Optimal Integrated Code Generation for Clustered VLIW Architectures

Christoph Kessler*
PELAB
Department of Computer Science
Linköping University
S-58183 Linköping, Sweden
chrke@ida.liu.se

Andrzej Bednarski
PELAB
Department of Computer Science
Linköping University
S-58183 Linköping, Sweden
andbe@ida.liu.se

ABSTRACT

In contrast to standard compilers, generating code for DSPs can afford spending considerable resources in time and space on optimizations. Generating efficient code for irregular architectures requires an integrated method that optimizes simultaneously for instruction selection, instruction scheduling, and register allocation.

We describe a method for fully integrated optimal code generation based on dynamic programming. We introduce the concept of *residence classes* and *space profiles*, which allows us to describe and optimize for irregular register and memory structures. In order to obtain a retargetable framework we introduce a structured architecture description language, ADML, which is based on XML. We implemented a prototype of such a retargetable system for optimal code generation. Results for variants of the TI C62x show that our method can produce optimal solutions to small but nontrivial problem instances with a reasonable amount of time and space.

Categories and Subject Descriptors

D.3.4 [**Programming languages**]: Processors—*code generation*

General Terms

Algorithms, Design, Languages, Experimentation, Performance

Keywords

Instruction scheduling, register allocation, instruction selection, integrated code generation, dynamic programming, space profile

1. INTRODUCTION

Today's market of communication and consumer electronic systems is growing continuously. Such systems are mainly embedded and require specialized, high performance processors such as digital signal processors (DSPs). Code generation for such architectures must aim at limiting the cost, for instance by decreasing the size of memory modules or the power consumption, or by an efficient utilization of registers.

Generating code consists in solving several subproblems. The most important ones are instruction selection, instruction scheduling and register allocation. First a front-end translate a source program to an *Intermediate Representation* (IR). *Instruction selection* maps IR nodes into semantically equivalent instruction of the target processor. *Instruction scheduling* reorders instructions, preserving data dependencies, to make the program execute more efficiently. Each instruction is mapped to a time slot when it is to be executed. *Register allocation* maps values corresponding to IR nodes to physical registers. An efficient mapping decreases the number of (slow) memory accesses by keeping frequently used values in registers.

The state of the art in production compilers adopts a phase-based approach where each subproblem is solved independently. However, most important subproblems of code generation are strongly interdependent, and this interdependence is particularly strong in the case of irregular processor architectures. In fact, techniques applied in compilers for general purpose processors do not produce expected performance for irregular architectures. This is due to the fact that hardware irregularities in data paths, multi-bank memory organization and heterogeneous register sets impose intricate constraints on the applicability of special combinations of instructions such as multiply and accumulate in the same clock cycle, depending on the location of operands and results, which in turn depend on the current scheduling situation. Hence, the main problems in code generation, namely instruction selection, instruction scheduling, and register allocation, can generally not be optimally solved in subsequent separate compiler optimization phases but must be considered simultaneously in an integrated optimization framework.

We consider an integrated framework that deals with instruction selection, scheduling, and register allocation simultaneously. Of course this increases the complexity of the problem. However, DSP software industry is willing to spend a considerable amount of time and space on optimization for critical program parts.

In previous work we introduced a dynamic programming method for an integrated code generator to produce, for a regular processor architecture, time-optimal code for a given number of registers.

In this paper we present the new concepts of residences and space profiles that allow us to precisely describe irregular register sets and memory structures. They are used as a basic data structure in a modified dynamic programming algorithm to produce time-optimal code for irregular architectures. The framework supports a generalized form of instruction selection for VLIW architectures

*Research partially supported by the CENIIT program of Linköping University, project 01.06.

LCTES'02–SCOPES'02, June 19-21, 2002, Berlin, Germany

that corresponds to forest pattern matching.

In this work we focus on optimality on the basic block level. We also discuss the generalization to extended basic blocks [12]. An extension to global code generation is planned.

Furthermore we introduce ADML, a structured architecture description language based on XML that we developed to make our framework retargetable.

We report first results for variants of the TI C62x that show that our method can produce an optimal solution to small but nontrivial problem instances with a reasonable amount of time and space. The method is intended for the final, aggressively optimizing production run of the compiler. It can also be used to assess the quality of faster heuristics. For instance, we could experimentally prove that the heuristically optimized code given by Leupers [10, Chap. 4] is indeed optimal.

2. FUNDAMENTALS

2.1 Modeling the target processor

We assume that we are given a superscalar or VLIW-like processor with f functional units $U_1, ..., U_f$.

The *unit occupation time* o_i of a functional unit U_i is the number of clock cycles that U_i is occupied with executing an instruction before a new instruction can be issued to U_i.

The *latency* ℓ_i of a unit U_i is the number of clock cycles taken by an instruction on U_i before the result is available. We assume that $o_i \leq \ell_i$.

The *issue width* ω is the maximum number of instructions that may be issued in the same clock cycle. For a single-issue processor, we have $\omega = 1$, while most superscalar processors and all VLIW architectures are multi-issue architectures, that is, $\omega > 1$.

2.2 Basic terminology

In the following, we focus on code generation for basic blocks where the data dependences among the IR operations form a directed acyclic graph (DAG) $G = (V, E)$. In the following, let n denote the number of IR nodes in the DAG.

2.2.1 IR-level scheduling

An *IR-schedule*, or simply *schedule*, of the basic block (DAG) is a bijective mapping $S : \{1, ..., n\} \mapsto V$ describing a linear sequence of the n IR operations in V that is compliant with the partial order defined by E, that is, $(u, v) \in E \Rightarrow S(u) < S(v)$. A *partial schedule* of G is a schedule of a subDAG $G' = (V', E \cap (V' \times V'))$ induced by a subset $V' \subseteq V$ where for each $v' \in V'$ holds that all predecessors of v' in G are also in V'. A partial schedule of G can be extended to a (complete) schedule of G if it is prefixed to a schedule of the remaining DAG induced by $V - V'$.

