Code Generation for Embedded Processors

Rainer Leupers

University of Dortmund
Dept. of Computer Science 12
44221 Dortmund, Germany
email: leupers@ls12.cs.uni-dortmund.de

Abstract– The increasing use of programmable processors as IP blocks in embedded system design creates a need for C/C++ compilers capable of generating efficient machine code. Many of today's compilers for embedded processors suffer from insufficient code quality in terms of code size and performance. This violates the tight chip area and real-time constraints often imposed on embedded systems. The reason is that embedded processors typically show architectural features which are not well handled by classical compiler technology. This paper provides a survey of methods and techniques dedicated to efficient code generation for embedded processors. Emphasis is put on DSP and multimedia processors, for which better compiler technology is definitely required. In addition, some frontend aspects and recent trends in research and industry are briefly covered. The goal of these recent efforts in embedded code generation is to facilitate the step from assembly to high-level language programming of embedded systems, so as to provide higher productivity, dependability, and portability of embedded software[1].

1 Introduction

Due to the increasing complexity of embedded systems and availability of deep submicron VLSI technology, there is a shift towards more abstract system specification and implementation methods. Today's VHDL or Verilog based specification methods are step by step being replaced by C/C++ based languages, which offer both a convenient abstraction level and high simulation speed. Examples are the SystemC [1] and SpecC [2] initiatives. In addition, hardware synthesis from C/C++ is becoming common, e.g. in products by Synopsys (CoCentric) and C Level Design.

C/C++ also offer an ideal interface to software synthesis for embedded systems. The building blocks of today's and future systems are complex intellectual property (IP) components, or cores, many of which are programmable processors. Obviously, this IP based implementation methodology requires compilers capable of mapping C specifications into assembly code for embedded processors.

[1]Publication: ISSS 2000, Madrid/Spain, Sept 2000, ©IEEE

This contribution mainly deals with *efficient* code generation for embedded processors. Efficiency of the generated code is very important for embedded systems, due to limited system-on-a-chip memory sizes, real-time constraints of embedded applications, and the need to minimize power consumption of mobile devices. The following processor classes are common in embedded systems:

Microcontrollers: These are tailored for control-intensive applications and typically show a CISC architecture. Microcontrollers allow for a high code density, but computational resources are usually very limited. Examples are the 8051 and 6502 CPUs.

RISC processors: These show a load-store architecture and a large file of general-purpose registers. Due to the simplified instruction set, the most effective code optimization technique for RISCs is global register allocation [3]. A well-known example for RISC cores is the ARM family.

DSP processors: These are tuned for arithmetic intensive applications and allow for fast execution of DSP routines such as FIR filters or FFT. This is achieved by dedicated hardware support (e.g. multipliers and address generation units) and DSP-specific data path architectures. DSPs exist in a very large variety of domain-specific architectures. Major vendors include Texas Instruments, Motorola, Analog Devices, and NEC.

Multimedia processors: These are a recent mixture of RISC and DSP processors. They use the VLIW programming paradigm, i.e., multiple functional units working in parallel with statically determined schedules. Thus, multimedia processors allow for very high performance, however at the expense of low code density and high power consumption. Frequently, there is support for vectorized (SIMD) instructions (see section 3). Examples are the TI C6x [5] and the Philips Trimedia [6] families.

Application-specific processors: ASIPs are a compromise between off-the-shelf processors and ASICs. They show application-specific data paths which sometimes can be customized w.r.t. register file sizes and word lengths. In this case, *retargetable* compilers are required (see section 6). Examples of ASIPs are Tensilica's Xtensa (RISC based) [7] and the AMS Gepard core (DSP based) [8].

Note that in practice, these processor classes may overlap, e.g., a RISC processor might have DSP extensions, and microcontrollers might have a general-purpose register file.

