Effective Compiler Generation by Architecture Description *

Stefan Farfeleder Andreas Krall Edwin Steiner Florian Brandner

Institut für Computersprachen
Technische Universität Wien
{stefanf,andi,edwin,brandner}@complang.tuwien.ac.at

Abstract

Embedded systems have an extremely short time to market and therefore require easily retargetable compilers. Architecture description languages (ADLs) provide a single concise architecture specification for the generation of hardware, instruction set simulators and compilers. In this article, we present an ADL for compiler generation. From a specification, we can derive an optimized tree pattern matching instruction selector, a register allocator and an instruction scheduler. Compared to a hand-crafted back end, the generated compiler produces smaller and faster code.

The ADL is rich enough that other tools, such as assemblers, linkers, simulators and documentation, can all be obtained from a single specification.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Retargetable compilers, code generation, translator writing systems and compiler generators

General Terms Algorithms, Languages

Keywords compiler generation, architecture description language, code generation

1. Introduction

Current embedded applications which deliver high performance at low cost require application specific architectures. Because of the size of the applications, these architectures have to be programmed efficiently in a high level programming language. Highly optimizing compilers are necessary to exploit all the features of the processor. An extremely short time to market, typical of embedded systems, requires that easily retargetable compilers be available.

A good approach is to use an architecture description language (ADL) to specify the complete architecture. A single specification can be used for hardware synthesis and for generation of simulators, compilers and other tools. However, specification of the compiler's code generator is difficult. An instruction specification has different requirements from a code generator. A proven technique for the generation of code generators, and the method which we advocate, is tree pattern matching. And the patterns needed for code generation are not easy to extract from an architecture specification where the focus is on hardware synthesis or simulator generation.

In this paper, we advocate the inclusion of rules for a tree pattern matching code generator in the architecture specification. It is not necessary to include a full specification. Only the basic tree patterns have to be specified. All other information can be extracted from the other parts of the specification, and optimizations can be generated automatically. We have developed an ADL based on XML and a generator for highly optimized code generators. We implemented both a manually developed tree pattern matching code generator and a code generator specified in our ADL for the xDSPcore digital signal processing architecture [KHPP04]. The ADL specification is less error prone (the rule specification has been reduced to one fifth the size), and it produces up to 3 percent smaller and up to 9 percent faster machine code.

The xDSPcore is a five-way variable-length VLIW (very long instruction word) load/store digital signal processor (DSP) with pipelined inorder execution. Up to five instructions, either short or long, are executed in each cycle. It supports some common extensions for the DSP domain, such as SIMD (single instruction multiple data) instructions, multiply-accumulate instructions, various addressing modes for loads and stores with simultaneous update of the address register, multiple memory banks, fixed point arithmetic, predicated execution, etc. The processor's register file consists of two banks, one for data registers, the other for address registers. Each data register is 40 bits wide, but can also be used as a 32 bit register, or as two registers of 16 bit width.

The architecture description language details are explained in section 2. In section 3 we present our code generator generator and in section 4 we give an experimental evaluation of the resulting code generator. In section 5 we discuss related work.

2. The Architecture Description Language

The design of the architecture description language (ADL) was driven by the following principles:

- a concise instruction set definition,
- avoidance of duplicate information,
- grouping of related information,
- extensibility,
- human readability, and
- usability for automatic processing.

With these principles in mind, a language was developed which is used for defining the instruction set and generating software components and development tools. The initial language design targeted automatic generation of documentation and instruction set simulators and was later extended for compiler generation. Extending the language was straight forward and did not impact the al-

* This work is supported in part by Infineon Technologies Austria and the Christian Doppler Forschungsgesellschaft.

LCTES'06 June 14–16, 2006, Ottawa, Ontario, Canada

ready existing tools. Currently, generators exist to build a compiling simulator (`csim_impl`) [KFH05], an interpreting simulator (`xsim_impl`), compiler components (`rule_gen`), and all documentation from the ADL.

The syntax is based on the widely used markup language XML, which eases editing, validation and automatic processing. For example, a document type definition is provided that enables editors and parsers to validate the ADL document and to offer editing aids like syntax highlighting and word completion.

2.1 Resource definitions

An architecture specification is split into two parts: the configuration and the instruction set definition. The configuration contains an enumeration of available resources, such as pipeline stages, execution units, registers, memories, caches (limited at the moment) and buses which connect the resources.

