Compiler Design Issues for Embedded Processors

Rainer Leupers
Aachen University of Technology

The growing complexity and high efficiency requirements of embedded systems call for new code optimization techniques and architecture exploration, using retargetable C and C++ compilers.

COMPILERS TRANSLATE high-level programming languages such as C and C++ into assembly code for a target processor. Used for decades to program desktop operating systems and applications, compilers are among the most widespread software tools.

For processor-based embedded systems, however, the use of compilers is less common. Instead, designers still use assembly language to program many embedded applications. Anyone who has programmed a processor in assembly knows the resulting problems: a huge programming effort and, compared with C or C++, far less code portability, maintainability, and dependability. So, why is assembly programming still common in embedded-system design? The reason lies in embedded systems' high-efficiency requirements.

Processor-based embedded systems frequently employ domain-specific or application-specific instruction set processors (ASIPs), which meet design constraints such as performance, cost, and power consumption more efficiently than general-purpose processors. Building the required software development tool infrastructure for ASIPs, however, is expensive and time-consuming. This is especially true for efficient C and C++ compiler design, which requires large amounts of resources and expert knowledge. Therefore, C compilers are often unavailable for newly designed ASIPs.

Of course, not only the processor architecture but also the embedded software executed on the processor must be efficient. However, many existing compilers for ASIPs and domain-specific processors such as DSPs generate low-quality code. Experimental studies[1] show that compiled code may be several times larger and/or slower than handwritten assembly code. Because this poor code is virtually useless for embedded systems under efficiency constraints, it requires a time-intensive postpass optimization. The cause of many compilers' poor code quality is the highly specialized architecture of ASIPs, whose instruction sets are incompatible with high-level languages and traditional compiler technology.

To boost designer productivity, a transition from assembly to C must take place in the embedded-system domain. That means the problems of compiler unavailability and poor code quality must be solved. As embedded applications increase in complexity, assembly programming will no longer meet short time-to-market requirements. Given the trend toward increasingly complex high-end processor architectures—with deep pipelining, predicated execution, and high parallelism—future human programmers are unlikely to outperform compilers in terms of code quality.

Moreover, designers increasingly employ compilers not only for pure application programming after an ASIP's architecture is fixed but also for architecture exploration. During exploration, the designer tunes the initial archi-

0740-7475/02/$17.00 © 2002 IEEE

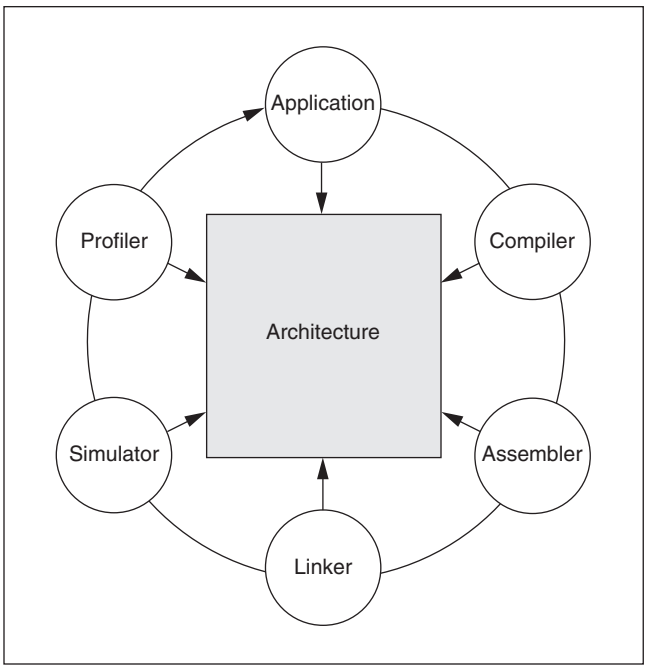

Figure 1. Iterative architecture exploration.

tecture for a given application or application set. This tuning requires an iterative, profiling-based methodology, by which the designer evaluates the cost-performance ratios of many potential architecture configurations. If C or C++ application programming is intended, the designer should apply a compiler-in-the-loop type of architecture exploration, as illustrated in Figure 1, thus avoiding a compiler and architecture mismatch.

