# Embedded Software: How To Make It Efficient?

Peter Marwedel
Universität Dortmund, Informatik 12
44221 Dortmund, Germany
Peter.Marwedel@udo.edu

## Abstract

*This paper stresses the importance of designing efficient embedded software and it provides a global view of some of the techniques that have been developed to meet this goal. These techniques include high-level transformations, compiler optimizations reducing the energy consumption of embedded programs and optimizations exploiting architectural features of embedded processors. Such optimizations lead to significant reductions of the execution time, the required energy and the memory size of embedded applications. Despite this, they can hardly be found in any available compiler.*

## 1. Introduction

Embedded systems are information processing systems, which are embedded into enclosing, larger products. Examples of embedded systems include information processing systems in cars, trains, airplanes, factories, communication equipment and consumer electronics. Embedded systems can be characterized by a number of common characteristics:

- The user is frequently not aware of this information processing, since embedded systems are not using well-known keyboards, mice and screens as their user interface. This characteristic has led to the term “the disappearing computer”.
- Embedded systems have to be dependable. Dependability includes the notions of reliability (the system should not break), maintainability (repairs should be easy), safety (it should not harm anyone) and security (it should keep private information private).
- Embedded systems are dedicated systems. This means: they are designed for a particular application or a small set of applications. Expandability is less an issue.
- Most embedded systems consist of a combination of special purpose hardware (which is necessary for the particular application) and processors and software. The latter is used to meet the flexibility requirements found for almost all of today’s applications.
- Many embedded systems are real-time systems, i.e. they have to meet real-time constraints.
- Portable systems are an important sub-class of embedded systems.
- Embedded systems have to be efficient. Efficiency concerns a number of different aspects:
  - Portable systems have to be energy efficient, due to the slow progress in battery technology.
  - Energy efficiency is an issue in general, due to the increasing performance requirements and cooling problems.
  - Embedded systems software should be compact (should have a small “memory footprint”). Embedded systems almost never come with large discs and space and size constraints also do not allow large memories.
  - Embedded software should be run-time efficient. If processor cycles are waisted, for example due to inefficient compilation, then faster processors are required in order to meet performance constraints. These are more expensive and require more electrical power.

Unfortunately, current design technology does, in many cases, result in efficient designs. Many compilers generate inefficient code [18], leading to a situation in which assembly language programming is being used [14]. The situation is slowly changing for standard DSP processors, but new problems are popping up with new VLIW processors. Software generation from higher-level specifications (for example in State-Charts or UML) is in an immature state.

These problems are not expected to disappear because of improvements in fabrication technology. Limited availability of energy and high performance requirement will impose tight constraints on the design technology. One result of these constraints is that compilers will be required which exploit the architectural features of such processors.

In the following, we will describe optimizations which aim at solving the problem of inefficient software. The results that will be presented provide a global view of the work performed in the embedded systems group at the University of Dortmund. In section 2 we will motivate the focus on C as the specification language. In section 3.1, we will provide an overview over techniques which can be applied before any compiler is used. In section 3.2. We will describe new techniques aiming at reducing the power consumption of embedded systems l. In section 3.3, we will provide a brief view on optimizations for digital signal processing. Section 3.4 is dedicated towards techniques for exploiting architectural features of modern embedded processors. Network processors will be briefly touched in section 3.5. Section 4 will contain conclusions.

## 2. Why focus on C?

There have been numerous discussions about the appropriateness of certain specification techniques and there have been many proposals for advanced specifications techniques and there have also been many proposals for advanced specification languages. An important observation is the following: Most existing design methodologies, if not based on assembly languages, use C as a specification language or use C as intermediate step. Fig. 1 is a graphical representation of the current situation.

