Instruction Scheduler Generation for Retargetable Compilation

Oliver Wahlen, Manuel Hohenauer, Rainer Leupers, and Heinrich Meyr
Aachen University of Technology

The availability of C compilers is crucial to the efficient design of embedded systems. Using virtual resources to automatically generate parts of a compiler's instruction scheduler from a formal processor description significantly reduces the overall scheduler generation time.

INCREASINGLY COMPLEX handheld and automotive devices impose high demands for robustness, performance, power efficiency, and flexibility at competitive prices. Embedded systems that combine programmable cores with dedicated hardware for the application on a single silicon chip are a good way to meet these demands. Embedded systems often contain application-specific instruction set processors comprising special-purpose functional units, instructions, and addressing modes. ASIPs combine the main attribute of processors—reconfigurability—with the main attribute of ASICs—high computational performance with low power and chip area requirements. ASIP design is challenging because besides the hardware model required for synthesis, the designer must create a software tool suite. Indispensable tools include an assembler, a linker, a simulator, and a profiler.

Recently, the software design entry language has been shifting from assembly to C. One reason is that nearly all algorithmic descriptions are written in (or can be converted to) C. If the design is reused for similar applications, manually converting the application from C to assembly repeatedly is unnecessary. A second reason is that there is increasing tool support for automatically retargeting a C compiler to a certain class of processor architectures from an architecture or compiler description.

The C compiler's back end generates assembly code from an internal representation of the original C program. One back-end component, the instruction scheduler, determines the sequence in which the instructions execute on the processor. Schedulers for architectures that use instruction-level parallelism (that is, the processor can execute instructions in parallel, and parallelism is specified in the assembly program) also decide which instructions execute in parallel. To address application-specific processors with their special hardware capabilities, current C compiler environments must become easily retargetable to new processor architectures. Retargeting a compiler to a new processor is defined as modifying an existing compiler to produce assembly code for a new processor. This usually means that the C compiler's front end and most of its optimization modules are reused, whereas the architecture-specific back end must be changed significantly. (An earlier article presents an overview of compiler design issues for embedded processors.[1])

The importance of retargetability becomes obvious in an ASIP design process involving the typical processor architecture exploration phase. Architecture exploration is the process of iteratively evaluating alternatives of the hardware implementation until a configuration that satisfies the design constraints is found. Each evaluation step requires verification of the consistency of the hardware model with the software tools. Furthermore, if a compiler is not used for the exploration, each step necessitates adaptation of the processor's assembler input. The duration of this tedious, error-prone process is one of the most important factors in the product's time to market, quality, and price.

To speed up the architecture exploration loop and

0740-7475/03/$17.00 © 2003 IEEE Published by the IEEE Computer Society

eliminate inconsistency, recent research focuses on automating the creation of the hardware model and the software tools. A common concept is to manually specify a single processor model in an architecture description language (ADL) and let a tool generate all or part of the software tool suite and the hardware model. FlexWare[2] is an environment based on this concept. Unfortunately, it still has inconsistency problems due to redundancies in the description. The Expression[3] and Peas[4] ADLs depend on a detailed semantic analysis of the processor model, thus imposing limits on the architectural features they can model. The Mimola[5] ADL describes the processor as a netlist, limiting modeling efficiency and slowing the exploration loop. Furthermore, Mimola does not directly support pipelined architectures. The Chess[6] environment, based on the nML ADL, is primarily useful for retargeting DSP compilers.

We have developed a technique that improves compiler retargetability by automatically generating parts of the instruction scheduler from a formal architecture description. We analyzed the quality and limitations of this approach by applying it to the architecture model of the STMicroelectronics/Hewlett-Packard ST200 very long instruction word (VLIW) processor.[7]

Software environment

The ST200 processor is modeled in the Language for Instruction Set Architectures (LISA),[8] an ADL that describes the behavior, structure, and I/O interfaces of a processor architecture. In addition to the ST200, designers have used LISA to model a wide variety of architectures, including the ARM7, C62x, C54x, and MIPS32 4K, and to develop ASIPs such as the ICore1, ICore2, and Alice.[9]

Several software tools based on LISA make up the LISA Processor Design Platform, commercially available from LISATek. Figure 1 illustrates the LPDP. To provide a consistent design flow for system-level processor architecture and software designers, the LPDP supports the generation of an assembler, a disassembler, a linker, a simulator with extensive profiling capabilities, and a synthesizable hardware description from a common LISA processor model. For seamless integration and verification of the LISA processor model in a system context, the simulator includes interfaces permitting cosimulation of the LISA model with other simulation environments—notably the CoCentric System Studio and the VSS VHDL simulator, both from Synopsys.

