Static Resource Models for Code-Size Efficient Embedded Processors

QIN ZHAO
Eindhoven University of Technology
BART MESMAN
Philips Research Laboratories Eindhoven, Eindhoven University of Technology
TWAN BASTEN
Eindhoven University of Technology

Due to an increasing need for flexibility, embedded systems embody more and more programmable processors as their core components. Because of silicon area and power considerations, the corresponding instruction sets are often highly encoded to minimize code size for given performance requirements. This has hampered the development of robust optimizing compilers because the resulting irregular instruction set architectures are far from convenient compiler targets. Among others, they introduce an interdependence between the tasks of instruction selection and scheduling. This so-called *phase coupling* is so strong that, in practice, instruction selection rather than scheduling is responsible for the quality of the schedule, which tends to disappoint. This lack of efficient compilation tools has also severely hampered the design space exploration of code-size efficient instruction sets, and correspondingly, their tuning to the application domain. In this paper, we present an approach that reduces the need for explicit instruction selection by transferring constraints implied by the instruction set to static resource constraints. All resulting schedules are then guaranteed to correspond to a valid implementation with given instructions. We also demonstrate the suitability of this model to enable instruction set design (-space exploration) with a simple, well-understood and proven method long used in High-Level Synthesis (HLS) of ASICs. Experimental results show the efficacy of our approach.

Categories and Subject Descriptors: D.3.4 [**Programming languages**]: Processors—*Code generation*; *compilers*; *Optimization*; C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures—*Parallel Processors*

General Terms: Algorithms, Design

Additional Key Words and Phrases: Static Resource Models, Scheduling, Phase Coupling, Convex Hull, Constraint Analysis

1. INTRODUCTION

Many embedded processors embody programmable processors as their core components. The machine code running on the embedded processors must meet tight timing constraints because of real-time requirements. Often the program memories reside on chip to obtain

Author's address: Q. Zhao, Department of Electrical Engineering, Eindhoven University of Technology, P.O. Box 513, NL-5600 MB Eindhoven, The Netherlands. q.zhao@tue.nl .
This work was supported in part by the IST-2000-30026 project Ozone.

small memory latency and low power dissipation. However, the cost of on-chip memories makes it necessary to severely restrict code size. This can be accomplished by exploiting characteristics of the application (domain), allowed by application-domain specific instruction-set processors (ASIPs) and digital signal processors (DSPs). These processors are frequently chosen as the core processors of systems on a chip due to their ability to balance flexibility and cost. There are roughly three ways to tune an ASIP core to an application, that can be used in combination:

—By synthesizing a communication (busses) and storage (registers) infrastructure just sufficient for the application.

—By hardware acceleration (see e.g., Xtensa, Tensilica Inc., http://www.tensilica.com/-technology.html); functional units are added to the data path that perform relatively coarse-grain functions characteristic of the application, such as a butterfly unit in an FFT processor.

—By minimizing the width of the instructions controlling a given data path [Woudsma 1994]. One way to achieve this is to encode frequently occurring (sequences of) operations with short instruction words. Another possibility is to limit the number of instructions by restricting certain combinations of operations that the data path can execute in parallel.

Hardware acceleration potentially offers the largest benefits on all accounts, especially for applications that contain much regularity. It is also the most complex method for the designer because it requires changes in the communication and storage hardware and the design of the dedicated functional units. The mentioned ways to tune the ASIP core all have the same severe drawback: They add the necessity to 'recognize' in the application, instructions supported in the instruction set. Often this task of recognition (code selection or instruction selection) is performed by the programmer himself, either by writing assembly or with the use of APIs in the source code to call the dedicated instructions or hardware. Both require a lot of source code rewriting and low-level programming, and both are detrimental for the maintainability and portability of the code. Alternatively, the compiler contains a code selection phase that recognizes valid instructions in a Data Flow Graph (DFG), often using pattern matching and graph covering techniques [Liem et al. 1994], [Leupers and Marwedel 1996]. Unfortunately the results of applying these techniques have been quite disappointing. They are for a large part responsible for the considerable overhead in both schedule length and code size (reported in the order of 800% [Paulin and Liem 1996]) of compiler-generated code compared to manually written assembly for ASIPs.

The task of instruction selection, i.e., to *cover* the DFG with processor instructions that implement the individual operations in the DFG, is depicted in Figure 1. The main issue introduced by an irregular instruction set is the issue of *phase coupling*: On the one hand, if instruction selection is performed prior to scheduling, the optimal schedule can easily be eliminated as a result of the choices made during instruction selection. On the other hand, if scheduling is performed first, the available instructions may not be able to implement the schedule. Traditional methods which perform these tasks in different phases might yield inferior schedules. This point is illustrated in Figure 1. The DFG on the left hand side has been covered with machine instructions which are listed below the DFG. The associated optimal schedule (6 clock cycles) for this selection of instructions is given in the right hand

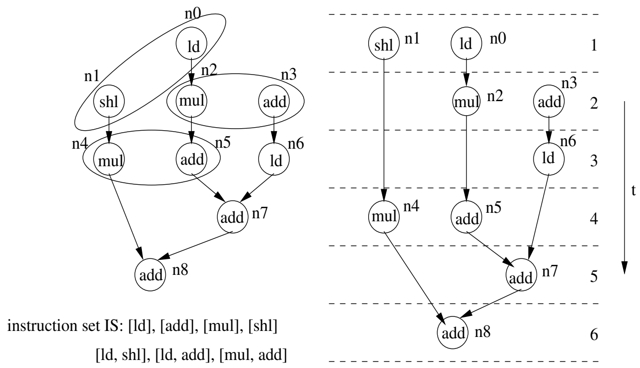


Fig. 1. Instruction selection prior to scheduling may yield inferior results

side of Figure 1. The imposed instruction selection is rather unfortunate, however, because Figure 2 depicts a valid schedule requiring only 5 clock cycles.

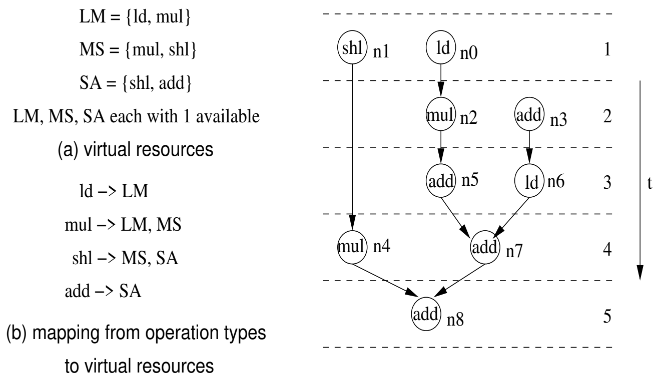


Fig. 2. With a static resource model optimal results can be obtained

A way out of this status quo is offered by the *Static Resource Model* (SRM) approach [Zhao et al. 2001]. The SRM approach targets instruction sets that are minimized by restricting certain combinations of operations that the data path can execute in parallel. Such restrictions might, for example, be caused by the number of issue slots and functional resources available in a VLIW architecture or wordlength constraints occurring in highly encoded ASIP instruction sets. Instruction sets restricting certain combinations of operations can be modeled in terms of *virtual* resources, easily interpreted by classic resource constrained schedulers such as the popular list-scheduling algorithm. This is illustrated in Figure 2 for the example in Figure 1. The instructions in the instruction set IS have been augmented by the use of the virtual resources $\{ld, mul\}$, $\{mul, shl\}$ and $\{shl, add\}$, which are abbreviated as LM, MS and SA. We have one instance available of each virtual resource. Each operation uses all the virtual resources that it is contained in, e.g., operation mul uses virtual resources $LM\ and\ MS$ simultaneously. In this way, the LM resource for example models the instruction set constraint that the ld and mul operations may not occur simultaneously. Using the so-called static resource model given in the left hand side of Figure 2, a list scheduler generates the schedule in the right part of this figure. The reader can verify that the operations at each clock cycle can be implemented with instructions from the original instruction set IS given in Figure 1. The resulting schedule has length 5, whereas the schedule in Figure 1 has length 6. This example demonstrates that better schedules can be obtained using a static resource model instead of covering the DFG with valid instructions. This alternative machine model with virtual resources therefore allows efficient resource constrained compilation with well understood and widely available compilation tools, rather than the poorly performing compilers based on instruction selection. In this paper, we further improve the approach of [Zhao et al. 2001] so that it can cope with a wider range of instruction set constraints.

The SRM approach also enables instruction set design (-space exploration) with an equally well-understood and proved method long used in High-Level Synthesis (HLS) of application specific integrated circuits (ASICs), for example supported in the A|RT tools (http://www.adelantetechnologies.com), illustrated in Figure 3. This method analyzes the time critical loops for bottlenecks in the availability of processor resources required to obtain the target schedule throughput. These bottlenecks can be identified by scheduling the loop and examining the *load diagrams* of the functional resources. The load on critical resources is then relieved by allocating additional resources. The potential use of this method for instruction set design is based on the observation that both extra real functional resources *and* extra virtual resources can be allocated when considering the SRM of an instruction set. The addition of virtual resources results in an extension of the instruction set. We presume therefore that the SRM view on an instruction set allows instruction set design in terms of allocating resources just sufficient to efficiently execute the critical loops.

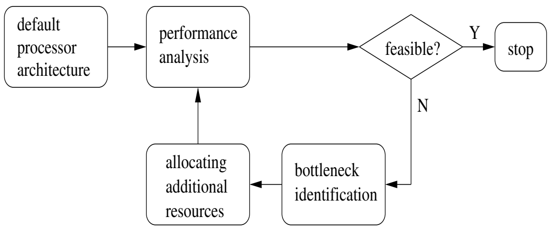


Fig. 3. Design flow of ASICs

The resulting instruction set can be implemented as an ASIC or an ASIP. ASIPs can be reprogrammed after fabrication. In order to tune the instruction set of a fabricated ASIP to an application, the instruction decoder should be reconfigurable to a certain degree. Note that the real (non-virtual) resources are considered fixed in this case.

This paper is organized as follows. Section 2 presents related work on instruction set selection. Section 3 details the problem statement and our approach. Section 4 addresses the construction of an SRM for an instruction set by using the convex hull approach known from computational geometry. In Section 5, we present the exploitation of instruction set design by evaluating the performance of an application through the SRM approach.

In Section 6, experimental results are given for benchmarks as well as real applications. Conclusions and future work are discussed in Sections 7 and 8.

