Energy aware Compilation for DSPs with SIMD Instructions

Markus Lorenz
University of Dortmund
Dept. of Computer Science 12
44221 Dortmund, Germany
Markus.Lorenz@uni-dortmund.de

Lars Wehmeyer
University of Dortmund
Dept. of Computer Science 12
44221 Dortmund, Germany
Lars.Wehmeyer@uni-dortmund.de

Thorsten Dräger
Techn. Universität Dresden
Vodafone Chair For Mobile Communication Systems
01062 Dresden, Germany
draeger@ifn.et.tu-dresden.de

ABSTRACT

The growing use of digital signal processors (DSPs) in embedded systems necessitates the use of optimizing compilers supporting special hardware features. In this paper we present compiler optimizations with the aim of minimizing energy consumption of embedded applications: This comprises loop optimizations for exploitation of SIMD instructions and zero overhead hardware loops in order to increase performance and decrease the energy consumption. In addition, we use a phase coupled code generator based on a genetic algorithm (GCG) which is capable of performing energy aware instruction selection and scheduling. Energy aware compilation is done with respect to an instruction level energy cost model which is integrated into our code generator and simulator. Experimental results for several benchmarks show the effectiveness of our approach[1].

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Code generation, Compilers, Optimization*

General Terms

Algorithms, Performance

Keywords

Vectorization, SIMD instruction, zero overhead hardware loop, energy minimization, DSP

1. INTRODUCTION

Digital signal processors (DSPs) are frequently used in *embedded systems* to flexibly account for specification modifications in late design phases. In order to meet given constraints with respect to execution time, code size and energy consumption, many programs are still written in assembly code. However, this is a very time consuming process which potentially leads to incorrect and hardly portable code. For this reason, there is an increased requirement for optimizing compilers which are adapted to special architectures and thus are capable of exploiting the irregular architecture features of DSPs.

Apart from the parallel execution of processor instructions (ILP = instruction level parallelism), DSPs like the TI C6201 [23], Philips TriMedia TM1000 [20] or the M3-DSP [7] support the execution of SIMD instructions (SIMD = single instruction multiple data). Exploitation of SIMD instructions (*vectorization*) exposes a potential for significant code improvements with respect to execution time and therefore also with respect to the energy consumption. However, there is still a need for suitable optimization techniques in this area.

It is a well known fact that DSP programs spend most of their execution time in loops. In order to reduce the existing loop overhead, DSPs usually provide executing a limited number of processor instructions in *zero overhead hardware loops* (ZOL). After initializing a special hardware register with the number of iterations to be executed, the instructions in the loop body can be executed without the conventional loop overhead. There is no need for additional instructions modifying the loop counter, checking the termination condition and jumping back to the first loop instruction. Thus, ZOLs lead to a reduction of execution time, code size and energy consumption.

The task of code generation can be subdivided into the subtasks *code selection* (CS), *instruction scheduling* (IS), *register allocation* (RA) and *address code generation* (ACG). Finding an optimal solution for each subtask usually means solving an NP-hard optimization problem. Due to the strong interdependencies among these subtasks, it is important for efficient code generation (particularly for DSPs) to perform all subtasks simultaneously by means of a complete *phase coupling.* With respect to an energy aware code generation, phase coupling is also important in order to avoid energy wasting decisions made at an early code generation phase.

In this paper we present loop optimizations for exploitation of SIMD instructions and for zero overhead hardware loops as well as a phase coupled energy aware code generator based

[1]This work has been sponsored by the German Research Foundation (DFG) and Agilent Technologies, USA.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
LCTES'02–SCOPES'02, June 19-21, 2002, Berlin, Germany.
Copyright 2002 ACM 1-58113-527-0/02/0006 ...$5.00.