A pipeline specification contains the names, the order and the width of each stage. Figure 1 shows the definition of an instruction fetch stage. The `<config>` element has two attributes: a unique name identifying the resource and a type specifying what is to be defined, in this case, a pipeline stage. Properties of the resources are defined by the `<data>` element. As can be seen in the example, the pipeline stages are ordered by the `pipeline_order` value. Two stages may not share the same position in the ordering, implying that it is currently not possible to define independent data paths with separate pipelines (e.g. one data path for integers and one for floating point instructions). The attribute `number_of_instructions` defines the number of instructions that may be executed in parallel in a stage.

```
<config type="PipelineStage" name="IF">
  <data name="phase" value="fetch"/>
  <data name="pipeline_order" value="1"/>
  <data name="number_of_instructions" value="5"/>
</config>
```

Figure 1. The fetch pipeline stage

Register sets are described by several register definitions, each containing basic information, such as bit width and 'endianness', as well as optional information, such as encoding and calling conventions. The ADL supports several register sets and banks that may share registers. It is also possible to define overlaps and other dependencies between registers. An example of a 32 bit wide data register set is shown in figure 2. The `<meta_data>` element defines optional attributes that are used by generators — in the figure the calling convention for the register allocator generator is given. The calling convention defines how registers are to be saved and restored upon function calls. It is possible to define a register to be an argument register, `saved` or `scratch`. Saved registers have to be saved and restored by the called function, while the caller has to take care of scratch registers. Argument registers are used to pass argument values to the called function. If an argument register is not used to pass a value, the `saved` calling convention is implied.

```
<config type="RegisterFile" name="Registers">
  <config type="Register" name="D0">
    <data name="width" value="32"/>
    <meta_data name="callingconvention"
        value="argument0"/>
  </config>
</config>
```

Figure 2. A simple register set

Memories are defined by size, cell size (size of an addressable data word), number of ports and the bus interface. It is possible to define banked memories using the most significant bits of the address to select the bank.

Currently only fully set associative caches are supported, they are defined analogously to memories. It is possible to specify the number of cache lines, the block size and the total size of the cache.

```
<config type="Memory" name="DataMemory">
  <data name="max_address" value="65535"/>
  <data name="mem_cell_size" value="16"/>
  <data name="mem_bus_size" value="32"/>
  <data name="number_of_memory_busses" value="2"/>
</config>
```

Figure 3. A memory definition

The configuration section also contains functional units that are solely used to model resource constraints. For example, figure 4 depicts a unit that may be used to implement integer arithmetic instructions. The resource constraints introduced here are used by the instruction scheduler. It is important to note that no functions or semantics for the units are specified here.

```
<config type="ExecutionUnit" name="ALU">
  <data name="order" value="1"/>
  <data name="instruction_type" value="INTALU"/>
</config>
```

Figure 4. Functional unit

2.2 Instruction set definition

The central part of an architecture description is the instruction specification. Each instruction of the target architecture is enumerated and its syntax, encoding, timing, resource constraints and semantics are described. To enable reuse, the description is decomposed into smaller pieces that are referenced within an instruction definition (see figure 5):

- The `<operands>` of an instruction are described via this section. Operands may be registers or immediate fields. It is important to note that the operands are derived from the instruction encoding (e.g. for register operands the index of the register is available from the instruction's encoding). Memory locations and registers that are addressed through a computed index are not considered to be operands.
- The `<opcode>` tag specifies the instruction's binary encoding. As can be seen, the encoding is built up using a string of digits and characters. The digits encode fixed values that do not depend on the instruction's operands, while the characters represent place holders that are replaced by concrete digits depending on the instruction's operands.
- The `<instruction_type>` is used to model resource constraints, as described for functional units. Instructions may only execute on units that support instructions of the given type.
- The `<execution_model>` references an abstraction of the execution phases of an instruction. It declares abstract operations that build up the instruction and assigns the operations to pipeline stages. These operations do not carry any functionality or semantics. The information defined here is used by the instruction scheduler for dependence analysis.
- The `<execution_cycle>` element maps the abstract operations defined in the execution model to concrete semantic specifications, the so called μ-instructions. In fact, the μ-instructions contain several specifications for different generator tools; the text snippets are written in tool specific languages, for example

```
<instruction name="add" >
  <opcode>10p00000aaaabbbbcccc</opcode>
  <instruction_type>INTALU</instruction_type>
  <execution_model>binary_model</execution_model>
  <operands>
    <operand char="a" order="1">
      <operand_type>DATA_REGISTER</operand_type>
    </operand>
    <operand char="b" order="2">
      <operand_type>DATA_REGISTER</operand_type>
    </operand>
    <operand char="c" order="3">
      <operand_type>DATA_REGISTER</operand_type>
    </operand>
  </operands>
  <execution_cycle>
    <map key="READ_OPERAND_1">
      <define_value>MACRO_READ_REGISTER_OPERAND_1(op1)
      </define_value>
    </map>
    <map key="READ_OPERAND_2">
      <define_value>MACRO_READ_REGISTER_OPERAND_2(op2)
      </define_value>
    </map>
    <map key="WRITE_OPERAND_3">
      <define_value>MACRO_ADD</define_value>
      <define_value>MACRO_WRITE_REGISTER(op3)
      </define_value>
    </map>
  </execution_cycle>
  <mnemonic>
    <define_value>add</define_value>
  </mnemonic>
  <syntax>
    <define_value>op1, op2, op3</define_value>
  </syntax>
</instruction>
```