To partially generate a C compiler from LISA descriptions, we used the CoSy compiler development system from Associated Computer Experts (http://www.ace.nl/). Compiler designers can use CoSy to retarget high-level language (such as C and C++) compilers to a broad range of architectures in a flexible, modular manner. The designer describes all aspects of the CoSy compiler back end in code generator description (CGD) files. After analyzing these files, the CoSy back-end generator produces the retargeted compiler back end.

To verify and profile our scheduler generation algorithm, we developed a tool that analyzes a LISA processor model and generates the CoSy CGD file describing an instruction scheduler. We combine this file with manually written CGD files describing the instruction selector and the register allocator. The back-end generator reads all the CGD files and then generates a C compiler for the LISA model.

CoSy scheduler descriptions have two parts. The first contains three tables that provide information about latencies between instructions: read-after-write latency, write-after-write latency, and write-after-read latency. To date, our scheduler generation technique applies only to generating the scheduler description's second part, which describes structural hazards (that is, potential resource conflicts) in the processor architecture. This part is similar to a reservation table: It defines which (exclusive) resources an instruction uses and in which cycle it uses them. The scheduler can also specify that an instruction can allocate alternative resources; for example, if the processor has several ALUs, the scheduler can decide which ALU will execute the instruction.

ST200 architecture

An important problem for VLIW architectures is that not all instructions can be combined in a single instruction word because of constraints on the coding format. We call a valid combination of instruction classes a *composition*. Figure 2 shows examples of such compositions. Either two instructions with register operands (composition 0) or a single instruction with an immediate operand (composition 1) can form an instruction word.

In LISA, we model such constraints on combining instructions into one instruction word by listing the valid cases of instruction combinations in a switch statement. Consequently, each case represents a composition. Figure 3 shows the LISA code corresponding to the examples in Figure 2. In the code, `reg8_op` and `imm16_op` are LISA operations that contain the information about coding format and syntax of the two types of instructions.