In this paper, we primarily deal with code generation for the latter three processor classes, since these are the most challenging ones from a C compiler design viewpoint. In fact, current compilers for many standard DSPs and multimedia processors have been empirically shown to produce significantly less efficient code (as much as 1000 % overhead) in terms of performance and code size, as compared to hand-optimized reference code [9, 10, 11, 12]. The reason is that their special instruction set architectures are not well exploited by conventional compiler technology. Therefore, a large part of the software for such processors still has to be developed in assembly language. This implies time-consuming programming, extensive debugging, and low code portability. The requirements of short time-to-market and dependable code are obviously much better met by using C/C++ instead of assembly.

The need for more efficient compilers for embedded processors has been recognized about a decade ago. The purpose of this contribution is to provide a survey of important techniques that have been developed for embedded code optimization, while also highlighting their key methodologies which reach beyond the scope of classical compiler technology for general-purpose systems.

In the following three sections, we give examples for code optimization techniques at different levels in the compilation flow, reaching from source level (section 2) to assembly level techniques (section 4). In section 5, we briefly discuss compiler frontend related issues, while sections 6 and 7 provide an overview of recent compiler trends in academia and industry. Finally, conclusions are given.

2 Source level optimization

The compilation process, i.e., the mapping of a given C program to a behaviorally equivalent assembly program, starts with source code analysis and source level optimization. Optimizing already at the C level is attractive, because the result is still machine-independent in the sense that it can be compiled to different target processors. In addition, also C based hardware synthesis can benefit from source level optimization. Several examples are given in the following.

Standard optimizations All reasonable compilers perform machine-independent standard optimizations, such as constant folding, common subexpression elimination, or jump optimization [13, 14, 15]. These techniques need only a minimum of machine-specific information and are beneficial for most programs. Frequently, such standard optimizations are also performed at the *intermediate representation* (IR) level, where complex source code constructs have already been split into a simple form, such as three-address code.

Address code transformation The high-level address code transformation techniques described in [16] can be regarded as an extension of the above standard optimizations, targeted towards memory-intensive applications. Here the goal is the simplification of array index expressions beyond the classical induction variable elimination technique (see e.g. [13]). At the expense of larger code size, this leads to a reduction of up to 50 % in instruction cycles for array-intensive nested loops.

Loop transformations In case of multimedia applications mapped to VLIW processors, loop transformations are a very effective means of code optimization. A simple example is *loop unrolling*, where loop iterations are duplicated, resulting in larger basic blocks and thereby in a higher potential for parallelization of instructions during scheduling. Its counterpart is *loop folding* or *software pipelining* [18, 19], where loop iterations are restructured in such a way, that the critical path length within the loop body is reduced. Again, these loop optimizations come at the price of an increased code size. Further loop restructuring techniques are described in [20].

Function inlining Function inlining is a well-known technique, in which function calls are replaced by copies of function bodies, so as to reduce the calling overhead. Usually, compilers use local heuristics in order to identify suitable candidate functions for inlining, while mostly neglecting code size constraints. In contrast, the inlining technique described in [17] aims at a maximum program speedup for a given global code size constraint, and thus better meets the demands of embedded processors. It is based on profiling information, code growth estimation, and a branch-and-bound optimization procedure. This technique also exemplifies a common concept in code generation for embedded processors: By the use of time-intensive optimization techniques, a larger search space can be explored, which in turn leads to better code than in traditional compilers. This approach is valid, since for embedded systems high code quality is much more important than high compilation speed.

3 Optimized instruction set mapping

After source level or IR level optimization, the machine-independent IR statements are mapped to assembly instructions. At this point all machine-specific features, such as special-purpose registers, complex instruction patterns, and inter-instruction constraints need to be taken into account. This is what makes efficient code generation for embedded processors generally difficult.

Tree pattern matching A very important basic technique for instruction set mapping, or *code selection*, is tree pattern matching [21]. IR statements are represented in the form of data flow trees (DFTs), where tree nodes correspond to variables, constants, and operations, while edges denote data dependencies. Also the machine instructions can be considered as small tree patterns. Thus, an optimum mapping is given by a minimum cover of a DFT by cost-attributed instruction patterns (fig. 1). A number of tools are available for automatic generation of tree pattern matchers from instruction set grammar specifications. These include IBURG [22], BEG [23], and OLIVE [21]. The generated matchers are given as C source code, and they compute optimum tree covers in linear time by means of dynamic programming.