Figure 5. example instruction specification

C++ or tree patterns. The user is responsible for writing correct macro specifications which are accepted by the tool generators. In the example the abstract operations for reading the operands are mapped to a register read (depicted in Figure 7), while the definition of the destination operand is mapped to an add operation and a register write (see Figure 8). The element `csim_impl` contains C++ code for the compiling simulator, while the `rule_gen` element contains rule patterns. Finally the `semantics` element is used for the instruction scheduler to define how operands are used by the instruction, for example if an operand is read, written, stored to memory, etc.

- The syntax of the instruction is defined by two tags, the `<mnemonic>` is a symbolic name for the instruction (e.g. add) that is placed at the beginning of the instruction's syntax. An instruction may have several mnemonics which select different variants of the instruction. In such a case, the `<mnemonic>` element contains a mapping which specifies the bits in the encoding to set for each variant. The rest of the syntax is given by `<syntax>`.

3. Compiler Generation

The compiler is based on the highly optimizing open compiler environment (OCE) from Atair. Nearly all the standard analyses and optimizations are available. The abstract syntax tree is augmented with static single assignment (SSA) information. Accurate alias and array dependence analyses provide the information for vectorization, scheduling and software pipelining. A classical graph color-

```
<define type="binary_model">
  <define type="READ_OPERAND_1">
    <value name="pipeline_stage">EX1</value>
    <value name="cycle_phase">begin</value>
  </define>
  <define type="WRITE_OPERAND_2">
    <value name="pipeline_stage">EX1</value>
    <value name="cycle_phase">end</value>
  </define>
</define>
```

Figure 6. execution model

```
<define type="MACRO_READ_REGISTER_OPERAND_1">
  <define_value name="semantics">
    use
  </define_value>
  <define_value name="csim_impl">
    %newtmp(operand1, $1)
    %tmp(operand1) = REG_READ_%reg_name($1)();
  </define_value>
  <define_value name="rule_gen">
    $op1 := $1
  </define_value>
</define>
```

Figure 7. A μ-instruction for reading a register operand

```
<define type="MACRO_ADD">
  <define_value name="csim_impl">
    %tmp(result) = %tmp(operand1) + %tmp(operand2);
  </define_value>
  <define_value name="rule_gen">
   $res := IrAdd($op1, $op2)
  </define_value>
</define>
```

Figure 8. A μ-instruction of an addition

ing register allocator and an optimal allocator based on partitioned boolean quadratic programming (PBQP) are included [HKS03]. Instruction selection is implemented by tree pattern matching. Using the OCE, a compiler for the xDSPCore [KHPP04] has been developed by writing the instruction selection rules manually.

The compiler is divided into a machine independent front end and a machine dependent back end. The front end does all the required analyses and machine independent optimizations. The back end does instruction selection, register allocation, if conversion, instruction combination for memory access instructions and instruction scheduling and grouping. Instruction selection takes an abstract syntax tree as input and transforms it into a list of machine instructions, assuming an unlimited number of virtual registers. Register allocation replaces the virtual registers by real registers. If the number of machine registers is not sufficient, some virtual registers are spilled to memory. Memory access instructions can be combined with address computations when the same register is used. If conversion transforms short basic blocks into guarded instructions to reduce the number of branches. Instruction scheduling reorders instructions and groups them into VLIW bundles. Finally, assembly language source code is emitted.

3.1 Tree Pattern Matching Instruction Selection

Tree pattern matching is a very powerful method for transforming the abstract syntax tree to machine operations in linear time. Tree pattern matching searches for an optimal covering (optimal in run time or code size) of the abstract syntax tree with instruction trees.