A number of in-house ASIP design projects conducted by system houses suffer from the fact that the architecture was fixed before anybody thought about compiler issues. If the system designers decide only afterward that a C compiler should be added to the software tool chain, the compiler designers often have difficulty ensuring good code quality because an instruction set designed primarily from a hardware designer's viewpoint fails to support their efforts. Therefore, architecture-compiler co-design is essential to good overall efficiency of processor-based designs.

Solving these problems is a topic of intensive research in academia and industry. To combat poor code quality, compiler designers need domain-specific code optimization techniques that go beyond classical compiler technology, which mainly supports machine-independent optimization and code generation for clean architectures (those with homogeneous register files) such as reduced-instruction-set computers (RISCs).

Compiler design

Because a compiler is a comprehensive piece of software (frequently 50,000 lines of source code or more), it is usually subdivided into numerous phases. Many code optimization problems are known to be NP-complete; that is, they (most likely) require exponential computation time for optimal solutions.[2] Compilers handle this computational complexity by separating the code optimization process into multiple passes, each with a limited scope, and using heuristics instead of optimal solutions. The compiler usually consists of a front end, an intermediate representation (IR) optimizer, and a back end. Figure 2 illustrates this structure.

Front end

The front end for the source language (C, for example) translates the source program into a machine-independent IR for further processing. The IR is frequently stored in a simple format, such as three-address code, in which each statement is either an assignment with at most three operands, a label, or a jump. The IR serves as a common exchange format between the front end and the subsequent optimization passes and also forms the back-end input. The front end also checks for errors such as syntax errors or undeclared identifiers and emits corresponding messages.

The front end's main components are the scanner, the parser, and the semantic analyzer. The scanner recognizes certain character strings in the source code and groups them into tokens. The parser analyzes the syntax according to the underlying source-language grammar. The semantic analyzer performs bookkeeping of identifiers, as well as additional correctness checks that the parser cannot perform. Many tools (for example, lex and yacc for Unix- and Linux-based systems) that automate the generation of scanners and parsers from grammar specifications are available for front-end construction.

IR optimizer

The IR generated for a source program normally contains many redundancies, such as multiple computations of the same value or jump chains. To a certain extent, these redundancies arise because the front end does not pay much attention to optimization issues. The human programmer might have built redundancies into the source code. These redundancies must be removed by subsequent optimization passes, which will do their job no matter what the source of the redundancies. For example, consider constant folding, which replaces compile-time constant expressions with their respective values. In the following C example, an element of array A is assigned a constant:

```
void f() {
  int A[10];
  A[2] = 3 * 5;
}
```

A front end might decompose the C source into a sequence of three-address code statements by adding temporary variables, and then it generates an unoptimized IR with two compile-time constant expressions. (Here I use the C-like three-address code IR notation of the Lance compiler system.[3]) One expression (3 * 5) was present in the source code; the other (2 * 4) has been inserted by the front end to scale array index 2 by the number of memory words occupied by an integer (here assumed to be 4):

```
void f() {
   int A[10], t1, t3, *t5;
   char *t2, *t4;
   t1 = 3 * 5;
   t4 = (char *)A;
   t3 = 2 * 4;
   t2 = t4 + t3;
   t5 = (int *)t2;
   *t5 = t1;
}
```

Now the IR optimizer can apply constant folding to replace both constant expressions by constant numbers, thus avoiding expensive computations at program runtime. Constant propagation replaces variables known to carry a constant value with the respective constant. Jump optimization simplifies jumps and removes jump chains. Loop-invariant code motion moves loop-invariant computations out of the loop body. Dead code elimination removes computations whose results are never needed in the given program.