2.2.2 Instruction selection

Naive instruction selection maps each IR operation v to one of a set $\Psi(v)$ of equivalent, single target instructions: A *matching target instruction* $y \in \Psi(v)$ for a given IR operation v is a target processor instruction that performs the operation specified by v. An *instruction selection* Y for a DAG $G = (V, E)$ maps each IR operation $v \in V$ to a matching target instruction $y \in \Psi(v)$.

Our framework also supports the case that a single target instruction y covers a set χ of multiple IR operations, which is quite common for a low-level IR. This corresponds to a generalized version of tree pattern matching. In particular, the covering pattern needs not have the shape of a tree but may even address disconnected nodes of the DAG, for instance different 16-bit operations that could be combined to a 32-bit MMX instruction.

The converse case of a single IR operation corresponding to multiple target instructions requires either lowering the IR or scheduling on the target level only, rather than on the IR level, if we do not want to compromise optimality by fixing a (prescheduled) sequence of subsequent target instructions for an IR operation.

A target instruction y may actually require time slots on several functional units.

2.2.3 Target-level scheduling

A *target-schedule* is a mapping s of the time slots in $\{U_1, ..., U_f\} \times \mathbb{N}_0$ to instructions such that $s_{i,j}$ denotes the instruction starting execution on unit U_i at time slot j. Where no instruction is started on U_i at time slot j, $s_{i,j}$ is defined as `NOP`. If an instruction $s_{i,j}$ produces a value that is used by an instruction $s_{i',j'}$, it must hold $j' \geq j + \ell_i$. Also, it must hold $j' \geq j'' + o_i$ where $s_{i',j''}$ is the latest instruction issued to $U_{i'}$ before $s_{i',j'}$. Finally, it must hold $|\{s_{i'',j'} \neq \texttt{NOP}, 1 \leq i'' \leq f\}| \leq \omega$.

For a given IR-schedule S and a given instruction selection Y, an optimal target-schedule s can be determined in linear time by a greedy method that just imitates the behavior of the target processor's instruction dispatcher when exposed to the instruction sequence given by $Y(S)$.

The *execution time* $\tau(s)$ of a target-schedule s is the number of clock cycles required for executing s, that is,

$$\tau(s) = \max_{i,j}\{j + \ell_i : \ s_{i,j} \neq \texttt{NOP}\}.$$

A target schedule s is *time-optimal* if it takes not more time that any other target schedule for the DAG.

2.2.4 Register allocation

A *register allocation* for a given target-schedule s of a DAG is a mapping r from the scheduled instructions $s_{i,j}$ to physical registers such that the value computed by $s_{i,j}$ resides in a register $r(s_{i,j})$ from time slot j and is not overwritten before its last use. For a particular register allocation, its register need is defined as the maximum number of registers that are in use at the same time. A register allocation r is optimal for a given target schedule s if its register need is not higher than that of any other register allocation r' for s. That register need is referred to as the *register need* of s. An optimal register allocation for a given target-schedule can be computed in linear time in a straightforward way [4].

A target-schedule is *space-optimal* if it uses no more registers that any other possible target-schedule of the DAG. For single-issue architectures with unit-time latencies, IR-schedules and target-schedules are more or less the same, hence the register allocation can be determined immediately from the IR schedule, such that space-optimality of a target-schedule also holds for the corresponding IR-schedule [9]. In all other cases, the register allocation depends on the target-schedule.

2.3 Basic method

A naive approach to finding an optimal schedule consists in the exhaustive enumeration of all possible schedules, each of which can be generated by topological sorting of the DAG nodes.

Topological sorting maintains a set of DAG nodes with indegree zero, the *zero-indegree set*, which is initialized to the set z_0 of DAG leaves. The algorithm repeatedly selects a DAG node v from the current zero-indegree set, appends it to the current schedule, and removes it from the DAG, which implies updating the indegrees of the parents of v. The zero-indegree set changes by removing v and adding those parents of v that now got indegree zero. This process is continued until all DAG nodes have been scheduled. Most