The ST200 is a configurable VLIW core for media-

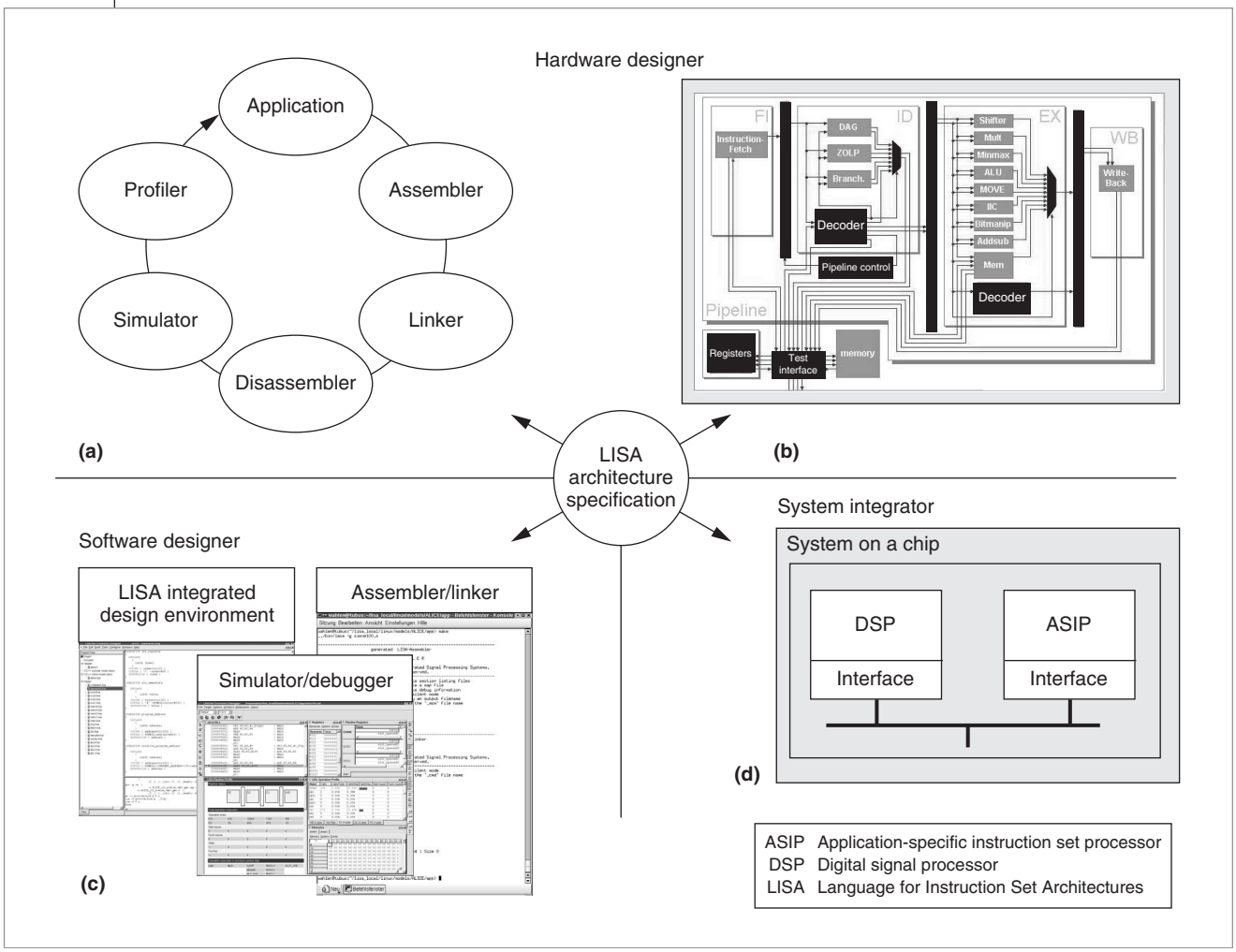


Figure 1. LISA processor design platform: architecture exploration tools (a), architecture implementation tools (b), software application design tools (c), and integration and verification tools (d). In (b), LISA has generated the processor control path (parts colored black). Generation of the memory and data path (gray parts) is not yet possible but is currently under research.

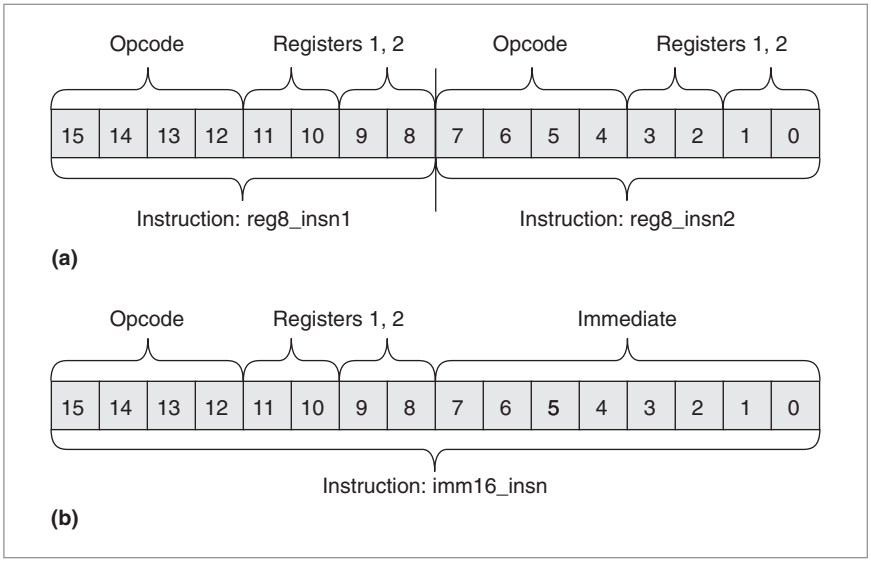


Figure 2. Coding formats for an instruction word: composition 0, two parallel instructions with register arguments (a); composition 1, a single instruction with register and immediate arguments (b).

oriented applications developed jointly by HP Labs and STMicroelectronics. After a predecoding pipeline stage, the instruction word splits into four pieces, which are assigned to corresponding lanes with attached (parallel) units. The assembly syntax determines the specific lane in which an instruction will execute. A lane may contain not only instructions; it may also contain an immediate operand of an instruction from a neighboring lane. Some instructions can execute only in certain lanes. For these reasons, the LISA decoder description is complex and contains 42 different valid compositions. Because the number of compositions plays an important role in the complexity of the scheduler generator algorithm, we chose the ST200 architecture as a complex real-life example to analyze the quality of our approach.