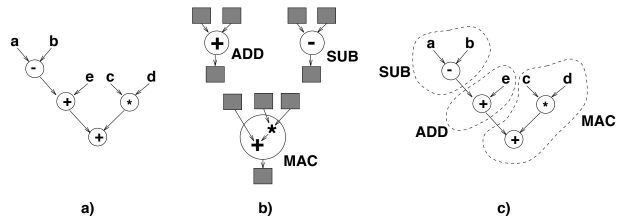

Figure 1: *Code selection: a) Data flow tree, b) available instruction patterns, c) optimal tree cover*

Phase-coupled code generation Besides code selection, the task of machine code generation also comprises register allocation and instruction scheduling. Traditionally, these phases, each of which is an NP-hard optimization problem in itself, are solved sequentially and heuristically. However, there is a cyclic dependence, since each of the three phases may impose possibly unnecessary and obstructive restrictions on the remaining ones. Therefore, one key method in embedded code generation is *phase coupling*. Early techniques in this area include [24, 25, 26], each of which integrated two out of three code generation phases. In [27], the above tree pattern matching technique has been coupled with register allocation and scheduling for a family of TI DSPs. Later, an additional heuristic for handling (more general) data flow graphs (DFGs) has been presented [28]. An improved approach, based on simulated annealing, has been described in [29]. Mutation scheduling [30] is another approach to complete phase coupling, where also algebraic transformations are exploited in order to explore alternative instruction set mappings. In [31, 32], phase-coupled compilers for certain classes of VLIW processors have been described. A *constraint logic programming* technique for DSPs with irregular architectures has been presented in [33]. In that approach, all binding decisions are delayed until they are really required, which yields a maximum degree of freedom in code generation. For several DSPs the code quality has been shown to be very close to assembly programs, however at the expense of high compilation times. Further, more theoretically-oriented, contributions include [34, 35, 36].

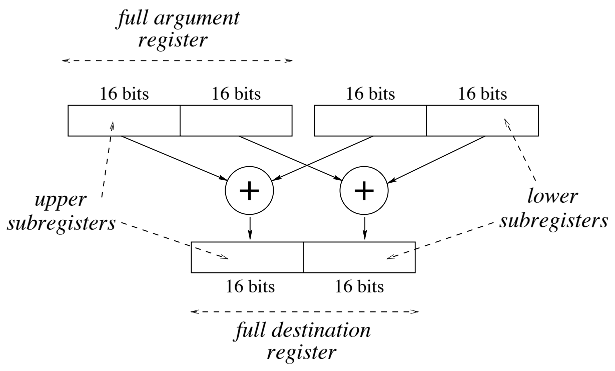

Figure 2: *SIMD instruction "ADD2" in the TI C6x processor*

Multimedia instruction sets Multimedia processors require different mapping techniques than DSPs. On one hand, their architecture is more regular and VLIW/RISC-like, on the other hand they show special new instruction types. An example are SIMD (single instruction multiple data) instructions (fig. 2). These instructions perform multiple identical operations in parallel, while splitting the data registers into subregisters, each of which then contains a separate data item. The power of SIMD instructions lies in faster execution of algorithms operating on "short" (16 or 8 bit) values. However, most current compilers for multimedia processors can exploit SIMD instructions only by means of *compiler-known functions*, i.e., C level macros. The reason is that SIMD instructions do not fit into the usual tree based code selection approach. This results in non-portable code, and the responsibility of exploiting the SIMD mode is still with the human programmer. In order to circumvent this problem, special language constructs [37] or C++ class libraries [38] have been proposed. The technique described in [39] is capable of generating SIMD instructions also from plain C code, based on an extension of the above tree pattern matching technique towards general graphs.

4 Assembly level optimization

Once assembly code has been generated, a significant optimization potential is still left w.r.t. memory access organization and instruction scheduling. This concerns the consideration of the concrete memory architecture, as well as exploitation of address generation hardware and instruction-level parallelism.