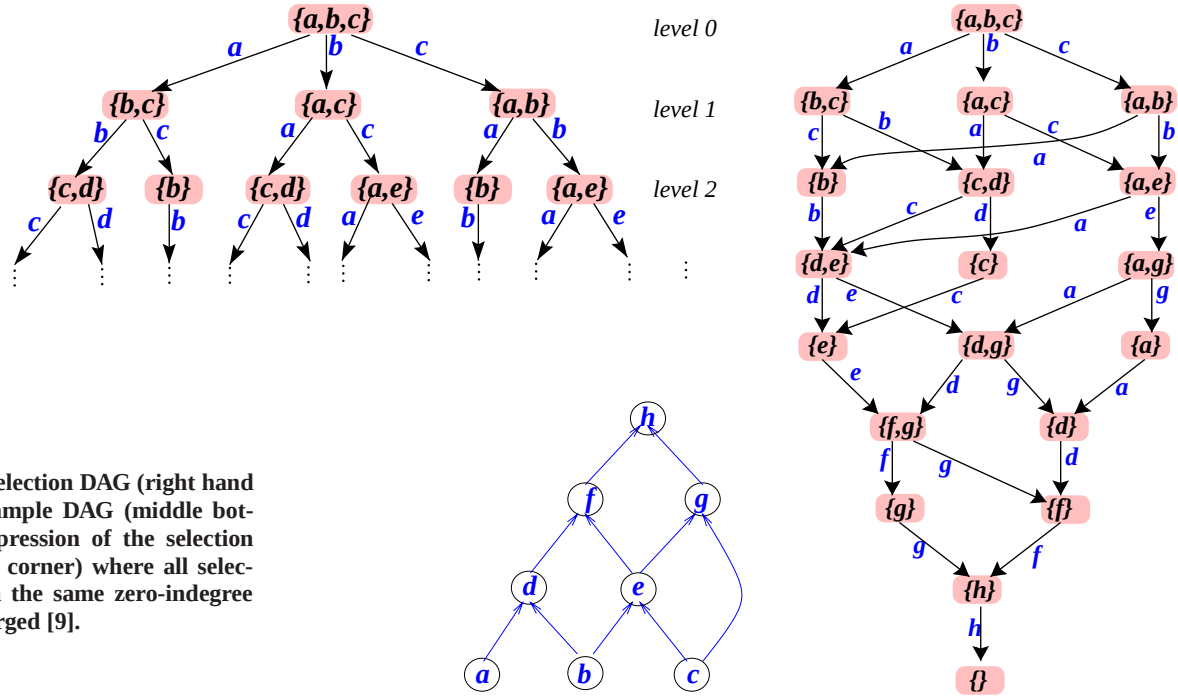

Figure 1: The selection DAG (right hand side) of the example DAG (middle bottom) as a compression of the selection tree (left upper corner) where all selection nodes with the same zero-indegree set could be merged [9].

heuristic scheduling algorithms differ just in the way how they assign priorities to DAG nodes that control which node v is being selected from a given zero-indegree set. If these priorities always imply a strict linear ordering of nodes in the zero-indegree set, such a scheduling heuristic is also referred to as *list scheduling*.

A recursive enumeration of all possibilities for selecting the next node from a zero-indegree list generates all possible IR-schedules of the DAG. This naive enumeration of all different topological sortings of a DAG generates an exponential number of valid schedules. As shown in [9], the naive enumeration cannot be applied for basic blocks larger than 15 instructions even if only space-optimization matters. For time optimization and especially in the context of irregular register sets the number of variants to be enumerated grows even faster.

Exhaustive enumeration of schedules produced by topological sorting implicitly builds a tree-like representation of all schedules of the DAG, called the *selection tree*, which is leveled. Each node of the selection tree corresponds to an instance of a zero-indegree set of DAG nodes during topological sorting. A directed edge connects a node u to a node v of a selection tree if there is a step in the selection process of topological sorting that produces the zero-indegree set v from u.

In previous work [8] we pointed out that multiple instances of the same zero-indegree set may occur in the selection tree. For all these instances, the same set $\mathit{scheduled}(z)$ of nodes in the same subDAG G_z of G below z has been scheduled. This leads to the idea that we could perhaps optimize locally among all the partial schedules corresponding to equal zero-indegree set instances, merge all these nodes to a single selection node and keep just one optimal partial schedule to be used as a prefix in future scheduling steps (see Figure 1). In [9] we have shown that this optimization is valid when computing space-optimal schedules for a single-issue processor. When applying this idea to all nodes of a selection tree, the selection tree becomes a *selection DAG*. In the same way as the selection tree, the selection DAG is leveled, where all zero-indegree sets z that occur after having scheduled $l = |\mathit{scheduled}(z)|$ DAG nodes appear at level l in the selection DAG, see Figure 1. This grouping of partial schedules is applicable to schedules that are *comparable* with respect to the optimization goal. Comparability also depends on the target architecture. For space-optimality on single-issue architectures with unit-time latencies it is sufficient to compare just the zero-indegree sets, since there is a one-to-one relationship between a zero-indegree set z and $\mathit{alive}(z)$, the set of values that reside in registers at the situation described by z:

$$\mathit{alive}(z) = \{u \in \mathit{scheduled}(z): \exists (u,v) \in E,\ v \notin \mathit{scheduled}(z)\ \}$$

The resulting compression of the solution space decreases considerably the optimization time and makes it possible to generate space-optimal schedules for DAGs of reasonable size [9].

2.4 Time profiles

For time-optimal schedules, comparability of partial schedules requires a more involved definition. In previous work [8] we introduced the concept of a *time profile* that represents the occupation status of all functional units at a given time. It records the information which instructions are currently being executed and have not yet completed on every functional unit, which may influence future scheduling decisions. Two partial schedules are comparable if they have the same zero-indegree set and the same time profile [8]. It is sufficient to keep, among comparable schedules, one with least execution time. Together with other optimizations [8], this compression makes the algorithm practical also for medium-sized DAGs with up to 50 nodes.

Formally, a *time profile* $P = (d, p)$ consists of a *issue horizon displacement* $d \in \{0, 1\}$ and a *profile vector*

$$\begin{aligned} p \quad = \quad & (\, p_{1,1}, \ldots, p_{1,\ell_1+d-1}, \\ & p_{2,1}, \ldots, p_{2,\ell_2+d-1}, \\ & \ldots, \\ & p_{f,1}, \ldots, p_{f,\ell_f+d-1}) \end{aligned}$$

of $\sum_{i=1}^{f}(\ell_i + d - 1)$ entries $p_{i,j} \in V \cup \{\texttt{NOP}\}$. An entry $p_{i,j}$ of

the profile vector denotes either the corresponding DAG node v for some instruction y issued to U_i, or a NOP (–) where no instruction is issued. Note that for a unit with unit-time latency there is no entry in p. The displacement d accounts for the possibility of issuing an instruction at a time slot where some other instruction has already been issued. For single-issue processors, d is always 0, thus we can omit d in P. For an in-order multi-issue architecture, we have $d \in \{0, 1\}$ where for $d = 1$ at least one and at most $\omega - 1$ of the entries corresponding to the most recent issue time, $p_{i,1}$ for $1 \leq i \leq f$, must be non-NOP. For out-of-order issue architectures, the displacement could be greater than one, but we need not consider this case as we aim at computing an optimal schedule statically. Of course, not all theoretically possible time profiles do really occur in practice.

