Compilation techniques for energy-, code-size-, and run-time-efficient embedded software

Peter Marwedel, Stefan Steinke, Lars Wehmeyer

Abstract— **This paper is motivated by two essential characteristics of embedded systems: the increasing amount of software that is used for implementing embedded systems and the need for implementing embedded systems efficiently. As a consequence, embedded software has to be efficient. In the following, we will present techniques for generating efficient machine code for architectures which are typically found in embedded systems. We will demonstrate, using examples, how compilers for embedded processors can exploit features that are found in embedded processors.**

Keywords—**Embedded processors, compilers for embedded processors, embedded software, low power compiler**

I. Introduction

IT is generally accepted that information technology will find more and more applications in so-called embedded systems. Embedded systems are systems in which information processing is embedded into a larger system. For such systems, information processing is typically not immediately visible to the user and the main reason the user is buying such a system is not information processing. Examples of embedded systems include:

- *information processing in vehicles,*
- *consumer electronics,*
- *electronics in smart homes,*
- *process control.*

One of the key characteristics of embedded systems is the need for efficient realization. In this context, the term *efficiency* includes the following aspects:

- *Low weight*
- *Low energy consumption*

This requirement is expected to exist in the long run, due to a variety of reasons. These include the facts that battery technology is expected to improve only slowly, the need for light-weighted batteries, and high costs for cooling power-hungry processors.

- *Efficient use of the available communication bandwidth*

It is expected that the increasing communication between various *information appliances* will make bandwidth a very scarce resource.

- *Low cost*

In the past, many embedded systems were implemented using specialized hardware. Currently, however, there is a trend for implementing most of the functionality of embedded systems in software. This trend is motivated by the need for flexible solutions (products that can be updated easily).

Considering trends and characteristics together, it is obvious that processors and software for embedded systems have to be efficient, at least for most of the applications. Accordingly, highly optimized processors are already being used in embedded systems. For such processors, code compatibility with legacy code is less important than for PC-like systems. Therefore, these processors use a larger percentage of their functional units for performing useful computations and a smaller percentage for providing code compatibility. Examples of such processors include the ARM processor (see www.arm.com) and many processors for digital signal processing.

Looking at software generation techniques, the situation is different: software generation techniques for embedded systems are still far from being satisfactory. Problems do already start at the specification level. How do we specify an embedded system? Many proposals have been made so far. Still, an ideal specification language meeting all requirements does not yet exist. A number of attempts for meeting the most important requirements are shown in fig. 1.

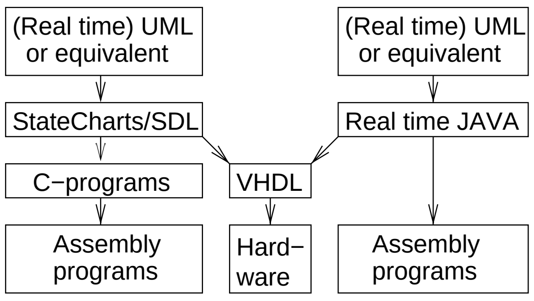


Fig. 1. Approaches for specifying embedded systems

According to Vaandrager [1], the size of embedded software is doubling every two years. Consequently, since the amount of embedded software engineers is limited, there is a need for specifying embedded software at a high level of abstraction, using compact specifications. This is reflected in the proposals for a real-time extension of the unified modeling language UML. At a slightly lower level, StateCharts and SDL are being used, as well as real-time extensions of Java. At still lower levels we find programs written in C, VHDL and assembly languages. Note that C is used as an intermediate step for most translations from higher level descriptions into machine languages[1]. Hence, the efficient generation of machine programs from higher level specifications requires that C-compilers generating efficient code for embedded processors are available.

However, C-compilers for embedded processors are notoriously known for their poor code quality. Detailed studies

The authors are with the Dept. of Computer Science, Informatik 12, University of Dortmund, 44221 Dortmund, Germany. E-mail: {marwedel|steinke|wehmeyer}@ls12.cs.uni-dortmund.de .