Scheduler generation using virtual resources

Our scheduler generator analyzes the instruction-coding format of a LISA processor model and emits a reservation table similar to that used in many retargetable compiler environments. In our case, the reservation table is the input to the CoSy back-end generator. Along with a manual specification of an instruction's latency, the back-end generator can produce a space- and time-efficient finite-state machine for a scheduler.[10] Each FSM state represents a constellation of scheduled instructions. Thus, the addition of an instruction to a combination of already scheduled instructions represents a new state. The addition is legal if there is a state transition in the FSM between the old and the new state.

Eisenbeis, Chamski, and Rohou[11] first proposed using virtual resources in an instruction scheduler. However, their approach does not generate FSMs and is therefore less efficient than the one we propose. Furthermore, their technique does not address architectures in which instructions occupy multiple issue slots. Our approach uses virtual resources to generate FSM-based schedulers from an ADL that supports processor architectures with several independent valid word compositions. The technique is not bound to a certain ADL or compiler environment. With LISA and the LPDP, we can address a broad range of architectures without having to manually specify structural hazards.

To illustrate our technique, we extend the example in Figure 2 to three compositions, as Figure 4 shows. In contrast to Figure 2, the squares in Figure 4 do not represent bits of the binary instruction word. Each box labeled v_0 to v_{23} represents one of 24 virtual resources

```
OPERATION decode_op
{
 DECLARE
 {
  ENUM compositions = {composition0,
    composition1};
  GROUP reg8_insn1, reg8_insn2 = { reg8_op };
  GROUP imm16_insn = { imm16_op };
 }
 SWITCH(compositions)
 {
  CASE composition0:
  {
   CODING AT (program_counter)
     { insn_reg == reg8_insn1 || reg8_insn2 }
   SYNTAX { reg8_insn1 "," reg8_insn2 }
  }
  CASE composition1:
  {
   CODING AT (program_counter)
     { insn_reg == imm16_insn }
   SYNTAX { imm16_insn }
  }
 }
}
```

Figure 3. Modeling coding constraints in LISA.

that have no direct correspondence to any hardware resource. A defined set of virtual resources is allocated if an instruction is scheduled into a certain instruction slot of a certain composition. This allocation is exclusive; that is, the constellation of allocated virtual resources allows or disallows other instructions to be scheduled. If we go back to the example in Figure 2, the purpose of this technique becomes obvious: If one **`reg8_insns`** is already scheduled in one of the two slots of composition 0, the schedule must prohibit the insertion of any **`imm16_insn`** into the same instruction word. On the other hand, it must allow the scheduling of an additional **`reg8_insn`**.

In Figure 4, composition 0 contains three instruction slots, composition 1 contains two slots, and composition 2 contains four. The vertical lines in the figure indicate the virtual resources allocated if an instruction is scheduled into a slot of a certain composition. We define $s_{i,j}$ as slot j of composition i, and we define $I(s_{i,j})$ as the set of instructions that can execute in this slot. A possible assignment of instruction sets to slots