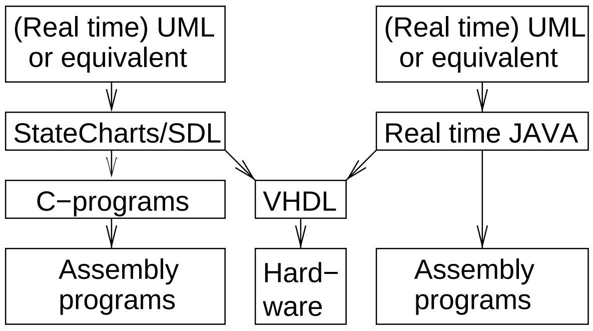

**Figure 1. C in the context of specifications**

Specifications in State-Chart-based languages as well as SDL specifications are first translated into C, before binary machine programs are generated. UML-based approaches typically need to be complemented with executable languages such as C++ or SDL and are also using C as an intermediate step. All these approaches cannot result in efficient code, unless the problem of efficiently translating C to machine languages is solved. Methodologies based on Java sometimes avoid C as an intermediate language. However, they do also benefit from efficient compilation technologies. For that reason, we will focus on efficient translation techniques from C in this paper. Of course, this does not mean that we do not need new tools and techniques for other languages.

## 3. Optimization techniques

### 3.1 High-level optimization techniques

Many software programs have not been written with maximum efficiency in mid. This applies especially to standards. For example, standards like the MPEG-2 standard use double precision floating point numbers, although this precision is actually not needed. The main reason is that the standards are intended just to serve as a reference specification. In other cases, arrays are much larger than required. This situation has led to the proposal of the so-called *software washing machine* [12]. The idea is that tools are required which remove all the "dirt" from the software. Our paper can be seen as a contribution in this direction.

First of all, we will present some high-level techniques. Currently, most of these are not available in any compiler and they will probably not be available in the majority of future compilers.

#### 3.1.1 Array folding

Frequently, only small portions of a arrays are needed at any point in time. Hence, large arrays can be mapped to much smaller arrays (see fig. 2).

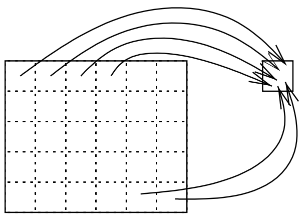

**Figure 2. Array folding**

Techniques for this have been studied in detail in the context of the data transfer and storage exploration (DTSE) project at IMEC [1].

#### 3.1.2 Loop splitting

Splitting loops into two can have a number of advantages. For example, some processors provide zero-overhead loop

(ZOL) instructions for loops of a limited size. Some loops may have to be split into two to take advantage of ZOLs. Loop splitting normally does not change the code of the loop body.

However, an extended form of loop splitting does allow modifying the loop body. Frequently, loop bodies contain if-statements and the tests performed by these statements can be simplified by separating out a few of the common cases. Margin handling with large arrays is an example of this. Many image algorithms use information in the immediate neighborhood of a pixel in order to improve the quality of the image. Special code is required at image borders, where such neighbourghs do not exist. It is then a good idea to split the loops in one which handles image borders and one which handles the remaining pixels.

Automatic detection of cases, in which loops should be split and the code adjusted accordingly, has been implemented by Falk [4]. His method is based upon a complex analysis of iterator spaces. Some results are shown in fig. 3.

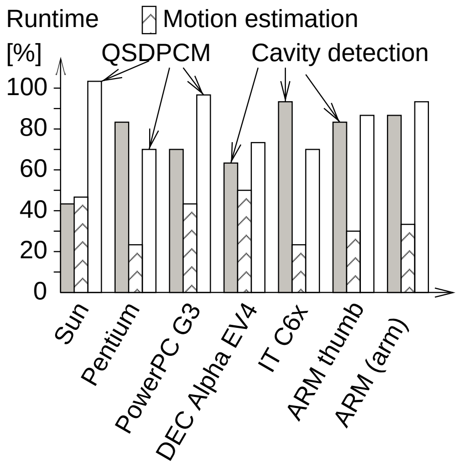

**Figure 3. Reduction of the run-time of 3 benchmarks by a automatic loop splitting and by simplification of loop bodies**

#### 3.1.3 Other forms of high level transformations

Other forms of high-level transformations have been analyzed in the ADOPT project at IMEC [3]. Also, Benini et al. describe the effect of some high-level transformations [2]. There are many additional contributions in this domain in the compiler community.