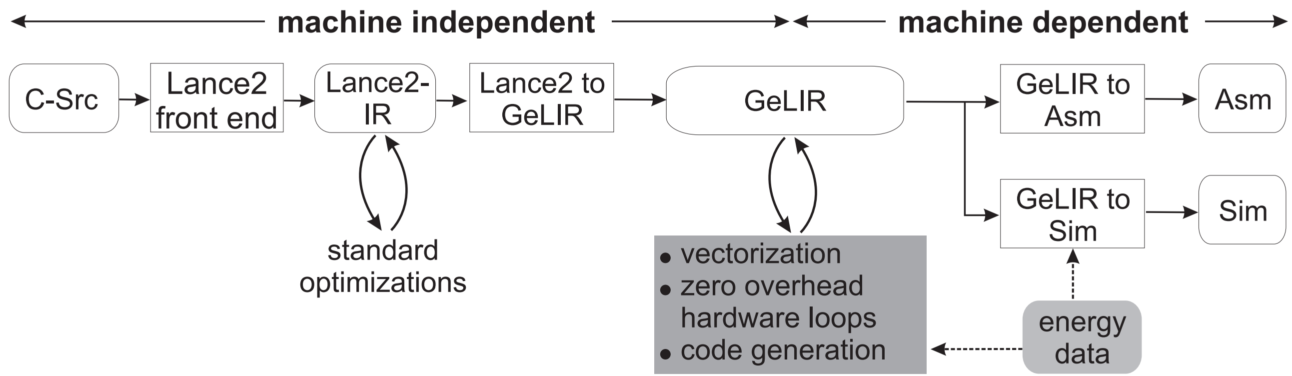

Figure 1: Compilation process and embedded optimizations

on a genetic algorithm (GCG). Due to an integrated instruction level energy cost model, our code generator is capable of reducing the energy consumption by suitable instruction selection and instruction scheduling.

The remainder of this paper is organized as follows: The next section gives an overview of our compiler framework. After that, the target architecture and the instruction level energy cost model is described. In section 4 we present the loop optimizations for exploitation of SIMD instructions (vectorization) and zero overhead hardware loops. The phase coupled energy aware code generator is described in section 5. Related work is given within these sections. We demonstrate the effectiveness of our approach in section 6 and conclude the paper with a summary.

2. COMPILER FRAMEWORK

An overview of the compilation process and our compiler framework is given in figure 1. At first, a front end (here: Lance2 [15, 13]) reads a given C source program and transforms it into a machine independent *intermediate representation* (IR). After running a set of standard optimizations, the Lance2-IR is mapped to our *generic low-level IR* (GeLIR) [9], which serves as exchange format for all succeeding transformations and optimizations. In order to propagate optimization results to other phases, GeLIR allows for storing machine dependent program and target architecture information like number and type of available registers, functional units, parallel execution possibilities and valid resource combinations of processor instructions. In order to represent the currently possible resource alternatives for a specific GeLIR graph node (or abstract operation (e.g. load or add) of the source program), it is possible to associate alternative resource combinations with such a graph node. In this representation, the process of code generation aims at restricting the set of resource alternatives by optimizing according to a specified cost function. This enables the implementation of modular and generic energy and performance aware optimization techniques like the vectorization of loops in order to exploit SIMD instructions, the exploitation of zero overhead hardware loops and finally the energy aware code generation using a phase coupled genetic code generator. Information about the energy consumption of processor instructions is stored in the GeLIR data structures and permits an efficient evaluation of different code sequences. Validation of optimizations can be performed by *compiled simulation* [10] of a specific GeLIR representation on different levels of abstraction. After validation, the current graph based GeLIR representation is mapped to sequential assembly code. Simulation is done by writing the given GeLIR program and target representation as C code which is then compiled using a traditional C compiler. Validation can also be done by compiling the original source program with a standard compiler and comparing the results of the generated binaries after execution. In order to assess the code quality, the GeLIR simulator provides information about the number of execution cycles, number of memory accesses and the energy consumption with respect to the specified energy data.

3. TARGET ARCHITECTURE

3.1 M3-DSP

The M3-DSP (figure 2) is an instance of a scalable DSP platform for mobile communication applications [7]. The platform permits a fast design of DSPs adapted to special applications. In order to meet constraints with respect to real-time processing, chip area, and energy dissipation, the platform supports among others, the following features:

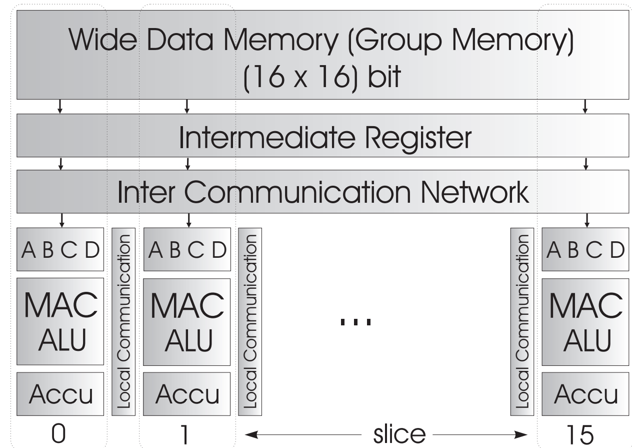

Figure 2: Coarse architecture of the M3-DSP

There is a scalable number of data paths that enable processing either on a single data path (slice 0) or on all data paths in parallel according to the SIMD principle. In the case of the M3-DSP there are 16 data path slices. In order to provide an effective use of all data path slices in parallel, the memory is organized as an on-chip *group memory*: Addressing one 16-bit data word means addressing an entire group of 16 such words. The addressed group is loaded into an intermediate register from which the values are distributed to the *group registers* in the data paths by an application-specific inter-communication network. The term group register denotes the set of those data path input registers in all slices with the same label (e.g. A or B in figure 2). Here, the greatest challenges in code generation are to take advantage of the full memory bandwidth and the SIMD data path instructions.
The loop overhead can be reduced by executing up to 256 processor instructions in a hardware loop whose number of iterations is restricted to a maximum of 32767.
Furthermore, the M3-DSP contains a 4-stage pipeline and is organized as a *very long instruction word* (VLIW) architecture which allows for an independent control e.g. for data manipulation, data transfer, program control, and the address generation unit.

