Electronic Notes in Theoretical Computer Science 65 No. 2 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume65.html 0 pages

Verified Code Generation for Embedded Systems

Sabine Glesner Rubino Geiß Boris Boesler

Institut für Programmstrukturen und Datenorganisation
Universität Karlsruhe, 76128 Karlsruhe, Germany
Tel.: +49-721-608-7399, Fax: +49-721-30047
Email: {glesner|rubino|boesler}@ipd.info.uni-karlsruhe.de

Abstract

Digital signal processors provide specialized SIMD (single instruction multiple data) operations designed to dramatically increase performance in embedded systems. While these operations are simple to understand, their unusual functions and their parallelism make it difficult for automatic code generation algorithms to use them effectively. In this paper, we present a new optimizing code generation method that can deploy these operations successfully while also verifying that the generated code is a correct translation of the input program.

1 Introduction

We address the problem of generating optimized as well as verified code for digital signal processors in embedded systems. Digital signal processors (DSPs) exhibit irregular architectures which allow in particular for simultaneous execution of multiple operations, typically SIMD (single instruction multiple data) instructions with the same operation performed on two or four independent pairs of operands. Such operations are specialized for a certain application area and perform complex computations. The goal of our work is a uniform method for generating optimized and correct (wrt. the input program) code for DSPs which is suitable not only for an individual case but for various kinds of DSPs.

In particular, we want our code generation method to meet the following criteria: It should exploit the parallelism of DSPs given in the form of SIMD instructions as much as possible by scheduling suitable computations in parallel, restricted only by the given data dependencies. Moreover, the result of this optimization should be verified. Especially in embedded systems, this is an important criterion since subsequent improvements are often impossible as the machine code of an application might be hardwired into the embedded system. In the field of compiler construction, most translation methods can

©2002 Published by Elsevier Science B. V.

be implemented in generators, thus simplifying life for compiler programmers. Therefore, we require our code generation method to be implementable as a generator. Finally, we want to use as much of the well-established methods of compiler construction as possible since they are best practice and simplify our life in two aspects: We can fall back on well-understood theoretical foundations instead of eventually redeveloping them from scratch, and we can integrate our method in existing compiler construction tools, i.e., compiler generators.

Verified and optimizing code generation is an important goal in the software development process for embedded systems. Embedded systems consist of hardware and software which are employed in a technical environment. Characteristically, they are specialized for a fixed task so that their hardware and their software can be tailored to their application area. This specialization is necessary to keep them competitive. Typically, complex computations need to be done very quickly. Such tasks are done by DSPs which are optimized with regard to special algorithms and which possess specialized irregular hardware structures. The deployment of such processors is cheaper than that of standard processors as one would need several standard processors to compete with the performance of a single DSP. Such arguments may not be neglected since the market for embedded systems is a mass market and very cost sensitive. In this market we see the trend that more and more parts of functionality are implemented in software instead of in hardware. This gives us the advantage of more flexibility. The same hardware can be adapted easily to slightly different environments. Consequently this implies that software should be written in higher programming languages (which is not the standard right now) to make it less error-prone. This, in turn, means that compilers for embedded system processors are necessary. We concentrate on DSPs which are the most complex case of processors in embedded systems. Compilers for DSPs need to be highly optimizing because the performance requirements of embedded systems are tight. Moreover, they need to be correct since maintenance in the sense of error correction and further developments is not possible as many programs are hardwired into the embedded system.

At first sight, the requirements on a compiling method to be verifying, optimizing, and to be implementable as a generator, especially by building up on already existing generators, seem to be incompatible since it would be much too expensive, if not impossible, to automatically verify the generator implementation or the generated code. Instead, we choose the method of program checking [2,7,3,38] which has already been applied successfully in the Verifix approach [17,13,19,20] which shows how to construct compilers for standard processors that check their results.

The process of code generation in a compiler starts after the computation of the intermediate representation of a program. In our approach, we choose a static single assignment intermediate representation since it shows nothing but the functional data dependencies which are the only constraints to be respected when scheduling computations in parallel. Typically, the code gen-

eration phase in a compiler is partitioned into several phases which are code selection, register allocation, instruction scheduling, register assignment, and resource assignment. Even though there is a connection between them, one tries to solve them independently from each other in order to keep the complexity within a reasonable scope. Since it is our premise to apply the well-understood methods of compiler construction whenever possible, we choose the same partitioning of tasks in the code generation phase.