### 3.2 Energy-aware compilation

Many embedded systems are mobile systems which have to run on batteries. While computational demands on mobile systems are increasing, battery technology is expected to improve only slowly. Hence, the availability of energy is a serious bottleneck for new applications. Energy and power are closely related, since the latter is the time-integral of the former. Hence, techniques for saving energy are also frequently called low-power design techniques, even though a minimum energy design does not need to be the one which consumes a minimum amount of power. In this paper, emphasis is on saving energy.

Saving energy can be done at various levels, including the fabrication process technology, the device technology, circuit design, the operating system and the application algorithms. Adequate translation from algorithms to machine code can also help. High-level optimization techniques such as the ones presented in section 3.1 do also reduce the energy consumption. In this section, we will look at compiler optimizations which can reduce the energy consumption. Power models are very essential ingredients of all power optimizations. Using these, the following compiler optimizations have been used for reducing the energy consumption:

- Energy-aware scheduling: the order of machine instructions can be changed as long as the meaning of the program does not change. This optimization can be performed on the output generated by a compiler and therefore does not require any change to the compiler.

- Energy-aware instruction selection: typically, there are different instruction sequences for implementing the same source code. In a standard compiler, the number of instructions or the number of cycles is used as a criterion (cost function) for selecting a good sequence. This criterion can be replaced by the energy consumed by that sequence. Steinke and others found that low-power instruction selection reduces the energy consumption by some percent.

- Replacing the cost function is also possible for other standard compiler optimizations, such as register pipelining, loop invariant code motion etc. Possible improvements are also in the order of a few percent.

- Exploitation of the memory hierarchy:

  Smaller memories provide faster access and consume less energy per access. Figs. 4 and 5 show the access time and the energy required per access as a function of the size of the memory [15]. The same behavior can be observed for larger memories.

  Therefore, using a memory hierarchy does not only improve the average memory access time, but it also reduces the energy consumption. Commonly used caches, however, are not the ideal components. Set associative and associative caches require accesses to

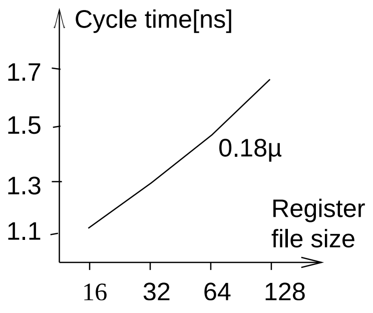

**Figure 4. Delay as a function of the memory size**

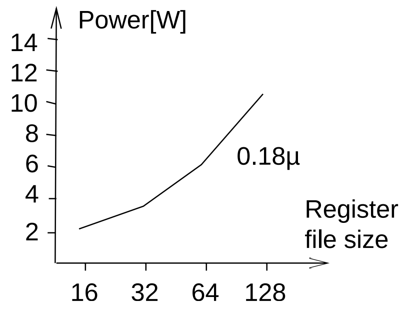

**Figure 5. Power consumption as a function of the memory size**

tags. These accesses consume energy. This energy increases with the set size and is largest for a fully associative cache. In contrast, scratch pad memories (SPMs) are mapped into the address space of the system. They can be accessed by just using the appropriate addresses. As a result, accesses to SPMs require just a fraction of the energy required for caches (see fig. 6).

Compilers can be designed such that the most frequently accessed variables and code segments are mapped to the SPM. This mapping can be a one-to-one mapping. An integer programming model leading to such a mapping was presented by Steinke et al. [16]. Results are shown in fig. 7.

Obviously, larger scratch pad memories lead to a reduced energy consumption in the main memory. Also, the energy required in the processor is also reduced, since less wait cycles are required. Supply voltages have been assumed to be constant.

