Phase Coupled Code Generation for DSPs Using a Genetic Algorithm

Markus Lorenz, Peter Marwedel
University of Dortmund, Dept. of Computer Science 12, Germany
email: {Markus.Lorenz, Peter.Marwedel}@uni-dortmund.de

Abstract

The growing use of digital signal processors (DSPs) in embedded systems necessitates the use of optimizing compilers supporting special hardware features. Due to the irregular architectures present in today's DSPs there is a need of compilers which are capable of performing a phase coupling of the highly interdependent code generation subtasks and a graph based code selection. In this paper we present a code generator which performs a graph based code selection and a complete phase coupling of code selection, instruction scheduling (including compaction) and register allocation. In addition, our code generator takes into account effects of the subsequent address code generation phase. In order to solve the phase coupling problem and to handle the problem complexity, our code generator is based on a genetic algorithm. Experimental results for several benchmarks and an MP3 application for two DSPs show the effectiveness and the retargetability of our approach. Using the presented techniques, the number of execution cycles is reduced by 51% on average for the M3-DSP and by 38% on average for the ADSP2100 compared to standard techniques[1].

I Introduction

Digital signal processors (DSPs) are frequently used in embedded systems to flexibly account for specification modifications in late design phases. Embedded processor based systems often have to meet real-time constraints while minimizing area and energy consumption. Thus, designers try to meet the given timing constraints by adding instruction level parallelism to processors. Unfortunately, the use of heterogeneous register files for reducing chip area and power consumption leads to irregular processor architectures which can rarely be handled by traditional compilers. 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 demand for optimizing compilers which are adapted to special architectures and thus are capable of exploiting the irregular architecture features of DSPs.

The task of code generation can be divided 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*. For instance, finding an optimal instruction schedule does not necessarily result in optimal code with respect to all subtasks together. Most traditional code generation techniques are based on *tree based* code selection algorithms [1]. These are runtime-efficient, but the main disadvantages are that using a tree based technique allows only a restricted phase coupling which usually results in superfluous memory accesses and potentially leads to inefficient assembly code.

In this paper we present a DSP code generator based on a genetic algorithm capable of performing a graph based code selection and a complete phase coupling of the subtasks CS, IS (including compaction) and RA. Effects of the subsequent address code generation phase are taken into account, too.

The remainder of this paper is organized as follows: The next section gives an overview of the related work in this area. In section III, the considered target architectures are explained. After that our genetic algorithm driven code generator is presented. Finally, we demonstrate the effectiveness of our code generator and conclude the paper with a summary.

II Related Work

Most traditional compilers perform a tree based code selection technique by using a *tree pattern matcher*. The tree based code selection techniques decompose data flow graphs (DFGs) into trees and perform a separate code selection for each tree. Graphs are often split at *common subexpressions* (CSEs)[2] to form trees. On general-purpose processors with large homogeneous register files, CSEs are normally kept in registers, and the register allocation phase in the compiler aims at avoiding register spills and reloads. However, this approach usually does not work for DSPs with an irregular, special-purpose register architecture. The reason is that DSP register files show a very small storage capacity (frequently only a single value), so that temporary values cannot be kept in registers during several instruction cycles. Sometimes DSP registers even cannot be spilled at all, so that from a compiler point of view, memory is the only "safe" resource for storing CSEs. Therefore, it is very common in DSP compilers that CSEs are stored in memory right from the beginning

[1]This work has been sponsored by the German Research Foundation (DFG).

[2]These are values which are used in more than one expression.

and are reloaded into registers only at the time of further CSE uses [2, 3].