[1] Java is an exception in this context, but it would also benefit from knowledge about efficient translation techniques from C to machine languages of embedded processors.

of the code quality of available compilers were made in the context of the DSPStone project at the Technical University of Aachen [2]. In this project, manually generated assembly language programs were compared with compiled code. Some of the results are shown in fig. 2.

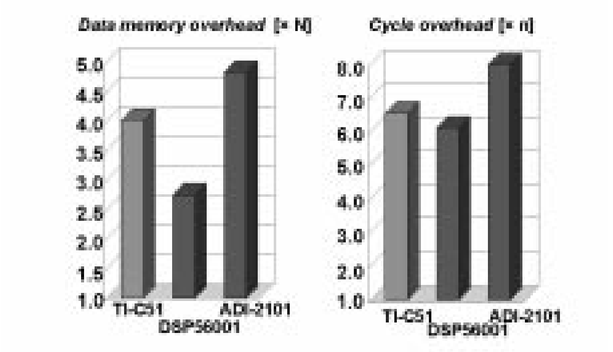


Fig. 2. Overhead of compiled code for ADPCM algorithm

According to fig. 2, data memory overhead for compiled code can almost reach a factor of 5. Even worse, cycle overhead can reach a factor of 8. That means, up to $\frac{7}{8}$ of the useful processor cycles can be needed just to compensate for the negative effect of the compiler. This is a serious problem for battery-operated systems. As a result, many embedded systems are still implemented in assembly languages, partially because the energy consumption of compiled code would be unacceptable.

Recently, many techniques aiming at avoiding the overhead of compiled code have been proposed. First techniques trying to reduce the energy consumption of programs will be described in section 2. In section 3 of this paper, we will provide examples of optimization techniques for DSP processors. Optimization techniques for multimedia- and very long instruction word (VLIW-) architectures will be presented in sections 4 and 5 respectively. Finally, there will be a conclusion.

II. Optimizations for low energy applications

Using electrical energy as efficiently as possible is extremely important for mobile applications. Energy saving compilers can help reducing the energy consumption. Possible compiler optimizations include the following:

• *Exploitation of the memory hierarchy:*
Small scratch pad memories are much more power efficient than larger memories. In order to compare the two types of memories, we have analyzed an ATMEL evaluation board containing an ARM7TDMI processor core. The core comes with a 4kB on-chip scratch pad memory. The board contains a 512 kB off-chip memory. For this configuration, we have measured current and energy requirements for executing instructions from the THUMB instruction set (16-bit instructions). Energy savings for using the small memory are significant. For example, 32-bit register LOAD instructions consume only 14.2% of the electrical energy if both the instruction as well as the data are fetched from the scratch pad memory (see table I[2]).

TABLE I
Energy consumed by loading 32-bit data using a 16-bit instruction

code	data	energy [nJ]	energy [%]
off-chip	off-chip	115.8	100
off-chip	on-chip	51.6	44.56
on-chip	off-chip	76.5	66.06
on-chip	on-chip	16.4	14.16

Therefore, a significant amount of energy can be saved if the existence of small scratch pad memories can be exploited by a compiler. For this purpose, we model each basic block and each variable as a memory segment i. For each segment, there is a corresponding size s_i. Using profiling, we can compute the gain g_i of moving segment i to the scratch pad memory. Let $x_i = 1$ if segment i is mapped to the scratch pad memory and $x_i = 0$ otherwise. Then, we try to maximize $\sum_i g_i \times x_i$ while respecting the size constraint $\sum_i s_i \times x_i \leq K$ where K is the size of the scratch pad memory. This problem is known as a knapsack problem. Hence, we added a solver for the knapsack problem to our compiler encc for the ARM processor. For some benchmark applications, we found energy reductions of about 45%, even though the size of the scratch pad memory was just a few percent of the total size of the application.