In this paper, we concentrate on code selection and instruction scheduling since they are the crucial phases in the code generation process of a compiler if one wants to exploit the SIMD parallelism. Since the static single assignment intermediate representation of a program is a graph whose nodes are operations and whose edges represent the data flow of the program, one can do the code selection via a graph rewrite system that maps subgraphs of the intermediate representation to instructions of the target processor. The special problem when generating DSP code are the SIMD instructions that can perform the same operation on different independent pairs of operands. This means that a purely graph rewriting based approach is not appropriate since then, suitable pairs of operands could only be found in a local context. We overcome this problem by transforming the processor architecture into an equivalent one which can be handled with an extended graph rewriting mechanism: Whenever an operational unit on a processor computes an SIMD operation, i.e., n operations in parallel, this unit is replaced by n operational units which perform the same operation but each only on one pair of operands. To specify that these n units show the same behavior as the original unit, we require that whenever one of them starts a computation, the others must start their computations simultaneously or wait until it is finished. This requirement can be expressed with a set of constraints. Whenever the graph rewriting mechanism finds a subgraph in the intermediate representation that can be computed with an SIMD operation, then the corresponding machine code is associated with that subgraph and, moreover, the corresponding set of constraints is generated. In the instruction scheduling phase, we need to take these constraints into account in order to get a valid schedule of instructions. Moreover, we use the constraints in the verification phase as proof obligations which must be fulfilled by the generated code.

This paper is structured as follows: In section 2 we give the foundations of our work which are program checking in the field of compiler construction, static single assignment representations, and graph rewrite systems. Section 3 shows how the mechanism of graph rewriting can be extended so that the application of rules induces additionally the generation of constraints. Furthermore, this section shows how this mechanism can be utilized in the code selection and instruction scheduling phase for DSPs. The result of this code generation can be verified as presented in section 4 using the generated constraints as proof obligations. In section 5, we show how this code generation method can be integrated into our existing compiler generator environment.

Thereafter, in section 6, we discuss related work. Finally, we conclude in section 7 with a characterization of future work.

2 Foundations

Foundations of our work are twofold since we are concerned with verification as well as optimization. Our verification approach is based on Blum's idea of program checking, shown in subsection 2.1. Foundations of the optimizing part of our work are a particularly suited static single assignment (SSA) form, implemented in the language FIRM, which we introduce in subsection 2.2 as well as graph rewrite systems in the code generation phase which we discuss in subsection 2.3.

2.1 Trust is good, control is better

The idea of program checking [2,7,3,38] can be adapted for compilers as shown in the Verifix project [39,17,13,19,20]. Here we summarize the relevant results of Verifix.

When defining the correctness of a compiler, one needs to consider two aspects: the correctness of the specification of a compiler and the correctness of its implementation. Both need to be correct. Given a source program π, a compiler specification C defines a target program $\pi' = C(\pi)$. The translation defined by C is correct if π' shows the same behavior as π. To define the notion of "same behavior", we look at the observable states of a program. These observable states are initial and final states as well as all states which are reached by input and output operations. If required, more states as e.g. procedure entries and exits may be defined as observable. A compiler specification C is correct iff each sequence of observable states in π' has a corresponding sequence of observable states in π and iff for each sequence of observable states $q_0, \ldots, q_k$ in π terminating with an error message after k steps, there is a corresponding sequence of observable states of length k in π.

Due to resource limitations, a compiler may not be able to translate programs of arbitrary length. Mathematically speaking: For nearly all programs a compiler does not work correctly due to resource limitations. This means that we cannot expect a compiler to be correct for all input programs. Instead we use the notion of a verifying compiler: A compiler is a verifying compiler if the translated target program preserves the observable behavior of the source program up to resource limitations. A verifying compiler does not need to produce a target program for every source program. To get a practicable method, the set of correctly translated programs must be sufficiently large.

To establish the correctness of a code generation tool, we need to do two major tasks: We need to prove first that the code generation algorithm is correct in the sense that it preserves the semantics of the transformed prorgams and furthermore that its implementation is correct. For the first task, namely

the establishment of the correctness of the transformation algorithm, we would need to define the semantics of the intermediate program representation as well as the semantics of the machine code formally. Based on these technical means, we could formally prove that the semantics is preserved. While we will deal with this problem in future work, we concentrate in this paper on the second task.

2.2 Static Single Assignment Representations

Static single assignment (SSA) form [10,11,9] has become the preferred internal program representation for handling all kinds of program analyses and optimizing program transformations prior to code generation. Its main merits comprise the explicit representation of def-use-chains and, based on them, the ease by which further dataflow information can be derived.

By definition SSA-form requires that a program is represented as a directed graph of elementary operations (jump, memory read/write, unary or binary operation) such that each "variable" is assigned exactly once in the program text. Only references to such variables may appear as operands in a unary or binary operation. Thus, an operand explicitly indicates the data dependency to its point of origin. The directed graph of an SSA-representation is an overlay of the control flow and the data flow graph of the program.

SSA-form is a very elegant and easily comprehensible program representation as long as we only concentrate on handling local variables of the current procedure. Handling accesses to non-local variables, arrays, record fields, object attributes, etc., in general: handling of memory accesses, leads to additional complexities.