In order to apply the traditional tree pattern matching technique to graphs, [3] proposes extensions which make this technique applicable to a strongly restricted class of architectures with heterogeneous register files. Since tree pattern matchers for architectures with heterogeneous register files usually store CSEs in memory locations and load this value for every use, in [4] a technique based on a simulated annealing algorithm is proposed which is capable of keeping (at least some) CSEs in special-purpose registers. However, there is still a demand for techniques supporting phase coupling and instruction covering of different tree nodes.
Phase coupling techniques for homogeneous VLIW-architectures (VLIW = *very long instruction word*) are presented in [5, 6]. But only one architecture considered in [5] shows a slight irregularity.
In order to take into account the phase coupling problem, code generation techniques have been proposed based on *integer linear programming* (e.g. [7, 8]). However, the complexity of the solver allows the computation of optimal solutions only for small benchmarks or only for some code generation subtasks.
A phase coupling technique for irregular architectures based on *constraint logic programming* is proposed in [9]. Restrictions w.r.t. the use of resources or the execution order of instructions are taken into account by the formulation of constraints. In addition, optimal graph based code selection and instruction scheduling are applied independently of each other. Phase coupling between the different code generation tasks is achieved by passing alternative solutions to subsequent optimization phases.
Genetic algorithms have been proven to be very effective in finding optimal or near optimal solutions in huge search spaces. For this reason, we are using a special list-scheduling algorithm in combination with a genetic algorithm as proposed in [10]. In contrast to earlier work using genetic algorithms for scheduling problems (e.g. [10, 11, 12]), we perform a graph based code selection in conjunction with a complete phase coupling of CS, IS and RA under consideration of the effects of the subsequent address code generation phase. In contrast to the genetic algorithm driven code generator presented in [13] we perform a stronger phase coupling by also integrating the code compaction phase and considering the effects of address code generation. Our chromosomal representation also allows the representation of data transfers and enables the representation of alternative machine programs which is essential for a complete phase coupling. Another contribution of this paper concerns a technique for merging different data paths (e.g. starting from CSEs) by *bypassing* and a new crossover operator (*CS-crossover*) which takes into account the tree/graph structure of the given optimization problem. Due to the generic specification of the target architecture, main parts of the code generator can be reused without modifications for other processors.

III Target Architectures

A M3-DSP

The M3-DSP is an instance of a scalable DSP platform for mobile communication applications [14]. The platform permits a fast design of DSPs adapted to special applications. In order to meet constraints with respect to performance, chip area, and energy dissipation, the platform supports among others, the following features: There is a scalable number of data paths that enables processing either on a single data path or on all data paths in parallel according to the SIMD principle. In the case of the M3-DSP, there are 16 data paths containing special-purpose register files.

The M3-DSP is organized as a VLIW architecture which permits independent control e.g. for data manipulation, data transfer, program control, and the address generation unit. The address generation unit (AGU) contains four address pointer registers which allow auto-increment addressing with a specific offset. If there is a need for larger offsets, the address pointer registers can be used orthogonally with one of four modify registers in auto-modify operations. In addition, page pointer relative addressing is possible. In order to reduce the loop overhead, support for executing up to 256 processor instructions in zero-overhead hardware-loops is available.

B ADSP

The ADSP2100 of the ADSP210X-family [15] contains three functional units (ALU, MAC and shifter) with special-purpose register files. A parallel execution of data transfers, memory accesses and data manipulations is only possible in a restricted manner. The memory is partitioned into two memory banks which allows accesses of two data words in parallel by using two AGUs. Like in the M3-DSP, support for execution of processor instructions in a zero-overhead hardware-loop is incorporated.

IV Phase Coupled Code Generation

Before presenting details of our genetic algorithm driven code generator, an overview of the code generation process is given. Code generation is started after the source program is transformed into our low-level intermediate representation (LIR), which serves as an exchange format for all succeeding transformations and optimizations. In order to propagate optimization results to other phases, it allows 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 resource alternatives for a specific graph node of the LIR or abstract operation of the source program (e.g. load or add), it is possible to associate alternative resource combinations with such a graph node. At this level of abstraction the source program is represented by a set of DFGs which are then separately mapped to assembly code by performing the following steps:

1. Preprocessing
   The main task of this step is the generation of alternative machine programs (or solutions) of the source program. This is done by inserting the maximum number of data transfers between two specific graph nodes which can potentially occur and generating an initial covering of the graph nodes with processor resources. Note that this task represents the only difference in generating the code for

the M3-DSP and the ADSP. Following this phase, it is possible to prune the set of resource alternatives by performing *constraint propagation* which is a basic technique in constraint logic programming. Applying the constraint propagation algorithm, the *node* and *edge consistency* of all graph nodes is ensured. Node consistency means that for each graph node it is made sure that only valid resource combinations according to the given operation specification can be chosen. In addition, edge consistency assures a valid data transfer path between two data dependent graph nodes.