3.2 Instruction level energy cost model

During the code generation process it is necessary to evaluate different code sequences. However, repeated simulations or measurements would mean an unacceptable overhead. Thus, a suitable cost model permitting a precise and quick evaluation of arbitrary instruction sequences is essential. Usually, an evaluation of a machine program with respect to execution time or code size can be done easily with the help of the instruction set. Unfortunately, energy data which is necessary for optimizing the energy consumption of an application is usually not available and thus has to be determined by simulation or measurement on the real chip. For instance, Tiwari [24] reports an instruction level energy model based on measurement of the energy consumption of a single instruction (*base energy cost*) and of the switching activities of successive instructions (*overhead energy cost*) which permits a quick and precise evaluation. Based on Tiwaris work, we have obtained a modified instruction level energy cost model for the M3-DSP with the help of measurements on the silicon chip [5]. A validation of our energy cost model shows a difference of less than 2% compared to measurements on the real hardware when executing the whole program.

As expected, the energy cost model shows that load and store instructions have a more significant contribution to the energy consumption than SISD instructions (SISD = single instruction single data) of the data path. SIMD instructions show the highest energy consumption (4-5 times more compared to SISD instructions) but perform up to 16 useful computations in parallel. For this reason, executing SIMD instructions will potentially reduce the total energy dissipation of an application. In addition, we expect a much smaller number of data transfers especially for handling the group memory. For example, storing a specific data word when processing is done in SISD mode means that there is a need for loading the correct memory group (containing 16 x 16 bit) into one of the group register files of the data paths, transporting the data word to a specific register element, and resaving the modified group into the memory.
This means that beside the reduction of execution time, optimization techniques which utilize all available data paths and reduce the number of memory accesses are worthwhile. In addition, the cost model shows that there are noticeable differences with respect to the energy consumption between different successive processor instructions. This leads to the assumption that the energy consumption can be reduced by performing a skillful instruction selection and instruction scheduling.

4. LOOP OPTIMIZATIONS

In this section we introduce the loop optimizations for the M3-DSP. They comprise the vectorization of loops in order to take advantage of the parallel data paths in section 4.1 and the exploitation of zero overhead hardware loops in section 4.2. As described before, both optimizations are integrated into the GeLIR framework. Thus, they read a given GeLIR representation and write back a modified representation.

4.1 Vectorization

The exploitation of parallel data paths and the handling of the group memory in order to take advantage of the SIMD capabilities of a processor is one of the greatest challenges in code generation for the M3-DSP.
To exploit SIMD instructions, hand written assembly code libraries may be available or *compiler known functions* in C programs can be applied. This has the advantage that the compiler can generate assembly code for the inserted SIMD instructions directly. The disadvantages are that the user potentially has to perform a lot of manual work and the programs are hardly portable to other target architectures. In order to avoid these disadvantages there are the following strategies:

Vectorization by loop analysis
The main idea of this classical strategy is to find loops which can be vectorized. If vectorization is possible, compiler known functions can be inserted into the source program through language extensions by the compiler [22]. In [19] this strategy is realized on the source code level by using a pattern matching based code transformation tool which performs a C to C transformation. However, the inserted assembly code instructions have a great impact on the code generation phase which is performed in a subsequent step.
Vectorized statements in loops are annotated as vectorized in the intermediate representation in [12] for the UltraSPARC VIS instruction set and in [4] for the T0 vector processor.
In all cases, loop transformations like *loop fission*, *strip mining*, *reduction recognition* or *scalar expansion* [2] are performed in order to increase the number of loops which can be vectorized. However, complex techniques for analyzing the source program are required.

Vectorization on basic block level
Due to the complex analysis, [12, 14] propose to perform the vectorization on basic block level. Here, the parallelism is increased by unrolling a loop n times. The loop unrolling factor n can be determined for instance by the number of parallel data paths of the processor. After that, instructions which can be executed as SIMD instructions are packed

into groups. The approach described in [16] makes use of SIMD instructions in conjunction with the instruction selection phase by formulating alternative covers of tree nodes with processor instructions as an integer linear programming problem. The disadvantage of this approach is the expected high runtime for solving complex systems of equations for large values of n. Thus, in [16] an unrolling factor between 1 and 3 is chosen, whereas in the case of the M3-DSP it would be necessary to use a factor of 16. Vectorization on basic block level by loop unrolling has the advantage that the analysis is less complex compared to the classical vectorization method. However, applying this method bears a risk of increasing code size for loops which cannot be vectorized.