2. RELATED WORK

2.1 Tree covering techniques for code selection

The phase of code selection has received a lot of attention in the software compiler community. Traditional compilers use the *template pattern base* to represent the target processor, which essentially enumerates the different partial instructions available in the instruction set. Each partial instruction is represented as a pattern, usually a data flow tree (DFT), which is expressed by means of the intermediate representation of an algorithm. In this pattern, nodes correspond to variables, constants, and operations, while edges denote data dependencies. It has been shown that code selection is an NP-complete problem for intermediate representations that take the form of a directed acyclic graph [Aho 1989]. However, optimal vertical code can be generated in polynomial time when the following conditions are satisfied:

(1) the intermediate representation of the algorithm is an *expression tree*;
(2) the template pattern base is restricted to contain only *tree patterns*, i.e., it can be represented as a *regular tree grammar*;
(3) the processor has a *homogeneous* register structure.

Several code-selector generators are based on a stepwise partitioning of the code selection problem using *dynamic programming*. It is assumed that conditions 1) and 2) of the above mentioned canonical problem are satisfied. In [Aho and Johnson 1976] tree pattern matching is done in a bottom-up traversal of the subject tree. For each node, the method computes the minimal cost to cover the subtrees rooted at that node. During the top-down traversal of the subject tree, the tree cover is finally found, by determining the minimal cost at the tree's root node. A number of tools are available for automatic generation of tree pattern matchers from instruction set grammar specifications, such as Twig [Aho 1989], Beg [Emmelmann et al. 1989], Iburg [Fraser et al. 1993] and Olive [Aho 1989]. Dynamic programming is also extended to target heterogeneous register structures, which is related to several DSP compilers. CBC [Fauth et al. 1995] is based on the hardware description language nML and the machine specification is transformed to an Iburg specification. In [Araujo et al. 1996], the DFGs are pruned to DFTs, which satisfy the RTG criterion and code selectors are generated by the code selector generator Olive.

2.2 Phase coupling in code generation

Besides code selection, the task of machine code generation 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 are mutual dependencies, since each of the three phases may impose possibly unnecessary and obstructive restrictions on the remaining ones. Therefore, the key method in embedded code generation to solve this problem is to integrate these phases. Early techniques of data routing incorporate register allocation for distributed register files and instruction scheduling [Rimey and Hilfinger 1988; Hartmann 1992; Wess 1991].

In [Araujo and Marlik 1995], the tree pattern matching technique has been coupled with register allocation and scheduling for architectures that satisfy the "RTG criterion", e.g.

a family of TI DSPs. Later, additional heuristics for handling data flow graphs (DFGs) are presented in [Araujo et al. 1996], in which the DFGs are pruned into DFTs in order to provide faster code selection. An improved method based on simulated annealing has been described in [Leupers 2000]. Mutation scheduling [Novack et al. 1995] is another approach considering phase coupling, where algebraic transformations are exploited in order to explore alternative instruction set mappings. A constraint logic programming technique for DSPs with irregular architectures has been presented in [Bashford and Leupers 1999]. In that approach, all binding decisions are delayed until they are really required, which yields a maximum degree of freedom in code generation. However, this is at the expense of high compilation time.

Another approach for code selection is based on constructing graphs which include complete information of the architecture. In CHESS [van Praet et al. 1994], the target processor is described in the nML language [Fauth et al. 1995] and is translated into an Instruction-Set-Graph (ISG), which models connectivity, encoding restrictions and structural hazards. Code selection covers the control-data flow graph with partial instructions (bundles) by searching valid paths in the ISG.

A common disadvantage of all the mentioned methods so far is that they add a complex step to the compiler chain and eliminate potentially interesting solutions from the schedule and register binding search spaces. Exact approaches for code generation are described in the form of constraints (generally linear equations and inequalities). The complete solution space is explored while all constraints are considered simultaneously, leading to a complete phase integration. The approach in [Leupers 1997] performs instruction scheduling based on the Integer Programming (IP) approach. Preliminary code selection and register allocation are performed in advance, based on Iburg generated code selectors, whereas scheduling is delayed. Complete integration is obtained in [Wilson et al. 1994; Gebotys 1997]. Wilson's approach leads to very high runtime due to large IP models. The approach of Gebotys describes constraints based on horn clauses. This can be mapped to Linear Programming (LP) problems, which can be solved more efficiently than IP problems. However, only restricted classes of architectures can be handled efficiently. Architectures comprising register files with multiple registers, typically lead to an explosion of generated models. In [Hanono et al. 1997], a covering approach for DFGs for finding a minimum set of (VLIW) instructions is specified. The approach is based on binate covering. Detailed register allocation is performed in a post processing phase.

2.3 Solving the phase coupling problem statically

A completely different approach to tackle the code generation problem is to exploit the instruction set constraints statically in the sense that the constraints provide some fixed upper bound on the combinations of operations allowed in the instructions. State diagrams or Finite State Automata (FSAs) are used to represent the set of all legal instruction schedulers for a processor [Bala and Rubin 1995; Proebsting and Fraser 1994]. They provide the advantage of space and time efficiency. However, they are not amenable to certain advanced scheduling techniques, such as iterative modulo scheduling [Rau 1996] and mutation scheduling [Novack et al. 1995]. The concept of reservation tables are used to detect conflicts for scheduling. Examples of compilers that adopt this approach include the Multiflow Trace Scheduling Compiler [Lowney et al. 1993] and the Trimaran (Elcor) Compiler [Gyllenhaal et al. 1996]. Eisenbeis [Eisenbeis et al. 1999] starts from placement conflicts by using reservation tables for VLIW architectures, such as the Trimedia processor.

[Timmer et al. 1995] discusses the modeling of instruction set constraints together with resource constraints for instruction scheduling, but the assumption that all the instruction set constraints can be transferred to resource constraints targets mostly instruction sets with a structure associated with so-called issue-slot machines. The virtue of these orthogonal instruction sets, incorporating for example the VLIW instruction sets [Rau and Glaeser 1981], is that they allow to a large extent the modeling of the instruction set constraints by means of a model like [Eisenbeis et al. 1999], [Timmer et al. 1995], or a static resource model. These models offer the scheduler more opportunity to satisfy the timing and resource constraints than with the use of an explicit instruction selection step. The scheduler rather than the instruction selector is considered the designated place for handling these constraints. The disadvantage of the orthogonal processors (especially VLIW processors) is the inherent problem of code size due to their associated instruction word widths.

Our approach to code generation is based on constraint analysis and is motivated by the fact that the irregularity of the architectures of ASIPs and DSPs and the timing objectives of certain tasks to be implemented on those processors can be represented as *constraints* and phase coupling can be tackled by pruning the search space defined by those constrains. In this way, wrong decisions are prevented by checking whether they violate any constraints. In [Mesman et al. 1999], data flows are expressed as precedence constraints. Latencies and initiation intervals are transferred to timing constraints. Register files are constrained by their capacities, and functional resources are associated with operation types. Constraint analysis is the kernel to combine all those constraints such that when one decision is made, it will have an impact on the whole search space.

The focus of this paper is on translating the instruction set constraints of highly encoded instruction sets to a static resource model, allowing the use of code size efficient processors in combination with efficient compilation tools based on constraint analysis. Our approach is not restricted to issue-slot tables with fixed bit-width for each issue slot; thus it can deal with more flexible encodings than the earlier approaches. In addition, the procedure of constructing a static resource model also provides useful information for instruction set design.

3. PROBLEM STATEMENT AND APPROACH

3.1 Definitions

A DSP algorithm can be expressed as a data flow graph, which describes the primitive operations performed in an algorithm and the dependencies between those operations.

Definition 3.1. A *data flow graph (DFG)* is a tuple (V, E), where V is the set of vertices (operations) and $E \subseteq V \times V$ is the set of precedence edges.

In a processor architecture, a functional resource can be used in different ways. E.g., a functional resource ALU can execute an operation *add* or a *subtract*, etc. For reasons of complexity, we do not wish to enumerate all possible uses of a functional resource in an instruction set. Therefore, we consider the collection of these uses of a functional resource and associate with it an *operation type*. E.g., on the functional resource ALU, operations *add* and *subtract* can be executed, which are associated with the operation type alu. We denote the set of operation types with T. An *instruction* is now defined as a combination of operation types that can be executed in a single clock cycle.

An operation type can appear multiple times in an instruction (multiple ALUs can be present in the data path). For an operation type *op* in an instruction I, we denote this number by $I(op)$. If for two instructions I_0 and I_1, $I_0(op)$ is always at most equal to $I_1(op)$ for each operation type *op*, we say that instruction I_0 is *contained* in instruction I_1. In this paper, we consider instruction sets IS where for each instruction all contained instructions are also in IS. We call these instruction sets *prefix closed*.

The code generation problem is to find a schedule of a DFG, i.e., to determine a start time $s(v)$ for each operation $v \in V$, that satisfies precedence constraints and architectural constraints. These architectural constraints can be modeled either as an instruction set or by introducing functional resources and associating a certain resource usage with each operation. The corresponding resource constraints are *static* in the sense that they only provide a fixed upper limit and any usage within the limit is valid. We say that such a problem has a static resource model.

Definition 3.2. A *Static Resource Model (SRM)* is a model generating static resource constraints that defines three aspects:

−a set of resources R,

−a mapping that associates each operation type with the resources in R that it needs, and

−the number of instances available of each resource $r \in R$, denoted by #r.

Our approach is motivated by the observation that both resource constraints and instruction set constraints can be expressed by inequalities. For example, if an architecture contains two ALUs and each ALU can be used as an adder or a subtractor, then the resource constraints can be expressed by the following inequality: $N(ALU) \leq 2$, which is equivalent to $N(A) + N(S) \leq 2$, where $N(A)$ and $N(S)$ denote the number of add and subtract operations allowed to execute in parallel. Any schedule satisfying at any time the above inequality indicates a valid resource usage. Similarly, if an instruction set contains instructions $[add, add]$, $[add, sub]$ and $[sub, sub]$, the operation type usage can also be expressed as an inequality: $N(add) + N(sub) \leq 2$, assuming any subinstruction is also a valid instruction.