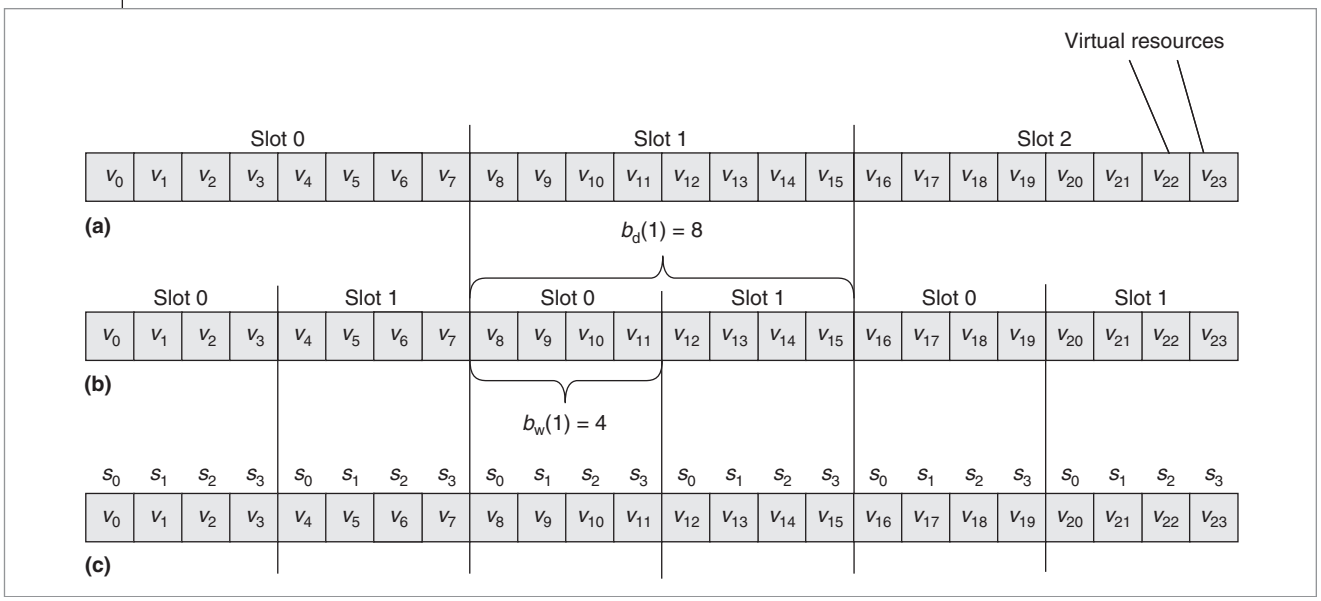


Figure 4. Allocation of virtual resources for three compositions: composition 0 (a), composition 1 (b), and composition 2 (c).

for Figure 4 is

$I(s_{0,0})$: **add**, **sub**
$I(s_{0,1})$: **mul**
$I(s_{0,2})$: **div**
$I(s_{1,0})$: **add_imm**, **sub_imm**
$I(s_{1,1})$: **mul**
$I(s_{2,0})$: **add**, **sub**
$I(s_{2,1})$: **add**, **sub**
$I(s_{2,2})$: **mul**
$I(s_{2,3})$: **mul**

The **div** instruction is only part of instruction set $I(s_{0,2})$. For such an instruction, the scheduler allocates all resources associated with slot 2 of composition 0—namely, the range from v_{16} to v_{23}. This means that composition 0 allows the execution of either **add** or **sub** in parallel with **mul** and **div**. It is not possible to schedule **add_imm** in parallel with **div** because **add_imm** would allocate v_0 to v_3, v_8 to v_{11}, and v_{16} to v_{19}. Virtual resources v_{16} to v_{19} are allocated by both **div** and **add_imm**, so they cannot be scheduled in the same cycle. If an instruction can be scheduled in several slots (potentially in different compositions), the scheduler can alternatively allocate any of the virtual resources associated with these slots. Thus, in the example, the scheduler can allocate all virtual resources associated with $I(s_{0,0})$, $I(s_{2,0})$, or $I(s_{2,1})$ for the **add** instruction. This example shows that the use of virtual resources instructs the scheduler to combine only instructions of the same composition in the instruction word and to allow only a single instruction in each instruction slot.

To generalize from the example, we must analyze how many virtual resources are needed for n_c compositions with $n_s(i)$ instruction slots (i is the composition's index). Furthermore, we must find a mapping function between the slots and the associated virtual resources. We can derive the formal solution from Figure 4 if we assume that only composition 2 is valid in the example processor. In this case, only four virtual resources were needed. They directly correspond to this composition's four instruction slots. If compositions 2 and 1 are valid, the allocation of a slot in composition 1 must be mutually exclusive with any slot allocation of composition 2: That is, the virtual resource sets associated with each slot of composition 1 must overlap with each virtual resource set of composition 2. This requires eight virtual resources: Slots 0 and 1 of composition 1 allocate ranges v_0 to v_3 and v_4 to v_7, respectively. Each of the four slots of composition 2 allocates two resources: one from each range. Because the resource sets associated with the slots overlap, the scheduler cannot use a slot from composition 1 and a slot from composition 2 at the same time.