Memory access optimization Several DSPs, e.g. Motorola 56k, Analog Devices 210x, and AMS Gepard show two memory banks (usually called X and Y), which are accessible in parallel. This raises the problem of partitioning the program variables between X and Y in such a way, that potential parallelism is maximized. Naive compilers use only one of the two banks, or leave the partitioning decision to the programmer by means of C language extensions. More advanced approaches use a dedicated optimization phase for

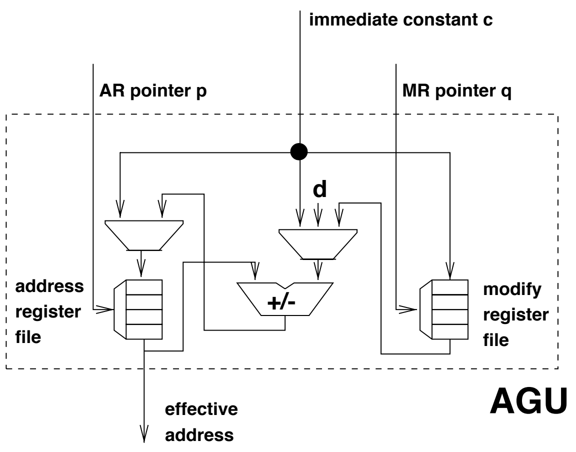

Figure 3: *Address generation unit (AGU) in DSPs*

X/Y partitioning based on pre-scheduled assembly code. An example is given by [41], where a simulated annealing algorithm has been successfully applied to Motorola DSPs. Further approaches are presented in [40, 42].

In [43] detailed knowledge about the memory interface and fast access modes is exploited in combination with processor pipeline timing information. Due to this "memory-aware" approach, the scheduling of instructions can be adapted to the concrete memory module in use. This generally results in fewer pipeline stalls, and for a TI C6x processor an average speedup of 24 % as compared to a traditional approach has been measured.

Address code optimization One area in DSP code optimization that has been thoroughly investigated is the utilization of dedicated address generation units (AGUs, fig. 3). Such AGUs generally contain address register and modify register files, where address register updates can take place in three ways: Adding a "wide" immediate constant c, adding a "short" immediate constant d, or adding the contents of a modify register. The latter two modes should be preferred, since they translate to efficient parallel auto-increment address computations. Exploitation of auto-increment modes depends on the layout of variables in memory. A relatively large number of techniques for computing good layouts are already available, including [44, 45, 46, 47, 48, 49, 50, 51, 52], which differ in the concrete AGU configurations (e.g., register file sizes and bound on d) that can be handled, as well as in the optimization method (e.g., heuristics or genetic algorithms). The achievements in this area can be considered satisfactory, and the techniques have already found their way into commercial compilers.

Instruction scheduling Instruction scheduling assigns generated machine instructions to control steps, which is important for VLIW-like multimedia processors as well as for DSPs with limited instruction-level parallelism. Local scheduling (or code compaction) algorithms, such as the list scheduling heuristic, are limited to the scope of a single basic block [53]. For DSPs also exact local scheduling techniques have been developed [54, 55]. In contrast, global techniques operate on an entire function [56, 57]. The latter result in a more effective scheduling of global critical paths, but frequently at the expense of larger code size. Recent scheduling techniques [58] aim at an extensive pre-analysis of scheduling constraints, so as to achieve shorter schedules as compared to pure heuristics.

5 Frontend issues

Besides the above code optimization techniques, C/C++ compiler development for embedded processors also requires frontend software, which performs lexical, syntactical, and semantical analysis of source programs. A popular commercial product is the EDG frontend [59], which however is relatively expensive. Research frontends include the Trimaran system [60], which applies to a class of VLIW processors, and Stanford's SUIF system [61]. For the latter, also a backend development system has been announced recently [62]. Also the GNU C compiler [63] is sometimes used in the context of embedded processors, event though its limitations make it rather difficult to adapt it to irregular processors architectures. The LANCE system [64] developed at Dortmund serves as a compiler infrastructure both for research and industrial projects. It comprises an ANSI C frontend, a library of standard IR optimizations, as well as a backend interface for data flow tree generation. One key feature is that new IR optimizations can be quickly added as "plug-and-play" components. An *executable IR* in the form of low-level C code supports validation of IR transformations and simultaneously allows for source-level optimizations.