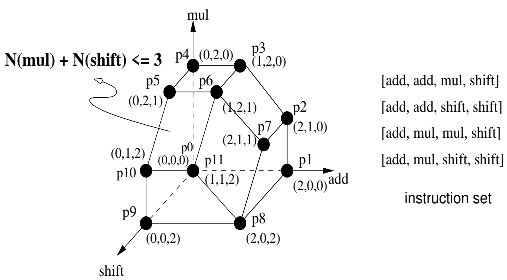


Fig. 4. Instructions expressed as points in the operation type space

In general, operation types are associated with axes in a multi-dimensional numerical space $\mathcal{R}^d$, where d is the dimension corresponding to the number of operation types, and instructions can be geometrically represented as points in this operation type space. An example is depicted in Figure 4. This figure gives an instruction set using three operation types. Only the “maximum” instructions are listed; instructions that are fully contained in other instructions are not explicitly represented. We will do so throughout the paper. Since the instruction set uses three operation types, the operation type space is 3-dimensional. As mentioned, all valid instructions correspond to points in this space. Instruction $I = [add, add, mul, shift]$, for example, is drawn as point $p7$ with coordinates $I(add)$, $I(mul)$ and $I(shift)$. For clarity, only a subset of all the instructions is shown in this figure. Because we assume prefix closedness of instruction sets, the instruction set corresponds to a well-defined, closed subspace of the operation type space. It is our aim to capture this subspace spanned by the instructions via inequalities. Inequality $N(mul)+N(shift) \leq 3$, for example, is one of the inequalities needed to capture this subspace. Inequalities can subsequently be translated to virtual resources in an SRM. The example inequality results in one virtual resource with three instances, and one instance needed by any mul or *shift* operation. In general, deriving the inequalities and the resulting virtual resources for an instruction set can be perceived as a *convex hull* problem [Avis et al. 1997] [Preparata and Shamos 1985]. Here we give some basic definitions concerning the convex hull problem.

Definition 3.3. Given a set of points $X = \{x_1, x_2, \ldots, x_k\}$, y is called an *affine combination* of X if y can be expressed as a linear combination of X:

$$y = \sum_{i=1}^{k} \lambda_i x_i \qquad and \qquad \sum_{i=1}^{k} \lambda_i = 1 \tag{1}$$

A *convex combination* of X is an affine combination such that each λ_i is non-negative. A *proper* convex combination is one where each λ_i is positive.

Definition 3.4. A subset S of a d-dimensional space $\mathcal{R}^d$ is called a *convex set* if and only if every convex combination of points in S is also in S. The *convex hull* of a set X, denoted as *conv(X)*, is the set of all convex combinations of X; it is the smallest convex set containing X.

Consider again Figure 4. The set of points bounded by the planes $mul = 0$, $add = 0$, $shift = 0$ and the other planes depicted is the convex hull of the original instruction set IS given in the right part of the figure. That is, it is the smallest convex set containing all the instructions. (Recall that all subinstructions of the listed instructions are also included in this instruction set.) Above, it has already been explained that the depicted subspace corresponds to the instruction set. Thus, in this example, the convex hull of all instructions captures precisely all the instructions in the instruction set. It does not contain a combination of operation types that is not a valid instruction. This property turns out to be crucial in translating instruction set constraints to resource constraints. Later in this paper, an example is given showing that not all instruction sets have the property that they are precisely captured by their convex hull.

As the example discussed so far already suggests, a convex set can be described by means of its boundary. In general, a convex set is determined by a set of halfspaces. A halfspace is the set of all points below or above some plane. A halfspace can be seen as a *constraint* on the elements of the set being described and it is defined via an *inequality*. Such an inequality describes part of the boundary of the convex set. Consider the example of Figure 4 again. The halfspace containing all points below the plane through points $p5$, $p6$, $p10$ and $p11$ corresponds to inequality $N(mul) + N(shift) \leq 3$. A convex set can be seen as an intersection of a set of halfspaces. The convex set in the example of Figure 4 is the intersection of nine halfspaces, namely $N(add) \geq 0$, $N(mul) \geq 0$, $N(shift) \geq 0$, $N(add) \leq 2$, $N(mul) \leq 2$, $N(shift) \leq 2$, $N(add) + N(mul) \leq 3$, $N(mul) + N(shift) \leq 3$, and $N(add) + N(mul) + N(shift) \leq 4$. The nine inequalities define the boundary of the convex set, and thus of the convex hull of the given instruction set. An interesting observation is that any set defined as the intersection of a set of halfspace is necessarily convex. In other words, it is not possible to describe a non-convex set through inequalities of the form given in this example.

Given a set of halfspaces $\mathcal{H}$ with intersection S, halfspace $H \in \mathcal{H}$ is *non-redundant* if and only if there is some point contained in the intersection of all the halfspaces in $\mathcal{H} \setminus \{H\}$ but not contained in S. Thus, a non-redundant halfspace really restricts the convex set. The notion of non-redundancy is useful in determining the minimal set of inequalities describing convex set S. The halfspace representation of the convex hull of some set X is denoted by $\mathcal{H}(X)$.

It is also possible to describe a convex hull via its *extreme* points. The convex hull in Figure 4, for example, can be described via the set $\{p0 = (0,0,0), p1 = (2,0,0), \ldots, p11 = (1,1,2)\}$. Given a convex set S, a point in S is an *extreme* point if and only if it is not a $proper$ convex combination of any two points in S. The notation $\mathcal{V}(X)$ denotes the vertex description of the convex hull of some set X, consisting of the set of extreme points.

There are two closely related computational problems concerning the two descriptions of the convex hull of a set X:

—The *vertex enumeration problem* is to compute $\mathcal{V}(X)$ from $\mathcal{H}(X)$.

—The *convex hull problem* is to compute $\mathcal{H}(X)$ from $\mathcal{V}(X)$.

In code generation for ASIPs and DSPs, deriving virtual resources from a given instruction set can be perceived as a variant of the convex hull problem, since the instructions form the vertex description and the virtual resources form the non-redundant halfspace description. On the other hand, in the instruction set design space exploration, one can optimize the instruction set by modifying the SRM to meet the real-time constraints, and subsequently calculating the corresponding instructions. This can be perceived as a variant of the vertex enumeration problem.

The precise complexity of the two problems is an interesting problem. The two problems are dual and are therefore of the same complexity. Two excellent references on the complexity of convex-hull related problems are [Fukuda 2000; Avis et al. 1997]. Recall that d is the dimension of the space in which we are operating; let n be the number of inequalities in case of the vertex enumeration problem and the number of vertices in case of the convex hull problem. It is known that the problems are efficiently solvable ($O(n\,log\,n)$) for $d \in \{2, 3\}$. For the general case, there is an algorithm of *optimal worst-case* complexity $O(n^{\lfloor d/2 \rfloor})$ [Chazelle 1993]. Thus, one could say that for fixed dimension d, the complexity is polynomial. However, for high dimensions, the degree of the polynomial tends to become large. In our applications, the dimension of the operation type space may become quite large, although it remains to be seen how large it will be in practical examples. Fortunately, in practice hardly any input to any of the two above problems ever causes the worst-case complexity. The average complexity is usually much better. The precise performance depends on the particular implementation. For a comparison of algorithms, the interested reader is referred to [Avis et al. 1997].

3.2 Problem statement

In this subsection, we generalize the problem of constructing the SRM for a given instruction set and present the convex hull approach for this problem. In addition, we present an approach for instruction set design (-space exploration) by tuning the corresponding SRMs.

Problem 3.5. The general problem can be defined as mapping a given instruction set to an SRM such that any schedule for a DFG satisfying the resource constraints posed by the SRM corresponds to a valid instruction selection. We say that the instruction set *has* an SRM.

We propose a solution strategy based on expressing the instruction set constraints as inequalities, like $N(add) + N(sub) \leq 2$ in the example in Section 3.1. We start from an initial instruction set IS. It is followed by a step called *operation-type statistics*, which calculates the number of times operation types as well as their combinations appear in an instruction set. In this way, we enumerate all the potential extreme points of the instruction set in the operation type space. We then search for an SRM by computing the convex hull, i.e., the smallest convex set containing all the instructions in IS. With the resulting inequalities, we can obtain a new instruction set NIS by enumerating all the instructions allowed under those inequalities. The equivalence of the instruction set constraints and the SRM is verified by comparing the new instruction set NIS to the initial instruction set IS. The SRM itself is derived directly from the inequalities, but is only accurate if the aforementioned test turns out positive. If NIS contains combinations of operation types that do not correspond to an original instruction in IS, then the constraints obtained from the SRM are not sufficiently tight. Since the convex hull is the *smallest* convex set containing all the original instructions, it even follows that it is impossible to capture the instruction set constraints via virtual resource constraints. The procedure is illustrated in Figure 5. In case the SRM is not accurate, it can be profitable to adapt the instruction set in such a way that it fits the SRM, because then all the advantages of the SRM approach can be exploited.

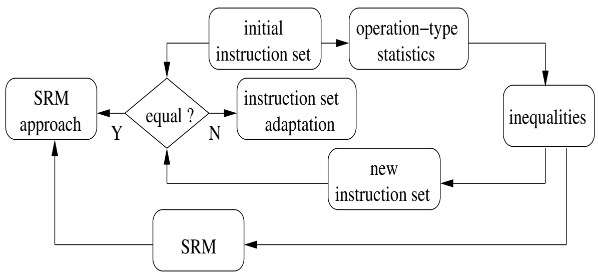


Fig. 5. Overview of the SRM approach

Since the SRM approach can be applied easily to evaluate the performance of an application, especially a loop kernel, it is natural to consider this approach for adapting the designed instruction set to the performance requirements of an application. We generalize the following instruction set design (-space exploration) problem.

Problem 3.6. Given a set of time critical loop kernels (belonging to a single application) with the corresponding throughput constraints and a target instruction width for the ASIP, design an instruction set and the corresponding SRM such that the throughput constraints can be satisfied.

In Figure 3, a practical and commonly used design flow is depicted for allocating functional resources in the context of High-Level Synthesis (HLS). Noticing the similarity between the functional resources and the virtual resources, we propose to apply this flow for allocating *virtual* resources in the context of designing an instruction set.

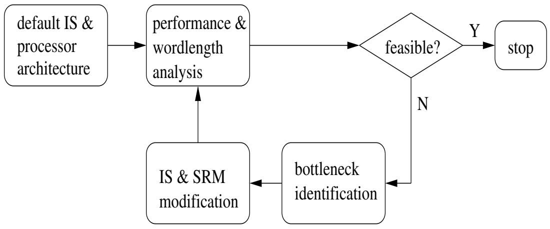


Fig. 6. Overview of the instruction set design flow

In this optimization flow, depicted in Figure 6, we start with a *default instruction set and processor architecture*. Subsequently, the performance of the critical loops is analyzed, as explained in more detail in Section 5. If the performance is insufficient, we look for the responsible (virtual) resources in a step called *bottleneck identification*. The SRM is subsequently modified by allocating additional instances of the virtual resources whenever necessary, which results in the modification of the original instruction set. The essential difference to the HLS flow in Figure 3 is that we also consider the instruction width as a criterion in the design process to evaluate the modifications made for the SRM model. This is performed iteratively until all the constraints are satisfied.

4. CONSTRUCTION OF AN SRM

In this section, we present in more detail the method for verifying whether an instruction set has an equivalent SRM outlined in the previous section. If it has, the method also gives this SRM.