As can be seen from fig. 7, substantial savings are possible. In all the cases observed by Steinke, the energy savings enabled by memory hierarchies are the largest.

Code can also be dynamically copied into the SPM, resulting in a many-to-one mapping. An integer pro-

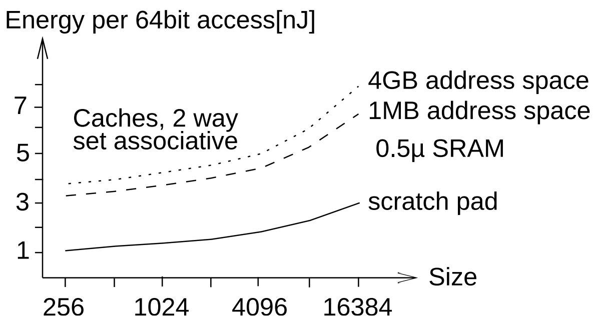

**Figure 6. Energy consumption for cache and scratch-pad memories**

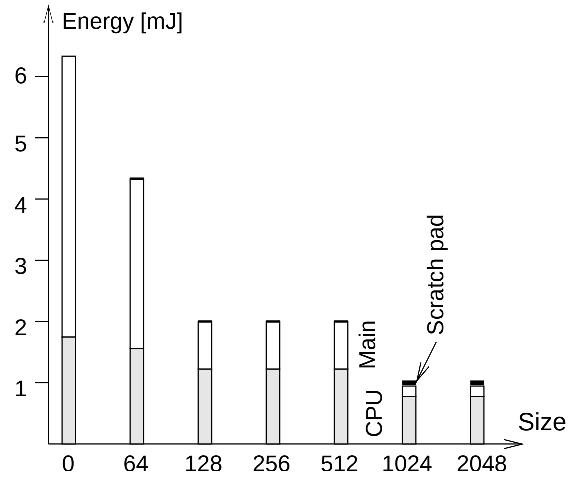

**Figure 7. Energy reduction by compiler-based mapping to scratch-pad for bubble sort**

gramming model reflecting this more general optimization problem was also proposed by Steinke [5]. Using this more general model, the energy gain can be increased.

## 3.3 Compilation techniques for digital signal processing

Many embedded systems include digital signal processing (DSP). DSP applications can be very efficiently implemented using special DSP processors. Such processors contain a number of specialized functions. These include [13]:

- special DSP instructions, such as multiply/add instructions,
- special addressing modes such as modulo addressing,
- saturating arithmetic,

- separate address generation units.

New optimization techniques take advantage of these features [9].

## 3.4 Compilation techniques for multimedia processors and VLIW

Registers and arithmetic units of many modern architectures are 64 bits wide. Therefore, two 32 bit data types, four 16 bit data types or eight 8 bit data types can be packed into a single register (see fig. 8).

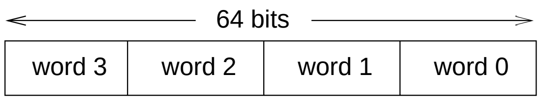

**Figure 8. Using 64 bit registers for packed data structures**

Arithmetic units can be designed such that they suppress carry bits at data type boundaries. Multimedia instruction sets exploit this fact by supporting operations on packed data types. This way, speed-ups of up to about eight over non-packed data types are possible. Data types are typically stored in packed form in memory. Unpacking and packing and packing is avoided if arithmetic operations on packed data types are used. Furthermore, multimedia instructions provide a more efficient form of overflow handling than standard instructions. Hence, the overall speed-up achieved with multimedia instructions can be significantly larger than the factor of eight enabled by operations on packed data types.

Frequently, however, packed data types are not supported by compilers and their use is therefore restricted to assembly language routines. In order to fully support packed data types, compilers must be able to automatically convert operations in loops to operations on packed data types. It is necessary not to ignore this potential for generating efficient software. Compiler algorithms exploiting operations on packed data types are extensions of vectorizing algorithms, originally developed for supercomputers.

