Andreas Fauth, Günter Hommel, Carsten Müller, and Alois Knoll. Global code selection of directed acyclic graphs. In *CC '94: Proceedings of the 5th International Conference on Compiler Construction*, pages 128–142. Springer, April 1994.

[8] Andreas Fauth, Johan Van Praet, and Markus Freericks. Describing instruction set processors using nML. In *EDTC '95: Proceedings of the 1995 European Design and Test Conference*, pages 503–507. IEEE Computer Society, March 1995.

[9] C. W. Fraser and D. R. Hanson. A retargetable compiler for ANSI C. Technical Report CS–TR–303–91, Princeton, N.J., 1991.

[10] Christopher W. Fraser, David R. Hanson, and Todd A. Proebsting. Engineering a simple, efficient code-generator generator. *ACM Letters on Programming Languages and Systems*, 1(3):213–226, September 1992.

[11] Christopher W. Fraser, Robert R. Henry, and Todd A. Proebsting. BURG – fast optimal instruction selection and tree parsing. *ACM SIGPLAN Notices*, 27(4):68–76, April 1992.

[12] David Goodwin and Darin Petkov. Automatic generation of application specific processors. In *CASES '03: Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, pages 137–147, New York, NY, USA, 2003. ACM Press.

[13] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE 4th Annual Workshop on Workload Characterization*, December 2001.

[14] J. Gyllenhaal. A machine description language for compilation. Master's thesis, 1994.

[15] Ashok Halambi, Peter Grun, Vijay Ganesh, Asheesh Khare, Nikil Dutt, and Alex Nicolau. Expression: a language for architecture exploration through compiler/simulator retargetability. In *DATE '99: Proceedings of the conference on Design, automation and test in Europe*, page 100, New York, NY, USA, 1999. ACM Press.

[16] Silvina Hanono and Srinivas Devadas. Instruction selection, resource allocation, and scheduling in the AVIV retargetable code generator. In *DAC '98: Proceedings of the 35th Design Automation Conference*, pages 510–515. ACM Press, June 1998.

[17] Dirk Lanneer, Johan Van Praet, Augusli Kifli, Koen Schoofs, Werner Geurts, Filip Thoen, and Gert Goossens. CHESS: Retargetable code generation for embedded DSP processors. In Peter Marwedel and Gert Goossens, editors, *Code Generation for Embedded Processors*, pages 85–102. Kluwer Academic Publishers, 1995.

[18] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *International Symposium on Microarchitecture*, pages 330–335, 1997.

[19] Rainer Leupers and Peter Marwedel. A BDD-based frontend for retargetable compilers. In *EDTC '95: Proceedings of the 1995 European Design and Test Conference*, pages 239–243. IEEE Computer Society, March 1995.

[20] Rainer Leupers and Peter Marwedel. Retargetable generation of code selectors from HDL processor models. In *EDTC '97: Proceedings of the 1997 European Design and Test Conference*, pages 140–145. IEEE Computer Society, March 1997.

[21] The gnu compiler collection. `http://gcc.gnu.org/`.

[22] Peter Marwedel. The Mimola design system: Tools for the design of digital processors. In *DAC '84: Proceedings of the 21st Design Automation Conference*, pages 587–593. IEEE Press, June 1984.

[23] Prabhat Mishra, Arun Kejariwal, and Nikil Dutt. Synthesis-driven exploration of pipelined embedded processors. In *VLSID '04: Proceedings of the 17th International Conference on VLSI Design*, page 921, Washington, DC, USA, 2004. IEEE Computer Society.

[24] Sanghyun Park, Eugene Earlie, Aviral Shrivastava, Alex Nicolau, Nikil Dutt, and Yunheung Paek. Automatic generation of operation tables for fast exploration of bypasses in embedded processors. In *Proceedings of the Conference on Design, Automation and Test in Europe, DATE 2006, Munich, Germany, March 6-10, 2006*, pages 1197–1202. European Design and Automation Association, Leuven, Belgium, 2006.

[25] David A. Patterson and John L. Hennessy. *Computer Organization & Design: The Hardware/Software Interface*. Morgan Kaufmann, 1994.

[26] Stefan Pees, Andreas Hoffmann, and Heinrich Meyr. Retargeting of compiled simulators for digital signal processors using a machine description language. In *DATE '00: Proceedings of the conference on Design, Automation and Test in Europe*, pages 669–673. IEEE Computer Society, March 2000.

[27] Wei Qin, Subramanian Rajagopalan, and Sharad Malik. A formal concurrency model based architecture description language for synthesis of software development tools. In *LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 47–56. ACM Press, June 2004.

[28] Johan Runeson and Sven-Olof Nyström. Retargetable graph-coloring register allocation for irregular architectures. In Andreas Krall, editor, *Software and Compilers for Embedded Systems, 7th International Workshop, SCOPES 2003, Vienna, Austria, September 24-26, 2003, Proceedings*, volume 2826 of *Lecture Notes in Computer Science*, pages 240–254. Springer, 2003.

[29] S. P. Seng, K. V. Palem, R. M. Rabbah, W. F. Wong, W. Luk, and P.Y.K Cheung. PD-XML: extensible markup language for processor description. In *Field-Programmable Technology*, pages 437–440, 2002.

[30] Aviral Shrivastava, Eugene Earlie, Nikil D. Dutt, and Alexandru Nicolau. Operation tables for scheduling in the presence of incomplete bypassing. In *Proceedings of the 2nd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2004, Stockholm, Sweden, September 8-10, 2004*, pages 194–199. ACM, 2004.

[31] Albert Wang, Earl Killian, Dror Maydan, and Chris Rowen. Hardware/software instruction set configurability for system-on-chip processors. In *DAC '01: Proceedings of the 38th conference on Design automation*, pages 184–188, New York, NY, USA, 2001. ACM Press.