2. Code generation (CS, IS, RA)
   In this step, the process of code generation aims at restricting the set of resource alternatives by optimizing according to a specified cost function. In addition, every graph node has to be assigned to a specific control step (or execution cycle). Unfortunately, the generation of optimal assembly code means solving an NP-hard optimization problem. Thus, there is a need for an optimization algorithm capable of finding optimal or near optimal solutions in polynomial time. For this reason, we are using an optimization algorithm based on a genetic algorithm which is described in more detail in section A. The result is a DFG with scheduled (and compacted) graph nodes covered by processor resources.

3. Address code generation (ACG)
   In this step, the address code for the given memory access sequence of a specific basic block is generated. The resulting sequential address code is inserted into the LIR-code and is then compacted by reusing the genetic algorithm driven code generator.

After these steps, the current graph based LIR representation is mapped to assembly code.

Due to the generic implementation of the code generation techniques, the code for steps 2 and 3 can be applied without modifications for the ADSP and the M3-DSP. The difference between the code generators is confined to the specification of the different target architectures in the first step.

A Overview of the Genetic Code Generator

In order to handle the problem complexity there is need for an efficient optimization technique which is capable of generating optimal or near optimal solutions according to a specified cost function. Genetic algorithms (GA) have proven to solve complex optimization problems by imitating the natural evolution process (see e.g. [16] 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 for the optimization problem. By applying genetic operators like *selection*, *mutation*, and *crossover* to the members of the population the fitness of the individuals will increase in the course of the generations. An overview of the main steps of the optimization process of our genetic algorithm driven code generator is given in fig. 1.

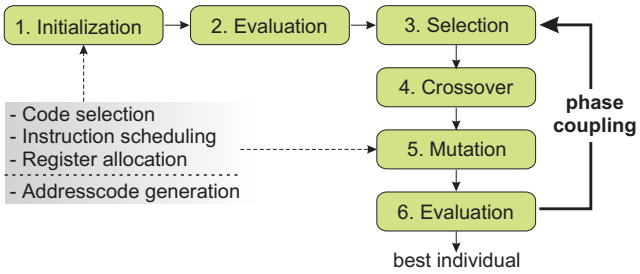

Fig. 1: Overview of the genetic code generator

At first all individuals of the population are initialized (1) by performing the tasks of code generation (CS, IS and RA) and afterwards evaluated according to a specified cost function (2). Individuals which should inherit their genes to the next generation are selected probabilistically in the following step (3). The crossover operator (4) performs a recombination of the genetic information by choosing two individuals and swapping genes between these individuals. Afterwards, mutation (5) creates new gene material by changing alleles. The resulting individuals are evaluated (6) again. The optimization process is iterated until a termination condition (here: maximum number of generations) is met. It is a very important characteristic of genetic algorithms that suitable gene material is passed to the subsequent generations. This permits one 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. However, one of the main problems using genetic algorithms is finding a suitable representation of the underlying optimization problem and using genetic operators which can be executed very quickly (because they are used many times).

In the next section we describe the coding mechanism and the initialization, evaluation, crossover, and mutation steps of our code generator in more detail.

B Chromosomal Representation

A specific individual represents the machine code sequence of one basic block (BB). For this purpose, the graph nodes of the source program are mapped to genes (s. fig. 2).

This means that every individual of a population has the same number of genes (in this case 23). The given graph node numbers represent the corresponding gene of the chromosome. Data transfers which can potentially occur are represented as CP-nodes (CP = copy). The available alternative resources (e.g. registers A, B, C, ...) are stored along whith each gene.