A good compiler consists of many such IR optimization passes, some of which are far more complex and require an advanced code analysis.[4] Because there are strong interaction and mutual dependence between these passes, some optimizations enable opportunities for other optimizations and should be applied repeatedly to be most effective. However, theoretical limitations prevent any compiler from achieving an optimal IR for arbitrary source code.[5]

Back end

The back end (or code generator) maps the machine-independent IR into a behaviorally equivalent machine-specific assembly program. For this purpose, most of the original, statement-oriented IR is converted into a more expressive control/dataflow graph representation. Front-end and IR optimization technologies are quite mature thanks to the achievements of classical compiler construction, but the back end is often the most crucial compiler phase for embedded processors.

The back end usually includes three major phases. Code selection maps IR statements into assembly instructions. Register allocation assigns symbolic variables and intermediate results to the physically available machine registers. Scheduling arranges the generated assembly instructions in time slots, considering interinstruction dependencies and limited processor resources. Naturally, all three phases aim at maximum code quality within their optimization scopes.

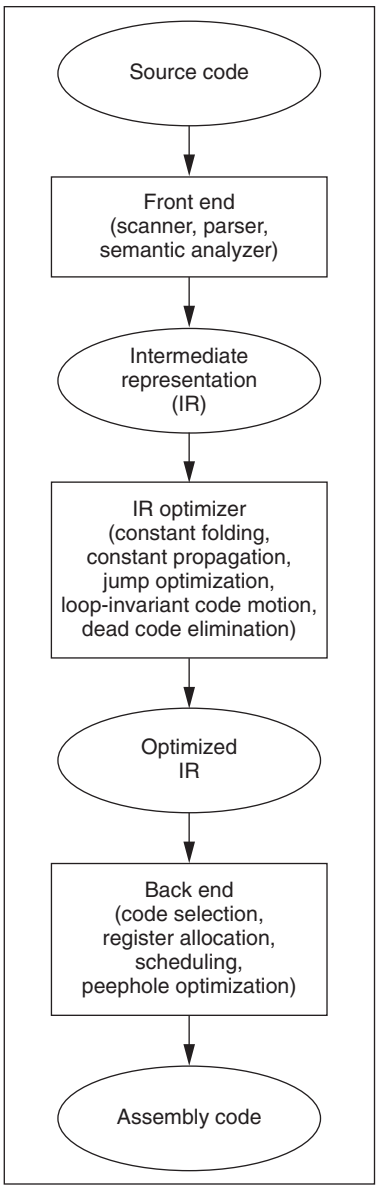

Figure 2. Coarse structure of a typical C compiler.

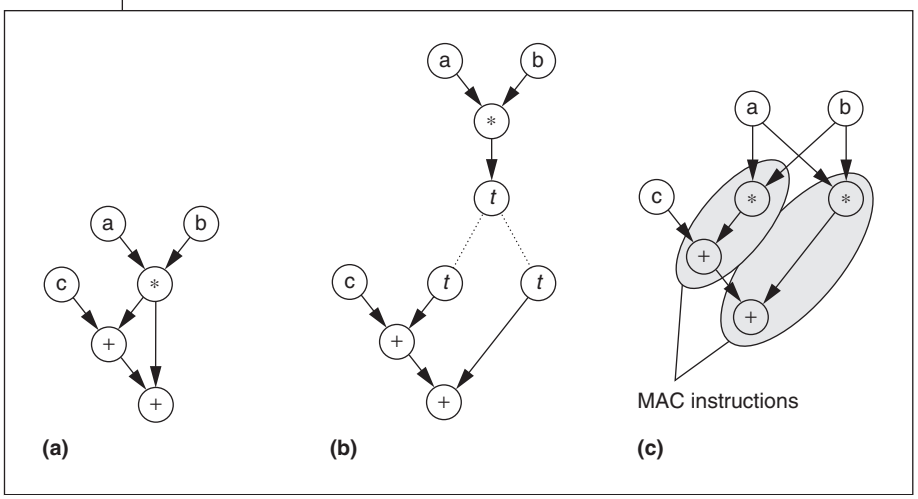

Figure 3. Code selection with multiply-accumulate (MAC) instructions: Dataflow graph (DFG) representation of a simple computation (a). Conventional tree-based code selectors must decompose the DFG into two separate trees (linked by a temporary variable *t*), thereby failing to exploit the MAC instructions (b). Covering all DFG operations with only two MAC instructions requires the code selector to consider the entire DFG (c).

