Retargetable Generation of Code Selectors from HDL Processor Models

Rainer Leupers, Peter Marwedel
University of Dortmund, Dept. of Computer Science 12, 44221 Dortmund, Germany
email: leupers|marwedel@ls12.informatik.uni-dortmund.de

Abstract—Besides high code quality, a primary issue in embedded code generation is retargetability of code generators. This paper presents techniques for automatic generation of code selectors from externally specified processor models. In contrast to previous work, our retargetable compiler RECORD does not require tool-specific modelling formalisms, but starts from general HDL processor models. From an HDL model, all processor aspects needed for code generation are automatically derived. As demonstrated by experimental results, short turnaround times for retargeting are achieved, which permits to study the HW/SW trade-off between processor architectures and program execution speed.

1 Introduction

Today, many designs of embedded VLSI systems are based on programmable processors. Compared to custom hardware, processor-based design offers increased reusability and flexibility. Many standard processors are currently available in form of cores, which can be instantiated like library components. However, certain applications do not require the full amount of capabilities of a standard processor. Thus, in order to avoid a possible waste of resources, system houses are starting to use customized processors, commonly called ASIPs. This paper focusses on retargetable compilation of ASIP machine code from high-level programming languages.

Due to the narrow application range of a particular ASIP, high-level language (HLL) compilers are often not available, but the largest part of ASIP software is still developed manually using assembly languages [1]. A promising approach to eliminate this bottleneck are retargetable compilers. We call a compiler retargetable, if it can be adapted, so as to generate code for different target processors (within a defined processor class) in such a way that the largest part of compiler source code is retained. According to Goossens’ classification scheme [2], our RECORD compiler generates code for ASIPs in the DSP domain, that satisfy the criteria in table 1.

<table>
<thead>
<tr>
<th>parameter</th>
<th>supported features</th>
</tr>
</thead>
<tbody>
<tr>
<td>data type</td>
<td>fixed-point</td>
</tr>
<tr>
<td>code type</td>
<td>time-stationary</td>
</tr>
<tr>
<td>instruction format</td>
<td>horizontal &amp; encoded</td>
</tr>
<tr>
<td>memory structure</td>
<td>load-store &amp; memory-register</td>
</tr>
<tr>
<td>register structure</td>
<td>post-modify addressing modes</td>
</tr>
<tr>
<td>program control</td>
<td>heterogeneous &amp; homogeneous</td>
</tr>
<tr>
<td></td>
<td>standard jump instructions</td>
</tr>
<tr>
<td></td>
<td>mode registers</td>
</tr>
</tbody>
</table>

Table 1: Target processor class in RECORD

From a code generation viewpoint, however, HDL processor models are less favorable. HDL models may comprise details of the hardware structure, which are irrelevant for code generation. In contrast, models intended for code generation should represent a processor as a black box implementing a certain instruction set, i.e., behavioral models are preferable. The purpose of this paper is to present techniques which bridge the gap between HDL processor models comprising structural details and behavioral models suitable for code generation. For the processor class from table 1, we show how an efficient processor-specific code selector, which maps source program operations to processor-specific machine operations, can be automatically constructed from an HDL processor model.

1.1 Related work

Frequently, ASIPs show inhomogeneous architectures, which exclude the use of general-purpose compilation
For instance, presence of special-purpose registers associated with specific functional units makes register allocation by graph coloring, developed for machines with large homogeneous register files, less useful for ASIPs. As a consequence, recent work on embedded code generation (cf. [5] for surveys) focuses on new compilation techniques tailored towards inhomogeneous architectures.

Approaches to modeling of ASIP architectures for code generation can roughly be divided into graph-based and tree-based techniques. Graph-based models closely reflect the actual target processor structure. Graph nodes represent hardware entities like registers and functional units, while edges represent either physical connections or data flow between hardware entities. A graph-based model has been used in the MSSQ compiler [6], which has been refined for the CHESS code generator [7].

In contrast, tree-based models reflect the target machine in a behavioral manner through a set of tree-shaped register transfer (RT) templates. An RT template represents a primitive RT-level processor operation. Compared to graph-based models, RT templates hide the detailed hardware structure and thereby permit more efficient pattern matching between source code operations and processor-specific RTIs. Unfortunately, current code generators operating on tree-based processor models, such as [8, 9, 10], do not well support automatic generation of the RT template base from more common processor models. For realistic target processors, the RT template base may be considerably large, and a local change in the processor data path or instruction decoder may have a global impact on many of the RT templates. Therefore, tree-based models are often less comfortable from a modeling viewpoint, in particular if the architecture of the target ASIP is not completely fixed beforehand. In the CBC compiler [11], the RT template base is derived from a processor model in the nML language, which, however, does not offer the expressiveness of an HDL.