The on-chip memory organization of the M3-DSP as a group memory provides a high memory bandwidth and can potentially speed-up the performance. But, due to the following reasons, exploitation of SIMD instructions for the M3-DSP is more complicated than for other processors:

- The on-chip group memory of the M3-DSP is partitioned into several groups, each of them containing 16 data words. Thus, addressing one single data word means to access that (partitioned) memory group the addressed data word belongs to.
- The M3-DSP contains no separate vector and scalar unit. For this reason, the SISD processing of scalar data has to be done in the special data path (slice 0) which is also used for SIMD processing.

Our vectorization strategy is based on the classical approach, and can be performed without the use of special language extensions. The communication with the code generator is done by performing restrictions with respect to the available machine operation alternatives which can cover a specific GeLIR graph node. This entails further restrictions e.g. with respect to the available register alternatives. The advantage is, that it is possible to make precise defaults by preserving potential alternatives for example with respect to register files.
Before starting the vectorization process all data transfers which *potentially* can occur between two graph nodes are inserted into the GelIR data structure and an initial cover of the GeLIR graph nodes with resource alternatives is generated. After that, those resource alternatives are automatically eliminated whose selection would lead to an invalid machine program. This concerns on the one hand the elimination of invalid resource combinations of a specific graph node and on the other hand the warranty of valid data transfer paths between two data dependent graph nodes. After this is done we employ the following vectorization strategy:

1. Compute an optimized address assignment of arrays and scalar variables. Different arrays are stored in different memory groups whereby the elements of a specific array are stored in consecutive memory addresses. Assigning scalar variables to memory groups is done by a special partitioning method based on a genetic algorithm that minimizes the number of memory accesses [18].
2. Perform a loop detection in order to find potential candidates for vectorization. This involves the tasks of finding the set of basic blocks in the loop, determining the loop bounds as well as the increment of the loop induction variable.
3. Determine dependencies between available array memory accesses in the loop by performing a *δ-array data flow analysis* [6]. After that, it is possible to decide whether a vectorization will preserve the data flow dependencies.
4. Check dependencies between scalar variables and array accesses. If this analysis fails it is possible to perform additional loop transformations like *loop fission* or *scalar expansion* to still enable vectorization.
5. Constrain the set of machine operation alternatives for the GeLIR graph nodes which can be vectorized. Note that this will usually also cause restrictions to other resource alternatives.

For illustration of the vectorization optimization, consider the following C source program, assuming arrays A and B have a size of 64 elements:

```
for(i = 0; i < 64; i++)
 {
   x = A[i];
   x = x + 2;
   B[i] = x;
 }
```

Figure 3 a) contains the (simplified) GeLIR code for our example program. The `for`-loop is split into several parts (relevant changes are marked bold). Existing operation alternatives are given as comments next to the respective code fragments[2]. Note that due to the group memory only an SIMD operation alternative (`SIMD_Load` or `SIMD_Store`) is given for each memory access. After performing the analysis mentioned above, it is recognized that this loop can be vectorized. Thus, for instance the set of operation alternatives for the first add-operation is restricted such that only an `SIMD_Add` may be chosen to cover the expression (s. figure 3 b)). Beside the restrictions of the machine operation alternatives, modifying the increment of the induction variable (here: i) is also necessary. Now, with each loop iteration the induction variable i is incremented by 16 instead of 1. This leads to valid code because with each memory access a group of 16 data elements is accessed and processed in subsequent instructions. Now, there is only need for 4 loop iterations instead of 64.

4.2 Zero overhead hardware loops

In order to reduce the loop overhead, it is desirable to perform as many instructions as possible in *zero overhead hardware loops* (ZOL). Strategies making use of ZOL can be partitioned into two sets, depending on whether this optimization is performed *before* or *after* the code generation process (e.g. [25, 21]). Usually, it is much easier to recognize loops before performing code generation since the program code is more structured. However, in this compilation phase there is no knowledge about the final number of processor instructions to be embedded within the hardware loop.

[2]For a better representation other resources like registers are not pictured.