The back end often includes a final peephole optimization pass—a relatively simple pattern-matching replacement of certain expensive instruction sequences by less expensive ones.

For each of the three major back-end passes, standard techniques, such as tree parsing, graph coloring, and list scheduling,[4] are effective for general-purpose processors with clean architectures. However, such techniques are often less useful for ASIPs with highly specialized instruction sets. To achieve good code quality, the code selector, for instance, must use complex instructions, such as multiply-accumulate (shown in Figure 3c) or load-with-autoincrement. Or it must use subword-level instructions (such as those used by SIMD and network processor architectures), which have no counterpart in high-level languages.

Likewise, the register allocator utilizes a special-purpose register architecture to avoid having too many stores and reloads between registers and memory. If the back end uses only traditional code generation techniques for embedded processors, the resulting code quality may be unacceptable.

Embedded-code optimization

C and its object-oriented extension C++ aim mainly at compiling for clean targets such as RISC processors with large homogeneous register files, rather than DSPs, for which there are sometimes more exceptions than rules in the programmer's manual. C and C++ are well known and widespread, and a large amount of legacy and reference code is written in these languages. Therefore, although there are other high-level languages, C and C++ will likely be dominant for programming embedded processors.

To support C and C++, compilers must be clever enough to map source code into highly efficient assembly code for ASIPs. Researchers have developed three main approaches to achieving this goal.

Dedicated code optimization techniques

Code optimization technology must keep pace with trends in processor architecture by continually extending the suite of known techniques. DSP code optimization research has progressed well in the past decade. For example, researchers have developed several variants of offset assignment techniques that exploit address generation units (AGUs) in DSPs and have successfully integrated these techniques in C compilers.[6] Although not all the available knowledge in this area has rippled down into production-quality compilers, it may now be time to switch research activities toward forthcoming embedded-processor families such as VLIW and network processors.

The following are promising dedicated optimization techniques for embedded processors:

- *Single-instruction, multiple-data instructions.* Recent multimedia processors use SIMD instructions, which operate at the subword level. SIMD instructions help ensure good use of functional units that process audio and video data, but they require special compiler support.
- *Address generation units.* Many DSPs are equipped with AGUs that allow address computations in parallel with regular com-

putations in the central data path. Good use of AGUs is mandatory for high code quality and requires special techniques not covered in classical compiler textbooks.

- *Code optimization for low power and low energy.* In addition to the usual code optimization goals of high performance and small code size, the power and energy efficiency of generated code is increasingly important. Embedded-system architects must obey sufficient heat dissipation constraints and must ensure efficient use of battery capacity in mobile systems. Compilers can support power and energy savings. Frequently, performance optimization implicitly optimizes energy efficiency; in many cases, the shorter the program runtime, the less energy is consumed. "Energy-conscious" compilers, armed with an energy model of the target machine, give priority to the lowest-energy-consuming (instead of the smallest or fastest) instruction sequences.[7] Since systems typically spend a significant portion of energy on memory accesses, another option is to move frequently used blocks of program code or data into efficient on-chip memory.[8]

New optimization methodologies

Despite the difficulties in compiler design for embedded processors, there is some good news: Unlike compilers for desktop computers, compilers for ASIPs need not be very fast. Most embedded-software developers agree that a slow compiler is acceptable, provided that it generates efficient code. Even overnight compilation of an application (with all optimization flags switched on) would make sense, as long as the compiler delivers its result faster than a human programmer.