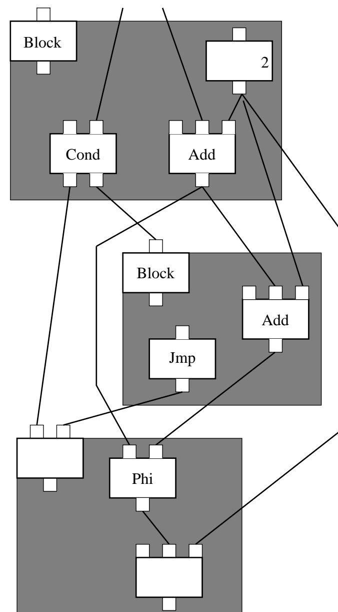

Figure 1. Firm Example

Martin Trapp [37] introduced *explicit dependency graphs*, a refined kind of SSA-form, for dealing with these additional problems. These explicit dependency graphs have been implemented as an abstract data type in our library FIRM, [36]. Its properties may be summarized as follows.

Viewed from the outside a program representation in FIRM may be successively generated by adding nodes of five different kinds: *Data nodes* represent unary and binary operations (arithemtic, logical, ...); incoming edges represent their operands; outgoing edges represent the use of the result. *Block nodes* represent basic blocks; all other nodes are associated with the basic block to which they belong. *Control nodes* represent jumps and procedure returns. Further, a control node may depend on a value which forces control to conditionally follow a selected path. E.g. in the implementation of a conditional statement the value of the condition forces control into the then- or the else-alternative. Each block node has one or more such control nodes as its predecessor. At entry to a basic block ϕ *nodes*, $x = \phi(x_1, \ldots, x_n)$, represent the unique value assigned to variable x as a selection amongst the values $x_1, \ldots, x_n$ where x_i represents the value of x defined on the control path through the i-th predecessor of the block node; n is the number of predecessors of the block node. *Memory nodes* represent memory accesses, i. e. accesses to non-local variables as discussed above. In a first approximation memory nodes may be considered as accesses to fields of a global state variable *memory*. But refinements of this picture allow for a more detailed analysis. This analysis may then exhibit which memory accesses address overlapping memory areas and thus are truly dependent on each other. Without such additional analysis all memory accesses must be held in strict temporal order; reordering may lead to wrong values being fetched from memory.

A FIRM representation may easily be generated during a tree walk through the attributed syntax tree as generated from a compiler front-end. Further operations of the FIRM library allow for navigation and queries on the representation and for transformations of the representation. Final code generation may be viewed as a pattern matching process on the SSA represenation of a program.

As an example figure 1 shows the FIRM representation for the program fragment:

```
a := a+2; if(..)  { a := a+2; } b := a+2
```

In the first basic block, the constant 2 is added to a. Then the *cond* node passes control flow to the “then” or to the “next” *block*, depending on the result of the comparison. In the “then” *block*, the constant 2 is added to the result of the previous *add* node. In the “next” *block*, the *Phi* node chooses which reachable definition of variable a to use, the one before the if statement or the one of the “then” *block*.

2.3 Graph Rewrite Systems in the Code Generation Phase

Rewrite systems are a known technique in compiler construction for code generation. In most cases the intermediate representation of a program is a set of trees which are rewritten in the code generation phase by techniques like Bottom Up Pattern Matchers (BUPM) [16,14,15] or Bottom Up Rewrite Systems (BURS) [28]. Both mechanisms use *rules* to rewrite a tree: E.g. $(+(r, c) \rightarrow r, code)$ is a rule to rewrite an addition of two operands by its result. The target code *code* for this rule is emitted simultaneously. We use graphs instead of trees because therewith we do not loose any information of the semantic analysis about the program which we would have to recompute again for optimizations when using trees.

In our intermediate SSA representation FIRM, the program is represented by a graph with explicit data and control flow dependencies. Therefore it is the method of choice to use a graph rewrite system when rewriting FIRM graphs during code generation. Our code generator generator (called CGGG) [8] reads a specification and generates a code generator which uses the BURS mechanism of graph rewriting. The code generator has two major steps: first to calculate all possible rule covers of a graph and then to find the most inexpensive one.

The first step is based on the rules mentioned above. A rule has an acyclic pattern which will be rewritten by a second acyclic pattern if the first pattern matches a subgraph (see example below). In the second step an A^*-search algorithm searches for the cover with the lowest cost. It returns a path in the search graph of rewrite rules. The code is emitted by traversing this path and executing the corresonding code at every node of the path which prints the target code to a file.

The search must take care of at least two problems: the first problem arises if a node (*Phi*) wants to use an input at the beginning of a basic block which is produced at the end of another basic block. This happens in all kinds of `for` loops. Secondly a node in a basic block wants to use an input which has not been produced yet because it is produced in another basic block which has not been handled yet. In both cases the search generates conditions to notify that there are nodes which have to produce a special result or else the code will be incorrect.