### 1.2 Overview of our approach

The processor modeling approach presented in this paper (fig. 1) is intended to combine the advantages of graph-based and tree-based processor modeling styles. The processor model visible to the user is an HDL model. This model optionally incorporates structural hardware details, and its granularity is determined by the user, dependent on the intended application and the available target processor documentation. From the HDL model, an internal graph model is constructed, which represents primitive processor entities according to the chosen model granularity as well as the interconnect structure. On the graph model, we perform instruction-set extraction in order to determine the set of available RT templates, while also taking into account possible restrictions due to instruction encoding. Exploiting semantical knowledge about hardware operators, the extracted RT template base is extended by further templates and is translated into a tree grammar. Tree grammar construction creates a behavioral view of the target processor, as required for efficient code selection.

From the tree grammar, we obtain a processor-specific code selector by existing compiler construction tools. The code selector is used for mapping expression trees in the intermediate source program representation to processor-specific RTIs. The remainder of this paper provides a more detailed description of code selector construction as well as an experimental evaluation.

### 2 Instruction-set extraction

Techniques for instruction-set extraction (ISE) have already been described in an earlier contribution [12]. In order to provide the necessary background for this paper, here we give a brief summary.

ISE operates on a netlist model of the target processor. Currently, the netlist model is constructed from a processor description in the MIMOLA HDL [13]. The concepts are, however, language independent, and a VHDL frontend is planned. The primitive netlist entities are modules. Module I/O ports are interconnected by wires or tristate busses. A module is described by its I/O interface and its behavior, which is given by a set of concurrent assignments to ports or local variables of the module. In contrast to the data path analysis technique in [14], ISE is not restricted to predefined component types, but the behavioral complexity of modules may range from primitive components like logic gates or registers to complete data paths. From the netlist model, ISE extracts the complete set of valid RT templates in the following two steps:

**Enumeration of data transfer routes:** For each RT destination (register, memory, port) in the netlist,
a backwards traversal in the netlist is executed. The netlist traversal searches for possible routes for transporting data from a set of source registers or ports through the data path to the destination within a single machine cycle. The examined transfer routes may cross module interconnections and combinational modules. When reaching multiple-input modules (e.g. ALUs, multiplexers) netlist traversal forks for each different input. In this way, all possible RT templates for a certain destination are enumerated, and each template is represented by a tree pattern.

**Analysis of control signals:** Each RT template is associated with an execution condition, i.e., the control signals for all modules involved in an RT template must be properly adjusted. Primary sources for control signals are the instruction memory and (optionally) mode registers, which store control signals that change only rarely. Analysis of control signals is performed by netlist traversal from module control ports back to primary control signal sources. Tracing back control signals may pass random logic components, e.g., instruction decoders. Thus, analysis of control signals requires support for Boolean manipulation of execution conditions. We model execution conditions by means of binary decision diagrams (BDDs), in which the Boolean variables correspond to the instruction word bits and mode register bits. The extracted execution conditions account for the required binary partial instructions and mode register states for each RT template. This information is used for code compaction and for revealing unsatisfiable execution conditions (e.g. due to instruction encoding conflicts or bus contentions), resulting in invalid RT templates, which are discarded from the template base.

### 3 Code selector generation

In order to increase the search space investigated during code selection, the RT template base delivered by ISE is extended by further templates, which cannot be directly derived from the processor model. Additional templates are created by exploiting algebraic properties of hardware operators, e.g. commutativity: For each RT template comprising a commutative operator, a complementary template with swapped arguments is added to the template base. Exploitation of commutativity avoids potential code quality overhead due to badly structured expression trees in the intermediate program representation. This is particularly important in the area of DSP, where sum-of-product computations are dominant. Optionally, additional templates are also created based on application-specific rewrite rules retrieved from an external transformation library.

#### 3.1 Tree grammar definition

In the next phase, the extended RT template base is translated into a tree grammar. Tree grammars, which are a special case of context-free grammars, are the formal basis of most contemporary code generation techniques operating on expression trees. This section describes systematic translation of an RT template base into a corresponding tree grammar representation. Formally, a tree grammar is a quintuple

\[ G = (\Sigma_T, \Sigma_N, S, R, c) \]

where \( \Sigma_T \) is an alphabet of terminals, \( \Sigma_N \) is an alphabet of non-terminals with \( \Sigma_N \cap \Sigma_T = \emptyset, S \in \Sigma_N \) is the start symbol, \( R \) is a finite set of rules, and \( c : R \rightarrow N \) is a cost function. All rules \( r \in R \) are of the form \( X \rightarrow \ell \), where \( X \in \Sigma_N \) and \( \ell \in TR(\Sigma_T \cup \Sigma_N) \). For an alphabet \( A, TR(A) \) denotes the tree language over \( A \) (cf. [15] for a formal definition). Let \( t_1, t_2 \in TR(\Sigma_T \cup \Sigma_N) \). \( t_1 \) derives \( t_2 \) in \( G \), if there exists a rule \( r : X \rightarrow t_3 \in R \), such that \( t_2 \) results from replacing a leaf labelled \( X \) in \( t_1 \) by \( t_3 \).