VLIW (very long instruction word) architectures contain a major number of functional units which can operate in parallel. Parallelism is explicit at the instruction set level and the compiler is responsible for explicitly describing the parallelism. Moving the responsibility for identifying the parallelism to the compiler avoids the hardware overhead that is necessary for identifying parallelism in hardware. Hence, VLIW processors can perform more useful computations per mm$^2$ of silicon than super-scalar processors. Compared to such processors, VLIW processors can also frequently run with decreased clock frequencies and still meet the time constraints, due to the parallelism exploited. Decreased clock frequencies, in turn, allow reducing the power supply voltage and thereby saving power.

Parallel execution in VLIW architectures requires the availability of a large number of arguments in the same clock cycle. Multi-port memories containing a sufficient amount of ports are extremely expensive and also tend to being slow. Hence, modern VLIW architectures use partitioned register files. For example, the C6xx processor family from Texas Instruments includes two register files. Functional units are attached to each of the register files. The compiler is in charge of mapping operations to one of the functional units. Algorithms performing this mapping have been proposed [10, 6].

As an example, we are considering the M3-DSP processor [11]. The M3-DSP processor is a VLIW processors containing (up to) 16 parallel data paths. These data paths are connected to a group memory, providing the necessary arguments in parallel (see fig. 9).

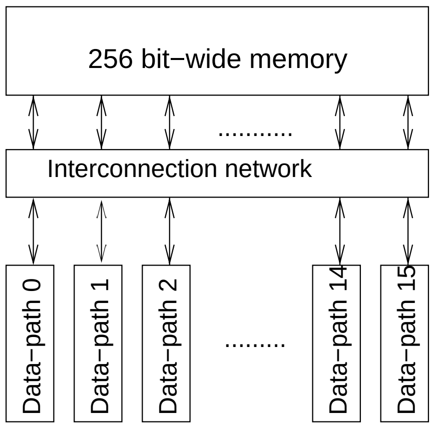

**Figure 9. M3-DSP (simplified)**

Automatic parallelization of loops for the M3-DSP requires the use of vectorization techniques [7, 8]. With such vectorization techniques, significant speedups (compared to the case of sequential operations) has been achieved (see fig. 10). For application dot_product_2, the size of the vectors was too small to lead to a speedup and no vectorization should be performed. The number of cycles can be reduced by 94 % for benchmark example if vectorization is combined with zero-overhead-loop detection.

## 3.5 Compilation techniques for network processors

Network processors are a new type of processors. They are optimized for high-speed Internet applications. Their instruction sets comprise numerous instructions for accessing and processing bit fields in streams of information. Typ-

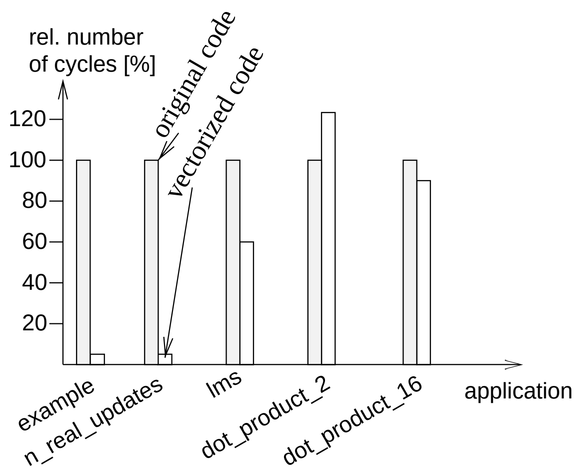

**Figure 10. Reduction of the cycle count by vectorization**

ically, they are programmed in assembly languages, since their throughput is of utmost importance. Nevertheless, network protocols are becoming more and more complex and designing compilers for such processors makes sense. The necessary bit-level details have been analyzed by Wagner [17]. Wagner obtained a 28 % performance gain by exploiting special bit-level instructions of a network processor.