An example for a rule from a code generator specification is:

```
RULE a:Add (b:Register b) -> s:Shl (d:Register c:Const);
RESULT d := b;
EVAL { ATTR(c, value) = 1; }
EMIT {}
```

This rule describes an addition of two operands. On the left hand side of the rule, the first operand is a register with short name `b`. The second operand is the first operand again, identified by the short name. The left-hand side of the rule is a directed acyclic graph. If the code generator finds this pattern in the

graph, it rewrites it with the right side of the rule. This could be a DAG again. After rewriting the `EVAL` code is executed. This code places the constant `1` in the attribute field `value` of the `Const` node. The `RESULT` instruction informs the register allocator that the register `d` equals register `b`.

3 Code Generation for DSPs

Characteristically, DSPs employ SIMD instructions which execute uniform complex computations on independent data in parallel. If one writes assembler code for such processors, one can make sure that uniform computations are combined properly in order to exploit the SIMD parallelism. In contrast, when using higher programming languages, one solves problems algorithmically, whereby it is much harder to associate independent computations in order to parallelize them. It is the job of a highly optimizing compiler - and the goal of the work described in this paper - to detect and correlate independent computations which can be executed in parallel. As technical means we introduce graph rewrite systems with verification constraints. During code selection, verification constraints are generated. These constraints are used to guide the instruction scheduling phase. Furthermore, the verification constraints are checked in the verification phase, confirm section 4.

3.1 Motivation

To motivate our method, let us undertake a thought experiment: Assume an SIMD operational unit that takes n pairs of operands and returns n results. Notionally, we can also think of n separate operational units, each taking one pair of operands and returning one result. These n operational units work either at the same time or do not work at all, i.e., their execution never overlaps. With this treatment, we have nearly the same situation as in the code generation for standard processors: Given the intermediate representation of a program, we look for local computation sequences that can be computed by the functionality of a DSP instruction. In technical terms this means that given the SSA graph representation of a program, we search for local rule covers for the n partial DSP instructions. To get a valid instruction sequence for the real DSP, we need to put these partial DSP instructions together globally by combining up to n partial independent instructions into a single DSP instruction. We use constraints to express this global connection. Constraints are typically the method of choice whenever one wants to express global correlations because the logical variables used in the constraints are either not instantiated or have the same value globally, thus carrying and transporting local information.

3.2 Graph Rewrite Systems with Verification Constraints

We use the BURS graph rewrite method described in subsection 2.3 and extend it by a constraint-generating mechanism. In the original form, the graph rewrite rules are triples $(lhs, rhs, code)$ where lhs and rhs are the left- and right-hand side of the rule and *code* is the code emitted when rewriting the graph with that rule. We use quadrupels $(lhs, rhs, code, vars)$ instead. lhs, rhs, and *code* have the same aforementioned meaning. $vars$ is a set of rule-specific variables and predicates which is generated and indexed with a unique number when rewriting the graph with that rule instance. This means that whenever the same rule is applied several times, a new version of $vars$ is generated for each application.

Given a DSP instruction set, we decompose each SIMD instruction that computes n separate results into n independent partial DSP instructions computing only a single result. For each such DSP operation *op*, we introduce the global variable t^{op}_{max} representing the amount of time which is needed at most to execute the DSP operation. When applying a rule describing a partial DSP instruction, i.e. with *code* being a partial DSP instruction *op*, we generate the variable t^{op}_i and the unary predicate $\chi^{op}_i(t)$ where i is the unique number for this rule application. The intended meaning is the following: t^{op}_i is the time at which execution of the partial DSP instruction on the decomposed DSP from our thought experiment starts. $\chi^{op}_i(t)$ is 1 if $t = t^{op}_i$ and 0 otherwise.

During code selection, we apply rules to nodes in the graph and thereby emit code as well as variables and predicates. The idea is to find an assignment to the generated variables that defines a parallel scheduling of partial DSP instructions. Such a parallel schedule must fulfill two conditions:

Simultaneousness Condition This condition states that the n partial DSP instructions either execute in parallel or sequentially, but not overlappingly:
$\forall i \forall j \forall op : t^{op}_i = t^{op}_j \vee \mid t^{op}_i - t^{op}_j \mid \geq t^{op}_{max}$

Booked up Condition This condition states that for each time t, no more than n partial DSP operations may be assigned to the same original DSP unit:
$\forall t \forall op : \sum_i \chi^{op}_i(t) \leq n_{op}$

During the instruction scheduling, partial DSP instructions must be merged such that these two conditions are true.

Remark: In future work, we also want to address the register assignment problem. The SIMD instructions in DSPs usually take two input registers carrying the n pairs of input values and return a single output register carrying the n results. Therefore it will be necessary to generate a further kind of variables to represent the registers and in particular the register portions which carry the input and output values of the SIMD instruction. It is the task of the register assignment to find a valid assignment to these register variables.

3.3 Code Selection and Instruction Scheduling for DSPs

Given a graph rewrite system with verification constraints, we can generate the code selection and instruction scheduling phases. The principle is the same as in the case of code generation for standard processors extended by the special treatment of the DSP instructions. Here, we give the algorithm for the code selection and instruction scheduling phases. Input to this algorithm is a FIRM graph as intermediate program representation as well as a graph rewrite system with verification constraints.