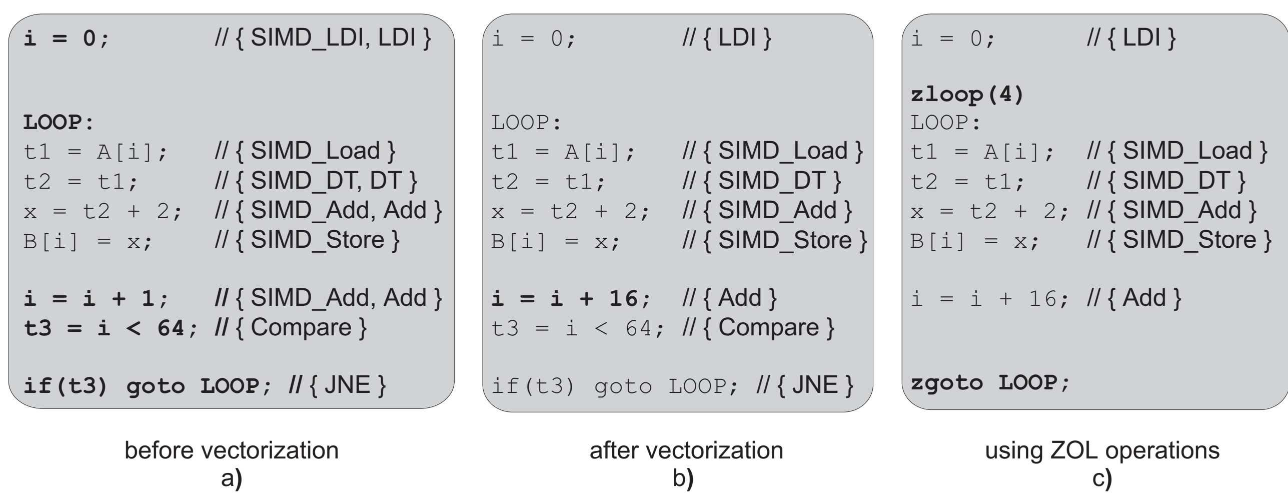

Figure 3: Loop optimizations: Vectorization and zero overhead hardware loops

In our case, the exploitation of zero overhead hardware loops is performed before the code generation phase. After performing loop detection, the number of iterations is determined with the help of the increment and the loop bounds, if possible. If the number of iterations is greater than an architecture specific limit, the use of a ZOL is not possible in that way. In the other case two specific predefined GeLIR operations (s. figure 3 c)) will be inserted into the intermediate representation: First, this concerns the machine operation `zloop` for initializing the hardware loop with the number of loop iterations (here: 4). Second, the machine operation `zgoto` is inserted which marks the end of the loop. No further processor instructions are necessary for testing the termination condition and modifying the loop counter. Finally, after insertion of these specific loop operations, a *dead code elimination* is performed, which eliminates the redundant loop control code. In figure 3 c) this concerns the conditional jump. At this point it is still possible that the maximum number of processor instructions embedded within the hardware loop exceeds the allowed size limit. In this case, the GeLIR data structure enables the user to continue the optimization process from an earlier point of compilation.

5. ENERGY AWARE CODE GENERATION

Due to the poor techniques for handling irregular processor architectures, the code generated by DSP compilers is often insufficient with respect to execution time, code size, and energy consumption. Since traditional techniques often perform a tree based code selection [26]. In contrast to that, performing a graph based code selection offers an immense optimization potential [3, 17]. In addition, traditional code selection techniques only achieve a restricted phase coupling: Tree based code selection techniques generate optimal code for trees but this concerns only sequential code which has to be compacted in a separate optimization phase. In addition, the generated code does not contain spill code[3] which is inserted by the register allocation step. Thus, energy aware optimizations made in early code generation phases are potentially nullified in subsequent optimization steps. For instance, the insertion of spill code in a separate phase drastically changes the switching activities of successive instructions. For this reason, there is much unused optimization potential using a traditional tree based code selection technique. However, performing a graph based code selection and a phase coupled code generation means solving a more complex problem. This makes the use of special optimization methods necessary which are capable of finding optimal or near optimal solutions in a huge search space in polynomial time.

Genetic algorithms (GAs) have proven to solve complex optimization problems by imitating the natural evolution process (see e.g. [1, 11] for an overview). A *population* of a GA consists of several individuals, each of them representing a potential solution of the optimization problem. The representation of an individual is given by a *chromosome* which is subdivided into *genes*. The genes are used to encode the variables of the optimization problem. This means that finding a suitable combination of *alleles* (concrete values) for the genes is the same as finding good solutions of the optimization problem.
It is a very important characteristic of genetic algorithms that suitable gene material is passed to the subsequent generations. This permits to revise unfavorable decisions made in a previous optimization phase. For this reason, genetic algorithms are adequate for solving non-linear optimization problems like phase coupled code generation.