Because composition 1 comprises two slots, eight (2×4) virtual resources are needed. If we add composition 0 with its three slots, 24 ($3 \times 2 \times 4$) virtual resources are needed. In formal notation, the number

of virtual resources (n_{res}) required for n_c compositions with $n_s(i)$ slots is

$$n_{res} = \prod_{i=0}^{n_c-1} n_s(i)$$

Each slot $s_{i,j}$ out of composition i allocates several sets of consecutive virtual resources. The number of virtual resources contained in such a set is $b_w(i)$. The distance between the starting point of two sets is $b_d(i)$. Figure 4 illustrates the meaning of these terms. We calculate $b_w(i)$ and $b_d(i)$ of composition i as

$$b_w(i) = \prod_{k=i+1}^{n_c-1} n_s(k)$$

and

$$b_d(i) = \prod_{k=i}^{n_c-1} n_s(k)$$

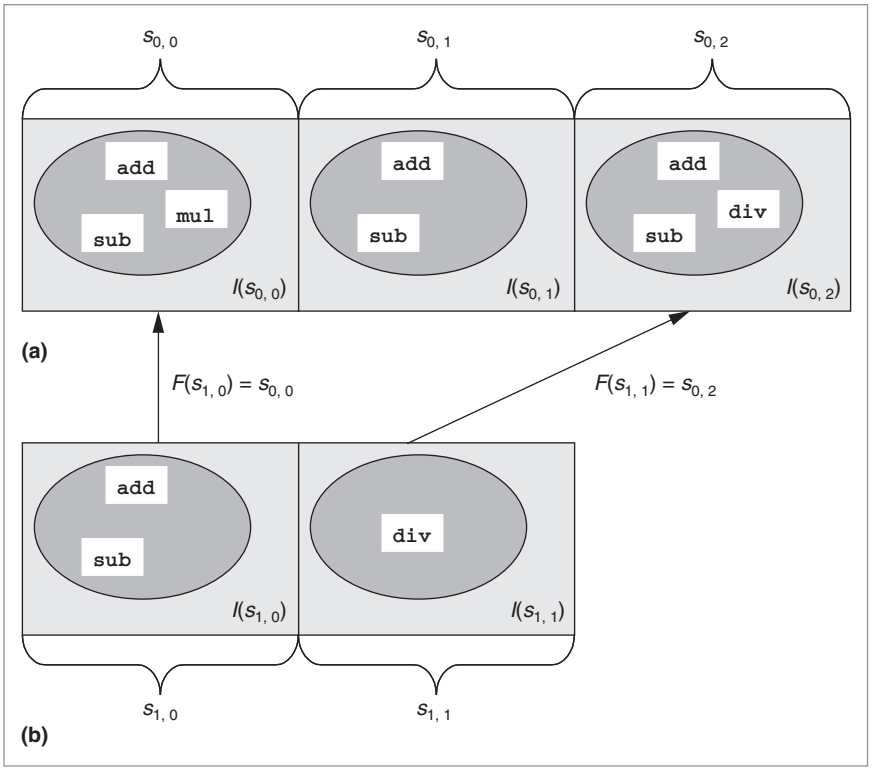


Figure 5. Composition c_0 (a) makes composition c_1 (b) irrelevant.

Slot $s_{i,j}$ allocates all virtual resources $v_k(k \in [0, n_{res} - 1])$ for which the following equation is true (mod is the modulus operator, and div the integer division without remainder):

$$j = [k \bmod b_d(i)] \operatorname{div} b_w(i) \quad (1)$$

For example, to determine which virtual resources must be allocated for slot $s_{1,1}$, we calculate $b_d(1) = n_s(1) \times n_s(2) = 2 \times 4 = 8$ and $b_w(1) = n_s(2) = 4$. Now we iterate over all $n_{res} = n_s(0) \times n_s(1) \times n_s(2) = 3 \times 2 \times 4 = 24$ resources and find that Equation 1 is true for virtual resources v_4 to v_7, v_{12} to v_{15}, and v_{20} to v_{23}.