4.1 Using inequalities for the construction of an SRM

In [Zhao et al. 2001], we presented an approach for deriving an SRM for a given instruction set by making use of inequalities. The basic idea is to generalize the maximum usage of operation types in an instruction set by representing it as a set of inequalities, and then transforming these inequalities into an SRM. The desired set of inequalities is derived from the *operation-type statistics*. An example of these statistics is depicted in the table of Figure 7 (b) for the instruction set IS_1 given in Figure 7 (a). In the table, rows correspond to the individual instructions listed in Figure 7 (a), and columns correspond to (combinations of) operation types. The numbers in the table indicate how many times an operation type (combination) is present in an instruction. For example, the operation type *add*, occurs twice in instruction (1), and the operation types *shift* and *mul* together occur three times in instruction (3). By looking at the largest frequency within each column, the inequalities in Figure 7 (c) are derived. Each inequality corresponds to one column. For example, the operation-type combination *shift* and *mul* occurs three times in instructions (3) and (4), and only two times in instructions (1) and (2). Thus, we can derive the constraint $N(\mathit{shift}) + N(\mathit{mul}) \leq 3$ for this combination of operation types. In general, for each combination of operation types $S = \{op_1, \ldots, op_n\}$, we have one inequality:

$$N(op_1) + \cdots + N(op_n) \leq \underset{I \in IS}{\mathrm{MAX}}\Big(\sum_{op \in I \cap S} I(op) \Big) \tag{2}$$

where $N(op_i)$ is the number of times an operation type appears in an instruction.

(1) [add, add, mul, shift]

(2) [add, add, shift, shift]

(3) [add, mul, mul, shift]

(4) [add, mul, shift, shift]

(a) instruction set IS1

	add	mul	shift	add mul	add shift	shift mul	add shift mul
(1)	2	1	1	3	3	2	4
(2)	2	0	2	2	4	2	4
(3)	1	2	1	3	2	3	4
(4)	1	1	2	2	3	3	4

(b) operation−type statistics

(1) N(add) <= 2

(2) N(mul) <= 2

(3) N(shift) <= 2

(4) N(add) + N(mul) <= 3

(5) N(add)+N(shift) <= 4

(6) N(mul) + N(shift) <= 3

(7) N(add) + N(mul) + N(shift) <= 4

(c) inequalities

#A = 2, #M = 2, #S = 2,

#AM = 3, #MS = 3, #AMS = 4

add −> A, AM, AMS

mul −> M, AM, MS, AMS

shift −> S, MS, AMS

(d) static resource model

Fig. 7. Example of an instruction set with an SRM

Based on the computation of the operation-type statistics, the instruction set constraints have been replaced by a set of inequalities. This set often contains redundancy, in the sense defined in Section 3.1; in the example of Figure 7 (c), inequality (5) can be removed because it is implied by inequality (7). Removing inequalities is advantageous because the complexity of the derived SRM is determined by the number of inequalities. In [Zhao et al. 2001], we provided several rules to remove the redundancy. From the remaining inequalities the SRM is derived, shown in Figure 7 (d). For example, inequality (6) is translated to the virtual resource $\{mul, \mathit{shift}\}$, which is abbreviated as MS, with three instances available. For reasons of convenience, we represent a virtual resource by combining the first letters of all those operation types which compose it. An operation type "uses" all the resources from the SRM that it is contained in, e.g., add uses A, AM and AMS at the same time. Note that virtual resource A corresponds to the real functional resource implementing the add operation, while virtual resources AM and AMS have functionalities similar to functional resources but are not real functional resources, explaining the term "virtual resource."

The complexity of the proposed algorithm to compute an SRM for a given instruction set is $O(2^{|T|})$, where T is the set of operation types introduced in Section 3.1. (Recall that $|T|$ is equal to the dimension d of the operation-type space that forms the basis of the convex-hull approach to computing an SRM.) A disadvantage of the approach in this subsection is that a resulting SRM often has redundancies, as the above example illustrates. These redundancies lead to a larger SRM than strictly necessary and, as a result, complicate constraint-based scheduling techniques using the SRM. Redundancies can be removed by formulating the redundancy-removal problem as a set of linear programs (LPs), one for each inequality [Fukuda 2000]. By successively solving the LPs for each inequality, one eventually obtains a set of non-redundant inequalities. Although the LP method is polynomial, this entire process can still be time consuming. Another weakness of the method proposed in this subsection is that it does not always give an SRM, even if an instruction set has one. It is for example not well suited for handling complex encodings. Figure 8 shows an example where methods from the literature [Eisenbeis et al. 1999] and the one presented above fail to express instruction set restrictions in an SRM-like manner. The difficulty is introduced by the *wordlength constraints* on instructions. There are three adders and two multipliers in the data path. Each add is encoded with 8 bits and each mul with 10 bits. The total wordlength for an instruction is limited to 24 bits. Thus, all the possible (maximum) combinations of operation types are: $[add, add, add]$, $[mul, add]$, $[mul, mul]$. The method above will produce the inequalities in Figure 8 (b). It contains one redundant inequality: inequality (1). Furthermore, inequality (3) allows combinations of operations such as $[add, mul, mul]$ and $[add, add, mul]$, which are in fact not valid because of the instruction encoding constraints. In contrast, the method explained below generates the new inequality (3) for the instruction set in Figure 8 (c), which correctly models the instruction encoding limitation. Using this new inequality, the instruction set is verified to have an SRM.

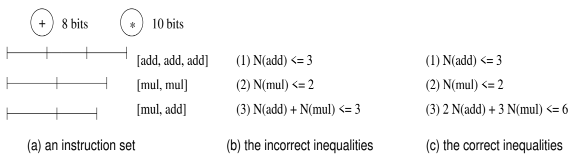


Fig. 8. Example of wordlength constraints

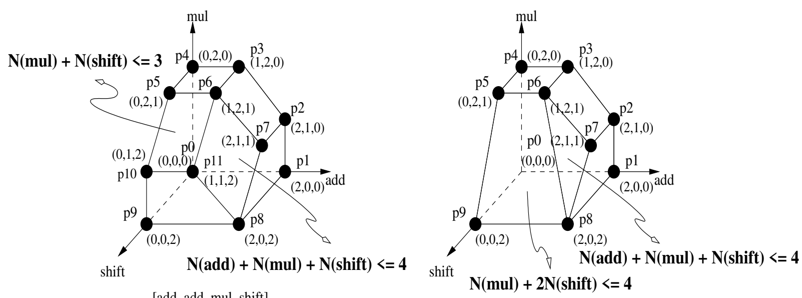
In the next subsection, we provide a general approach for the construction of SRMs that improves the method of [Zhao et al. 2001]. It solves all the mentioned problems and it gives a valid SRM for the instruction set in Figure 8 (a).

4.2 A general approach for deriving inequalities

In order to find a method that produces a valid SRM for a broader class of instruction sets, we observe the following restriction in the approach of Section 4.1: The inequalities do not have weights on the left hand side for each operation type. Assume IS_1 in Figure 9 (a) is the full instruction set of a certain VLIW architecture, while IS_2 in Figure 9 (b) is a similar but smaller instruction set. Note that instruction set IS_1 is the one of Figure 7. The method above will generate the same SRM for the two instruction sets. Because IS_2 is smaller than IS_1, this SRM is not accurate for IS_2. However, by adding inequality (6') to the set of inequalities in Figure 9 (d), we solve this problem: Instruction set IS_2 now has a valid SRM, different from the SRM corresponding to IS_1.

If the operation types are associated with the axes in a multi-dimensional numerical space and the instructions are represented as points in this operation type space geometrically, then the inequalities derived for the instruction set can be perceived as boundaries enclosing those points and the problem is transformed into a *convex hull* problem.

The instruction sets IS_1 and IS_2 and their convex hulls are illustrated in Figure 9 (a) and (b). In Figure 9 (c), we obtain the same set of inequalities as in Figure 7, which in Figure 9 (e) leads to the same SRM as in Figure 7. Notice that a standard convex hull algorithm does not yield the redundant inequality (5) of Figure 7 (c) in its output. Because the last instruction in IS_1 is not valid for IS_2, points $p10$ and $p11$ of Figure 9 (a) are not present in Figure 9 (b), and the convex hull and the set of inequalities in Figure 9 (d) are slightly different from IS_1. Notice that inequality (6'), $N(mul) + 2N(shift) \leq 4$, is generated with the weight on the left hand side automatically. Because of this weight, inequality (3) becomes redundant, which means virtual resource S is not required any more. Also notice that for the same reason, the *shift* operation uses virtual resource MS twice as is reflected in the SRM of Figure 9 (f).



[add, add, mul, shift]
[add, add, shift, shift]
[add, mul, mul, shift]
[add, mul, shift, shift]

(a) instruction set IS1

[add, add, mul, shift]
[add, add, shift, shift]
[add, mul, mul, shift]

(b) instruction set IS2

(1) N(add) <= 2
(2) N(mul) <= 2
(3) N(shift) <= 2
(4) N(add) + N(mul) <= 3

(6) N(mul) + N(shift) <= 3
(7) N(add) + N(mul) + N(shift) <= 4

(c) inequalities for IS1

(1) N(add) <= 2
(2) N(mul) <= 2

(4) N(add) + N(mul) <= 3

(6') N(mul) + 2 N(shift) <= 4
(7) N(add) + N(mul) + N(shift) <= 4

(d) inequalities for IS2

#A = 2, #M = 2, #S = 2
#AM = 3, #MS = 3
#AMS = 4

add –> A, AM, AMS
mul –> M, AM, MS, AMS
shift –> S, MS, AMS

(e) SRM for IS1

#A = 2, #M = 2
#AM = 3, #MS = 4
#AMS = 4

add –> A, AM, AMS
mul –> M, AM, MS, AMS
shift –> MS, MS, AMS

(f) SRM for IS2

Fig. 9. Examples of SRMs calculated from convex hulls

The basis for the general problem of constructing an SRM for an instruction set IS is the following: given a set of points (instructions) in the d-dimensional operation type space, determine the convex hull $conv(\mathrm{IS})$ expressed as a set of m linear inequalities.

$$conv(IS) = \{\vec{x} \mid A\vec{x} \leq b\} \tag{3}$$

where $d = \mid T \mid$, $A \in \mathcal{R}^{m \times d}$ and $b \in \mathcal{R}^{m}$.