6 New research directions

Most existing code optimization techniques focus on code size and performance. However, in the context of embedded systems, additional goals may be relevant. These include low power for mobile applications and retargetability for support of architecture exploration.

Low power Power minimization has been a design goal in hardware synthesis for several years. However, only recently the impact of compiler techniques on power consumption has been investigated. It is important to note, that power optimization in compilers may be contrary to the traditional goals, code size and performance, and therefore requires new techniques. Existing work, e.g. [65, 66], so far mainly focused on the impact of instruction selection and scheduling on power consumption of machine code. However, also more general approaches, such as alternative instruction and data encodings and utilization of power-efficient on-chip memories should be further investigated. A special session at the

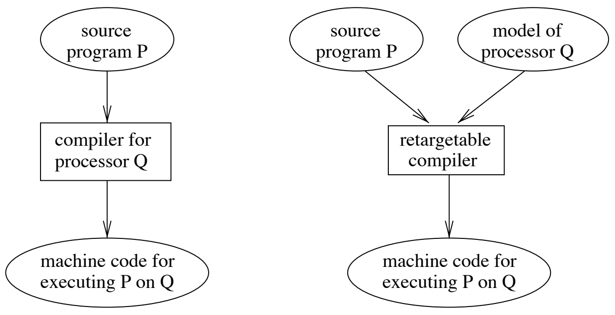

Figure 4: *Traditional vs. retargetable compiler*

Design Automation Conference 2000 also stressed the importance of low power embedded software.

Retargetability Retargetable compilers can generate code for different target processors, based on external (e.g. HDL) machine models (fig. 4). Their application area is twofold: Retargetable compilers are useful in the context of parameterizable ASIPs (see section 1), because a single compiler is sufficient to explore different configurations of a given ASIP, thereby making hardware/software trade-offs. A significant amount of retargetable compiler technology for embedded processors is already available, including MSSQ [67] RECORD [68], SPAM [28], CHESS [69], CodeSyn [70], and AVIV [31]. Currently, retargetable compilers are receiving renewed interest also due to the need for architecture exploration at the system level: Retargetable compilers, in combination with simulators, can provide an early estimation of the performance of different target processors for a given application. A recent research effort into this direction is the EXPRESS project at UCI [71, 43].

7 Industrial trends

With the increasing amount of technology available for embedded code generation, several retargetable optimizing compilers for embedded processors have become commercially available. These include CoSy (ACE, [72]), CHESS (Target Compiler Technologies, [73]), and Archelon [74]. However, these tools still have their limitations w.r.t. certain processor classes, and also the retargeting mechanism should be further improved, so as to better fit into the system design flow. Concerning high-performance "off-the-shelf" processors, currently a trend towards VLIW machines can be observed. This trend is mainly enabled by increasing chip integration scales, which allows for integration of many parallel functional units. The VLIW trend is expected to continue [75], which will also facilitate the construction of more efficient compilers. However, also new problems are introduced, such as mapping to SIMD or conditional instructions, or scheduling for clustered VLIW data paths. Simultaneously, more and more ASIPs and customizable processors are in use. A promising new entry in this area is Tensilica's Xtensa core [7]. Using a special description language, this RISC core can be extended by application-specific instructions, while both the HDL core model and the corresponding software tools, including a C compiler, are automatically retargeted.

8 Conclusions

We have provided a brief survey of methods and techniques in the area of code generation for embedded processors. More extensive overviews and literature references can be found e.g. in [76, 70, 77]. With the growing importance of compilers for embedded processors, research results are increasingly turned into products. In the future, we expect a higher importance of VLIW and ASIP processors, where also non-conventional compiler design goals like low power and retargetability will be of high interest.