Reducing the number of virtual resources

The problem with the virtual resources approach is its complexity, which grows almost exponentially with the number of coding compositions in the LISA model. If we used this approach for the ST200 model with its 42 compositions, we would end up with $n_{res} \approx 44 \times 10^{15}$ virtual resources.

Fortunately, not every coding composition is relevant for the scheduler. A composition, c_1, is irrelevant if there is at least one other composition, c_0, that allows all the instruction combinations that c_1 allows (or even more). In that case, the virtual resources algorithm does not take c_1 into account. Figure 5 shows an example of an irrelevant case. We can formulate the definition of irrelevancy more suitably for an algorithmic implementation: Composition c_0 makes composition c_1 irrelevant if, for each slot $s_{1,j}$ of c_1, there is exactly one dedicated slot $s_{0,i}$ in c_0 such that all instructions that fit into $s_{1,j}$ also fit into $s_{0,i}$. Figure 5 shows this injective function, F. (Function $f: X \rightarrow Y$ is *injective*, or *one on one*, if and only if $\forall x \in X \, \forall z \in X \, [f(x) = f(z) \Rightarrow x = z]$.) In the Figure 5 example, $s_{1,0}$ maps to $s_{0,0}$, and $s_{1,1}$ maps to $s_{0,2}$. The mapped slots comprise a superset of the instructions in the original slots.

We implemented a recursive algorithm that detects irrelevant compositions in polynomial time. Eliminating all irrelevant compositions made it possible to generate a CoSy scheduler description from the LISA model within a reasonable time. We reduced the number of relevant coding cases from 42 to 11. That reduced the number of required virtual resources from $\approx 44 \times 10^{15}$ to 139,967. Although this is still a very large number, the CoSy back-end generator produced a scheduler from this description within six minutes on a desktop PC. The resulting scheduler state machine consists of 30 states and 25 state transitions.

Results

The scheduler description presented so far limits the scheduler to valid instruction compositions of a word. It does not yet contain information about dataflow dependencies or other structural hazards. However, it is possible to enrich the description with such information and to support architectures with complex dependency patterns and multicycle resources, such as Philips's Trimedia or Fujitsu's FR-V.

We manually wrote a complete ST200 scheduler description, from which CoSy generated a scheduler FSM with 831 states and 830 state transitions in less than five seconds. To compare the manually written scheduler description with the generated version, we reduced the manual description to reflect only the valid instruction combinations based on the coding. The resulting scheduler had 28 states and 27 transitions. Hence, the manual scheduler's complexity is very close to that of the generated version.

The manual scheduler description is far smaller and can be processed much faster because for the ST200 we can associate instruction slots 0, 1, 2, and 3 with bits in the word's binary coding. The ST200 instructions always allocate one or more of these slots, which are fixed for all instruction compositions. Our approach associates a dedicated set of slots with each composition. The advantage is that instructions can allocate an arbitrary set of word bits depending on the composition the word bits are used in. Nevertheless, both descriptions provide the same information about which instruction can be scheduled into a set of already scheduled instructions, so it is not surprising that both descriptions result in similar FSMs.

The current state of our project requires that we manually add the instruction latency information to the scheduler description to produce a correct scheduler. The latency description imposes the same constraints on both the manual and the automated approaches. Thus, the scheduler generation technique presented here, as compared with a manual scheduler specification, in no way affects the generated scheduler's code quality and performance. Generating the scheduler from the generated specification with virtual resources takes longer than generating it from a manual specification. However, our automated approach reduces the time of the overall scheduler generation process for a given processor model by orders of magnitude.