In our case, the code generation process is subdivided into the following steps:

1. Perform the subtasks of instruction scheduling (including compaction), code selection and register allocation simultaneously with respect to a given cost (or *fitness*) function for each basic block of a function using the genetic algorithm driven code generator (GCG) which is described in more detail in [17].
2. Generate address code for each memory access sequence

[3]Spill code is necessary in order to move variables into the memory locations if the available number of physical registers is exceeded.

of a basic block by making maximal use of auto-increment and auto-modify instructions which can be performed in parallel to other machine operations.

3. Perform a compaction of the inserted address code with respect to a given cost function using GCG.

In each step, the optimization first reads a given GeLIR data structure and writes back a modified GeLIR data structure after finishing the optimization.

The *fitness* function of a genetic algorithm represents the objective function of the underlying optimization problem and thus has an essential impact on the optimization progress of the genetic algorithm. Hence, GCG enables code generation according to different objectives by specifying a suitable fitness function:

- Minimization of execution time can be done by counting the number of execution cycles. A high number of cycles corresponds to low fitness.
- Energy aware optimization is possible by computing the energy consumption with respect to our instruction level energy model for each individual.

Furthermore, energy aware optimizations can be performed for given real-time constraints (e.g. maximum number of execution cycles of a basic block) by adding a penalty for every constraint violation. Thus, solutions not meeting the constraints will be assigned a lower fitness than others.

6. EXPERIMENTAL RESULTS

In the following, experimental results for the loop optimizations vectorization and zero overhead hardware loops (ZOL) as well as results of the energy aware code generator GCG are presented. All presented data was generated using GCG and the GeLIR simulator with the energy cost model of the M3-DSP.

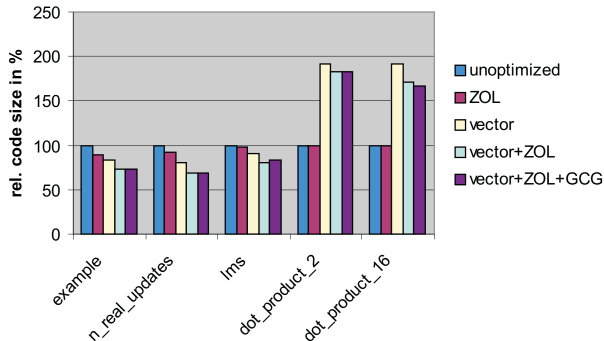

Figure 4: Vectorization: Code size

The benchmarks are taken from the DSPstone benchmark suite [27], with the existing pointer based memory accesses converted into array accesses by the pointer conversion algorithm described in [8]. In all cases, we present results for a specific optimization technique by enabling this optimization and comparing it with the unoptimized results. In order

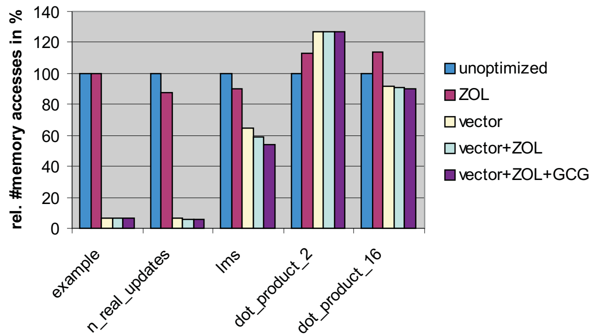

Figure 5: Vectorization: Memory accesses

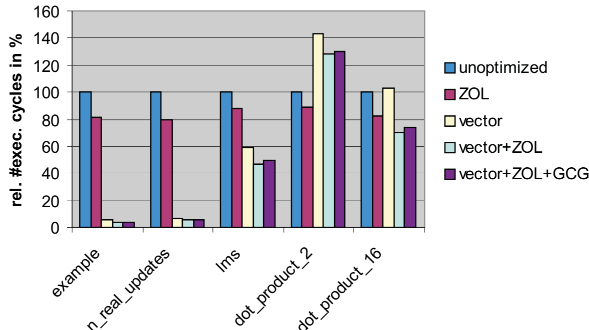

Figure 6: Vectorization: Execution cycles

to show the impact of the energy optimization performed in GCG all energy optimized results are marked by the suffix GCG. The runtime of the whole compilation process lies between 47 seconds for the `n_real_updates` and 210 seconds for the `lms` benchmarks. For a better representation, the optimization results are given in relation to the unoptimized case (= 100%).

Results for benchmarks which were vectorized by our compiler are presented in figures 4 to 7. The benchmarks comprise the `example` program given in section 4, the `n_real_updates` and the `lms` benchmark which can be vectorized without additional loop transformations. In order to vectorize the `dot_product`[4] benchmark, we first applied a *reduction recognition* loop transformation.