The time profile $P = \mathit{profile}(s)$ of a given target-schedule s is determined from s as follows: Let $t \geq 0$ denote the time slot where the last instruction is issued in s. Then $P = (d, p)$ is the time profile that is obtained by just concatenating (in reverse order to determine p) the DAG nodes (or NOPs) corresponding to the $\ell_i + d - 1$ latest entries $s_{i,t-d-\ell_i+2}, ..., s_{i,t}$ in s for the units U_i, $i = 1, ..., f$, where $d = 1$ if another instruction may still be issued at time t to a unit U_i with $s_{i,t} = p_{i,1} = \mathsf{NOP}$, and 0 otherwise. Entries $s_{i,j}$ with $j < 0$ are regarded as NOPs. t is called the *time reference point* of P in s. Note that t is always smaller than the execution time of s.

Hence, a time profile contains all the information required to decide about the earliest time slot where the node selected next can be scheduled: For determining a time-optimal schedule, it is sufficient to keep just one optimal target-schedule s among all those target-schedules s' for the same subDAG G_z that have the same time profile P, and to use s as a prefix for all target-schedules that could be created from these target-schedules s' by a subsequent selection step [8].

3. SPACE PROFILES

A space profile for a zero-indegree set z describes in which register classes the values in $\mathit{alive}(z)$ reside. We introduce the concept of a *residence class* as a generalization of a register class, by modeling memory modules as a special kind of registers.

3.1 Registers and residences

We are given a DSP architecture with k registers $R_1, \ldots, R_k$ and κ different (data) memory modules $M_1, \ldots, M_\kappa$. Standard architectures with a monolithic memory have $\kappa = 1$ memory module.

A value can reside simultaneously in several *residence places*, for instance, in one or several registers and/or in one or several memory modules. For simplicity of presentation we assume that the capacity of each register is one value. Note that this is not generally the case for most DSPs. For instance, for the TI-C62x family DSPs the ADD2 instruction performs two 16-bit integer additions on upper and lower register halves with a single instruction. A corresponding generalization is straightforward. Furthermore, we assume the capacity of a memory module to be unbounded.

Let $\mathcal{R} = \{R_1, \ldots, R_k\}$ denote the set of register names and $\mathcal{M} = \{M_1, \ldots, M_\kappa\}$ the set of memory module names. Then, $\mathcal{RM} = \mathcal{R} \cup \mathcal{M}$ denotes the set of all *residence places*.

The set of the residence places where a certain value v resides at a certain point of time t is called the *residence* of v at time t.

An instruction takes up to two operands (called operand 1 and operand 2 in the following) and generates up to one result (called operand 0 in the following). For each instruction y and each of its operands q, $0 \leq q \leq 2$, the instruction set defines the set of possible residence places, $\mathit{Res}(y, q) \subseteq \mathcal{RM}$.

For instance, for load-store architectures with a single memory module, binary arithmetic operations expect their operands 1 and 2 in various registers and write their result to a register again. An ordinary Load instruction expects of course an address value as operand 1, which should reside in some register, but the value being loaded actually resided in memory. For simplicity of presentation we assume that the loaded value is implicitly given as operand 2 of a Load instruction. Similarly, an ordinary Store instruction takes an address value as operand 1 and a register holding the value to be stored as operand 2, and creates implicitly an operand 0 with residence in memory. There may even be special instructions, called *transfer instructions* in the following, that move values directly between different memory modules or register files. Load and Store could be interpreted as transfer instructions as well. The artificial memory operands allow to model data dependences in memory. Table 1 gives a summary of instruction types for such a simple example architecture.

3.2 Residence classes and versatility

We derive general relationships between registers and register classes by analyzing the instruction set.

For two different residence places R_i and R_j in $\mathcal{RM}$, we denote by $R_i \leq R_j$ (read: R_j is at least as *versatile* as R_i) that for all instructions y, $R_i \in \mathit{Res}(y, q) \Rightarrow R_j \in \mathit{Res}(y, q)$ for $0 \leq q \leq 2$. In other words, wherever R_i can be used as operand, one may use R_j as well.

We denote $R_i \equiv R_j$ for $R_i \leq R_j \wedge R_j \leq R_i$, and we say $R_i < R_j$ iff $R_i \leq R_j \wedge \neg R_j \leq R_i$.

For a given set I of instructions of the target processor, a *register class* is a maximum-size subset of $\mathcal{RM}$, containing registers that can, in all instructions in I, be used interchangeably as operand 1, as operand 2, or as result location. I could be the entire instruction set, or may be narrowed to the set of instructions that are applicable at a certain scheduling situation.

Note that register classes are just the equivalence classes of the "equally versatile" relation ($\equiv$).

EXAMPLE: For the TI-C62x (see Figure 6) there are two register classes A and B consisting of 16 registers each, each one connected to a memory bank.

For the Hitachi SH7729 SH3-DSP [6] with its eight DSP registers, we find that $M_0 \equiv M_1$, $A_0 < A_1$, $Y_1 < Y_0$, $X_1 < X_0$ and $X_0 < A_1$. Via transitivity of $\leq$ this means also $X_1 < A_1$.

Hence, M_0 and M_1 form one register class M, while all other register classes contain just one register and are thus named the same way. □

The versatility relation among registers can be lifted to register classes in a natural way: A register class RC_2 is at least as versatile as a register class RC_1 iff $RC_1 \subseteq RC_2$ and for all $R_1 \in RC_1$, $R_2 \in RC_2$ holds that $R_1 \leq R_2$.

Following the generalization of registers to residences, we obtain the straightforward generalization of register classes to *residence classes*, where the residence class of a memory module residence is just that memory module name itself. Let $\mathcal{P}$ denote the set of all residence classes. For now we do not consider residence classes that overlap.