It remains to determine how to precisely compute (3) for a given IS. The basic idea is to use a standard convex hull algorithm applied to the *extreme points* in IS. Note that we could simply apply such an algorithm to the entire instruction set without influencing the result. However, recall from Section 3.1 that the complexity of the convex hull problem depends on the number of points n in the set of which the convex hull needs to be computed. Thus, the convex hull computation can be sped up by minimizing the input to the algorithm. Because we assume prefix closedness of instruction sets, any subinstruction is also a valid instruction. Although a subinstruction is not explicitly listed in the given example instruction sets, it is quite possible to be an extreme point of a convex hull. For example, the four maximum instructions in IS_1 can be drawn as points $p7$, $p8$, $p6$ and $p11$ in Figure 9 (a). These four points are not sufficient to build $conv(IS_1)$. Point $p1$, for example, represents a subinstruction $[add, add]$, which is obviously an extreme point for this convex hull. In order to construct the complete convex hull, we have to find all the extreme points in the space, which is as complex as the convex hull problem itself. A compromise between using the entire instruction set or only its extreme points for the convex hull computation follows from the following observation. As we can see in the example, extreme point $p1$ with coordinates $(2, 0, 0)$ is the projection of point $p8$ with coordinates $(2, 0, 2)$ on the plane $shift = 0$. Any point *between* them has no contribution to the convex hull, meaning it is redundant. From this intuition, we can simply calculate the points projected from the maximum instructions onto the planes with dimensions $d - 1$, $d - 2$, ..., 1 obtained by assuming $1, 2, \ldots, d - 1$ coordinates to be zero respectively. The complexity of this projection in the worst case is $O(k\ 2^p)$, where k is the number of maximal instructions and p is the number of operation types appearing maximally in one such maximal instruction. Of course, this approach might still result in quite a number of non-extreme points. However, we do not think that this will cause any problems in the convex hull computations. If necessary, the input to the convex hull computation may be further reduced using the techniques explained in [Fukuda 2000].

4.3 Deriving resources

So far, we have explained by means of examples how instruction set constraints can be captured in an SRM based on an approach computing the convex hull of the instruction set. In this subsection, we give a theorem formulating a necessary and sufficient condition for verifying whether an instruction set has an SRM, thus solving problem 3.5 as defined in Section 3.2. The argument showing that the condition is sufficient includes a transformation from the inequalities describing the convex hull of the instruction set to an SRM. The basic idea of the theorem is that the convex hull of an instruction set defines a new instruction set consisting of all the integer points in the convex hull. If this new instruction set is equal to the original instruction set, then the convex hull provides the basis for an SRM of the original instruction set because it captures precisely all the instruction set constraints. If the new instruction set contains integer points that are not valid instructions in the original set, then the convex hull does not provide an appropriate SRM because the constraints are not sufficiently tight.

Theorem. Let IS be an instruction set. If NIS is the set of all the integer points contained in $conv(\text{IS})$, then IS has an SRM if and only if NIS equals IS.

We prove the sufficiency of the condition in this theorem by giving the following SRM. Recall Definition 3.2 introducing the notion of an SRM and equation (3) that describes the convex hull of an instruction set in terms of inequalities. We need to define three aspects:

- the set of virtual resources R of the SRM contains all the sets of operation types corresponding to an inequality in the convex hull description of (3);
- each operation type needs w instances of all the virtual resources that it is contained in, where w is the weight of the operation type in the inequality in (3) corresponding to the virtual resource;
- the number of instances of a resource equals the bound in the corresponding inequality in (3).

To clarify this construction, consider the example instruction set IS_2 of Figure 9 (b). Figure 9 (d) gives the inequalities describing the convex hull of IS_2. The convex hull consists of five inequalities, which means that the derived SRM has five virtual resources. Inequality (4), for example, corresponds to virtual resource $\{add, mul\}$, as before abbreviated AM. Inequalities (1), (2), (6′) and (7) correspond to virtual resources A, M, MS and AMS, respectively. The number of instances of resource AM is determined by the right hand side of inequality (4), being 3. The other four resources have 2, 2, 4 and 4 instances, respectively. Finally, operation type mul uses one AM resource, because 1 is the weight of mul in inequality (4); it further uses one M resource, one MS resource and one AMS resource. Operation type add uses one A, one AM and one AMS resource; *shift* uses two MS resources and one AMS resource. Figure 9 (f) summarizes the SRM.

It still needs to be shown that the condition in the theorem that NIS equals IS is sufficient to guarantee that the SRM defined above is an appropriate representation of the instruction set constraints of IS. The construction of the SRM guarantees that a schedule of all the operations in a DFG constrained by the SRM satisfies all the inequalities describing the convex hull of IS. Recall that NIS contains exactly all the integer points satisfying these inequalities. Thus, if NIS equals IS, the inequalities only allow valid instructions which means that any schedule constrained via the above SRM satisfies all instruction set constraints. Thus, IS has an SRM, namely the one provided above.

As an example consider again instruction set IS_2 of Figure 9 (b). The inequalities in Figure 9 (d) precisely capture all these instructions, i.e., they do not allow any integer solutions that are not instructions in IS_2. Thus, we conclude that IS_2 has an SRM. Figure 9 (f) provides an SRM, which is constructed as explained above.

The necessity of the condition that NIS equals IS in the above theorem immediately follows from the following fact: The convex hull of a set of points is the *smallest* convex set containing all these points (Definition 3.4). In other words, NIS is the smallest set of integer points containing all the instructions in IS that can be described by inequalities of the form given in (3). Thus, if NIS contains a point that is not an element of IS, then it is impossible to describe IS via such inequalities. Since there is a one-to-one correspondence between inequalities and SRMs, IS cannot have an SRM.

To illustrate that there are instruction sets without an SRM, consider the example in Figure 10. Figure 10 gives an instruction set, the inequalities describing its convex hull, and the SRM that can be derived from these inequalities following the above construc-

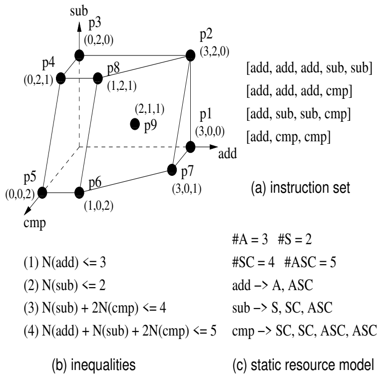


Fig. 10. Verifying an instruction set having no SRM

tion. The inequalities allow instructions $[add, add, add, sub, sub]$, $[add, add, add, cmp]$, $[add, sub, sub, cmp]$, $[add, add, sub, cmp]$, $[add, cmp, cmp]$ plus all their subinstructions. Thus, unfortunately, the inequalities allow instruction $[add, add, sub, cmp]$ that is not part of the initial instruction set; it is depicted as point p9 in Figure 10 (a). As a result of the above theorem, we may conclude that the initial instruction set has no SRM. Point p9 is part of the plane through points p2, p6, p7 and p8. Thus, it is straightforward to verify that we cannot tighten the corresponding constraint expressed by inequality (4) in such a way that it excludes point p9 but none of the other points that do correspond to valid instructions. Although this is a negative result, it provides us useful information for instruction set design. If we could extend the initial instruction set with instruction $[add, add, sub, cmp]$, the resulting instruction set is guaranteed to have an SRM, which implies all the advantages discussed in this paper.

Note that for certain important classes of instruction sets, it is always possible to derive an SRM using our method. One such class is the class of VLIW instruction sets, which serve as the interface for VLIW architectures that are used quite often in media processing. A VLIW instruction is an instruction that is encoded with several issue slots that can be executed in parallel. Each issue slot controls the execution of one operation that possibly can be mapped onto several functional units; the control between different issue slots is completely independent. As a final remark, our method is a generalization of the methods presented in [Eisenbeis et al. 1999], [Timmer et al. 1995], and [Zhao et al. 2001].

5. DESIGN OF THE INSTRUCTION SET ARCHITECTURE FROM AN SRM

Now that we have explained the SRM model, we are going to use that model to do instruction set design. The method explained in this section is based on [Zhao et al. 2002].

Usually the instruction set design process is performed independently from the compiler. Thus it could happen that although a good processor architecture is generated, it cannot produce the desired performance for applications even if a lot of effort is put on generating an efficient compiler. It is a challenge to design an instruction set for an ASIP that can be encoded using a restricted number of instruction bits, while still offering a sufficient degree of parallelism for critical functions in the target application.

5.1 Performance and bottleneck analysis

Similar to the high-level synthesis approach of Figure 3, performance analysis in our case can be done either fast or accurate. An accurate analysis is obtained by actually scheduling the critical loops and examining the load diagrams. These load diagrams enable the designer to identify critical resources. Alternatively, the performance on the critical loops can be estimated in a fast way by considering a well-known lower bound based on available (virtual) processor resources, which is explained next.

When applying software pipelining techniques to loop kernels, the *initiation interval (II)* is an important criterion for measuring the performance. An II is the period between the start times of the execution of two successive loop-body iterations. The *minimum initiation interval (MII)* is a lower bound of the II. The MII can be determined either by a critical resource that is fully utilized or a critical chain of dependencies running through the loop iterations. One lower bound, the *resource constrained MII (ResMII)*, is derived by calculating, in total, the usage requirements for each resource imposed by one iteration of the loop.

Suppose a loop containing 14 add operations is mapped on a data path containing three adders and each adder takes one clock cycle to execute. Then we need at least $\lceil \frac{14}{3} \rceil = 5$ clock cycles to execute the loop iteratively. By doing this calculation for every available resource, we obtain the lower bound ResMII on the initiation interval II of a pipelined schedule of the loop. The general experience is that this bound is very tight. The lower bound indicates the critical functional resource in the data path. This lower bound estimate can therefore be used for bottleneck identification.

In case of instruction set constraints and the corresponding SRM, virtual resources should be taken into account for the performance estimation. The ResMII is no longer simply the totaling of resource usages for each resource because some operations are mapped onto more than one instance of virtual resources. Thus the ResMII has to be updated with respect to the multiple usage of the virtual resources. Suppose a virtual resource M with b instances available, and M is used by a set of operation types $S = \{op_i, i = 1, \cdots, n\}$. Also assuming that the number of operations mapped to operation type op_i is n_i, then ResMII is updated as:

$$\mathrm{ResMII} = \underset{M \in R}{\mathrm{MAX}}\left(\frac{\sum_{i=1}^{n} a_i n_i}{b}\right) \tag{4}$$

where R is the set of virtual resources and a_i is the i-th element in the vector in A in equation (3) corresponding to the resource M.