References

[1] G. Arnout: *SystemC Standard*, Asia South Pacific Design Automation Conference (ASP-DAC), 2000

[2] D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, S. Zhao: *SpecC: Specification Language and Methodology*, Kluwer Academic Publishers, 2000

[3] J.L. Hennessy, D.A. Patterson: *Computer Architecture – A Quantitative Approach*, Morgan Kaufmann Publishers Inc., 1990

[4] ARM home page: www.arm.com, 2000

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

[6] Philips Semiconductors: www.trimedia.philips.com, 2000

[7] Tensilica Inc.: www.tensilica.com, 2000

[8] Austria Mikro Systeme International: www.amsint.com/databooks/digital/gepard.html, 2000

[9] V. Zivojnovic, J.M. Velarde, C. Schläger, H. Meyr: *DSPStone – A DSP-oriented Benchmarking Methodology*, Int. Conf. on Signal Processing Applications and Technology (ICSPAT), 1994

[10] P. Paulin, M. Cornero, C. Liem, et al.: *Trends in Embedded Systems Technology*, in: M.G. Sami, G. De Micheli (eds.): *Hardware/Software Codesign*, Kluwer Academic Publishers, 1996

[11] M. Levy: *C Compilers for DSPs flex their Muscles*, EDN Access, Issue 12, www.ednmag.com, 1997

[12] M. Coors, O. Wahlen, H. Keding, O. Lüthje, H. Meyr: *TI C62x Performance Code Optimization*, DSP Germany, 2000

[13] A.V. Aho, R. Sethi, J.D. Ullman: *Compilers - Principles, Techniques, and Tools*, Addison-Wesley, 1986

[14] S.S. Muchnik: *Advanced Compiler Design & Implementation*, Morgan Kaufmann Publishers, 1997

[15] A.W. Appel: *Modern Compiler Implementation in C*, Cambridge University Press, 1998

[16] S. Gupta, R. Gupta, M. Miranda, F. Catthoor: *Analysis of High-Level Address Code Transformations for Programmable Processors*, Design Automation & Test in Europe (DATE), 2000

[17] R. Leupers, P. Marwedel: *Function Inlining under Code Size Constraints for Embedded Processors*, Int. Conference on Computer-Aided Design (ICCAD), 1999

[18] M. Lam: *Software Pipelining: An Effective Scheduling Technique for VLIW machines*, ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 1988

[19] G. Goossens, J. Vandewalle, H. De Man: *Loop Optimization in Register-Transfer Scheduling for DSP Systems*, 26th Design Automation Conference (DAC), 1989

[20] U. Banerjee: *Loop Transformations for Restructuring Compilers – The Foundations*, Kluwer Academic Publishers, 1993

[21] A.V. Aho, M. Ganapathi, S.W.K Tjiang: *Code Generation Using Tree Matching and Dynamic Programming*, ACM Trans. on Programming Languages and Systems 11, No. 4, 1989

[22] C.W. Fraser, D.R. Hanson, T.A. Proebsting: *Engineering a Simple, Efficient Code Generator Generator*, ACM Letters on Programming Languages and Systems, vol. 1, no. 3, 1992

[23] H. Emmelmann, F.W. Schröer, R. Landwehr: *BEG – A Generator for Efficient Backends*, ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), SIGPLAN Notices 24, no. 7, 1989

[24] K. Rimey, P.N. Hilfinger: *Lazy Data Routing and Greedy Scheduling for Application-Specific Signal Processors*, 21st Annual Workshop on Microprogramming and Microarchitecture (MICRO-21), 1988

[25] R. Hartmann: *Combined Scheduling and Data Routing for Programmable ASIC Systems*, European Conference on Design Automation (EDAC), 1992

[26] B. Wess: *Automatic Code Generation for Integrated Digital Signal Processors*, IEEE Int. Symp. on Circuits and Systems (ISCAS), 1991

[27] G. Araujo, S. Malik: *Optimal Code Generation for Embedded Memory Non-Homogeneous Register Architectures*, 8th Int. Symp. on System Synthesis (ISSS), 1995