## 4 Conclusion

In this paper, we have shown that software for embedded systems must be efficient (due to, for example, the limited availability of energy). For this reason, embedded processors contain a number of features which make them more efficient than standard processors used, for example, in PCs. This, in turn, means that advanced machine code generation techniques are needed, which exploit these architectural features of embedded processors.

We have shown that advanced translation techniques exist, which perform the necessary optimizations. Some high-level techniques are quite complex; they will possibly never be integrated into compilers. A new area is that of energy-aware compilation. The memory hierarchy enables the largest potential for energy-aware compilation. In order to exploit this hierarchy, it is no longer sufficient to provide compilers with a description of the instruction set. They also need information about the memory hierarchy.

Finally, we have briefly discussed compilation for VLIW and multimedia instructions sets. The use of such architectures is motivated by their inherent hardware efficiency. Exploiting such architectures does, however, require complex compilation strategies. These strategies enable significant performance and energy benefits.

We expect that more optimization features will be required in the future, due to the features of recently introduced processors, such as scalable voltages, speculation, memory hierarchies etc.

## Acknowledgment

This paper includes results from various members of the embedded system group at the University of Dortmund. Their contributions are gratefully acknowledged. Comprehensive information about these contributions is available from our web site: //ls12.cs.uni-dortmund.de.

## References

[1] F. Catthoor, S. Wuytack, E. D. Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle. *Custom memory management methodology*. Kluwer Academic Publishers, 1998.

[2] E.-Y. Chung, L. Benini, and G. D. Micheli. Source code transformation based on software cost analysis. In *ISSS*, pages 153–158, 2001.

[3] I. Desics group. Adopt. *http://www.imec.be/desics*.

[4] H. Falk. Control flow optimization by loop nest splitting at the source code level. Technical report, University of Dortmund, Dept. of CS XII, 2002.

[5] N. Grunwald, L. Wehmeyer, R. Banakar, M. Balakrishnan, and P. Marwedel. Reducing energy consumption by dynamic copying of instructions onto onchip memory. *ISSS*, 2002.

[6] M. Jacome, G. de Veciana, and V. Lapinksi. Exploring performance tradeoffs for clustered VLIW ASIPs. *International Conference on Computer-Aided Design (ICCAD)*, 2000.

[7] A. Krall. Compilation techniques for multimedia extensions. *International Journal of Parallel Programming*, 28, 2000.

[8] S. Larsen and S. Amarasinghe. Exploiting superword parallelism with multimedia instructions sets. *Programming Language Design and Implementation*, 2000.

[9] R. Leupers. *Code Optimization Techniques for Embedded Processors - Methods, Algorithms, and Tools Kluwer Academic Publishers*. Kluwer Academic Publishers, 2000.

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

[11] M. Lorenz, L. Wehmeyer, T. Draeger, and R. Leupers. Energy aware compilation for DSPs with SIMD instructions. *LCTES/SCOPES '02*, 2002.

[12] H. D. Man. Keynote session at DATE'02. *http://www.date-conference.com/conference/keynotes/index.htm*, 2002.

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

[14] P. Paulin, C. Liem, T. May, and S. Sutarwala. DSP design tool requirements for embedded systems: A telecommunications industrial perspective. *Journal of VLSI Signal Processing*, pages 23–47, 1995.

[15] S. Rixner, W. J. Dally, B. J. Khailany, P. J. Mattson, and U. J. Kapasi. Register organization for media processing. *HPCA*, 2000.

[16] S. Steinke, L.Wehmeyer, B.-L. Lee, and P. Marwedel. Assigning program and data objects to scratchpad for energy reduction. *DATE*, 2002.

[17] J. Wagner and R. Leupers. Advanced code generation for network processors with bit packet addressing. *Workshop on Network Processors (NP1)*, 2002.

[18] 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.