In order to be able to traverse our solution space in a grid manner, we need a relation that classifies a set of partial schedules according to the location of the operands. We define a function $\mathit{rpot} : V \mapsto \mathbf{N}$ that sum up the numbers of different residence classes for each node in a set ζ of IR nodes:

$$\mathit{rpot}(\zeta) = \sum_{v \in \zeta} |\mathit{residence}(v)|$$

instruction	operand 0	operand 1	operand 2	operand 3	meaning
COMP1	register	register	–	–	unary arithmetic operation
COMP2	register	register	register	–	binary arithmetic operation
MOV	register	register	–	–	add another register residence
LOAD	register	register	(memory)	–	load a value from a memory module
STORE	(memory)	register	register	–	store a value to a memory module
MMOV	(memory)	register	register	(memory)	direct memory–memory move

Table 1: Overview of compute and data transfer instruction types for a simple RISC architecture with a single memory module.

For a given residence class $\alpha \in \mathcal{P}$ we define for a given set ζ of IR nodes

$$rpot(\alpha, \zeta) = \sum_{v \in \zeta} \begin{cases} 1 \text{ it iff } v \text{ resides in } \alpha \\ 0 \text{ otherwise} \end{cases}$$

Since $scheduled(z)$ increases in size at each scheduling step, $rpot(scheduled(z))$ is monotonically increasing as topological sorting proceeds. Additionally, a transfer instruction may move a value to a residence in another residence class where that value was still missing, and thus increases the "residence potential" of that value even if $scheduled(z)$ is not affected by such transfer instructions.

In contrast, $rpot(alive(z))$ is not monotonic since some nodes may leave $alive(z)$ at a scheduling step. Therefore we define the *residence potential* as a function *RPot* that computes, for a given zero-indegree set z and a level $l = |scheduled(z)|$ an index on the residence potential axis of the solution space:

$$RPot(z, l) = l \cdot (|\mathcal{P}| \cdot |V| + 1) + \sum_{\alpha \in \mathcal{P}} rpot(\alpha, alive(z))$$

The maximum residence potential that may occur for a given DAG is $|\mathcal{P}| \cdot |V|$. That means that all nodes of the DAG were present in all residence classes. This gives a simple lower bound of $l \cdot (|\mathcal{P}| \cdot |V| + 1)$ for *RPot* each time we schedule a node (and thus increase the level l) or a transfer instruction that does not increase the level (but the residence potential). Thus, $RPot(z, l)$ is a monotonically growing function in terms of our algorithm.

4. SYNTHESIS

4.1 Time-space profiles

We extend the dynamic programming algorithm of Section 2.3 to cope with irregular register sets and memory structures by combining time profiles and space profiles to a joint data structure, *time-space profiles*. For the comparability of partial schedules the following holds:

THEOREM 1. *For determining a time-optimal schedule, it is sufficient to keep just one optimal target-schedule s among all those target-schedules s' for the same subDAG G_z that have the same time profile P and the same space profile R and to use s as a prefix for all target-schedules that could be created from these target-schedules s' by a subsequent selection step.*

The proof is a straightforward extension of our proof in [8].

4.2 Structuring of the solution space

We structure the solution space as a three dimensional grid, as shown in Figure 2. The grid axes are the level (i.e., length of the partial schedule), execution time, and the residence potential (in terms of our adapted function *RPot*).

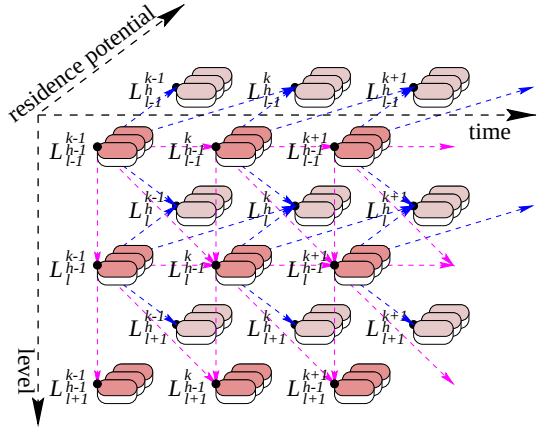

Figure 2: Structuring the space of partial solutions as a three-dimensional grid.

This structure supports efficient retrieval of all possible candidates for comparable partial solutions: They are located within a finite rectangular subgrid whose offset is determined by the schedule length, the time profile's reference time, and the residence potential, and whose extents are determined by the architectural parameters such as maximum latency or number of residence classes.

Another advantage of this grid structure is that, by taking the precedence constraints for the construction of the partial solutions into account, we can change the order of construction as far as possible such that the more promising solutions will be considered first while the less promising ones are set aside and reconsidered only if all the initially promising alternatives finally turn out to be suboptimal. This means that we proceed along the time axis as first optimization goal, while the axes for length and residence potential have secondary priority.

Obviously $rpot(\alpha, \emptyset) = 0$ for all $\alpha \in \mathcal{P}$. A complete schedule ends at level n (after n selection steps). This allows us to optimize the look-up for final solutions by simply checking in the solution space, at a given computation time k, the level n. Note that $RPot(n, \emptyset) = n(n \cdot |\mathcal{P}| + 1)$, called h_m in the following.

4.3 Implications to instruction selection

An instruction y that may match a node v is only *applicable* (and thus, selectable) in a certain scheduling step if its operands are available in the "right" residence classes. Further instructions may become applicable after applying one or several data transfer instructions that broaden the residence of some values.

We denote by χ the set of IR nodes that are covered by a selected instruction y. Note that applying an instruction y that covers $p = |\chi| > 1$ IR nodes skips $p - 1$ levels in the extended selection DAG. The topological sorting algorithm is extended accordingly to handle selections with $p > 1$.