A compiler can exploit an increased amount of compilation time by using more-effective (and more time-consuming) optimization techniques. Examples are genetic algorithms, simulated annealing, integer linear programming, and branch-and-bound search, which are beyond the scope of traditional desktop compilers. Researchers have successfully applied these techniques to code optimization for challenging architectures such as DSPs, and further work in this direction seems worthwhile.

Phase coupling

Many modern high-performance embedded processors have very long instruction word architectures. A VLIW processor issues multiple instructions (typically four to eight) per instruction cycle to exploit parallelism in application programs. Because all parallel functional units must be fed with operands and store a result, a VLIW processor normally requires many register file ports, which are expensive from a cost-performance viewpoint. Clustering the data path, with each cluster containing its own local units and register file, can circumvent this expense. Obtaining high code quality for clustered VLIW processors requires phase coupling—close interaction between code generation phases in a compiler—which is not implemented in traditional compilers.

The multiple phases of compilers must execute in some order, and each phase can impose unnecessary restrictions on subsequent phases. A phase-ordering problem exists between register allocation and scheduling:

- If register allocation comes first, false dependencies between instructions, caused by register sharing among variables, may occur, restricting the solution space for the scheduler.
- If scheduling comes first, the register pressure (the number of simultaneously required physical registers) may be so high that many spill instructions must be inserted in the code.

Obviously, both cases will negatively affect code quality. There are many other examples of phase-ordering problems. In fact, any phase ordering can lead to suboptimal code for certain compiler input programs. The separation into phases was introduced to meet the high-speed-compilation requirements of desktop systems. Clearly, if more compilation time is available, the compiler can use more-complex code generation algorithms that perform multiple tasks in a phase-coupled fashion.

Figure 4 (next page) shows the coarse architecture of a contemporary clustered VLIW DSP. Clustering a VLIW data path leads to an efficient hardware implementation but incurs a potential communication bottleneck between

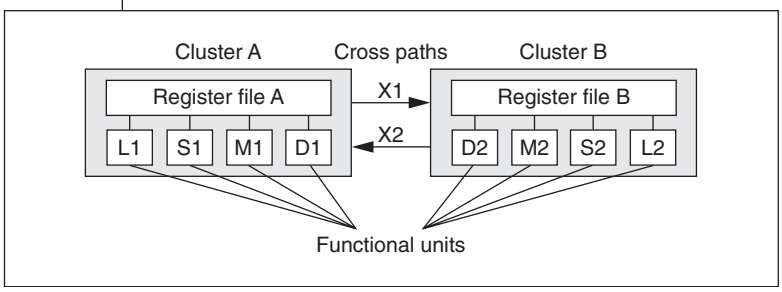

Figure 4. Clustered VLIW architecture of a TI C6201 DSP.

clusters. The instruction scheduler aims at balancing the computational load between clusters, but clusters must exchange values and intermediate results through a limited interconnection network. Hence, the compiler must take the required intercluster communication into account during scheduling through a tight interaction between the instruction assignment and scheduling phases.[9] This coupling requires more time-consuming code generation algorithms, but the result is good code, one of the most important goals for embedded software.

Retargetable compilers

To support fast compiler design for new processors and hence support architecture exploration, researchers have proposed retargetable compilers. A retargetable compiler can be modified to generate code for different target processors with few changes in its source code. To implement a retargetable computer, the designer provides it with a formal description (in a modeling language) of the target processor. In contrast, traditional compilers have a fixed, built-in processor model. A retargetable compiler reads the processor model as an additional input and reconfigures itself to emit assembly code for the current target machine. Thus, a new compiler can be generated quickly after each change in the ASIP architecture under design.

Retargetable compilers are cross-compilers. They run on a desktop host and generate assembly code for a certain target machine. Thus, besides a source program, they need a model of that machine as an input. There are different degrees of compiler retargetability:

- *Developer retargetability.* A programmer can port the compiler to a new target by changing most of its back-end source code. This requires in-depth knowledge of the compiler, usually possessed only by compiler developers.
- *User retargetability.* An experienced user can retarget the compiler by making only a few source code modifications. An external target description specifies most of the retargeting information in a special description language.
- *Parameter retargetability.* The compiler can be retargeted only within a narrow class of processors, so adjusting parameters such as register file sizes, word lengths, or instruction latencies is sufficient.