For example, for the instruction set in Figure 9 (b), inequality (6') in Figure 9 (d) indicates that the *shift* operation uses two instances of the resource MS, of which four are available. For a DFG in Figure 11 (a), the number of operations using each operation type add, mul and *shift* can be expressed as $n_{add} = 2$, $n_{mul} = 6$, $n_{shift} = 4$ respectively. Assume functional resources $adder$, *multiplier* and *shifter* are available, each with two instances. Thus the estimated lower bound for the initiation interval with respect to the functional resources in the data path is 3, shown in Figure 11 (b). Assuming the DFG in Figure 11 (a) is executed by instruction set IS_2, the corresponding virtual resources in Figure 9 (d) modify the estimation as depicted in Figure 11 (c); the result is one clock cycle higher than the original estimation based on only the functional resources. This lower bound is very tight; we can apply the loop folding technique directly by mapping the operations to the virtual resources, as depicted in Figure 9 (f). Figure 12 shows the optimal scheduling result and the virtual resource usages in the loop kernel. Each boxed multiset of operations in Figure 12 (c) gives the virtual resource usage of the corresponding operations in Figure 12 (b) when reading from left to right.

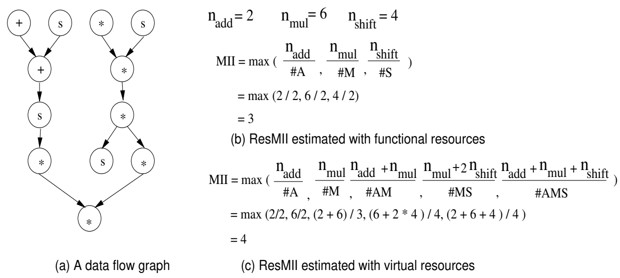


Fig. 11. DFG and the MII estimation

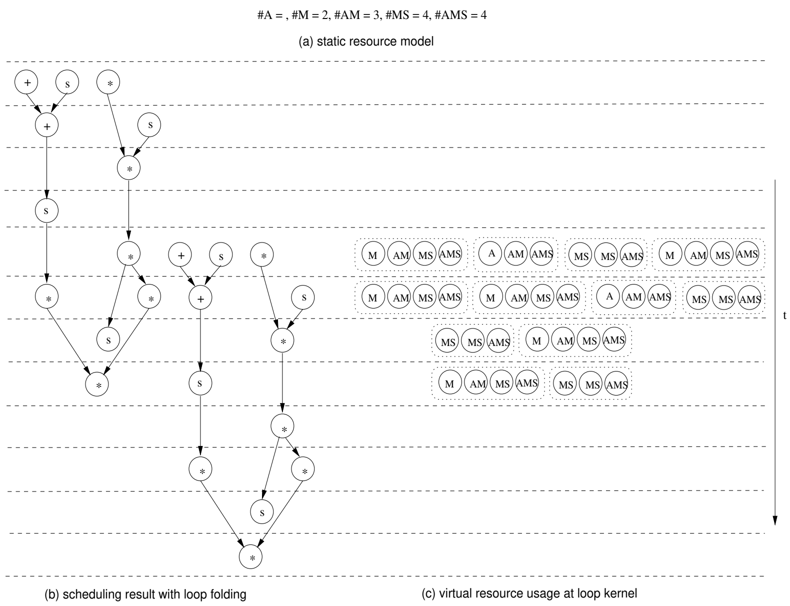


Fig. 12. Scheduling result with loop folding and the virtual resource usage in the kernel

Like in the bottleneck analysis of high-level synthesis of ASICs in Figure 3, the bottleneck of performance now can be identified from the updated lower bound estimation and can be relieved by allocating additional (virtual) resources. In the example of Figure 11, resource MS is the bottleneck causing the ResMII of four. In addition to the allocation of additional resources, in our instruction set design flow we also have the possibility to decrease the resource usage of a critical resource in order to relieve the bottleneck. Therefore it is more convenient to consider the inequalities instead of the resources, because they describe both the resource availability and, for each operation type, the usage of that resource. The way that we relieve the bottleneck is explained in the following subsection.

5.2 Modification of the SRM

Consider again the example of Figure 11. In the previous subsection, virtual resource MS has been identified as the bottleneck. We consider two possibilities to modify the SRM given in Figure 9 (f). Consider inequality $(6')$ in Figure 9 (d).

—One possibility is to increase the number of instances (i.e., the right hand side of the inequality) of virtual resource MS. This change means that possibly more parallel operations are allowed to be exploited under the relaxed constraints; it might result in an increase of the total instruction width.

—A second possibility is to decrease the largest usage (the $shift$ operation) of the resource MS. We consider the largest usage because it has the most impact on the ResMII lower bound. Also this change might result in an increase of the total instruction width; it allows a better balancing of the different operations within the possibly new instruction width constraints because it tends to equalize the weights in the left hand side of an inequality.

These two possibilities to adapt an instruction set by modifying its SRM are generally applicable. The idea is always to first identify the critical virtual resources causing a performance bottleneck and then to relieve the bottleneck through the proposed techniques. An example, given in Figure 13, illustrates the design process. In this example, we consider a loop with 6 add operations and 9 mul operations. The initial instruction set and the corresponding inequalities are shown in Figure 8. The performance requirement of the loop under consideration is given as $\mathrm{II} = 5$.

	initial design	modifying right side	modifying left side
inequality	2 N(add) + 3 N(mul) <= 6	2 N(add) + 3 N(mul) <= 7	2 N(add) + 2 N(mul) <= 6
SRM	#A = 3, #M = 2, #AM = 6	#A = 3, #M = 2, #AM = 7	#A = 3, #M = 2, #AM = 6
mapping	add –> A | AM | AM mul –> M | AM | AM | AM	add –> A | AM | AM mul –> M | AM | AM | AM	add –> A | AM | AM mul –> M | AM | AM
IS	[add, add, add] [add, mul] [mul, mul]	[add, add, add] [add, add, mul] [mul, mul]	[add, add, add] [add, add, mul] [add, mul, mul]
MII	max(6/3, 9/2, (2*6+3*9)/6) = 7	max(6/3, 9/2, (2*6+3*9)/7) = 6	max(6/3, 9/2, (2*6+2*9)/6) = 5
wordlength	24 bits	26 bits	28 bits

Fig. 13. Modification of the SRM

In Figure 13, the first column corresponds to the initial design depicted in Figure 8 (c). The virtual resource AM corresponding to the third inequality is identified as the bottleneck that lower bounds the II to 7, as calculated in the sixth row. In the second column, we evaluate the decision to modify the right hand side of the bottleneck by increasing the number of instances to 7. As a result of this SRM modification, II is now lower bounded by 6. A new instruction $[add, add, mul]$ is added to the instruction set, thereby increasing the instruction width to 26 bits. In the third column, we evaluate the decision to modify the left hand side coefficients unevenly by decreasing the larger one. The lower bound on the II is now 5 and the instruction width amounts to 28 bits. From this example, we see that the design in the third column meets the performance requirements, although it increases the wordlength to 28 bits. We consider a more elaborate example in the next section.

6. EXPERIMENTAL RESULTS

In this section, we provide several experiments of constructing the SRM for certain instruction sets and applying the constructed SRM in an existing resource-constrained scheduler. These experiments are automated when all the maximum instructions are provided. For computing the convex hulls underlying the SRMs, we used the cdd package (available from ftp://ftp.ifor.math.ethz.ch/pub/fukuda/cdd/); for computing schedules we use the tool FACTS which is explained in a little more detail in the next subsection.

The first experiment we performed shows that our method can deal with very flexible designs using complex instructions in DSPs and ASIPs. The second experiment proves that the SRM approach can be used to evaluate the restrictiveness of an instruction set. This provides quite useful information for instruction set design. Loop folding can be applied efficiently to loop kernels with the SRM approach, as shown by the third example. Finally, we also provided a small case study on instruction set design; this last experiment was performed manually.

6.1 Research tool FACTS

Our experiments are carried out with our research tool FACTS [van Eijk et al. 2000]. FACTS has been developed with the aim to deal with the tight combination of timing constraints and resource constraints in signal processing applications. Limited resource availability (e.g. registers) poses a severe problem for traditional methods for code generation. As explained, many of them perform code generation in separate phases. This separation often results in suboptimality (or even infeasibility). While dealing with all these phases takes a large amount of design time and excessive knowledge of the architecture, FACTS takes the approach to deal with all constraints during scheduling. By exploiting these constraints to prune the schedule search space, the scheduler is prevented from making a decision that inevitably violates one or more constraints. After analyzing the feasibility of a scheduling problem, scheduling is performed using branch-and-bound algorithms to satisfy resource constraints and register binding is performed using graph coloring algorithms guaranteeing the satisfaction of capacity constraints. Since the constraints generated by an SRM are straightforward resource constraints, FACTS could be used for our experiments without any adaptations.

6.2 Complex instructions in the TMS320C25

Often in an instruction set for a VLIW architecture, operation types are represented by a reservation table and are constrained by issue slots. In this case, approaches in [Eisenbeis et al. 1999] [Timmer et al. 1995] can translate the issue slot constraints to static resource constraints and apply the results for a scheduler. However, the instruction sets of DSPs or ASIPs are designed with many irregularities. For instance, in the TMS320C25 processor, complex instructions $[lt, pac]$, $[mpy, pac]$, $[lt, apac]$ are introduced to exploit the limited parallelism in the data path. Such complex instruction constraints cannot be expressed as a reservation table, because not all the combinations of the operation types are valid, e.g., $[mpy, apac]$ is not a valid instruction. Thus virtual resources cannot be derived directly using the reservation table approaches; however, our method can still deal with these complex constraints.

[lt]
[mpy]
[pac]
[apac]
[sacl]
[lt, pac]
[lt, apac]
[mpy, apac]

(a) (complex) instructions

#SLM = 1, #SMP = 1, #SPA = 1 #SLMPA = 2
lt –> SLM, SLMPA
mpy –> SLM, SMP, SLMPA
pac –> SMP, SPA, SLMPA
apac –> SPA, SLMPA
sacl –> SLM, SMP, SPA, SLMPA

(b) SRM for this instruction set

Fig. 14. Instructions for TMS320C25 and the SRM

Instructions for the TMS320C25 processor are shown in Figure 14 (a). They indicate potential parallelism to be exploited. This instruction set can be proven to have an SRM; this SRM is given in Figure 14 (b). We have implemented the UTDSP benchmarks (available via URL http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html) for the TMS320C25 processor and the result is shown in Table I.