[28] G. Araujo, S. Malik, M. Lee: *Using Register Transfer Paths in Code Generation for Heterogeneous Memory-Register Architectures*, 33rd Design Automation Conference (DAC), 1996

[29] R. Leupers: *Register Allocation for Common Subexpressions in DSP Data Paths*, Asia South Pacific Design Automation Conference (ASP-DAC), 2000

[30] S. Novack, A. Nicolau, N. Dutt: *A Unified Code Generation Approach using Mutation Scheduling*, chapter 12 in [76]

[31] S. Hanono, S. Devadas: *Instruction Selection, Resource Allocation, and Scheduling in the AVIV Retargetable Code Generator*, 35th Design Automation Conference (DAC), 1998

[32] B. Rau, V. Kathail, S. Aditya: *Machine Description Driven Compilers for EPIC and VLIW Processors*, Design Automation for Embedded Systems, Vol. 4, No. 2/3, Kluwer Academic Publishers, 1999

[33] S. Bashford, R. Leupers: *Phase-Coupled Mapping of Data Flow Graphs to Irregular Data Paths*, Design Automation for Embedded Systems, Vol. 4, No. 2/3, Kluwer Academic Publishers, 1999

[34] M. Mahmood, F. Mavaddat, M.I. Elmasry: *Experiments with an Efficient Heuristic Algorithm for Local Microcode Generation*, Int. Conf. on Computer Design (ICCD), 1990

[35] M. Langevin, E. Cerny: *An Automata-Theoretic Approach to Local Microcode Generation*, European Conference on Design Automation (EDAC), 1993

[36] T. Wilson, G. Grewal, B. Halley, D. Banerji: *An Integrated Approach to Retargetable Code Generation*, 7th Int. Symp. on High-Level Synthesis (HLSS), 1994

[37] R.J. Fisher, H.G. Dietz: *Compiling for SIMD Within a Register*, 11th Annual Workshop on Languages and Compilers for Parallel Computing (LCPC98), 1998

[38] Intel: *Coding Techniques for the Streaming SIMD Extensions With the Intel C/C++ Compiler*, developer.intel.com/vtune/newsletr/methods.htm, 2000

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

[40] D.B. Powell, E.A. Lee, W.C. Newman: *Direct Synthesis of Optimized DSP Assembly Code from Signal Flow Block Diagrams*, Proc. International Conference on Acoustics, Speech, and Signal Processing, 1992

[41] A. Sudarsanam, S. Malik: *Memory Bank and Register Allocation in Software Synthesis for ASIPs*, Int. Conf. on Computer-Aided Design (ICCAD), 1995

[42] M. Saghir, P. Chow, C. Lee: *Exploiting Dual Data-Memory Banks in Digital Signal Processors*, 7th International Conference on Architectural Support for Programming Languages and Operating Systems, 1996

[43] P. Grun, N. Dutt, A. Nicolau: *Memory Aware Compilation through Accurate Timing Extraction*, 37th Design Automation Conference (DAC), 2000

[44] D.H. Bartley: *Optimizing Stack Frame Accesses for Processors with Restricted Addressing Modes*, Software – Practice and Experience, vol. 22(2), 1992

[45] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, A. Wang: *Storage Assignment to Decrease Code Size*, ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 1995

[46] R. Leupers, P. Marwedel: *Algorithms for Address Assignment in DSP Code Generation*, Int. Conference on Computer-Aided Design (ICCAD), 1996

[47] A. Sudarsanam, S. Liao, S. Devadas: *Analysis and Evaluation of Address Arithmetic Capabilities in Custom DSP Architectures*, Design Automation Conference (DAC), 1997

[48] B. Wess, M. Gotschlich: *Optimal DSP Memory Layout Generation as a Quadratic Assignment Problem*, Int. Symp. on Circuits and Systems (ISCAS), 1997

[49] B. Wess, M. Gotschlich: *Constructing Memory Layouts for Address Generation Units Supporting Offset 2 Access*, Proc. ICASSP, 1997