(i) **Search locally for all possible rule covers.**
Result of this step: Each node in the graph has assigned all rules whose left-hand side matches the directed acyclic graph starting at this node.

(ii) **Search for a global rule cover** guided by a cost function implementing the following two heuristics:
Prefer DSP Instructions: We assume that DSPs are utilized in problem areas which can be solved more efficiently with the complex DSP operations than with the operations of a standard processor. This means that we need to prefer the use of DSP instructions. When searching for a global rule cover, we prefer rules whose emitting code are DSP partial operations.
Prefer More Complex DSP Instructions: Moreover, the idea of DSPs is to encapsulate complex computations into a single operation. This means that we need to prefer those graph rewrite rules whose left-hand side covers larger parts of the graph than those of other rules.
Result of this step: An annotated FIRM graph such that a rule is assigned to some nodes in the graph. Note that if a node is in the inner part of a matched region then it does not need to have a rule assigned to it since it will be rewritten by the rule associated to one of its predecessors.
Remark: Besides the special design of the cost function, there is no difference between this step and the corresponding step in the code generation for standard processors.

(iii) **Compute a sequential schedule** of the selected instructions.
Starting at the node in the FIRM graph that represents the result of the program all nodes are collected inductively whose results are needed to compute the result of the program. In doing so we can put the nodes into a sequential schedule.
Result of this step: A valid sequential schedule.
Remark: Again, there is no difference between this step and the corresponding step in the code generation for standard processors.

(iv) **Compute a parallelized schedule** by putting several data independent partial DSP instruction into a single DSP instruction.
Note that two operations are data independent iff there is no directed

path between their corresponding nodes in the FIRM graph. In this step we go through the linear schedule and put partial DSP operations together whenever they are data independent and whenever all operations scheduled inbetween them are also data independent. Thereby we must take care that the simultaneousness condition and the booked up condition are fulfilled. To ensure the second condition, we must take care not to schedule more than n partial operations in parallel. The first condition is ensured by the bevavior of the DSP since in a linear schedule, an instruction is only executed if the predecessing instructions are completed. (At least, this is the behavior which the processor shows to the outside, no matter if instructions are scheduled out of order or in a superscalar form.) It may be the case that we are not able to always schedule exactly n but only a number smaller than n partial DSP operations in parallel due to data dependencies in the FIRM graph.

Example: DSP Operations in the TriMedia Processor
The TriMedia processor [33] developed by Philips is a DSP especially designed for multimedia applications. It incorporates some instructions for the processing of videos according to the MPEG standard. During the decoding of videos, a special instruction of the TriMedia, `ume8uu`, can be used to speed up the computation. It computes the unsigned sum of absolute values of unsigned 8-bit differences.

This instruction `ume8uu` is a typical SIMD operation computing the four results of four independent pairs of operands. To generate code for the TriMedia, we need to model this instruction by four independent partial instructions computing exactly the same result for a single pair of inputs. During instruction scheduling we need to search for data independent partial `ume8uu` instructions which can be scheduled in parallel.

4 Verification Using Program Checking

In this section, we show how to establish the correctness of a code generator implementing the method proposed in section 3. Thereby we assume that the underlying graph rewrite system is correct. (The proof of such an assumption is subject to future work.) A transformed program is correct if the following conditions are satisfied:

- I The global rule cover is a correct rule cover of the FIRM intermediate program representation.
- II The graph rewrite rules used to generate the machine code must have been applied correctly.
- III Parallelization of the sequential schedule does not change the sequential order of data-dependent instructions and fulfills the booked up as well as the simultaneousness condition.
- IV Memory and registers are written correctly.

For the verification of the code generation implementation, we use a checker approach. Practically, it would be impossible to verify the code generator generated by a code generator generator or the code generator generator itself. We avoid this by applying program checking for verifying the results of the code generation phase. By restating the correctness conditions I to III, we define the following five verification tasks for the checker:

(i) Check if the input FIRM graph and the annotated FIRM graph are identical if one forgets about the graph rewrite rule annotations.

(ii) Check if the applied graph rewrite rules indeed match the corresponding subgraphs of the FIRM graph.

(iii) Check if the schedule is valid, i.e., check if all subgraphs are evaluated according to the BURS method.

(iv) Recompute the code generator result by applying the graph rewrite rules again. Thereby check if the recomputed result and the original result are identical.

(v) Check if the data-dependencies are sustained during the parallelization of the sequential schedule: Therefore check if the order of the parallelized schedule is a topological order of the FIRM graph. Moreover, check that no more than n partial DSP operations are scheduled in parallel so that the booked up condition is valid.

The first task implements condition I, the second, third and fourth implement condition II, and the fifth implements condition III. In future work we will investigate how to check condition IV. (The simultaneousness condition is ensured automatically because of the sequential operational semantics of the target processors: Operations in the sequential instruction schedule are executed in that order or in a reordering showing exactly the same effect.)