The results for the `example`, `n_real_updates` and `lms` benchmark demonstrate the huge optimization potential for vectorization. Thus, for instance the energy consumption is reduced by 94% for the `example` and `n_real_updates` routines, and by 49% for the `lms` routine. Results for the `dot_product` benchmarks show the overhead caused by the reduction recognition loop transformation. However, in the case of the `dot_product_16` benchmark, the subsequent vectorization and ZOL leads to an improvement with respect to

[4]The appendix (2 and 16) given by these benchmarks reflects the number of array elements to be processed.

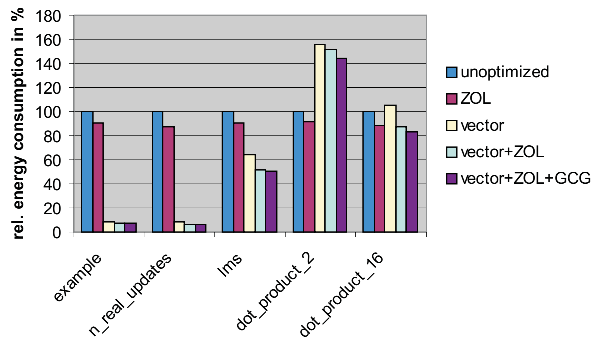

Figure 7: Vectorization: Energy consumption

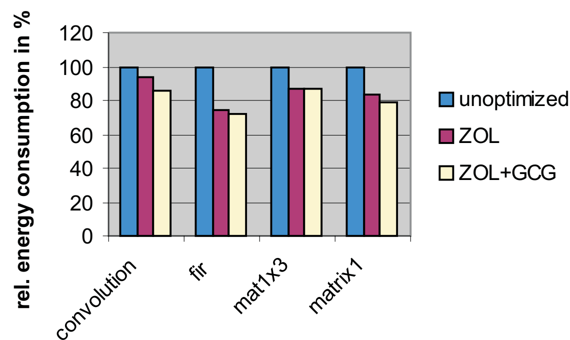

Figure 9: ZOL: Energy consumption

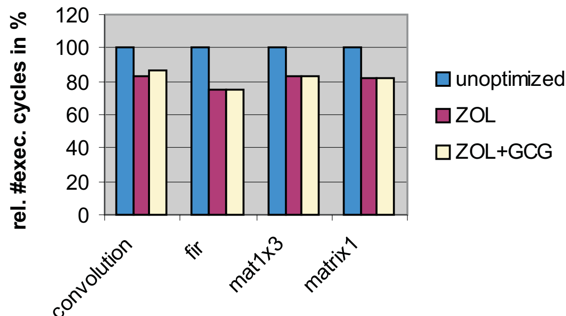

Figure 8: ZOL: Execution cycles

the optimization criterions memory accesses, execution time and energy consumption. The use of GCG (vector+ZOL-+GCG) leads to a further improvement of 4% on average.

In figure 8 and 9, results for ZOL are given for those benchmarks which cannot be vectorized by our compiler.
The number of execution cycles by making use of ZOL is reduced by 16% to 25% for these benchmarks. This trend is also confirmed with respect to energy consumption (c.f. figure 9). In addition, our energy aware code generator (ZOL+GCG) can reduce the already optimized energy consumption by an average of 5% compared to the results by using only ZOL. The code size is reduced on average by 7%, whereas the number of memory accesses did not change for these benchmarks.

7. CONCLUSIONS

The growing use of DSPs in embedded systems necessitates optimizing compilers supporting special hardware features. The organization of the M3-DSP on-chip memory as group memory provides a high memory bandwidth, but makes exploitation of SIMD capabilities more complicated compared to other processors. Apart from the traditional optimization goals of minimizing execution time and code size, low energy is also an important criterion today. In this paper we have presented loop optimizations for exploitation of SIMD instructions and for zero overhead hardware loops with the goal of minimizing the execution time and the energy consumption of embedded applications. In addition, we proposed a phase coupled code generator (GCG) based on a genetic algorithm. Due to an integrated instruction level energy cost model, GCG is capable of performing an energy aware instruction selection and scheduling.
Results show that our optimizations can drastically improve the code quality. Thus, for the vectorized benchmarks the total energy consumption is reduced by 42% on average and for the other benchmarks by 19% on average.
However, up to now vectorization can be performed only for a few benchmarks. For this reason, future work will deal with extending the number of programs which can be vectorized by our compiler.
All techniques are integrated into a common compiler framework which allows adaptations of the presented techniques for other processor architectures in an easy way.

8. ACKNOWLEDGMENTS

The authors would like to thank Martin Horst for his work made in the diploma thesis concerning the loop optimizations.

9. ADDITIONAL AUTHORS

Additional author: Rainer Leupers (Integrated Systems for Signal Processing, Aachen University of Technology, email: `leupers@iss.rwth-aachen.de`)

10. REFERENCES

[1] T. Bäck. *Evolutionary Algorithms in Theory and Practice*. Oxford University Press, 1996.