4.4 The entire algorithm

We extend the definition of selection nodes [9, 8] accordingly:

An *extended selection node*, or *ESNode* for short, is identified by a quadruple $\eta = (z, t, P, R)$, consisting of a zero-indegree set z, a space profile R, a time profile P, the time reference point t of P in the schedule s which is stored as an attribute $\eta.schedule$ in that node. Technically, ESNodes can be retrieved efficiently, e.g. by hashing (as applied in the current implementation).

According to Theorem 1 it is sufficient to keep as the attribute $\eta.schedule$ of an ESNode η, among all target-schedules with equal zero-indegree set, equal time-profiles and equal space profiles, one with the shortest execution time.

The overall algorithm, *timeopt*, is depicted in Figure 3. The function *update_space* inserts a new ESNode in the extended selection DAG if there is no ESNode with same zero-indegree set, same time profile, same space profile and lower reference time. Otherwise, the new ESNode is discarded immediately. If such an ESNode with higher reference time exists, that ESNode is removed and the new ESNode is inserted instead.

4.5 Example

Figure 4 illustrates the resulting extended selection DAG for the example DAG shown in the lower right corner of Figure 4. The example assumes a two-issue target processor with two functional units, U_1 with latency $\ell_1 = 1$ and U_2 with latency $\ell_2 = 2$. Instructions that could be selected for DAG node *b* are to be executed on unit U_2, those for all other DAG nodes on U_1. For simplicity we assume that the target processor has only two residence classes, namely a general purpose register file and the memory. We assume that the result of U_2 can be written only to memory.

For better readability each node of Figure 4 is organized into three layers. The top layer of each node represents the zero-indegree set z and the execution time in terms of clock cycles of the associated schedule. For example, the top node contains the set of DAG leaves as zero-indegree set, $\{a, b\}$, and the execution time is $0cc$ as the schedule is initially empty. The second layer represents the time profile and its reference time. In our example the time profile has one entry for each functional unit since we have a multi-issue architecture. The dashes in between parentheses denote empty (not yet used) time slots for a functional unit. Initially all functional units are empty and the reference time is 0. The bottom layer shows the space profile, i.e. the mapping of alive nodes to residence classes, and their associated residence potential $RPot(z, |scheduled(z)|)$. Initially the nodes reside in none of the residence classes. We could have started with a preset residence profile, but in this example we assume that the leaves are constants and do not need to reside in any residence class.

In Figure 4 the ESNodes are only grouped according to their level (increasing from top to bottom). The dashed arrows represent transfer instructions inserted by the algorithm, which add a progress in residence potential but not in length. The crossed ESNodes are pruned by the algorithm. ESNodes marked "not selectable" are not expanded further because their space profile does not match the requirements of the candidate instructions that could be selected there.

4.6 Improvement: Exploiting symmetry

To cope with the combinatorial explosion, we exploit symmetry properties similar to those characterized by Chou et al. [2] which allows us to reduce the solution space and thus the optimization time. The main difference is that our integrated framework includes instruction selection, while Chou et al. assume that the instruction selection has already been fixed in an earlier compiler phase.

```
int maxtime ← 0;

function timeopt ( DAG G with n nodes and set z_0 of leaves)
 List<ESNode> L_{h,k,l} ← empty list ∀h ∀k ∀l;
 η_0 ←new ESNode(z_0, P_0, R_0, t_0);
 η_0.schedule ← ∅
 L_{0,0,0}.insert(new List<ESNode> (η_0));
 h_m ← n(|V||P| + 1);
 for k from 0 to infinity do   // outer loop: over time axis
    checkstop(n(|V||P| + 1), |V||P|, k);
    for level l from 0 to n − 1 do
       for residence potential h from l(|V||P| + 1)
                                  to (l + 1)(|V||P| + 1) − 1 do
          for all η = (z, t, P, R) ∈ L_{h,k,l} do
             for all v ∈ z do
                for all target-instructions y ∈ Ψ(v) that are
                      selectable given z and space profile R
                   let χ = { nodes covered by y };
                   z' ← selection(v, z);
                   update_space(η, z', y, χ, l, k, |χ|)
             for all v ∈ alive(z) do
                for all possible transfers T of v do
                   if T selectable then
                      update_space(η, z, T, {v}, l, k, 0)
end function timeopt

function update_space ( ESNode η = (z, t, P, R),
    set z' (zero-indegree set),    target-instruction y,
    set χ of IR nodes covered by y,
    level l,    time k,    integer p (progress) )
 (s', R') ← (η.schedule ⋈ y, R);
 k' ← τ(s');
 P' ← profile(s');
 t' is the time reference point of (P', R') in s'
 η' ← new ESNode (z', t', P', R');
 h' ← RPot(l + p, alive(z'));
 for all L_{h',j,l+p} with k ≤ j ≤ maxtime do
    if η'' ← L_{h',j,l+p}.lookup(z', P', R') exists then
       break;
 if η'' = (z', t'', P', R') exists then
    if k' < j then
       L_{h',j,l+p}.remove(η'');  L_{h',k',l+p}.insert(η');
    else forget η' end if
 else L_{h',k',l+p}.insert(η');
    maxtime ← max(maxtime, k');
 if l + p = n then checkstop(h', 0, k);
end function update_space

function checkstop(h_1, h_2, k)
 for all h from h_1 to h_1 + h_2 do
    if L_{h,k,n}.nonempty() then
       exit with solution η.schedule for some η ∈ L_{h_m,k,n};
end function checkstop
```

Figure 3: The algorithm for determining a time-optimal schedule, taking instruction selection and transfers into account.