• *Modified cost function in instruction selection*:
Compilers typically use code size or cycle count as a cost function during various optimizations. For energy saving compilers, this cost function can be replaced by an estimate of the required energy. According to our experience, this typically results in a replacement of multiply instructions by shifts and adds (since multipliers are very power hungry), in a replacement of word load and store instructions by byte load and store instructions and a more aggressive use of the register file to avoid energy consuming memory accesses.

• *Exploitation of register pipelining*:
Register pipelining is an example of a technique that has been used in compilers for a couple of years and that can find a new application in low energy compilers. This particular optimization technique achieved an energy reduction of up to 26 % for the ATMEL board [3]. It should be worthwhile to also analyze the effect of other compiler optimizations on the energy consumed at run-time. Note that optimizations that reduce execution time typically also reduce the consumed energy (since energy = power × time). However, this is not necessarily true in all cases. We found optimizations that increase run-time and still decrease energy by reducing the power by a major factor.

• *Very long instruction word processors*:
VLIW architectures are using very long instruction words ($>$ 64 bits) for controlling multiple functional units in par-

[2]Units have been corrected in the current, revised version.

allel. With VLIW architectures, the task of detecting parallelism is moved from run time to compile time, resulting in more efficient system implementations. VLIW parallelism can be used either to improve the performance over sequential architectures or to reduce the clocking frequency. The latter is extremely important for low power applications, since the reduction of the clocking frequency allows a reduction of the supply voltage, which in turn reduces the power consumption.

• *Analysis of the influence of architectural decisions on energy consumption*:
An analysis of the influence of architectural parameters on the energy consumption can be used for optimizing architectures for certain applications or for application domains. The influence of the size of the register file on the energy consumption was studied by Jain et al. [4]. Similar studies for other architectural parameters will be very useful.

Compiler optimization for low energy applications is still a new topic and we expect significantly more results in this area.

III. Compilers for DSP processors

Processors for digital signal processing (DSP) come with a variety of special characteristics [5] including, for example:

- multiply/accumulate instructions,
- specialized register files,
- multiple arithmetic units,
- specialized DSP addressing modes,
- specialized address generation units,
- saturated arithmetic.

Compilers for DSP processors have to exploit these features in order to generate efficient code. Techniques for doing this can be very nicely demonstrated using address generation units as examples. Many DSP processors contain address generation units (AGUs) which are directly connected to the address input of the data memory (see fig. 3).

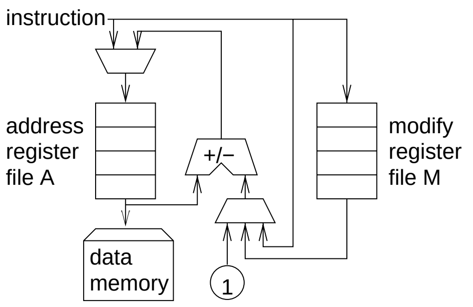
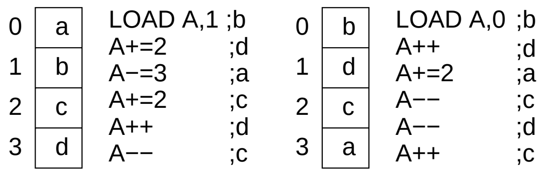


Fig. 3. AGU using special address registers

Addresses which are available in address registers do not need to be generated. They can be used in register-indirect addressing modes. This saves machine instructions, cycles and energy. In order to increase the usefulness of address registers, instruction sets typically contain auto-increment and -decrement options for most instructions using address registers. This possibility of generating addresses "for free" has an important impact on how variables should be laid out in memory. Fig. 4 shows an example.



```
LOAD A,1 ;b        LOAD A,0 ;b
A+=2     ;d        A++      ;d
A-=3     ;a        A+=2     ;a
A+=2     ;c        A--      ;c
A++      ;d        A--      ;d
A--      ;c        A++      ;c
```

Fig. 4. Comparison of memory layouts

We assume that in some basic block, variables `a` to `d` are accessed in the sequence (`b,d,a,c,d,c`). Accessing these variables with register-indirect addressing requires, first of all, loading the address of `b` into an address register (see fig. 4, left). Assuming that, for the time being, we just have a single address register `A`, we will then have to update the address register to point to variable `d`. This requires adding 2 to register `A`. For the next access we have to subtract 3. Auto-increment and -decrement modes can only be used for the last two accesses. In total, 4 instructions for calculating addresses are needed.