The tree pattern matcher included in the open compiler environment is an extension of lburg, which is more powerful than many other tree pattern matchers and makes sharing of information easy.

```
%MATCH  d = IrAdd(l, IrConstant)
%IF     { IrConstant->const >= 0 }
%COST   { 1 }
%OUT    { some lines of C++ for emitting code }
```

Figure 9. tree pattern rule

Fig. 9 describes a tree pattern rule for an addition with a constant. The `%MATCH` directive describes a subtree for an addition of a constant. The `%IF` directive allows the specification of conditions for rule selection. The `%COST` directive specifies the cost of selecting this rule, for example the latency of the instruction or the code size. The `%OUT` directive contains C++ code for entering the selected instruction in the machine instruction list. Writing these rules manually is time consuming and error prone.

3.2 Rule Generation

The aim of specifying the compiler by extending the ADL specification was to reduce the development effort of the compiler. A lot of the information necessary for the compiler is already contained in the specification for the instruction set simulator. The cost of an instruction can be computed from the information in an execution model (see fig. 6). The C++ code for generating a machine code instruction list and emitting assembly language source code is generated. Conditions like checks for the size of immediate operands in the instruction selection rules are generated from the operand size specifications. Additional conditions can be added. In the ADL it is no longer necessary to specify the `%COST` and `%OUT` directives (see fig. 8).

In the ADL generated compiler, only a subset of the specification is necessary. In the following, we describe how information is collected from the ADL specification, how a set of instruction selection rules is generated and how the rule set is optimized by exploiting algebraic laws.

The rule generator analyzes the instruction set to detect instructions with the same semantics. Instructions that have the same micro operations with the same timings are considered to be equivalent. Such a group of instructions is called a *semantic group*. The reason for instructions with the same semantics is that there might be several encodings with different sizes for immediate values, a short instruction for small numbers and a long version for larger ones. The rule generator creates rules for the most generic instruction in a semantic group. The code generator will choose the shortest instruction that fits in the last stage.

The rule generator iterates over the instruction elements in the instruction set that have not been deselected by the analysis described above. For each instruction, it inspects the micro operation list. The list of micro operations is traversed, and the rules for each one are looked up in the micro operation-set and parsed according to the grammar presented earlier. If a micro operation lacks the rules, the operation is skipped and no rules are generated.

The alternatives found in all rules are collected and all valid combinations are considered. Let m be the number of micro operations of the current instruction and A_i the set of rule-alternatives of micro operation i. Then an ordered sequence

$$\langle a_1, \ldots, a_m \rangle$$

where $a_i \in A_i$ is called a path. It follows that there are

$$\prod_{i=1}^{m} |A_i|$$

possible paths. Figure 10 shows a fictitious instruction with three micro operations. The first one has only a single alternative, the second one has three and the last micro operation has two alternatives. There are six possible paths, they are listed on the right side of the figure.

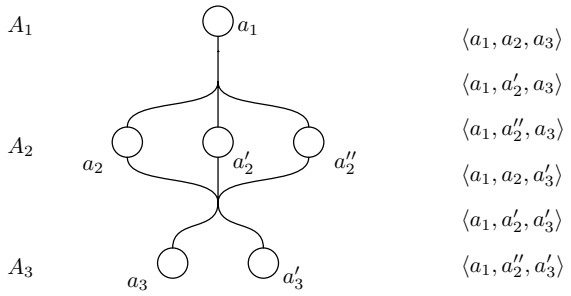

Figure 10. Multiple paths caused by several alternatives

In the next step, mode directives are evaluated on each path. Modes (see Fig. 12) are used to invalidate certain combinations of alternatives. The semantics of a mode is determined by the prefix of its identifier.

The paths that are invalidated by the above rules are discarded, they don't play any role in the further processing.

As an example, we consider a saturated addition instruction where we provide an example of usage and an attempt at a rationale for the mode directives. Figure 11 lists the micro operations used in a normal addition instruction (without saturation) on the left side and the associated data for the rule generator on the right side. The saturated addition instruction has an additional `SATURATION` micro operation after the `ADD`. This causes a problem because, at the level of the abstract syntax tree, the saturated addition is represented by a single operator `IrSatAdd`. At this point, the mode directives come into play. The solution is to offer both alternatives — the normal `IrAdd` and the saturated `IrSatAdd` — in the rules of the micro operation `ADD`. Then, with the help of mode directives, only one is allowed depending on whether `SATURATION` follows the addition or not. Figure 12 shows this idea. For a normal addition. only the first alternative will be chosen; saturated addition is handled by the second alternative.