We define an *equivalence relation* on IR nodes such that, whenever two equivalent nodes u, v are simultaneously in the zero-indegree set z, we need to consider only one order, say u before v, instead of both, because the other order would finally not make any difference on time and space requirements. An in-depth formal de-

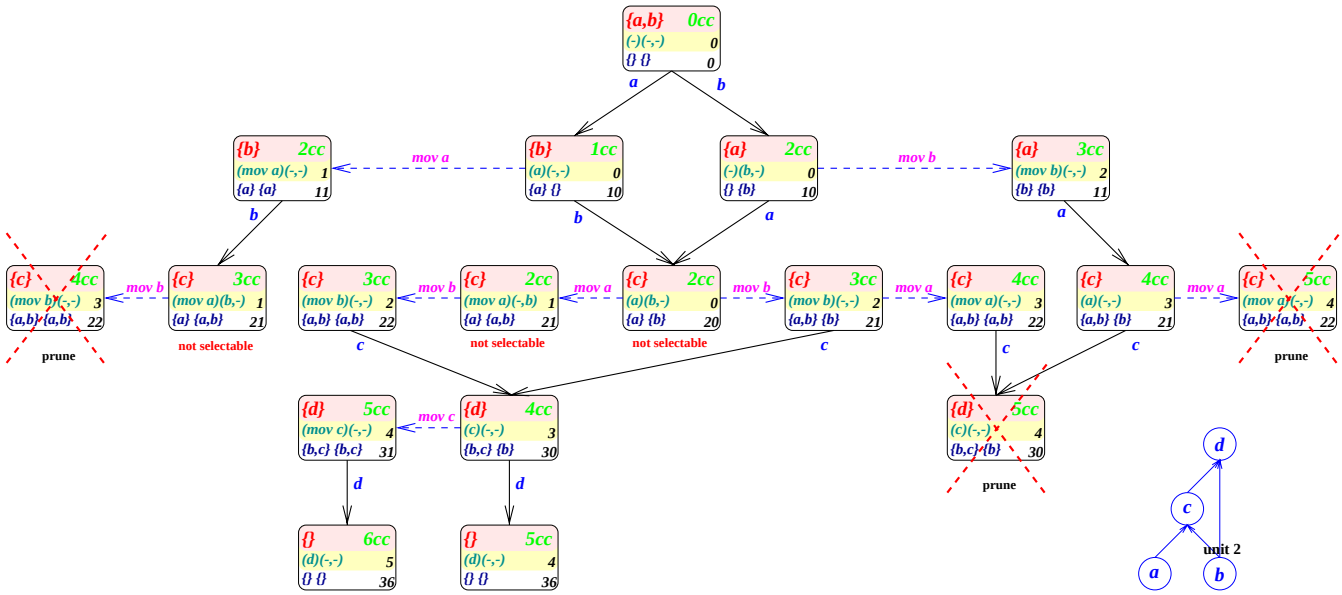

Figure 4: Solution space (extended selection DAG) for the example DAG in the right lower corner.

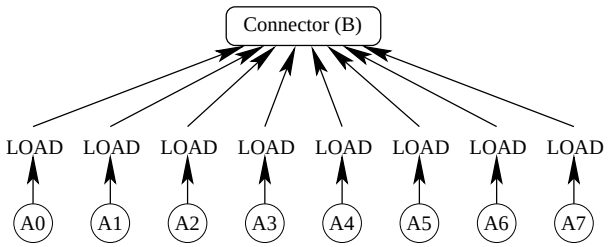

Figure 5: Example DAG [10, Chap. 4]

scription of equivalence in terms of graph isomorphism constraints would go beyond the space limits of this paper. A special case of equivalence is shown in the DAG in Figure 5 where all Load nodes are equivalent to each other.

We improve the *timeopt* algorithm in Figure 3 by modifying the loop **forall** $v \in z$, such that a node $v \in z$ is just skipped if it is equivalent to any other $v' \in z$ that was already tried.

4.7 Extension to global code generation

For a given basic block, we create an artificial entry node with all DAG leaves as successors, and an artificial exit node with all DAG roots as predecessors, to specify values and their residence classes that are alive on entry or at exit from the basic block. These nodes correspond to empty instructions that take no resources and no time but carry a time and space profile. The entry node may be preset with a certain time and space profile. After optimization, the exit node holds the resulting profiles at the end of the basic block. These nodes can serve as connectors to propagate profiles along control flow edges.

The algorithm described above can be immediately applied to *extended basic blocks* [12], by applying it first to the leading basic block, then propagating the time and space profiles of B_1's exit node to the entry nodes of its immediate control flow successors, optimizing for these, and so forth. As there is no join of control flow in an extended basic block, there is no need to merge profiles.

In acyclic regions of the control flow graph, basic blocks with multiple entries require merging of profiles over the ingoing edges, which may lead to a loss of precision. For loops, this would additionally require a fixpoint iteration. Loop unrolling may enlarge the scope of local code generation. However, other code generation techniques for loops, such as software pipelining, should also be taken into account. This is an issue of future research.

5. HARDWARE DESCRIPTION

5.1 Example: TI-C62x DSP

In the experiments with our prototype framework we generated code for several simplified variants of the TI-C62x DSP [13] as shown in Figure 6. The simplifications consist in that we do not cover all possible instructions and have less constraints.

TI-C62x is a load-store architecture with two separate register files A and B, each of which contains 16 general purpose registers of 32 bit length. Eight functional units, including two multipliers, allow a maximum issue rate of eight instructions per clock cycle. The eight functional units are divided into two symmetric groups of four units, where L1, S1, M1 and D1 are connected to register file A,and L2, S2, M2 and D2 are connected to register file B. Most instructions have zero delay slots (MOVE, ADD). The LOAD instruction has a delay of four. Functional units work on their local register file, with the exception that there are two cross-paths that allow up to two values to flow from one register file to another as an operand of an instruction. There are move instructions that allow to transfer data between register files. Those moves use the cross path X1 (resp., X2) to transfer data from register file B to register file A (resp., from register file A to register file B).

Additional constraints, relevant for our following specifications and examples, consist in that the processor may issue up to two LOAD/STORE instructions only if memory addresses are located in different register files.