In contrast, for the layout in fig. 4 (right), 4 address calculations are auto-increment and -decrement operations which will be executed in parallel with some operation in the main data path. Only 2 cycles are needed for other address calculations.

How do we generate such clever memory layouts? Algorithms doing this typically start from an access graph (see fig. 5).

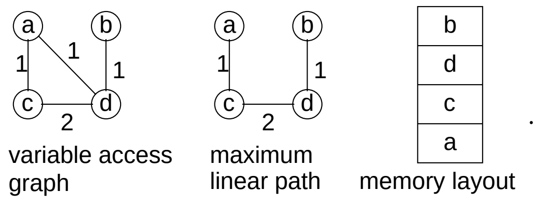


Fig. 5. Memory allocation for address register `A`

Such access graphs have one node for each of the variables and have an edge for every pair of variables for which there are adjacent accesses. The weight of such edges corresponds to the number of adjacent accesses to the variables connected by that edge.

Variables connected by an edge of a high weight should preferably be allocated to adjacent memory locations. The number of address calculations saved this way is equal to the weight along the edge. For example, if c and d are allocated to adjacent locations, then the last two accesses in the sequence can be implemented with auto-increment and -decrement operations.

The overall goal of memory allocation is to find a linear order of variables in memory maximizing the use of auto-increment and -decrement. This corresponds to finding a linear path in the variable access graph of maximum weight. Unfortunately, the maximum weighted path problem in graphs is NP-complete. Hence, it is common to use heuristics for generating such paths [6], [7]. Most of them are based on Kruskal's spanning tree heuristic. They start with a graph with no edges and then incrementally add edges with decreasing weight, always keeping the out-degree of all nodes to at most 2. The order of the variables in memory will then correspond to the order of the variables along the linear path.

The algorithm just sketched only covers a simple case. Extensions of this algorithm cover more complex situations, such as:
- $n > 1$ address registers [8],
- also using modify registers present in the AGU [8], [9],
- extension to arrays [10],
- larger auto-increment and -decrement ranges [7].

Memory allocation, as described above, improves both the code-size and the run-time of the generated code. Other optimization algorithms that have been proposed exploit other architectural features of DSP processors, such as:
- multiple memory banks [11],
- heterogenous register sets [12],
- modulo addressing,
- instruction level parallelism [13],
- multiple operation modes [14].

In addition, phase coupling techniques are required in order to cope with the irregular data paths of DSPs. Combining the optimization algorithms developed so far, it has been possible to come up with compilers with zero (see fig. 6 [15]) or even negative overhead of compiled code, at least for some processors.

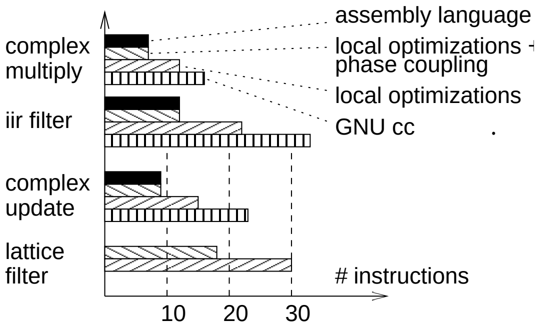


Fig. 6. Improvements for DSP code generation

IV. Optimizations for Multimedia-Architectures

Multimedia architectures exploit the fact that most processors come with a word-length corresponding to several color or audio values. For example, in the Pentium MMX architecture, four 16-bit entities can be stored and processed in a single 64-bit register. Four variable pairs are added at a time, instead of just one (see fig. 7).