`READ_OP_1(o1)`	`$op1 := $1`
`READ_OP_2(o2)`	`$op2 := $1`
`ADD`	`$res := IrAdd($op1, $op2)`
`WRITE_RES(o2)`	`$1 := $res`

Figure 11. micro operations for the add instruction and associated rules

`ADD`	`$res := IrAdd($op1, $op2)` `%MODE IFNOT_saturated` \| `$res := IrSatAdd($op1, $op2)` `%MODE IF_saturated`
`SATURATE`	`%MODE SET_saturated`

Figure 12. Extensions to allow saturated additions

The next step after the evaluation of the mode directives is to build a single tree by combining all the trees found in the rule alternatives. It is important to note that the operand nodes until now refer to the arguments of the micro operation their tree belongs to.

Thus the operand-node `$1` possibly represents different operands in different trees for the same instruction. Therefore, the operand nodes must now be associated with the instruction operands they point to. Following this step, all trees of the form

variable-node := *tree*

are considered. Such an assignment binds the tree t on the right-hand-side to the identifier v of the variable-node. For all other trees not having this structure, occurrences of a variable-node with the identifier v are replaced with the tree t until no variable-nodes are left.

This substitution process is illustrated with the add instruction that was already mentioned a few times and whose micro operations were listed in Figure 11. Figure 13 shows the steps of the substitution, beginning with the only tree that doesn't define a variable-node and substituting one variable-node with the associated tree in each row. Additionally, in the operand-nodes, $\$n$ was replaced with the real operands of the add instruction.

If these substitutions do not produce exactly one tree, no match-rules can be generated. Multiple trees mean that more than one result is generated. This situation occurs with instructions that write results to several registers, for instance, load and store instructions which automatically increment an address register at the same time. Such instructions are simply incompatible with tree-grammar code selection.

op_2 := $result
op_2 := IrAdd($oper1, $oper2)
op_2 := IrAdd(op_1, $oper2)
op_2 := IrAdd(op_1, op_2)

Figure 13. Tree substitution for the add instruction

One advantage of generating match rules automatically is that arbitrary transformations can be performed on the match trees. Currently the only transformation implemented is the exploitation of commutativity laws. The generator iterates over the tree and looks for subtrees with the form $a(b,c)$. If a is a commutative operator, an additional rule with swapped operands is generated. Rules covering the same patterns will be removed before being emitted.

The cost of a rule is determined automatically. Currently the cost is computed by the formula

$$cost = size + cycles$$

where $size$ is the number of memory cells the instruction's encoding occupies and $cycles$ is the number of cycles the instruction needs to compute the result. The number of cycles is computed by scanning the μ-instructions for the last write access to the register file. $size$ tries to minimise the code size, $cycles$ the critical path. The arguments of eventual cost-directives are appended to the generated cost.

3.3 Instructions with multiple results

Instructions with multiple results, such as memory access instructions with simultaneous update of the address register or divide instructions which generate both a quotient and a remainder, cannot be handled by tree pattern matching. A tree can only have one result. Furthermore, the suboperations of such instructions are in different expression trees. A separate pass is required where combinable instructions inside a basic block are discovered and then combined.

The instruction combiner constructs the data dependence graph of a basic block. Then it makes a forward scan over the graph for memory access operations and searches for an address computation operation which uses the same address register, and combines these two operations into one memory access instruction with post-address update. In a second backward scan, operations are combined into memory access instructions with pre-address update. Currently, we do not combine instructions across basic block boundaries and we do not employ program transformations to increase the number of combined instructions. Our greedy algorithm is so effective that we did not evaluate more costly algorithms which search for all possible combinations.

Currently the instruction combiner is hardcoded. We are developing a generator which emits the C++ code for the matching function from a concise matching table.

3.4 Register File Model

The register specification (see Fig. 2) is quite flexible. There is support for multiple register files with shared registers. While reading the architecture configuration, the compiler builds up an internal model of the register file. The model is basically a forest, where registers that are contained within other registers become child nodes of their containing parent nodes, and each independent hardware register becomes the root of a tree. The trees are augmented by edges indicating where sign- or zero-extension has to be performed when a child register is written. Parts of registers that cannot be accessed independently from their parent, like the guard bits of an accumulator, are represented by specially marked nodes. Figure 14 shows an example of a 40-bit accumulator containing a 32-bit long register that is itself divided into two 16-bit short registers.