Many existing compilers are developer retargetable. Retargeting them requires an enormous supply of labor and expertise, normally not available in companies concentrating on ASIP and system development. Therefore, for architecture exploration, compilers should be user retargetable.

The most common way to achieve user retargetability is to employ a mixture of processor description languages and custom components. An example is the GNU (a recursive acronym for "GNU's not Unix") C compiler (GCC). Users can retarget GCC using a machine description file in a special format with parameters, macros, and support functions that capture all the machine details the GCC's description format doesn't cover (http://gcc.gnu.org). With this mechanism, users have ported the GCC to numerous targets, but due to the compiler's preference for RISC-like machines, few are embedded processors.

A similar restriction applies to the Little C Compiler. The LCC (http://www.cs.princeton.edu/software/lcc) has a more concise machine description format than the GCC but fewer IR-level code optimizations. The LCC's retargeting mechanism embodies no idea about the machine instruction semantics and lets the user translate primitive C source language operations into assembly language, as Figure 5 shows in an excerpt from the LCC's Intel x86 target machine description file.

Each line in the excerpt describes the assembly implementation of a certain C-level conditional jump statement. For instance, `LEI4`

denotes "jump on ≤," which compares two 4-byte integers. The two operands of **LEI4** are a memory variable (**mem**) and a placeholder for a register or constant (**rc**). Following this operation pattern, the user specifies an appropriate assembly instruction sequence (such as **cmp** and **jle**) within a string. This string, used in the assembly code emission phase, contains placeholders (**%0**, **%1**, **%a**) that are later filled with concrete operand names or constants and the jump target address. Finally, a numerical value (here 5) specifies the instruction "cost," which controls the optimization process in the code selector.

The user-retargetable class includes many other systems: commercial tools such as CoSy (http://www.ace.nl) and Chess (http://www.retarget.com) and research compilers such as SPAM (http://www.ee.princeton.edu/spam/) and Record (http://LS12-www.cs.uni-dortmund.de/). Some of these generate compilers from processor models in a hardware description language instead of custom machine description formats. This allows a closer coupling between compilation and hardware synthesis models and hence reduces the number of processor models needed. On the other hand, extracting the detailed instruction set from a potentially low-level HDL model is difficult.

Operational parameterizable compilers include the Trimaran (http://www.trimaran.org) and Tensilica's Xtensa (http://www.tensilica.com), both providing compiler retargeting in very short turnaround times. Thus, retargetable compilation for architecture exploration is somewhat available. However, because such compilers focus on special processor classes, they support exploration of only a limited architecture space. Leupers and Marwedel provide a comprehensive overview of retargetable compilers.[10]

```
stmt: EQI4(mem,rc) "cmp %0,%1\nje %a\n"   5
stmt: GEI4(mem,rc) "cmp %0,%1\njge %a\n"  5
stmt: GTI4(mem,rc) "cmp %0,%1\njg %a\n"   5
stmt: LEI4(mem,rc) "cmp %0,%1\njle %a\n"  5
stmt: LTI4(mem,rc) "cmp %0,%1\njl %a\n"   5
stmt: NEI4(mem,rc) "cmp %0,%1\njne %a\n"  5
```

Figure 5. Assembly language excerpt from LCC's Intel x86 target machine description file.

Outlook

About a decade ago, it was not yet clear that ASIPs would become important design platforms for embedded systems. By now, it is obvious that efficiency demands create a need for a large variety of ASIPs, which serve as a compromise between highest flexibility (general-purpose processors) and highest efficiency (ASICs). ASIPs require tool support for architecture exploration based on application-driven profiling.