C Initialization

The initialization is performed for each individual separately with the goal of generating a population of different solutions (machine code sequences). Obviously, it is necessary to solve

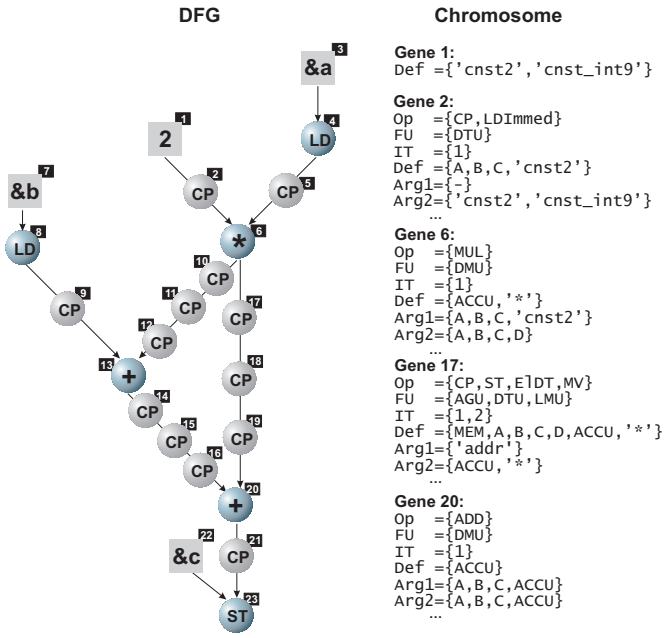

Fig. 2: Chromosomal DFG representation

the tasks of code generation CS, IS and RA. The main initialization procedure can be sketched as follows:

WHILE ready_set $\neq$ empty **DO**
// Perform constraint propagation after steps (2) to (5)
(1) $gene$ = `InstructionScheduling(`ind`)`;
(2) op_{gene} = `CodeSelection(`$gene$`)`;
(3) fu_{gene} = `SelectFunctionalUnit(`$gene$`)`;
(4) it_{gene} = `SelectInstructionType(`$gene$`)`;
(5) def_{gene} = `RegisterAllocation(`$gene$`)`;
(6) cs_{gene} = `Compaction(`$gene$`)`;
(7) `Update(`$ready\_set$`)`
END;

The algorithm is based on a variant of the well-known *list-scheduling* algorithm [17]. While the `ready_set` is not empty, the steps (1) to (7) are performed in a loop. In the first step (1), the next gene to be scheduled from `ready_set` is chosen. After that, the processor operation $\texttt{op}_{\texttt{gene}}$, the functional unit $\texttt{fu}_{\texttt{gene}}$, the instruction type[3] $\texttt{it}_{\texttt{gene}}$, and the destination register resource $\texttt{def}_{\texttt{gene}}$ is determined probabilistically. Choosing one specific resource alternative means erasing all other resources of the corresponding set. In order to avoid the selection of invalid resource combinations and invalid data transfer paths, constraint propagation is performed after steps (2) to (5) respectively.

In order to assign the current gene to a control step, the set of alternatives is determined: If cs_{max} is the highest assigned control step, the gene can potentially be assigned to the following control steps:

- Next control step $cs_{max} + 1$.
- Parallel execution with an operation (which is assigned to a control step $\leq cs_{max}$). This is only possible under the assumption of no resource conflicts.
- In the presence of CSEs it is possible that the result of an operation is already computed and is still available in a register resource. If the current operation can define (write) the same resource, an execution of this operation can be avoided. This is called *bypass* and is described in more detail below.

Finally, the `ready_set` is updated in step (7). Let $|V|$ and $|E|$ be the number of graph nodes and edges respectively. The complexity of this step is $O|V^3|$.