The only restriction this model puts on the register specification is that, whenever two registers share bits, one of them must be completely contained in the other one.

The relations between symbolic registers created by the compiler must be isomorphic to subgraphs of the register model. Usually symbolic registers are isolated nodes, but there are cases with a more complex structure, for example when the low and high word of a register are loaded by separate machine instructions.

Calling conventions specify register usage for function calls. The register model is the input both for the graph coloring and the PBQP based register allocator and it is used in all calculations of data dependency.

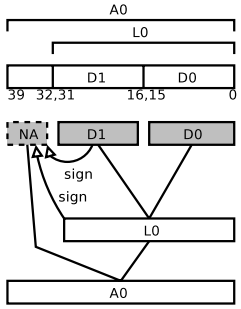

Figure 14. Register model of a 40-bit accumulator

3.5 USE/DEF Semantics

The instruction scheduler and VLIW grouper (based on list scheduling) and the software pipeliner (based on iterative module scheduling) depend on two aspects of the architecture description. On the one hand, information about definitions and uses of operands is needed for calculating exact data dependencies. On the other hand, hardware resources required by each instruction have to be considered.

For the purpose of determining data dependencies, the description of each μ-instruction contains a `semantics` value that specifies whether the μ-instruction defines or uses an operand. Figure 7 shows an example of a μ-instruction *using* an operand. Table 1 lists possible USE/DEF semantics of μ-operations. `use` and `def` have the obvious meaning for register operands. The other keywords are needed for assembling USE/DEF information for memory operands.

`use`	use as value
`def`	define value
`useaddr`	use as address
`defaddr`	define address
`absaddr`	immediate absolute address
`memoffset`	add immediate relative offset
`load`	memory load from the address in the operand
`store`	memory store to the address in the operand

Table 1. USE/DEF semantics of μ-operations

The compiler combines these semantic specifications with the execution model of the instruction to build a table which lists which operands are defined or used by the instruction at each pipeline stage. This information is used for calculating data dependence distances for scheduling and by the interference graph builder of the register allocator.

3.6 Resource Requirements

The resource requirements of each instruction are represented by a vector of integers with one component for each type of resource. A global vector gives the number of available resources for each type.

This simple model is based on the assumption that all instructions using a resource R require it in the same pipeline stage and for the duration of a single cycle. A natural extension would be to move the resource specification into the individual μ-instruction descriptions and have the compiler build a resource matrix for each instruction with one row vector for each pipeline stage. This would allow modeling instructions that use the same resource in different pipeline stages or for more than one cycle.

4. Empirical Evaluation

In the empirical evaluation, we compare the manually developed compiler for the xDSPcore with the generated one. Although much effort has been devoted to optimize the rule set for the manually developed compiler, some optimization opportunities have been missed.

Table 2 shows the number of handwritten rules before and after preprocessing and the sizes of them for the old and the new ADL matcher (comments were stripped before computing sizes). As already mentioned, preprocessing macros are used to decrease the number of rule copies for very similar rules. The number of rules in the ADL μ-instructions is down to almost a fifth of the original number.

	old	ADL
Number of handwritten rules	144	30
after preprocessing	169	53
Size (in kB)	98.3	19.3

Table 2. Comparison of handwritten rules

Table 3 shows statistical data about the generated rules. It lists the number of μ-instructions for which rule-data had to be written, the size of that data, the number of rules that are generated and the number of instructions for which rules are generated.

Number of μ-instructions with rule-data	66
Size of rule-data (in kB)	15.2
Number of generated rules	373
Number of instructions	94

Table 3. Statistics for the generated rules

The code selector was evaluated for code size and run time of the generated code. Table 4 compares the size of the generated code for a number of applications. Among them is the architecture's C library (clib), a few cryptographic programs and some typical DSP algorithms. Next to the column that lists the benchmark name, the number of instruction words of the code generated by the old handwritten and the new generated matcher are presented. The right-most column shows the comparison. For all benchmarks the machine code generated by the new matcher needs less instruction memory. The improvements range from one tenth of a percent up to more than three percent.

The second comparison concerns the run-time performance of the generated code. Table 5 compares the number of cycles needed for the execution of the same set of benchmarks. The C library is missing because it consists of a number of independent functions. The cycle numbers were obtained by running the generated code on a simulator. The improvements here are below one percent for all but four benchmarks. The best results are achieved in the cmac benchmark where the new code is almost ten percent faster.