On first sight, one might think that the checking tasks are as expensive as the original computation but this is not true: When checking the correctness of code generation, we are not interested at all if the result is optimal. We only care about correctness. This means that the code implementing the search for an optimized or even optimal instruction sequence does not need to be verified. Only certain intermediate steps as well as the result need to be verified, thus simplifying the verification task considerably.

The verification tasks are independent of concrete source or target languages but do only depend on an SSA intermediate representation. Therefore we do not need to implement the checkers for different pairs of source and target languages but instead, we can generate them. In doing so we only need to verify the checker generator implementation once instead of verifying each checker implementation separately.

5 Integration

The current version of our code generator generator reads a specification for a code generator and generates a graph rewrite system based on BURS. The code generator calculates all possible covers and searches for the most inexpensive one. When applying a rewrite rule in the most inexpensive cover, the corresponding code is not emitted directly but instead, `Emit` code written in C is executed. This code prints the target (e.g. assembler) code to a file.

We will extend this mechanism to generate the variables, predicates, and constraints by extending the `EMIT` code to print the additional constraint information. This extension can be done easily. The result of the code generation computed by the generated code generator is a sequential schedule with partial DSP operations together with the constraints.

In the next phase, we check the existing FIRM graph for data independent partial DSP operations and combine them with other suitable partial DSP operations to become a full DSP operation, as described in subection 3.3. Remember that two operations in the FIRM graph are independent iff there is no directed path between them. For a single pair of operations, this check can be done in quadratic time $\mathcal{O}(n^2)$ where n is the number of nodes in the FIRM graph. It will be the task of future research to find algorithms and data structures checking this property as efficiently as possible.

To implement a checker, we need to implement the steps 1. to 5. as described in section 4. This can be done straightforwardly. Moreover, we can also implement a checker generator as discussed also already in section 4. These implementations are subject to future work.

As we have shown, the modifications necessary to implement the methods described in this paper are simple so that we can add them easily to our existing tools. In future work we will investigate if the methods from [26] may be used to generate DSP parallelizers automatically.

6 Related Work

Related work concerns two aspects of compiler research: the construction of provably correct as well as optimizing compilers for embedded systems.

Verifix [17,13,19,20] is a common research project of the german universities in Karlsruhe, Kiel, and Ulm developing methods for the construction of correct compilers translating sequential real-world programming languages into the machine code of standard processors. This project has achieved progress by establishing the claim that it is possible to build provably correct compilers within the traditional framework of compiler construction, especially by deploying generators. In particular, a notion of correctness has been given which is based on the observable behavior of a program. Program checking [2,7,3,38] is used to keep the verification cost within a reasonable limit. The verifix project did not address the problem of irregular target ar-

chitectures. In [27], it is shown how some backend optimizations of the GCC can be checked. Proof-carrying code [29,30,31,12] is another weaker approach to the construction of correct compilers which guarantees that the generated code fulfills certain necessary correctness conditions. During the translation, a correctness proof for these conditions is constructed and delivered together with the generated code. A user may reconstruct the correctness proof by using a checking program which implements basically a syntax-oriented type checking algorithm. Pnueli [35,34] also addresses the problem of constructing correct compilers, but only for very limited applications. Only those programs consisting of a single loop with loop-free body are considered and translated without the usual optimizations of compiler construction. Thereby, such programs are translated correctly such that certain safety and liveness properties of reactive systems are sustained. In more recent work [40], he proposes a theory for validating optimizing compilers which is similar to the method developed in the Verifix project, cf. for example [39,19].

Optimizing code generation methods can be based successfully on term rewriting mechanisms: In [16,14,15], the generation of the code selection phase in a compiler is shown. [28] reformulate the BURS (Bottom Up Rewrite Systems) approach originally developed by [32]. BURS is the basis of [8], a generator for graph-rewriting code generators, as already explained in more detail in subsection 2.3.

In the context of embedded systems, there are several approaches tackling the optimizing code generation for DSPs: The Joses project (Java and CoSy Technology for Embedded Systems) [1] develops a compiler environment for the use of Java in embedded systems. AJACS (Applying Java to Automotive Control Systems) [18] is a project within the *Information Societies Technology (IST) Programme* and investigates the use of Java in automobile applications. A survey of code generation for embedded processors can be found in [25]. [23] examines the utilization of genetic algorithms and [4,5] the use of constraint-based approaches for the generation of optimized code for embedded systems (which is restricted to optimizations within basic blocks). Moreover, it is studied how compilers can be generated such that they can be adapted to different target architectures. If it is possible to simulate these target architectures [24], then the results can help in the design of new hardware structures. A survey of these approaches can be found in [6]. Last but not least there are approaches [22,21] using integer linear programming in the optimizing (but not verifying) code generation for irregular target architectures. None of these works has the goal to generate provably correct code.