A TYPICAL SCHEDULER description contains two parts: The first part eliminates dataflow hazards (in other words, it describes instruction latencies). The second part avoids structural hazards (instructions' conflicting resource requirements). The virtual resources technique addresses the second part. To allow the generation of complete scheduler descriptions, we are working on a method of extracting instruction latency tables from LISA processor descriptions. The availability of a complete scheduler generator would significantly speed up the tedious and error-prone compiler design process. Furthermore, it would eliminate the architecture exploration engineer's need to have knowledge about scheduling concepts. ■

References

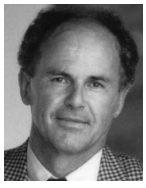
1. R. Leupers, "Compiler Design Issues for Embedded Processors," *IEEE Design & Test*, vol. 19, no. 4, July 2002, pp. 51-58.
2. P. Paulin and M. Santana, "FlexWare: A Retargetable Embedded-Software Development Environment," *IEEE Design & Test*, vol. 19, no. 4, July 2002, pp. 59-69.
3. P. Grun et al., "RTGEN: An Algorithm for Automatic Generation of Reservation Tables from Architectural Descriptions," *Proc. 12th Int'l Symp. System Synthesis* (ISSS 99), IEEE CS Press, Los Alamitos, Calif., 1999, pp. 44-50.
4. M. Itoh et al., "PEAS-III: An ASIP Design Environment," *Proc. Int'l Conf. Computer Design* (ICCD 00), IEEE CS Press, Los Alamitos, Calif., 2000, pp. 430-436.
5. R. Leupers and P. Marwedel, "Retargetable Code Compilation Based on Structural Processor Descriptions," *Design Automation for Embedded Systems*, vol. 3, no. 1, Jan. 1998, pp. 75-108; http://www.kluweronline.com/issn/0929-5585/contents.
6. D. Lanner et al., "Chess: Retargetable Code Generation for Embedded DSP Processors," *Code Generation for Embedded Processors*, P. Marwedel and G. Goosens, eds., Kluwer Academic, Norwell, Mass., 1995, pp. 85-102.
7. P. Faraboschi et al., "Lx: A Technology Platform for Customizable VLIW Embedded Processing," *Proc. 27th Int'l Symp. Computer Architecture* (ISCA 2000), ACM Press, New York, 2000, pp. 203-213.
8. A. Hoffmann et al., "A Novel Methodology for the Design of Application-Specific Instruction Set Processors (ASIPs) Using a Machine Description Language," *IEEE Trans. Computer-Aided Design*, vol. 20, no. 11, Nov. 2001, pp. 1338-1354.
9. O. Wahlen et al., "Application Specific Compiler/Architecture Codesign: A Case Study," *Proc. Joint Conf. Languages, Compilers, and Tools for Embedded Systems: Software and Compilers for Embedded Systems*, ACM Press, New York, 2002, pp. 185-193.

10. V. Bala and N. Rubin, "Efficient Instruction Scheduling Using Finite State Automata," *Proc. 28th Ann. Int'l Symp. Microarchitecture* (MICRO-28), IEEE CS Press, Los Alamitos, Calif., 1995, pp. 46-56.
11. C. Eisenbeis, Z. Chamski, and E. Rohou, "Flexible Issue Slot Assignment for VLIW Architectures," *Proc. 4th Int'l Workshop Software and Compilers for Embedded Systems* (SCOPES 99), ACM Press, New York, 1999.

Oliver Wahlen is pursuing a PhD in electrical engineering at Aachen University of Technology, Germany. His research interests include retargetable code generation for ASIP and DSP architectures, focusing on automated scheduler generation from architectural descriptions. Wahlen has a diploma in electrical engineering from Aachen University of Technology.

Manuel Hohenauer is pursuing a PhD in electrical engineering at Aachen University of Technology. His research interests include retargetable code generation for embedded processors, with a focus on machine description generation for retargetable compilers from architectural descriptions. Hohenauer has a diploma in electrical engineering from Aachen University of Technology.

Rainer Leupers is a professor of software for systems on silicon in the Institute for Integrated Signal Processing Systems at Aachen University of Technology. His research interests include software development tools for embedded systems, with emphasis on efficient compilers. Leupers has a diploma and a PhD, both in computer science, from the University of Dortmund, Germany.



Heinrich Meyr is a professor of electrical engineering at Aachen University of Technology, where he heads the Institute for Integrated Signal Processing Systems. Meyr has MS and PhD degrees in electrical engineering from the Swiss Federal Institute of Technology, Zurich.

■ Direct questions and comments about this article to Oliver Wahlen, Sommerfeldstr. 24, Room 24A305, 52074 Aachen, Germany; wahlen@ert.rwth-aachen.de.