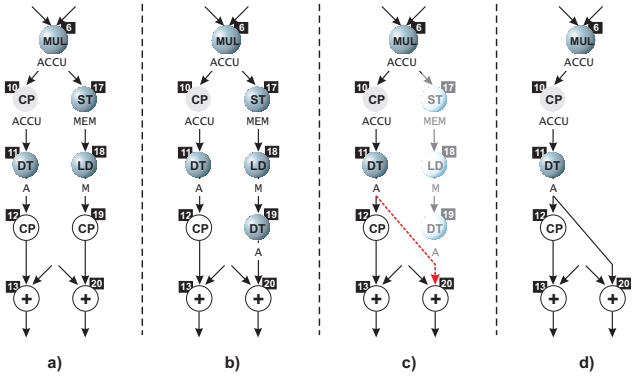

Fig. 3: Example for bypassing

The principle of bypassing
As mentioned before, the execution of a graph based code selection instead of a tree based approach bears an immense optimization potential. Thus, besides a more effective realization of an instruction covering, storing of CSEs can be potentially avoided. But, after insertion of all potential data transfer paths between two graph nodes, the result of a CSE (s. multiplication node in fig. 2) is transported on different data paths to succeeding nodes. As we will demonstrate using the example in fig. 3, bypassing helps to make multiple use of CSEs. All genes which already have been processed are shaded and are marked by the assigned operation and by the destination register resource (s. genes 6, 10, 11, 17 and 18 in fig. 3 a)). Gene 10 performs a *virtual* copy function which just transfers the data (from ACCU) without consuming a real processor instruction (to ACCU). Gene 11 performs a concrete data transfer (DT) from register resource ACCU to register A. In the right part, the CSE is stored into the memory (s. gene 17) and is then loaded from memory (gene 18). Part b) of the figure shows that gene 19 is assigned to a data transfer and the register resource A. Since the corresponding data resides already in register A (s. gene 11) the execution of this operation can be avoided. For this reason, a bypass is set from gene 11 to gene 20 (s. fig. 3 c)). This means that the memory accesses of gene 17 and 18 can also be avoided. This is dead code and can be eliminated (s. fig. 3 d)). In this way it is possible to eliminate data transfers and also memory accesses. The determination of whether specific genes define the same value is decided

[3] Assigning different instruction types for two specific genes means that these graph nodes must be scheduled to different execution cycles.

with the help of *value numbers* (s. e.g. [18]) which are computed once before starting the code generation process. Thus, two genes get the same value number if they define the same value.

D Evaluation

The evaluation of the individuals of a population is required to differentiate the solutions. Without loss of generality, we assume that individuals causing less costs represent better solutions than individuals with higher costs. The evaluation function has an essential impact on the optimization progress of the genetic algorithm because these values serve as a basis for the subsequent selection step. For example minimization of execution time is possible by counting the number of execution cycles. A high number of cycles corresponds to a low fitness. Effects of the subsequent address code generation phase can be taken into account by extending the evaluation function. Since execution of the address code generation for each individual would be a very time consuming process an approximation is needed. In our case we consider the number of necessary memory accesses as an approximation of the required address code. For this reason, among the solutions with minimal number of execution cycles those are preferred which cause the smallest number of memory accesses. The evaluation step can be done in $O(1)$.

E Crossover and Mutation

The *crossover* operator deals with generating new individuals by probabilistically swapping genes between two selected individuals. Unfortunately, standard crossover operators like *one point* or *uniform* crossover do not consider the special graph/tree structure of our optimization problem. For this reason, we have developed a special crossover operator *CS-crossover* which takes into account the execution order of the genes. Experimental results have shown a much faster convergence using the CS-crossover operator than standard operators. The basic idea is that all genes of a specific individual which are assigned before a specific control step cs_{cross} are left unchanged. This leads to valid partial solutions up to this control step. Only those genes are swapped which are assigned to a greater control step than cs_{cross}. The parameter cs_{cross} is determined for each individual again. In fig. 4 this is demonstrated for $cs_{cross} = 2$.

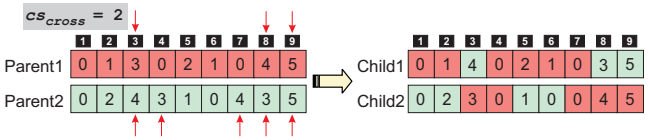

Fig. 4: CS-crossover

This step depends only on the number of graph nodes (genes) and can be done in $O(|V|)$. The results are two individuals consisting of recombined information of the parents. However, the large number of constraints (e.g. data dependencies or resource constraints) which have to be handled can lead to invalid solutions. In order to avoid such invalid solutions we combine the subsequent mutation operator with a correctness check. So, the main tasks of the *mutation* operator are to check the correctness of the actual allele and to generate the new gene material by changing alleles. Performing mutation for an individual is nearly the same task as initializing an individual. In analogy to the initialization step we determine a new allele of a gene by choosing an allele whose selection potentially leads to a valid solution. Thus, the complexity of the mutation step is $O(|V^3|)$.

V Experimental Results

In this section, experimental results of our genetic algorithm driven code generator are presented for two DSPs (the M3-DSP and the ADSP). Columns 1 and 2 of table 1 give an overview of the considered benchmarks. Except the MP3 application, all benchmarks are taken from the DSPstone benchmark suite [19].