For a given RT template base, the tree grammar \( G \) must be constructed in such a way, that exactly the entities of the intermediate program representation can be derived from the start symbol \( G \). In our approach, these entities are expression trees (ETs), each associated with a destination. ETs are unary or binary trees, where inner nodes represent operators and leaves represent program variables, primary program inputs, or constants. The destination into which an expression tree is evaluated is explicitly taken into account, because for inhomogeneous data paths the instruction cost for moving the result of an ET to its destination may have impact on the code selected for the ET itself. We assume that all primary source program inputs and program variables are a priori bound to certain memory or register resources, or are mapped to primary processor ports. The same holds for the destinations, to which the results of ETs are assigned. The grammar components are constructed as follows:

**Terminals:** Let \( SEQ \) denote the set of all sequential processor components (capable of storing data), \( PORTS \) the set of primary processor ports, \( OP \) the set of operators available in hardware, and \( CONST \) the (possibly empty) set of hardwired constants. Furthermore, let \( TERM(x) \) denote an auxiliary function that returns a unique terminal symbol for any object \( x \). Then, \( \Sigma_T \) is defined as

\[
\{ASSIGN\} \cup \{TERM(x) \mid x \in SEQ \cup PORTS \cup OP \cup CONST\}
\]

The designated terminal ASSIGN is used to capture the actual assignment of ET results to a destination, which is explained below.

**Non-terminals:** Intuitively, non-terminals in \( G \) represent hardware entities capable of temporarily storing data, e.g., registers holding intermediate results during ET evaluation. Since for inhomogeneous architectures, such "temporary locations" cannot be distinguished from those locations that store primary ET inputs or ET results, all components in \( SEQ \) must also appear in non-terminal form, in order to permit their use for intermediate results as well. Let \( \text{NONTERM}(x) \) denote a function that returns a unique non-terminal symbol for \( x \). Then, \( \Sigma_N \) is defined as

\[
\{START\} \cup \{\text{NONTERM}(x) \mid x \in SEQ \cup PORTS\}
\]
START is the designated grammar start symbol. Besides SEQ components, also the primary processor ports appear as non-terminals in $\Sigma_N$. This enables a uniform derivation mechanism for ETs, independent of whether the destination is a sequential component or a port, which is explained in the following.

Rules: The rule set $R$ of $G$ consists of three groups:

1. Start rules. The destination of an ET can be any sequential component or processor output port. Therefore, the start symbol for $G$ must be "generic", i.e., it must match any possible ET destination. This can be achieved by introducing designated start rules of the form
   \[ \text{START} \to \text{ASSIGN} (\text{TERM}(\text{dest}), \text{NONTERM}(\text{dest})) \]

   for each destination $\text{dest} \in \text{SEQ} \cup \text{PORTS}$. Start rules ensure that for any ET with destination $\text{dest}$ and having a derivation from $\text{NONTERM}(\text{dest})$, this derivation is always found independently of $\text{dest}$. Furthermore, it is ensured that the cost of the derivation includes the cost for moving the ET result to $\text{dest}$.

2. RT rules. RT rules correspond to the elements of the RT template base, i.e., RT rules serve the purpose of actually deriving ETs. For each RT template of the form "$\text{dest} := \text{exp}$" a grammar rule
   \[ \text{NONTERM}(\text{dest}) \to L(\text{exp}) \]

   is constructed, where the left hand side $L(\text{exp})$ is defined according to table 2.

   \begin{center}
   \begin{tabular}{ll}
   \hline
   $\text{exp}$ & $L(\text{exp})$ \\
   \hline
   constant $\in \text{CONST}$ & $\text{TERM}(\text{exp})$ \\
   reference to $\text{SEQ}$ & $\text{NONTERM}(\text{exp})$ \\
   reference to $\text{PORTS}$ & $\text{TERM}(\text{exp})$ \\
   unary expression & $\text{TERM}(\text{exp})$ \\
   $\text{op}(\text{exp}_1)$ ($\text{op} \in \text{OP}$) & $\text{TERM}(\text{op})(L(\text{exp}_1))$ \\
   binary expression & $\text{TERM}(\text{op})(L(\text{exp}_1), L(\text{exp}_2))$ \\
   \hline
   \end{tabular}
   \end{center}

   Table 2: Specification of left hand sides of rules

3. Stop rules: For each $\text{REG} \in \text{SEQ}$ a rule of the form
   \[ \text{NONTERM}(\text{REG}) \to \text{TERM}(\text{REG}) \]

   is incorporated. Such "stop rules" permit to terminate derivations from $\text{REG}$, whenever ET leaves are reached during derivation.