7 Conclusions

Compilers for DSPs in embedded systems need to generate optimized as well as provably correct code. In this paper, we have shown how to reach these two goals: When translating a program into the machine language of a DSP,

we schedule as many computations as possible in parallel, restricted only by the data dependencies given in the source program and by the number of available operational units. As technical means, we extended graph rewrite systems to graph rewrite systems with verification constraints which identify local program parts computable by SIMD operations and which connect these local program parts by constraints that must be fulfilled in the generated code. To ensure the correctness of this translation, we use program checking which verifies that the result of the translation is correct by proving the verification constraints. In particular, it is checked that the functional dependencies of the source program are sustained and that the operational units of the processor are used correctly. Using program checking gives us the advantage that we do not need to verify neither the implementation of the code generator nor the code generator generator producing it. We only need to verify that the result of the code generation is correct. Such an approach is helpful whenever the correctness check of a solution is much easier than the computation of the solution itself. Program checking allows us to stay within the existing compiler construction framework, especially by allowing us to use unverified generators when computing verified results. The presented code generation method succeeds not only for a specific DSP but instead offers a methodology of correct code generation that is applicable for a wide range of target architectures.

This paper presents work in progress which we want to complete and improve in several aspects: In future work we will investigate how to design the cost functions that control the utilization of DSP instructions during code generation. Furthermore, as already discussed, when generating code for SIMD instructions, we need to detect data dependencies by exploiting the structure of the FIRM graph. We will investigate which data structures and algorithms are suitable for computing this task efficiently. Moreover we want to study the remaining phases of code generation, i.e. register allocation, register assignment, and resource assignment, with particular emphasis on our special case of DSP operations where independent input values need to be stored in the same register. Last but not least we want to implement the presented methods in our existing compiler tool environment. The presented methods do not only work for SIMD instructions but seem to be applicable also to VLIW processors. We will exploit these possibilities in future work. Finally, we want to test our code generation method in real-world applications.

Acknowledgment: We want to thank Gerhard Goos, Götz Lindenmaier, Florian Liekweg, and Wolf Zimmermann for many valuable discussions and comments on the paper.

References

[1] U. Aßmann, D. Genius, P. Fritzson, H. Sips, R. Kurver, R. Wilhelm, H. Schepers, and T. Rindborg. Java and cosy technology for embedded systems:

The joses project. In *Proc. of the European Multimedia, Microprocessor Systems, Technologies for Business Processing and Electronic Commerce Conference (EMMSEC'99)*, Stockholm, Sweden, 1999.

[2] M. Blum and S. Kannan. Program correctness checking ... and the design of programs that check their work. In *Proceedings 21st Symposium on Theory of Computing*, 1989.

[3] M. Blum and S. Kannan. Designing programs that check their work. *Journal of the ACM*, 42(1):269–291, 1995. Preliinary version: *Proc. 21st ACM Symp. Theory of Computing (1989)*, pp. 86-97.

[4] Steven Bashford and Rainer Leupers. Constraint Driven Code Selection for Fixed-Point DSPs. In *36th Design Automation Conference*, New Orleans (USA), June 1999.

[5] Steven Bashford and Rainer Leupers. Phase-Coupled Mapping of Data Flow Graphs to Irregular Data Paths. *Design Automation for Embedded Systems*, 4(2/3), June 1999.

[6] Shuvra S. Bhattacharyya, Rainer Leupers, and Peter Marwedel. Software Synthesis and Code Generation for Signal Processing Systems. Technical Report Technical report UMIACS-TR-99-57, Institute for Advanced Computer Studies, University of Maryland, College Park 20742, USA, September 1999.

[7] M. Blum, M. Luby, and R. Rubinfeld. Self-Testing/Correcting with Applications to Numerical Problems. *Journal of Computer and System Sciences*, 47(3):549–595, 1993. Preliminary version: *Proc. 22nd ACM Symp. Theory of Computing (1990)*, pp. 73-83.

[8] Boris Boesler. Codeerzeugung aus Abhängigkeitsgraphen. Diplomarbeit, Universität Karlsruhe, June 1998.

[9] R. Cytron and J. Ferrante. Efficiently Computing Φ-Nodes On-The-Fly. *ACM Transactions on Programming Languages and Systems*, 17(3):487–506, 1995.

[10] R. Cytron, J. Ferrante, and B. K. Rosen. An efficient method of computing static single assignment form. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages (POPL'89)*, pages 25–35. ACM-SIGACT, ACM Press, 1989.

[11] R. Cytron, J. Ferrante, and B. K. Rosen. Efficiently computing static single assignment form and the control dependence graph. *Sigplan Notices*, 13(4):451–490, 1991.

[12] Christopher Colby, Peter Lee, George C. Necula, Fred Blau, Mark Plesko, and Kenneth Cline. A Certifying Compiler for Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'00)*, 2000.

[13] Axel Dold, Thilo Gaul, and Wolf Zimmermann. Mechanized verification of compiler backends. In *Proceedings of the International Workshop on Software Tools for Technology Transfer (STTT'98)*, 1998.