[50] N. Kogure, N. Sugino, A. Nishihara: *Memory Address Allocation Method for a DSP with $\pm$ 2 Update Operations in Indirect Addressing*, European Conference on Circuit Theory and Design (ECCTD), 1997

[51] R. Leupers, F. David: *A Uniform Optimization Technique for Offset Assignment Problems*, 11th Int. System Synthesis Symposium (ISSS), 1998

[52] A. Rao, S. Pande: *Storage Assignment using Expression Tree Transformations to Generate Compact and Efficient DSP Code*, ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 1999

[53] S. Davidson, D. Landskov, B.D. Shriver, P.W. Mallett: *Some Experiments in Local Microcode Compaction for Horizontal Machines*, IEEE Trans. on Computers, Vol. 30, No. 7, 1981

[54] R. Leupers, P. Marwedel: *Time-Constrained Code Compaction for DSPs*, 8th Int. System Synthesis Symposium (ISSS), 1995

[55] A. Timmer, M. Strik, J. van Meerbergen, J. Jess: *Conflict Modelling and Instruction Scheduling in Code Generation for In-House DSP Cores*, 32nd Design Automation Conference (DAC), 1995

[56] J.A. Fisher: *Trace Scheduling: A Technique for Global Microcode Compaction*, IEEE Trans. on Computers, vol. 30, no. 7, 1981

[57] A. Aiken, A. Nicolau: *A Development Environment for Horizontal Microcode*, IEEE Trans. on Software Engineering, no. 14, 1988

[58] B. Mesman, C. Alba Pinto, K. van Eijk: *Efficient Scheduling of DSP Code on Processors with Distributed Register Files*, 12th Int. Symp. on System Synthesis (ISSS), 1999

[59] Edison Design Group: www.edg.com, 2000

[60] Trimaran – An Infrastructure for Research in Instruction-Level Parallelism, www.trimaran.org, 2000

[61] The Stanford Compiler Group: suif.stanford.edu, 2000

[62] Machine SUIF: www.eecs.harvard.edu/hube/research/machsuif.html, 2000

[63] Free Software Foundation: www.gnu.org, 1999

[64] LANCE C compiler system: ls12-www.cs.uni-dortmund.de/∼leupers, 2000

[65] M. Lee, V. Tiwari, S. Malik, M. Fujita: *Power Analysis and Minimization Techniques for Embedded DSP Software*, IEEE Trans. on VLSI Systems, Vol. 5, No. 2, 1997

[66] C. Gebotys: *Low Energy Memory and Register Allocation Using Network Flow*, 34th Design Automation Conference (DAC), 1997

[67] P. Marwedel: *Tree-based Mapping of Algorithms to Predefined Structures*, Int. Conf. on Computer-Aided Design (ICCAD), 1993

[68] R. Leupers, P. Marwedel: *Retargetable Generation of Code Selectors from HDL Processor Models*, European Design & Test Conference (ED & TC), 1997

[69] D. Lanneer, J. Van Praet, A. Kifli, K. Schoofs, W. Geurts, F. Thoen, G. Goossens: *CHESS: Retargetable Code Generation for Embedded DSP Processors*, chapter 5 in [76]

[70] C. Liem: *Retargetable Compilers for Embedded Core Processors*, Kluwer Academic Publishers, 1997

[71] N. Dutt, P. Grun, A. Halambi: www.cecs.uci.edu/∼aces, Center for Embedded Computer Systems, UC Irvine, 2000

[72] ACE Associated Compiler Experts: www.ace.nl, 2000

[73] Target Compiler Technologies: www.retarget.com, 2000

[74] Archelon Inc.: www.archelon.com, 2000

[75] P. Faraboschi, G. Desoli, J.A. Fisher: *VLIW Architectures for DSP and Multimedia Applications – The Latest Word in Digital and Media Processing*, IEEE Signal Processing Magazine, March 1998

[76] P. Marwedel, G. Goossens (eds.): *Code Generation for Embedded Processors*, Kluwer Academic Publishers, 1995

[77] R. Leupers: *Retargetable Code Generation for Digital Signal Processors*, Kluwer Academic Publishers, 1997