Table I. Scheduling result of UTDSP benchmarks on TMS320C25 processor

		no ILP	ILP	with SRM	
	|V|	#instruction	#instruction	#instruction	T(ms)
fft	24	24	20	20	920
fir filter	5	5	5	5	10
iir filter	17	17	14	14	350
lattice filter	14	14	11	11	60
lms fir filter	7	7	6	6	10

The second column shows the number of operations in each application. The column "no ILP" reports the code selection result by a conventional compiler. Since no instruction-level parallelism (ILP) is exploited yet, the number of instructions equals the number of operations in each application. The "ILP" column shows the result when ILP is exploited after the conventional code selection, which is usually based on integer linear programming, a well-known time-consuming approach. For the benchmarks in Table I the run-time is in the order of seconds. Column "with SRM" reports the number of instructions by constructing the SRM first and applying the SRM in our resource-constraint-based scheduler FACTS. Since the complex instructions are quite small, the runtime of constructing the SRM can be ignored. The time reported is the time required to obtain a schedule. As we can see in all the examples, the SRM approach can generate very compact code equivalent to the optimal results in less than one second, which is very efficient.

6.3 Evaluating the restrictiveness of an instruction set

One of the merits of the SRM approach is that by performing instruction scheduling for applications with different SRMs, one can directly see from the results how restrictive an IS is. Experiments below perform the scheduling and register binding for benchmarks with the two instruction sets IS_1 and IS_2 of Section 4. Assume that the hardware resources are ALU, MUL and $SHIFT$, each with two instances available. Each resource has a delay of one clock cycle. Table II lists the results for 'fdct', 'idct', 'ar' for AR filter and 'wdf' for fifth-order digital elliptical wave filter. The second and third column show the latency and register requirements (one register file for allocating all the values) assuming that all the given functional resources can be used in parallel maximally; the fourth and fifth column give these results for IS_1 and the sixth and seventh column for IS_2. #in is the number of inequalities of the SRM, which corresponds to the number of (virtual) resources.

Table II. Scheduling and register binding results for benchmarks

	FS		IS1		IS2	
	L	RF	L_{SRM}	RF_{SRM}	L_{SRM}	RF_{SRM}
fdct	13	12	16	12	21	14
idct	14	11	15	11	20	13
ar	10	6	11	7	14	8
wdf	16	8	16	8	19	11
#in	3		6		4	

From this table, we can see that both the latency and register requirements are minimal for all the examples when the functional resources can be used maximally. Although using the same resources, IS_2 is obviously more restrictive than IS_1, since all the scheduling results are increased quite substantially. Register requirements are also increased because when some operations are postponed during scheduling in order to meet the instruction set constraints, the values they consume have to be stored in registers for a longer time, which increases the register pressure.

6.4 Loop folding with the SRM approach

VLIW instruction sets have an SRM because of their orthogonal instruction format. An example VLIW architecture is the TMS320C62x architecture. The C62x has two identical data paths with four functional units each. Each data path has 16 32-bit registers. Table III shows part of the instructions for one data path supported by the TMS320C62x.

Using the convex hull approach, virtual resources can be created for this instruction set. SRMs can be applied to software pipelining easily, since they only provide static boundaries without modifying the scheduling and register binding algorithms themselves. Table IV shows the scheduling and register binding results for GSM speech-coding algorithms. 'Convolution' and 'viterbi' are the half-rate convolutional encoder and one_viterbi_step for viterbi decoder in channel codec. 'weight' and 'inverse' are the weighting_filter and APCM_inverse_quantization algorithms for regular pulse excitation encoding in speech codec. 'reflect' is the reflection_coefficients for lpc code.

Table III. TMS320C62x instruction set

	L	S	D	M
Integer Adder	add	add	add	
	sub	sub	sub	
	mov	mov	mov	
	cmpgt			
	cmplt			
Logic	and	and	and	
	or	or	or	
	not	not	not	
Shift		shl		
		shr		
			ld	
			st	
Multiplier				mpy

Table IV. Scheduling and register binding results for GSM speech-coding algorithms

	|V|	II	MII	RF	T(ms)
invers	6	2	2	5	10
convol	14	5	5	8	20
reflect	19	7	8	6	170
weight	39	11	11	7	640
viterbi	55	19	19	10	15570

In Table IV, $|V|$ is the number of operations in each application. II is the minimal initiation interval calculated using our research tool FACTS with the SRM obtained for the TMS320C62x as input. The tool starts from an initial initiation interval such that a feasible solution is guaranteed. By reducing the initial value and analyzing the feasibility iteratively, it finally obtains the minimum one. Column MII gives the initiation interval estimated using the method explained in Section 5. RF reports the register requirements under II. The last column reports the total runtime including scheduling and register binding. The results show that our FACTS tool can work with SRMs and obtain good results for software pipelining applied to industrial algorithms in acceptable runtimes. The runtime for "viterbi" is large compared to the others because FACTS is not optimized for handling large applications with many operations in the DFG using more than one virtual resource. As we can also see, all the minimum initiation invervals calculated by FACTS are equal to the estimated values using the method in Section 5 except for "reflect"; these results indicate that the estimation method in Section 5 is quite accurate and could therefore be used for quick bottleneck analysis during instruction set design.

6.5 Instruction set design

We also performed a case study for the instruction set design using the SRM approach.

[a, m, m]
[a, s, s, m]
[a, a, s, l]
[a, a, s, m]
[a, a, a, m]
[a, a, a, s, s]

(a) instruction set

	inequality	SRM	#	II
(1)	N(a) + N(l) <= 3	AL	3	5
(2)	N(s) + N(l) <= 2	SL	2	5
(3)	N(m) + 2N(l) <= 2	ML	2	8
(4)	N(s) + 2N(m) + 3N(l) <= 4	SML	4	8
(5)	N(a) + 2N(m) + 3N(l) <= 5	AML	5	7
(6)	N(a) + N(s) + 2N(m) + 2N(l) <= 5	ASML	5	7

(b) SRM and the estimated initiation intervals

Fig. 15. An instruction set and the SRM

For the instructions given in Figure 15 (a) an SRM can be obtained using the approach of Section 4. We use the fast method for performance evaluation explained in Section 5 rather than performing detailed scheduling. Since the topology of the DFG of the loop is irrelevant for this analysis, we give only the number of operations and their resource usages. We assume the loop contains 9 add operations, 3 sub operations, 4 mul operations and 6 ld operations. add and sub are encoded with 8 bits each, mul and ld with 16 bits. For reason of convenience, we abbreviate add, sub, mul and ld as a, s, m and l. The instruction width is constrained to 40 bits. The fourth column gives the number of instances of each virtual resource and the fifth column lists the estimated initiation interval for each virtual resource. From this column, we obtain that the lower bound MII is 8. Assuming the objective II is set to 6, this design is far below the performance requirements.

	new inequality	II	extra instructions	WL
3l4l	N(m) + N(l) <= 2 N(s) + 2N(m) + 2N(l) <= 4	5 6	[m, l]	40
3r4r	N(m) + 2N(l) <= 3 N(s) + 2N(m) + 3N(l) <= 5	6 6	[m, l] [s, m, m]	40
3l4r	N(m) + N(l) <= 2 N(s) + 2N(m) + 3N(l) <= 5	5 6	[m, l] [s, m, m]	40
3r4l	N(m) + 2N(l) <= 3 N(s) + 2N(m) + 2N(l) <= 4	6 6	[m, l]	40

Fig. 16. Modification for candidates (3) and (4)

Figure 16 shows the different results by applying the modification methods in Section 5 to the bottleneck candidates (3) and (4). The first column refers to the design decision under evaluation. For example, '3l4r' represents the decision to modify the left hand side of inequality (3) and the right hand side of inequality (4). The second column presents the modified inequalities. The third column estimates the II according to the new SRM. Because of the modification, the resource constraints are relieved, and subsequently more instructions are allowed. The fourth column gives the new instructions besides those already provided in Figure 15 (a). The fifth column calculates the wordlength with the new instruction set.

From Figure 16 we can see that all designs sufficiently reduce the bottleneck. We choose the design of row three for the next iteration because it gives the best combination of per-

	new inequality	II	extra instructions	WL
5l6l	N(a) + 2N(m) + 2N(l) <= 5 N(a) + N(s) + 2N(m) + N(l) <= 5	6 6	[s, m, m], [a, m, l]	40
5r6r	N(a) + 2N(m) + 3N(l) <= 6 N(a) + N(s) + 2N(m) + 2N(l) <= 6	6 6	[a, m, l] [a, s, m, m], [a, a, m, m] [a, a, s, s, m], [a, a, a, s, m]	48
5l6r	N(a) + 2N(m) + 2N(l) <= 5 N(a) + N(s) + 2N(m) + 2N(l) <= 6	6 6	[a, m, l] [a, s, m, m] [a, a, s, s, m], [a, a, a, s, m]	48
5r6l	N(a) + 2N(m) + 3N(l) <= 6 N(a) + N(s) + 2N(m) + N(l) <= 5	6 6	[s, m, m], [a, m, l]	40

Fig. 17. Modification for candidates (5) and (6)

formance improvement and extra instructions. The next identified bottlenecks are virtual resources (5) and (6) in Figure 15 and the same procedure is repeated and shown in Figure 17. From this figure, we can see that the second and third design exceed the wordlength limitation and have to be omitted. The first and fourth design meet both the timing and code size constraints and are acceptable.

This example shows that the SRM approach provides a good basis for tuning performance and code size during instruction set design. This administrative approach is quite efficient for current embedded processor design because without looking at the detailed encoding and performing exact scheduling for the applications the designer can already obtain the knowledge of which adjustment should be made.

7. CONCLUSIONS

In this paper, we propose a method for code generation for issue slot architectures as well as for highly encoded instruction set processors; the method also supports instruction set design. The purpose of the method is to deal with the strong phase coupling between instruction selection and scheduling, caused by the instruction set. We replace the instruction set constraints with a set of virtual resources that implement the individual operations contained in the instruction set. This reduces the need for explicit instruction selection, because any schedule satisfying the resulting static resource constraints can be implemented by the instruction set. With this static resource model, the scheduler has the opportunity to generate better schedules in terms of timing and register requirements. Furthermore, software pipelining can be applied more effectively to exploit the available ILP. Our experiments demonstrate that a static resource model of an instruction set offers the possibility to measure the effectiveness of an instruction set with respect to exploiting the processor resources. This enables instruction set designers to make tradeoffs between the performance and the code size associated with an instruction set.

The SRM model also allows instruction set design via the allocation of extra (virtual) functional resources for the purpose of relaxing critical instruction set constraints, similar to a practical method used in the HLS of ASICs. We have described an iterative design flow comprising a bottleneck analysis based on fast performance estimation of the instruction set on a set of performance-critical loops. The availability of critical virtual resources is increased to relieve the bottleneck, resulting in an extension of the instruction set. A small case study demonstrates the practical applicability of the SRM approach to instruction set design. The resulting instruction set is supported by efficient classic resource constrained compilation, thereby preventing the considerable performance overhead introduced by explicit instruction selection, currently used in ASIP compilers.