[2] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler Transformations for High-Performance Computing. *ACM Computing Surveys*, 26(4), Dec. 1994.

[3] S. Bashford and R. Leupers. Constraint driven Code Selection for Fixed-Point DSPs. In *36th Design Automation Conference (DAC)*, 1999.

[4] D. J. DeVries. *A Vectorizing SUIF Compiler*. PhD thesis, University of Toronto, June 1997.

[5] T. Dräger and G. Fettweis. Energy Savings with Appropriate Interconnection Networks in Parallel

DSP. In *3. Workshop zum DFG-Verbundprojekt "Grundlagen und Verfahren verlustarmer Informationsverarbeitung VIVA", Chemnitz*, 2002.

[6] E. Duesterwald, R. Gupta, and M. Soffa. A Practical Data Flow Framework for Array Reference Analysis and its Use in Optimizations. In *Proceedings of SIGPLAN Conference on Programming Languages Design and Implementation*, pages 68–77, June 1993.

[7] G. Fettweis, M. Weiss, W. Drescher, U. Walther, F. Engel, and S. Kobayashi. Breaking new grounds over 3000 MOPS: A broadband mobile multimedia modem DSP. In *Proc. of ICSPAT'98*, pages 1547–1551, Toronto, Canada, 1998.

[8] B. Franke and M. O'Boyle. Towards Automatic Parallelisation for Multi-Processor DSPs. In *Workshop on Software & Compilers for Embedded Systems SCOPES*, 2001.

[9] Generic Low Level Intermediate Representation (GeLIR). ls12-www.cs.uni-dortmund.de/research/gelir/.

[10] A. Hoffmann, T. Kogel, A. Nohl, G. Braun, O. Schliebusch, O. Wahlen, A. Wieferink, and H. Meyr. A Novel Methodology for the Design of Application-Specific Instruction-Set Processors (ASIPs) Using a Machine Description Language. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1338 – 1354, 2001.

[11] J. H. Holland. *Adaption in Natural and Artificial Systems*. MIT Press, 1992.

[12] A. Krall. Compilation Techniques for Multimedia Extensions. In *International Journal of Parallel Programming*, volume 28, 2000.

[13] LANCE2. http://ls12-www.informatik.uni-dortmund.de/lance/.

[14] S. Larsen and S. Amarasinghe. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, Vancouver, Canada, 2000.

[15] R. Leupers. *Code Optimization Techniques for Embedded Processors*. Kluwer Academic Publishers, Boston, 2000.

[16] R. Leupers. Code Selection for Media Processors with SIMD Instructions. In *Design Automation and Test in Europe (DATE)*, 2000.

[17] M. Lorenz, T. Dräger, R. Leupers, P. Marwedel, and G. P. Fettweis. Low-Energy DSP Code Generation Using a Genetic Algorithm. In *Proceedings of the IEEE International Conference on Computer Design 2001*, Austin, Texas, 2001.

[18] M. Lorenz, D. Kottmann, S. Bashford, R. Leupers, and P. Marwedel. Optimized Address Assignment for DSPs with SIMD Memory Accesses. In *Asia and South Pacific Design Automation Conference*, Yokohama, Japan, 2001.

[19] R. Manniesing, I. Karkowski, and H. Corporaal. Automatic SIMD Parallelization of Embedded Applications Based on Pattern Recognition. In *6th International Euro-Par Conference, Munich*, pages 349–356, 2000.

[20] Philips Semiconductors. www.trimedia.philips.com, 2000.

[21] J. Sias, H. Hunter, and W. Hwu. Enhancing loop buffering of media and telecommunications applications using low-overhead predications. In *Proceedings of the 34th Annual International Symposium on Microarchitecture*, December 2001.

[22] N. Sreraman and R. Govindarajan. A Vectorizing Compiler for Multimedia Extensions. In *International Journal of Parallel Programming*, 2000.

[23] Texas Instruments. www.ti.com/sc/c6x, 2000.

[24] V. Tiwari, S. Malik, and A. Wolfe. Power Analysis of Embedded Software: A First Step towards Software Power Minimization. In *Transactions on VLSI Systems*. IEEE, December 1994.

[25] G.-R. Uh, Y. Wang, D. Whalley, S. Jinturkar, C. Burns, and V. Cao. Effective Exploitation of a Zero Overhead Loop Buffer. In *Proceedings of Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 10–19, May 1999.

[26] R. Wilhelm and D. Maurer. *Compiler Design*. Addison Wesley, 1995.

[27] V. Zivojnovic, J. M. Velarde, C. Schläger, and H. Meyr. DSPstone - A DSP-oriented Benchmarking Methodology. In *International Conference on Signal Processing Applications and Technology (ICSPAT)*, 1994.