Table 1: Benchmark characteristics

	benchmark	#CSEs	#CSE uses	CPU[s]
cm	complex_multiply	4	8	19
bos	biquad_one_section	3	7	49
cu	complex_update	4	8	37
nru	n_real_updates	1	2	21
conv	convolution	1	2	41
dot	dot_product	1	2	14
f2d	fir2dim	8	16	360
lms	lms	8	16	128
ncu	n_complex_updates	16	42	507
bNs	biquad_N_sections	10	31	631
fir	fir	5	10	107
m1	matrix1	5	10	128
m2	matrix2	8	16	285
m1x3	mat1x3	2	4	42
mp3	mp3	27	54	1034

Columns 3 and 4 give some characteristics about the number of common subexpressions in the source programs and their uses respectively. Finally, column 5 shows the CPU time requirements of our code generator on a 2.7 GHz Intel Pentium 4 processor. The number of generations to be executed by our code generator during code generation and address code compaction is set to 8 times the number of graph nodes. For example, the MP3 application has 56 basic blocks and 1105 IR operations to be compiled by our code generator. In order to compile a basic block with 50 operations, 400 (8×50) generations are performed. Note that in most cases the best result is already found early in the optimization process. For this reason, the runtime of the genetic algorithm could be drastically reduced in most cases without code quality losings.

All results are generated using a *steady-state* genetic algorithm with the following parameters[4]: population size: 30; number of individuals in the population to be replaced by the offspring: 4; mutation rate: 1/(number of graph nodes); crossover rate: 0.9.

Results for the M3-DSP and the ADSP w.r.t. execution time are given in figures 5 and 6. The code quality using a tree based code selection and a restricted phase coupling as common in traditional code generators is set to 100%. Results

[4]The implementation of the base genetic algorithm uses the genetic algorithm library PGAPack [20].

labeled `graph` mean that a graph based code selection technique without bypassing is performed. `graph+bypassing` shows the effectiveness of the bypassing technique. Finally, `graph+bypassing+pc` indicates that a phase coupling of CS, IS and RA instead of a restricted phase coupling is done by also integrating the compaction phase into the genetic algorithm and taking into account the effects of address code generation.

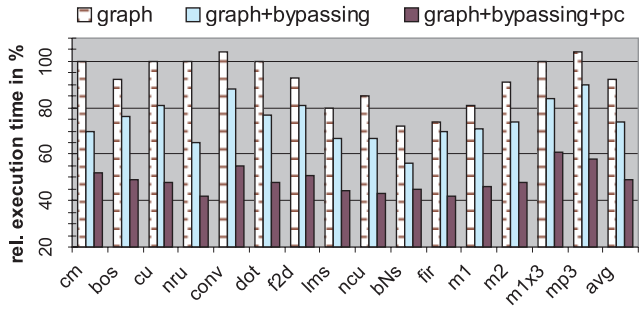

Fig. 5: M3-DSP: Results w.r.t. execution time

The pictured results show that the execution time can be dramatically reduced for both DSPs by using the genetic algorithm driven code generator. Performing a graph based CS technique with bypassing (s. `graph+bypassing`) for the M3-DSP already leads to an average (`avg`) improvement of 26% and 23% for the ADSP. Compared to `graph`, this also shows the benefit of the bypassing technique for reducing the execution time. The code quality is further improved by performing a complete phase coupling (s. `graph+bypassing+pc`). In contrast to `graph+bypassing`, instruction compaction is integrated into the phase coupling of the genetic code generator and the effects of address code generation are taken into account. Compared to `graph+bypassing` this leads to an average improvement of 34% for the M3-DSP and 19% for the ADSP and demonstrates the benefit of the proposed techniques. Altogether the execution time is reduced by 51% on average for the M3-DSP and 38% for the ADSP.