The dct32 benchmark is the only one that shows a performance degradation. A manual inspection of the code revealed that, even though the selected code was more efficient, the differences led to performance losses in the compiler's register allocation and instruction scheduling passes which are executed after instruction selection.

Name	ADL Matcher	old Matcher	% Improvement
clib	10594	10739	1.369
adpcm	710	721	1.549
cmac	3740	3744	0.107
g721	1735	1747	0.692
ghs	2971	3001	1.010
dct32	1142	1157	1.313
dct8x8	933	945	1.286
viterbi	1029	1049	1.944
rijndael	3437	3545	3.142
serpent	4390	4453	1.435
twofish	2900	2934	1.172
aan dct	5741	5783	0.732
hadamard	5321	5343	0.413
hpme	5831	5856	0.429
pixel	6504	6541	0.569

Table 4. Code-size comparison

5. Related Work

5.1 Architecture Description Languages

Pees et al. developed the architecture description language LISA which simplifies the specification of pipelined processors and is used to automatically generate interpreted and compiled instruction set simulators [PHM00]. To support the generation of compilers, Braun et al. added a semantic extension to the operations in LISA [BNS+04]. There, *micro-operations* are used to define an operator's meaning. The semantics of a micro-operation are kept in a separate library. In [CHL+05], Ceng et al. show

Name	ADL Matcher	old Matcher	% Improvement
adpcm	619357	619367	0.002
cmac	1101426	1206827	9.570
g721	18127364	18147825	0.113
ghs	5974	6029	0.921
dct32	2783	2763	-0.719
dct8x8	65628	65886	0.393
viterbi	2899659	2904415	0.164
rijndael	184397	185382	0.534
serpent	1292231	1297372	0.398
twofish	2026935	2028915	0.098
aan dct	660752286	671556087	1.635
hadamard	203425511	203430073	0.002
hpme	52166566	52470332	0.582
pixel	572051	599409	4.782

Table 5. Run-time performance comparison

how rules for a tree pattern matcher are generated from the 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 (eg. SIMD instructions), compiler *intrinsics* are provided. Cengs *micro-operations* can easily be mapped to rule patterns similar to the patterns presented in this work. Our approach of using the patterns directly offers more power and flexibility. We are not bound to an abstract semantic, but may express any rule patterns.

EXPRESSION [HGG+99] is an architecture description language that takes a mixed approach between specification of structure and behavior. The EXPRESS compiler [HSDN01] uses the information from the EXPRESSION language for its configuration. For the front-end, the GNU Compiler Collection is utilized. The compiler features *resource directed loop pipelining* and *trailblazing percolation scheduling*. The *transmutation framework* recognizes regions with different characteristics in the code and optimizes the code according to the current region. The framework allows a specific order of optimization passes for each identified region type.

The AVIV compiler [HD98] uses an ISDL machine description. In this compiler, the statements of a basic block are converted into split-node directed acyclic graphs. In such a split-node DAG, the operation nodes are duplicated for each functional unit that is able to perform the operation. Then the compiler does operation grouping, functional assignment, a preliminary register allocation and instruction scheduling at once based on several heuristics.

Qin et al. introduced the Mescal Architecture Description Language [QRM04] (MADL). MADL supports the generation of instruction set simulators and its information is also used in the register allocation and instruction scheduling modules of a compiler.

In MIMOLA [Mar84], a processor is described by a netlist of hardware modules. In [LM95], Leupers et al. present a BDD-based technique to extract an instruction set from a MIMOLA processor model. The RECORD compiler [LM97] 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 non-terminals. `iburg` [FHP92a] is then used to generate a code selector from the grammar. The grammar generated by the RECORD system contains a large number of nonterminals and rules, that have to be evaluated at runtime. Many of these rules model intermediate results that may only be used in a limited context. Our approach combines these intermediate results into one single rule for each instruction, thus reducing the number of rules and the runtime overhead.

Fauth et al. developed the architecture description language nML [FPF95]. Two compilers have been built around nML. The CHESS code generation environment [LPK+95] is retargetable only to a limited set of processors: it targets fixed-point digital signal processors. The nML model is translated into an instruction set graph for its internal usage. CHESS does not use common tree matching algorithms but a special *bundling* algorithm for its code selection. The CBC compiler [FHMK94] also retargets itself by a nML description. A phase called *macro expansion* transforms the IR into operations of the target processor. To cope with graphs, and not just expression trees, the *heuristic node duplication* technique is performed on the IR. CBC also uses the `iburg` matcher generator.