[14] H. Emmelmann. Code selection by regularly controlled term rewriting. In R. Giegerich and S.L. Graham, editors, *Code Generation - Concepts, Tools, Techniques, Workshops in Computing.* Springer-Verlag, 1992.

[15] Helmut Emmelmann. *Codeselektion mit regulär gesteuerter Termersetzung.* PhD thesis, Universität Karlsruhe, Fakultät für Informatik, May 1994.

[16] H. Emmelmann, F.-W. Schrer, and R. Landwehr. BEG - A Generator for Efficient Back Ends. In *ACM Proceedings of the Sigplan Conference on Programming Languages Design and Implementation* , June 1989.

[17] Thilo Gaul, Andreas Heberle, Wolf Zimmermann, and Wolfgang Goerigk. Construction of verified software systems with program-checking: An application to compiler back-ends. In *Proceedings of the Workshop on Runtime Result Verification (RTRV'99)*, 1999.

[18] Thilo Gaul and Antonio Kung. AJACS: Applying Java to Automotive Control Systems. In *Conference Proceedings of Embedded Intelligence Conference, Nrnberg.* Conference Proceedings Embedded Intelligence 2001, Feb 2001.

[19] Gerhard Goos and Wolf Zimmermann. Verification of Compilers. In E.-R. Olderog and B. Steffen, editors, *Correct System Design*, pages 201–230. Springer-Verlag, Lecture Notes in Computer Science, vol. 1710, 1999.

[20] Gerhard Goos and Wolf Zimmermann. Verifying Compilers and ASMs or ASMs for uniform description of multistep transformations. In *ASM-2000.* Springer-Verlag, Lecture Notes in Computer Science, 2000.

[21] Daniel Kstner. PROPAN: A Retargetable System for Postpass Optimisations and Analyses. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*, Vancouver, CA, June 2000.

[22] Daniel Kstner. *Retargetable Postpass Optimisation by Integer Linear Programming.* PhD thesis, Universität des Saarlandes, 2000.

[23] Birger Landwehr. A Genetic Algorithm based Approach for Multi-Objective Data-Flow Graph Optimization. In *Asia South Pacific Design Automation Conference (ASP-DAC)*, Hong Kong, China, January 1999.

[24] Rainer Leupers, Johann Elste, and Birger Landwehr. Generation of Interpretive and Compiled Instruction Set Simulators. In *ASP-DAC'99*, Hong Kong, January 1999.

[25] Rainer Leupers and Peter Marwedel. *Retargetable Compiler Technology for Embedded Processors.* Kluwer Academic Publisher, 2001.

[26] Thomas Müller. *Effiziente Verfahren zur Befehlsanordnung.* PhD thesis, Universität Karlsruhe, jun 1995.

[27] George C. Necula. Translation Validation for an Optimizing Compiler. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'00)*, 2000.

[28] A. Nymeyer and J.-P. Katoen. Code generation based on formal BURS theory and heuristic search. *Acta Informatica 34*, pages 597–635, 1997.

[29] George C. Necula and Peter Lee. Proof-Carrying Code. Technical Report CMU-CS-96-165, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, November 1996.

[30] George C. Necula and Peter Lee. Proof-Carrying Code. In *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)*, January 1997.

[31] George C. Necula and Peter Lee. The Design and Implementation of a Certifying Compiler. In *Proc. PLDI'98*, 1998.

[32] Eduardo Pelegri-Llopart. Rewrite Systems, Pattern Matching and Code Generation. Technical Report UCB/CSD 88/423, University of California, Berkeley, jun 1988.

[33] Data Book TM-1300 Media Processor, September 2000.

[34] A. Pnueli, O. Shtrichman, and M. Siegel. The code validation tool (cvt.). *International Journal on Software Tools for Technology Transfer*, 2(2):192–201, 1998.

[35] A. Pnueli, M. Siegel, and E. Singermann. Translation validation. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems*, pages 151–166. Springer, Lecture Notes in Computer Science, vol. 1384, 1998.

[36] Martin Trapp, Götz Lindenmaier, and Boris Boesler. Documentation of the Intermediate Representation FIRM. Technical Report 1999-14, Universitt Karlsruhe, Fakultt fr Informatik, Dec 1999.

[37] Martin Trapp. *Optimierung objektorientierter Programme.* PhD thesis, Universität Karlsruhe, 1999. Published by Springer Verlag, 2001.

[38] Hal Wasserman and Manuel Blum. Software Reliability via Run-Time Result-Checking. *Journal of the ACM*, 44(6):826–849, November 1997.

[39] Wolf Zimmermann and Thilo Gaul. On the Construction of Correct Compiler Backends: An ASM Approach. *Journal of Universal Computer Science*, 3(5):504–567, 1997.

[40] L. Zuck, A. Pnueli, and R. Leviathan. Validation of Optimizing Compilers. Technical Report MCS01-12, Faculty of Mathematics and Computer Science, The Weizmann Institute of Science, August 2001.