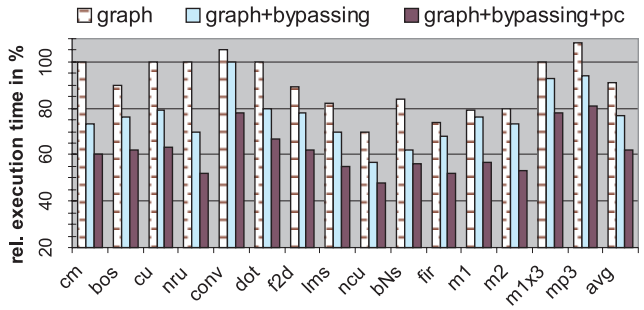

Fig. 6: ADSP: Results w.r.t. execution time

VI Conclusions

The growing use of DSPs in embedded systems necessitates optimizing compilers which are capable of making use of the irregular processor architecture features. Due to the strong interdependencies between code selection, instruction scheduling, register allocation and address code generation, there is a demand for code generation techniques performing phase coupling of these subtasks. In this paper we have presented a genetic algorithm driven code generator which performs graph based code selection and complete phase coupling of code selection, instruction scheduling (including compaction) and register allocation. In addition, our code generator takes into account effects of the subsequent address code generation phase. Results for several benchmarks and an MP3 application for two DSPs show the effectiveness and the retargetability of our genetic code generator. The number of execution cycles is reduced by 51% on average for the M3-DSP and by 38% on average for the ADSP compared to standard techniques. Due to the generic implementation of our techniques on a common compiler framework, these techniques can be applied to other processor architectures, too.

References

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

[2] S. Liao, S. Devadas, K. Keutzer, and S. Tjiang. Instruction Selection Using Binate Covering for Code Size Optimization. In *Proc. of the ICCAD*, 1995.

[3] G. Araujo, S. Malik, and M. Lee. Using Register Transfer Paths in Code Generation for Heterogeneous Memory-Register Architectures. In *Proc. of the DAC*, 1996.

[4] R. Leupers. Register Allocation for Common Subexpression in DSP Data Paths. In *Proc. of the ASP-DAC*, 2000.

[5] S. Novack and A. Nicolau. Mutation Scheduling: A Unified Approach to Compiling for Fine–Grain Parallelism. In K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, LNCS. Springer–Verlag, 1994.

[6] S. Hanono and S. Devadas. Instruction Selection, Resource Allocation, and Scheduling in the AVIV Retargetable Code Generator. In *Proc. of the DAC*, San Francisco, California, USA, 1998.

[7] T. Wilson, G. Grewal, B. Halley, and D. Banerji. An Integrated Approach to Retargetable Code Generation. In *Proc. of the International Symposium on High-Level Synthesis*, Ontario, Canada, 1994.

[8] D. Kästner and M. Langenbach. Integer Linear Programming vs. Graph-Based Methods in Code Generation. Technical Report A/01/98., Universität des Saarlandes, 1998.

[9] S. Bashford and R. Leupers. Constraint driven Code Selection for Fixed-Point DSPs. In *Proc. of the DAC*, 1999.

[10] S.J. Beaty. *Instruction Scheduling Using Genetic Algorithms*. PhD thesis, Department of Mechanical Engineering, Colorado State University, Fort Collins, Colorado, USA, 1991.

[11] T. Zeitlhofer and B. Wess. Operation Scheduling for Parallel Functional Units Using Genetic Algorithms. In *Proc. of the ICASSP*, Phoenix, Arizona, USA, 1999.

[12] S. Fröhlich. *Codegenerierung für Signalprozessoren mit Hilfe genetischer Algorithmen*. PhD thesis, Technische Universität Wien, 2001.

[13] M. Lorenz, T. Dräger, R. Leupers, P. Marwedel, and G.P. Fettweis. Low-Energy DSP Code Generation Using a Genetic Algorithm. In *Proc. of the ICCD*, Austin, Texas, USA, 2001.

[14] 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 the ICSPAT*, Toronto, Canada, 1998.

[15] Analog Devices. *ADSP-2001 User's Manual*, 1991.

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

[17] K.R. Baker. *Introduction to Sequencing and Scheduling*. Wiley, 1974.

[18] S.S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[19] V. Zivojnovic, J.M. Velarde, C. Schläger, and H. Meyr. DSPstone - A DSP-oriented Benchmarking Methodology. In *Proc. of the ICSPAT*, Dallas, Texas, USA, 1994.

[20] D. Levine. Users Guide to the PGAPack Parallel Genetic Algorithm Library. Technical Report ANL-95/18, Argonne National Laboratory, 1996.