Software development tools play an important role in mapping application programs to ASIPs. Although there is a clear trend toward C and C++ programming of ASIPs, compilers are not enough. A complete compiler-in-the-loop exploration methodology also requires assemblers, linkers, simulators, and debuggers. Hence, the retargetability of these tools is equally important for architecture exploration. There are already software systems that generate powerful low-level software development tools from a single, consistent ASIP model. For example, LisaTek's Edge tool suite (http://www.lisatek.com) supports the architecture exploration loop shown in Figure 1.[11] Users control simulation, debugging, and profiling via a graphical user interface.

Researchers are attempting to adapt the modeling languages used in such tool generators to C and C++ compilers, HDL models for synthesis, and hardware-software codesign and cosimulation. This adaptation will allow a fully integrated ASIP architecture exploration solution based on a single "golden" processor model, thereby avoiding the model consistency problems of current design flows.

An open issue is the tradeoff between retargetability and code quality for C compilers. In the past, a major argument against retargetable compilers was that they could not guarantee sufficient code quality and therefore were useless. Today, however, embedded-processor designers have widely recognized the need for architecture exploration and architecture-compiler codesign—and how retargetable compilers can meet that need.

Ultimately, what counts is not only the architecture's efficiency but also the overall hardware-software efficiency. This means that so-called superoptimizing retargetable compilers are not essential—at least not during archi-

tecture exploration. Instead, for complex applications, it is more important to explore many potential design points in little time. This is not possible with assembly programming. Even a nonoptimizing, yet flexible, compiler (if it is available quickly) significantly reduces software development time; the programmer can manually tune the few critical hot spots at the assembly level. Dedicated optimizations can be added to the compiler later to reduce the amount of hand tuning required.

RETARGETABLE COMPILERS for embedded processors are a promising yet challenging technology for system design. The first commercial tools are already in industrial use. Meanwhile, researchers are developing new processor-specific code generation techniques that continually narrow the code quality gap between C compilers and assembly programming. The approaches that achieve the right balance of flexibility, code quality, retargeting effort, and compatibility with existing design tools will be successful. ■

■ References

1. V. Zivojnovic et al., *DSPStone—A DSP-Oriented Benchmarking Methodology*, tech. report, Aachen Univ. of Technology, Germany, 1994.
2. M.R. Gary and D.S. Johnson, *Computers and Intractability—A Guide to the Theory of NP-Completeness*, W.H. Freeman & Co., New York, 1979.
3. R. Leupers, *Code Optimization Techniques for Embedded Processors*, Kluwer Academic, Boston, 2000.
4. S.S. Muchnik, *Advanced Compiler Design and Implementation*, Morgan Kaufmann, San Mateo, Calif., 1997.
5. A.W. Appel, *Modern Compiler Implementation in C*, Cambridge Univ. Press, Cambridge, UK, 1998.
6. S. Liao et al., "Storage Assignment to Decrease Code Size," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation* (PLDI 95), ACM Press, New York, 1995, pp. 186-195.
7. M. Lee et al., "Power Analysis and Minimization Techniques for Embedded DSP Software," *IEEE Trans. VLSI Systems*, vol. 5, no. 1, Mar. 1997, pp. 123-135.
8. S. Steinke et al., "Assigning Program and Data Objects to Scratchpad for Energy Reduction," *Proc. Design, Automation and Test in Europe* (DATE 02), IEEE CS Press, Los Alamitos, Calif., 2002, pp. 409-415.
9. J. Sánchez and A. Gonzales, "Instruction Scheduling for Clustered VLIW Architectures," *Proc. 13th Int'l Symp. System Synthesis* (ISSS 00), IEEE CS Press, Los Alamitos, Calif., 2000, pp. 41-46.
10. R. Leupers and P. Marwedel, *Retargetable Compiler Technology for Embedded Systems—Tools and Applications*, Kluwer Academic, Boston, 2001.
11. 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.

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

■ Direct questions and comments about this article to Rainer Leupers, Aachen Univ. of Technology, Inst. for Integrated Signal Processing Systems, SSS-611920, Templergraben 55, 52056 Aachen, Germany; leupers@iss.rwth-aachen.de.