Cost function: Since we assume single-cycle RTs, we set $c(r) := 1$ if $r \in R$ is an RT rule. Start and stop rules are only needed for consistency, so that for these rules $c(r)$ is set to zero.

### 3.2 Tree parser generation

Optimal code selection for an expression tree $T$, i.e., covering $T$ by a minimum set of RT templates, is equivalent to computing a minimum cost derivation of $T$ in the tree grammar $G$. This process is called tree parsing. Several tree parser generators have been developed in the compiler community. Currently, we use the iburg tree parser generator from Princeton University [16]. iburg reads a Backus-Naur specification of a tree grammar $G$ and emits C code for an efficient grammar-specific tree parser for $G$, based on the dynamic programming paradigm. Tree parsers generated by iburg show the following characteristics:

- The computation time is approximately linear in the number of ET nodes, with a constant factor determined by the underlying grammar. In practice, several hundred RT templates per CPU second are emitted on the average.
- The computed tree derivations are guaranteed to be optimal with respect to the accumulated costs of selected RTs. Simultaneously, the costs for pure data transport operations are minimized. Since special-purpose registers appear as grammar non-terminals, also allocation of special-purpose registers for intermediate results is implied by the constructed parse trees. Furthermore, also chained operations, e.g., multiply-accumulate or add-with-shift operations, are optimally exploited.

Limitations of tree parsing mainly concern incorporation of register spills and instruction-level parallelism in the cost function. We use an extension of the scheduling technique from [8] in order to minimize register spills. Exploitation of potential parallelism is performed in a subsequent code compaction phase [17].

### 4 Experimental results

The retargeting procedure presented in this paper has been implemented and has been applied to a number of different target processors. These include simple examples (demo, ref), educational purpose machines (manocpu [18], tanenbaum [19]) an industrial ASIP (bass boost [20]) and a standard DSP (Texas Instruments TMS320C25 [21]). Experimental results are listed in table 3. The number of RT templates in the extended template base is shown in column 2. Column 3 gives the total retargeting time, including ISE, grammar construction, parser generation by iburg, and parser compilation by a C compiler.

The results indicate, that our approach works for realistic machines, and that retargeting—once a new HDL processor model is available—at most takes some CPU minutes. Such short turnaround times permit to explore different target processor architectures by means of a retargetable compiler.

<table>
<thead>
<tr>
<th>target processor</th>
<th>number of RT templates</th>
<th>retargeting time</th>
</tr>
</thead>
<tbody>
<tr>
<td>demo</td>
<td>439</td>
<td>356</td>
</tr>
<tr>
<td>ref</td>
<td>1763</td>
<td>84</td>
</tr>
<tr>
<td>manocpu</td>
<td>207</td>
<td>6.3</td>
</tr>
<tr>
<td>tanenbaum</td>
<td>232</td>
<td>11.7</td>
</tr>
<tr>
<td>bass boost</td>
<td>89</td>
<td>3.7</td>
</tr>
<tr>
<td>TMS320C25</td>
<td>356</td>
<td>165</td>
</tr>
</tbody>
</table>

Table 3: Experimental results: retargeting time
Efficient code selection for expression trees based on tree parsing also forms the basis for generation of high quality (compacted) code. The chart in Fig. 2 shows results for basic program blocks (taken from the DSP- Stone benchmark suite [22]) and the TMS320C25 DSP. The columns show the relative code size (hand-written code set to 100 %) achieved by T1's C compiler (left) and RECORD (right). In many cases, RECORD achieves a lower overhead compared to hand-written code and outperforms the target-specific compiler.

5 Conclusions

The growing diversity of application-specific programmable processors creates a need for retargetable compilers. For a defined processor class, the presented retargeting procedure provides an automated path from a structural or behavioral HDL processor model to an efficient code selector for expression trees. In this way, short retargeting times are achieved, which support HW/SW co-design at the processor level. Furthermore, retargetability does not necessarily contradict high code quality, which was demonstrated for a representative DSP processor.

References