A very challenging task is to use this feature in compilers. Only few algorithms have been described so far [16], [17], [18].

V. Optimizations for VLIW-Architectures

VLIW architectures require special compiler optimizations:
- Due to the supported parallelism, parallel access to register files must be available. This is not easy, since processors

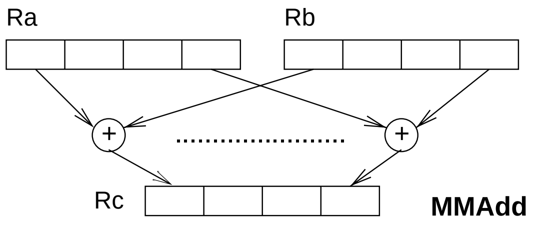


Fig. 7. Functionality of `MMAdd` instruction

such as the TMS C6xx family allow up to 8 operations to be started per cycle. Feeding these operations with a sufficient amount of register contents from the same register file would require a register file with many ports. These can hardly be implemented. Hence, the register file is split into two and each of the simpler register files has to provide the arguments for up to four operations at a time (which still is quite challenging). Four functional units are connected to each of the two simpler register files (see fig. 8).

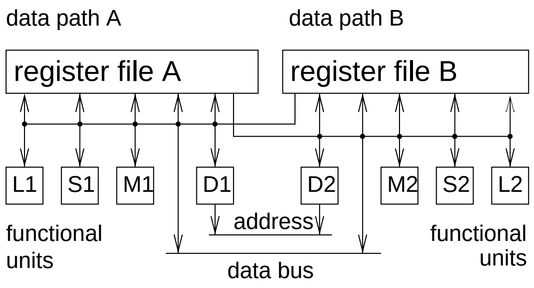


Fig. 8. Partitioned data paths of C6xx processors

A key optimization required for C6xx compilers is to allocate, at compile time, the functional unit that should execute a certain operation. This implies a partitioning of the operations into two sets [19], [20].
- VLIW processors frequently come with branch delay slots. For VLIW processors, the branch delay penalty is significantly larger than for other processors, because each of the branch delay slots could hold a full instruction packet, not just a single instruction. For example, for the C6xx, the branch delay penalty is $5 \times 8 = 40$ instructions. In order to avoid this branch delay penalty, most VLIW processors support predicated execution for a large number of condition code registers. Predicated execution can be employed for efficiently implementing small IF-statements. For large IF-statements, however, conditional branches are more efficient, since these allow mutual exclusion of THEN- and ELSE-branches to be exploited in hardware allocation. The precise tradeoff between the two methods for implementing IF-statements can be found with proper optimization techniques [21], [22], [23]
- Due to the large branch delay penalty, in-lining is another optimization that is very useful for VLIW processors. Traditional in-lining techniques rely on the user identifying functions to be in-lined. This is a problem in systems on a chip. For such systems, the size of the instruction memory is very critical. Hence, it is important being able to constrain the size of the instruction memory and to let

the compiler find out automatically which of the functions should be in-lined for a certain size of the memory. Two approaches for doing this are known [24], [25].

VI. Conclusion

Currently available compilers typically generate poor code for DSP, multimedia- and VLIW processors since they mostly do not exploit the special hardware characteristics for these processors. In this paper, we have provided a brief overview over new optimization techniques which aim at making assembly language programming of embedded systems obsolete. In addition to reducing the code size (which is very important for systems on a chip), new techniques for reducing the energy consumption of mobile software are being designed.

References

[1] F. Vaandrager, "Lectures on embedded systems," *in Rozenberg, Vaandrager (eds), LNCS, Vol. 1494*, 1998.

[2] V. Zivojnovic, J. Martinez, C. Schläger, and H. Meyr, "DSP-stone: A DSP-oriented benchmarking methodology," *Proc. of the Intern. Conf. on Signal Processing and Technology*, 1994.