5.2 Tree Pattern Matching

Aho and Johnson [AJ76] were the first to apply a dynamic programming algorithm to select code for expression trees. Later, Aho et al. improved on and simplified these ideas and developed the `twig` tool [AGT89]. They separated register allocation from code selection as this did not adversely affect the code quality and provided more flexibility. `twig` uses a modified version of the Aho-Corasick algorithm to match trees.

Balachandra et al. [BDB90] made an important discovery that increases the performance of the tree pattern matching algorithm. By precomputing *itemsets* at code selector generation time and storing the results in tables, the cost analysis can be avoided at matching time. This makes the algorithm run in time which is linear in the size of the expression tree.

The tool `BEG` [ESL89] is another code selector generator. Each rule in `BEG` can be supplied with a condition that is evaluated when the produced matcher is run and that prevents a rule from matching if it evaluates to false. `BEG` has an optional built-in register allocator.

Fraser et al. developed a tool called `burg` [FHP92b]. It reads a tree grammar and produces a tree pattern matcher in the C programming language. Its main development goal was to be fast. Thus it uses the technique from Balachandra et al. to compute the costs at generation time.

The tool `iburg` [FHP92a] reads the same input as `burg`. The difference is that `iburg` uses the standard dynamic algorithm that computes the costs at code selection time. While this approach is slower, it is easier to debug and smaller. Also delaying cost computation until the time the tree matching is done has the advantage that properties of the IR tree can be used when computing the cost (e.g. the value of a constant).

Ertl et al. [ECG06] save the computed states for tree nodes in a hash table. This approach retains the flexibility of dynamic cost computations at nearly the speed of precomputed states.

6. Conclusion

In this paper, we have presented an Architecture Description Language for compiler generation that allows the generation of an instruction selector, a register allocator and an instruction scheduler from a single specification. This specification is 5 times more concise than the usual specification. The resulting compiler achieves similar and, in several cases, even better code quality in terms of code size and performance than an existing highly optimized, handcrafted, back end.

Acknowledgments

This work is supported in part by Infineon Technologies Austria and the Christian Doppler Forschungsgesellschaft. We like to thank

Nigel Horspool and the anonymous reviewers for their helpful suggestions.

References

[AGT89] Alfred V. Aho, Mahadevan Ganapathi, and Steven W. K. Tjiang. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems*, 11(4):491–516, October 1989.

[AJ76] Alfred V. Aho and Stephen C. Johnson. Optimal code generation for expression trees. *Journal of the ACM*, 23(3):488–501, July 1976.

[BDB90] 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.

[BNS+04] 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.

[CHL+05] 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.

[ECG06] M. Anton Ertl, Kevin Casey, and David Gregg. Fast and flexible instruction selection with on-demand tree-parsing automata. In *PLDI 2006: Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, June 2006.

[ESL89] Helmut Emmelmann, Friedrich-Wilhelm Schröer, and Rudolf Landwehr. Beg - a generator for efficient back ends. In *PLDI '89: Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 227–237, June 1989.

[FHMK94] 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.

[FHP92a] 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.

[FHP92b] 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.

[FPF95] 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.

[HD98] 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.

[HGG+99] 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*, pages 485–490. IEEE Computer Society, March 1999.

[HKS03] Ulrich Hirnschrott, Andreas Krall, and Bernhard Scholz. Graph–coloring vs. optimal register allocation for optimizing compilers. In Laszlo Böszörmenyi, editor, *Proceedings of the Joint Modular Language Conference (JMLC) 2003*, LNCS, Klagenfurt, August 2003. Springer.

[HSDN01] Ashok Halambi, Aviral Shrivastava, Nikil Dutt, and Alex Nicolau. A customizable compiler framework for embedded systems. In *SCOPES '01: Proceedings of 5th International Workshop on Software and Compilers for Embedded Systems*. Springer, March 2001.

[KFH05] Andreas Krall, Stefan Farfeleder, and Nigel Horspool. Ultra fast cycle-accurate compiled emulation of inorder pipelined architectures. In Jarmo Takala, editor, *SAMOS 2005*, LNCS 3553, pages 222–231, Samos, July 2005. Springer.

[KHPP04] Andreas Krall, Ulrich Hirnschrott, Christian Panis, and Ivan Pryanishnikov. xDSPcore: A Compiler-Based Configureable Digital Signal Processor. *IEEE Micro*, 24(4):67–78, July/August 2004.

[LM95] 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.

[LM97] 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.

[LPK+95] 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.

[Mar84] 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.

[PHM00] 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.

[QRM04] 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.