8. FUTURE WORK

The key of the SRM approach is to represent some architectural constraints, e.g. encoding constraints, with static resource constraints; such resources only provide a fixed upper limit and any usage within this limit is valid. It is natural to extend this work to support other architectural constraints, e.g. interconnection constraints and bus constraints. In order to avoid using a large register file because of the large area and long access time, designers tend to partition the architecture into clustered data paths. As a consequence, explicit move operations have to be inserted between different clusters for the communication. In order to make use of the register files in a flexible way and reduce explicit move operations as much as possible, some registers are maintained as a common part, which can be written by all the functional units, while others are clustered for access of local functional units. We believe such an architecture can be modeled by an SRM. Another application of SRMs is to support SIMD instruction sets with some improvement of register allocation, since in that context multiple values have to be stored in one register, which is similar to the situation in this paper where multiple operations are executed at the same clock cycle in a single instruction.

In high-level synthesis, multiple-precision specifications are used to be implemented on hardware in a simple way, i.e., operations are mapped onto functional resources that are often wider than necessary, filling the operands with 0's and discarding some bits of the solutions produced. Some work [Ercegovac et al. 1999] suggests the implementation in a more efficient way, e.g., the execution of an operation during several cycles or the execution of multiple operations on a unique resource during the same cycle. Also this resource selection and binding has to be integrated with scheduling to obtain high performance. Efficient results can be obtained using the SRM approach for multiple-precision systems because the data path precisions resemble instruction set constraints very closely.

The experiments discussed in this paper are relatively small. Future experiments should clarify the performance of the SRM approach compared to other techniques, both in the applications discussed in this paper and in possible future applications. Such experiments require that the various techniques are implemented in one environment so that they can be compared under identical circumstances and on the same set of examples. It is clear that it is not straightforward to perform such experiments. However, the current set of experiments shows that the approaching is promising.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their comments which helped to improve the presentation of this paper. Mark de Berg is thanked for his input on the complexity of convex-hull related problems.

REFERENCES

AHO, A. V. 1989. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems 11,* 4 (Oct.), 491–516.

AHO, A. V. AND JOHNSON, S. C. 1976. Optimal code generation for expression trees. *Journal of the ACM 23,* 3 (July), 488–501.

ARAUJO, G. AND MARLIK, S. 1995. Optimal code generation for embedded memory non-homogeneous register architectures. In *Proceedings of the 8th International Symposium on System Synthesis*. IEEE Comput. Soc. Press, Los Alamitos, CA, USA, 36–41.

ARAUJO, G., MARLIK, S., AND LEE, M. 1996. Using register-transfer paths in code generation for heterogeneous memory-register architectures. In *Proceedings of the 33rd Design Automation Conference*. ACM, New York, NY, USA, 591–596.

AVIS, D., BREMNER, D., AND SEIDEL, R. 1997. How good are convex hull algorithms? *Computational Geometry: Theory and Applications 7*, 265–301.

BALA, V. AND RUBIN, N. 1995. Efficient instruction scheduling using finite-state automata. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*. IEEE Comput. Soc. Press, Los Alamitos, CA, USA, 46–56.

BASHFORD, S. AND LEUPERS, R. 1999. Phase-coupled mapping of data flow graphs to irregular data paths. *Design Automation for Embedded Systems 4,* 2/3, 119–165.

CHAZELLE, B. 1993. An optimal convex hull algorithm in any fixed dimension. *Discrete Comput. Geom. 10*, 377–409.

EISENBEIS, C., CHAMSKI, Z., AND ROHOU, E. 1999. Flexible issue slot assignment for VLIW architectures. In *Conference Record of the 4th International Workshop on Software and Compilers for Embedded Systems*.

EMMELMANN, H., SCHROER, F. W., AND LANDWEHR, R. 1989. Beg-a generator for efficient back ends. *SIGPLAN Notices 24,* 7 (July), 227–237.

ERCEGOVAC, M., KIROVSKI, D., AND POTKONJAK, M. 1999. Low-power behavioral synthesis optimization using multiple precison arithmetic. In *Proceedings of the 36th Design Automation Conference*. ACM, New York, NY, USA, 568–573.

FAUTH, A., VAN PRAET, J., AND FREERICKS, M. 1995. Describing instruction set processors using nML. In *Proceedings of the European Design and Test Conference*. IEEE Comput. Soc. Press, Los Alamitos, CA, USA, 503–507.

FRASER, C. W., HENRY, R., AND PROEBSTING, T. A. 1993. Engineering a simple efficient code-generator generator. *ACM Letters on Programming Languages and Systems 1,* 3 (Sept.), 213–226.

FUKUDA, K. 2000. Frequently asked questions in polyhedral computation. Version 16 October 2000. http://www.ifor.math.ethz.ch/˜fukuda/fukuda.html.

GEBOTYS, C. H. 1997. An efficient model for DSP code generation: performance, code size, estimated energy. In *Proceedings of the 10th International Symposium on System Synthesis*. IEEE Comput. Soc. Press, Los Alamitos, CA, USA, 41–47.

GYLLENHAAL, J. G., HWU, W. W., AND RAU, B. R. 1996. Optimization of machine description for efficient use. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Comput. Soc. Press, Los Alamitos, CA, USA, 349–358.

HANONO, S., HADJIYIANNIS, G., AND DEVADAS, S. 1997. Aviv: a retargetable code generator using ISDL. In *Proceedings of the 34th Design Automation Conference*. ACM, New York, NY, USA, 510–515.

HARTMANN, R. 1992. Combined scheduling and data routing for programmable ASIC systems. In *Proceedings of the European Conference on Design Automation*. IEEE Comput. Soc. Press, Los Alamitos, CA, USA, 486–490.

LEUPERS, R. 1997. *Retargetable code generation for digital signal processors*. Kluwer Academic Publishers, Dordrecht.

LEUPERS, R. 2000. Register allocation for common subexpression in DSP data paths. In *Asia South Pacific Design Automation Conference*. IEEE, Piscataway, NJ, USA, 235–240.

LEUPERS, R. AND MARWEDEL, P. 1996. Instruction selection for embedded DSPs with complex instructions. In *Proceedings of the European Design Automation Conference with EURO-VHDL*. IEEE Comput. Soc. Press, Los Alamitos, CA, USA, 200–205.

LIEM, C., MAY, T., AND PAULIN, P. 1994. Instruction-set matching and selection for DSP and ASIP code generation. In *Proceedings of the 7th International Symposium on System Synthesis*. IEEE Comput. Soc. Press, Los Alamitos, CA, USA, 31–37.

LOWNEY, P. G., FREUDENBERGER, A. M., KARZES, T. J., LICHTENSTEIN, W. D., NIX, R. P., AND O'DONNELL, J. S. 1993. The multiflow trace scheduling compiler. *Journal of Supercomputing 7,* 1-2 (May), 51–142.

MESMAN, B., TIMMER, A. H., VAN MEERBERGEN, J. L., AND JESS, J. A. G. 1999. Constraint analysis for DSP code generation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 18,* 1 (Jan.), 44–57.

NOVACK, S., NICOLAU, A., AND DUTT, N. 1995. A unified code generation approach using mutation scheduling. In *Code generation for embedded processors*, P. Marwedel and G. Goossens, Eds. Kluwer Academic Publisher, Dordrecht. Chapter 12.

PAULIN, P. AND LIEM, C. 1996. Embedded systems: tools and trends. Tutorial at the European Design and Test Conference, Paris, France, March 1996.

PREPARATA, R. P. AND SHAMOS, M. I. 1985. *Computational geometry: an introduction*. Springer, Heidelberg, Germany.

PROEBSTING, T. A. AND FRASER, C. W. 1994. Detecting pipeline harzards quickly. In *Conference Record of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, New York, NY, USA, 280–286.

RAU, B. R. 1996. Iterative modulo scheduling. *International Journal of Parallel Programming 24,* 1 (Feb.), 3–64.

RAU, B. R. AND GLAESER, C. D. 1981. Some scheduling techniques and an easily scheduable horizontal architecture for high performance scientific computing. In *Proceedings of the 14th Workshop on Microprogramming*. IEEE, New York, NY, USA, 183–198.

RIMEY, K. AND HILFINGER, P. N. 1988. Lazy data routing and greedy scheduling for application-specific signal processors. In *Proceedings of the 21st Annual Workshop on Microprogramming and Microarchitecture*. IEEE Comput. Soc. Press, Washington, DC, USA, 111–115.

TIMMER, A. H., STRIK, M. T. J., VAN MEERBERGEN, J. L., AND JESS, J. A. G. 1995. Conflict modelling and instruction scheduling in code generation for in-house DSP cores. In *Proceedings of the 32nd Design Automation Conference*. ACM, New York, NY, USA, 593–598.

VAN EIJK, C. A. J., MESMAN, B., ALBA PINTO, C. A., ZHAO, Q., BEKOOIJ, M., VAN MEERBERGEN, J. L., AND JESS, J. A. G. 2000. Constraint analysis for code generation: basic techniques and applications in FACTS. *ACM Transactions on Design Automation of Electronic Systems 5,* 4 (Oct.), 774–793.

VAN PRAET, J., GOOSSENS, G., LANNER, D., AND DE MAN, H. 1994. Instruction set definition and instruction selection for ASIPs. In *Proceedings of the 7th International Symposium on System Synthesis*. IEEE Comput. Soc. Press, Los Alamitos, CA, USA, 11–16.

WESS, B. 1991. Automatic code generation for integrated digital signal processors. In *Proceedings of the IEEE International Symposium on Circuits and Systems*. IEEE, New York, NY, USA.

WILSON, T., GREWAL, G., HALLEY, B., AND BANERJI, D. 1994. An integrated approach to retargetable code generation. In *Proceedings of the 7th International Symposium on System Synthesis*. IEEE Comput. Soc. Press, Los Alamitos, CA, USA, 11–16.

WOUDSMA, R. 1994. EPICS, a flexible approach to embedded DSP cores. In *Proceedings of the International Conference on Signal Processing, Application and Technology*. DSP Associates, Waltham, MA, USA, 506–511.

ZHAO, Q., BASTEN, T., MESMAN, B., VAN EIJK, C. A. J., AND JESS, J. A. G. 2001. Static resource models of instruction sets. In *Proceedings of the 14th International Symposium on System Synthesis*. ACM, New York, NY, USA, 159–164.

ZHAO, Q., MESMAN, B., AND BASTEN, T. 2002. Practical instruction set design and compiler retargetability using static resource models. In *Proceedings of Design Automation and Test in Europe*. IEEE Comput. Soc. Press, Los Alamitos, CA, USA, 1021–1026.

Received February 2002; accepted July 2002