[3] S. Steinke, Rüdiger Schwarz, Lars Wehmeyer, and Peter Marwedel, "Low power code generation for a RISC processor by RISC pipelining," Tech. Rep. 754, University of Dortmund, Dept. of CS XII, http://ls12-www.cs.uni-dortmund.de/ publications/ global_index.html, 2001.

[4] M. K. Jain, L. Wehmeyer, S. Steinke, P. Marwedel, and M. Balakrishnan, "Evaluating register file size in ASIP design," *CODES'01*, 2001.

[5] P. Marwedel, "Introduction," *in: P. Marwedel, G. Goossens (ed.): Code Generation for Embedded Processors*, Kluwer, 1995.

[6] S. Liao, S. Devadas, K. Keutzer, S. Tijang, and A. Wang, "Storage assignment to decrease code size," *Programming Language Design and Implementation (PLDI)*, 1995.

[7] A. Sudarsanam, S. Liao, and S. Devadas, "Analysis and evaluation of address arithmetic capabilities in custom DSP architectures," *Design Automation Conference*, 1997.

[8] R. Leupers and P. Marwedel, "Algorithms for address assignment in DSP code generation," *ICCAD*, 1996.

[9] R. Leupers and Fabian David, "A uniform optimization technique for offset assignment problems," *ISSS*, 1998.

[10] A. Basu, R. Leupers, and P. Marwedel, "Array index allocation under register constraints," *Int. Conf. on VLSI Design, Goa/India*, 1999.

[11] A. Sudarsanam and S. Malik, "Memory bank and register allocation in software synthesis for ASIPs," *Intern. Conf. on Computer-Aided Design (ICCAD)*, pp. 388–392, 1995.

[12] G. Araujo and S. Malik, "Optimal code generation for embedded memory non-homogenous register architectures," *8th Int. Symp. on System Synthesis (ISSS)*, pp. 36–41, 1995.

[13] R. Leupers and P. Marwedel, "Time-constrained code compaction for DSPs," *Int. Symp. on System Synthesis (ISSS)*, 1995.

[14] S. Liao, S. Devadas, K. Keutzer, and S. Tijang, "Code optimization techniques for embedded DSP microprocessors," *32nd Design Automation Conference*, pp. 599–604, 1995.

[15] S. Bashford and R. Leupers, "Phased-coupled mapping of data flow graphs to irregular data paths," *Design Automation for Embedded Systems*, vol. 4, pp. 1–50, 1999.

[16] R.J. Fisher and H. G. Dietz, "Compiling for SIMD within a single register," *Annual WS on Lang. & Compilers for Parallel Computing (LCPC)*, 1998.

[17] R. J. Fisher and H. G. Dietz, "The Scc compiler: SWARing at MMX and 3DNow!," *Annual WS on Lang. & Compilers for Parallel Computing (LCPC)*, 1999.

[18] R. Leupers, "Code selection for media processors with SIMD instructions," *DATE*, 2000.

[19] M. F. Jacome and G. de Veciana, "Lower bound on latency for VLIW ASIP data-paths," *International Conference on Computer-Aided Design (ICCAD)*, 1999.

[20] R. Leupers, "Instruction scheduling for clustered VLIW DSPs," *PACT, Philadelphia*, 2000.

[21] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, "Effective compiler support for predicated execution using the hyper-block," *MICRO-92*, 1992.

[22] D. I. August, W. W. Hwu, and S.A. Mahlke, "A framework for balancing control flow and predication," *MICRO-97*, 1997.

[23] R. Leupers, "Exploiting conditional instructions in code generation for embedded VLIW processors," *DATE*, 1999.

[24] J. Teich, E. Zitzler, and S.S. Bhattacharyya, "3D exploration of software schedules for DSP algorithms," *CODES'99*, 1999.

[25] R. Leupers, and P. Marwedel, "Function in-lining under code size constraints for embedded processors," *ICCAD*, 1999.