5.2 ADML

Our goal is to provide a retargetable code generator for various hardware architectures. The structure of a processor, as far as relevant for the generation of optimal code, is specified in a structured

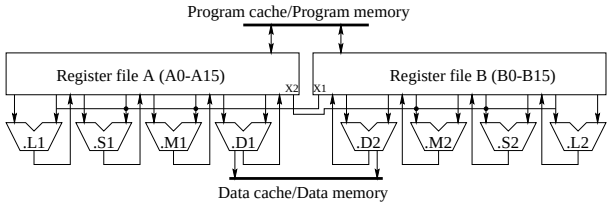

Figure 6: TI-C62x family DSP processor.

architecture description language, called *Architecture Description Mark-up Language* (ADML), which is based on XML.

An ADML document contains four sections:

(1) registers

(2) residence classes

(3) functional units

(4) instruction set

In the following example description ellipses are only used for brevity and are not part of the specification.

```
<architecture omega="8">
  <registers> ... </registers>
  <residenceclasses> ... </residenceclasses>
  <funits> ... </funits>
  <instruction_set> ... </instruction_set>
</architecture>
```

The field `omega` of the `architecture` node specifies the issue width of the architecture.

The `registers` section enumerates all registers present on the architecture. Each register is mapped to a unique name.

```
<registers>
  <reg id="A0"/> <reg id="A1"/> ...
</registers>
```

The `residenceclasses` section defines the different residence classes of the architecture. In our example, the TI-C62x processor has three residence classes: register file `A`, register file `B`, and the memory.

```
<residenceclasses>
  <residenceclass id="A">
    <reg id="A0"/> <reg id="A1"/> ...
  </residenceclass>
  <residenceclass id="B">
    <reg id="B0"/> <reg id="B1"/> ...
  </residenceclass>
  <residence id="MEM" size="32"/>
</residenceclasses>
```

An optional parameter `size` specifies the size of memory modules.

The `funits` part describes the characteristics of each functional unit of the processor in terms of occupation and latency cycles.

```
<funits>
  <fu id="L1" occupation="1" latency="1"/>
  <fu id="S1" occupation="1" latency="1"/>
  ...
</funits>
```

Finally, the `instruction_set` section defines the instruction set for the target processor. Each instruction defines in which residence classes the source operands (`op1` and `op2`) should reside, and in which residence class the result (`op0`) is produced. This defines the constraints for the selectability of a given instruction at a given scheduling step of our dynamic programming algorithm.

An abstract IR node may generally correspond to several semantically equivalent target instructions. For a given IR operator we enumerate the set of all semantically equivalent instructions that cover that operator. Each IR operator is identified by an integer value that is specified as `op` field of the `instruction` node in our description. The `id` field is only for readability and debugging purposes. In the following instruction set specification an IR addition (`ADDP4`) can be computed using either the instruction `ADD .L1`, or `ADD .L2` in TI-C62x notation. That is, an addition can be performed using the functional unit `L1` or `L2`. The choice of the instruction depends on the residence of operands: Instruction `ADD .L1` is selectable if its operands (`op1` and `op2`) are in the residence class `A`, and it produces the result back to the residence class `A`.

```
<instruction_set>
  <instruction id="ADDP4" op="4407">
    <target id="ADD .L1" op0="A"
       op1="A" op2="A" use_fu="L1"/>
    <target id="ADD .L2" op0="B"
       op1="B" op2="B" use_fu="L2"/>
    ...
  </instruction>
  ...
  <transfer>
    <target id="MOVE" op0="r2" op1="r1">
      <use_fu="X2"/>
      <use_fu="L1"/>
    </target>
    ...
  </transfer>
</instruction_set>
```

As part of the instruction set, we additionally specify transfer instructions that move data between different residence classes.

For future extensions of ADML we plan to express additional information, such as the immediate use of a value by multiple functional units via the cross path. Finally, we would like to add a more flexible way of expressing constraints that cannot be expressed in the basic ADML structure.

6. IMPLEMENTATION, FIRST RESULTS

The current implementation is based on our previous framework [8]. We use the LEDA [11] library for the most important data structures and graphs. We use LCC [3] as C front-end. The rest of the system is implemented in C++. For parsing ADML files we use the XML parser Xerces.

The order of constructing the solution space influences considerably the space requirements of our optimizer. The residence potential gives information about the space occupation. We experimented with two different orders of the main loops along the level axis and the residence potential axis. If the innermost loop is along the residence potential axis, the resulting time-optimal schedule exhibits multiple moves that do not affect the total time but that are inserted because higher residence profiles are generally preferred. This however can have a negative impact on code size and energy consumption. For that reason we prefer to have the innermost loop

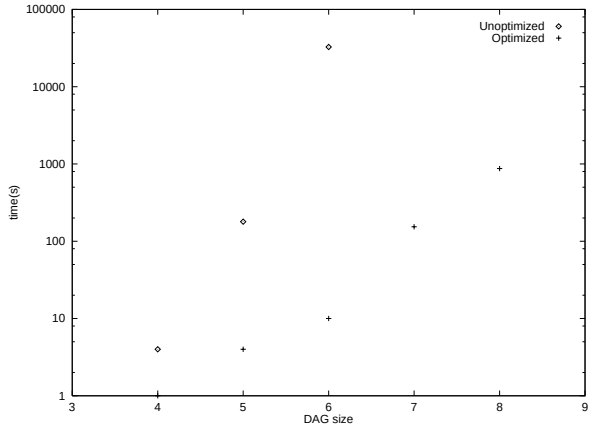

Figure 8: Optimal code generation for parallel loads: optimization time without and with exploiting symmetry.

Basic block	$\|V\|$	time[s]	space[MB]	#Impr.	#merged	#ESNodes
cplxinit	14	201	38	0	140183	44412
vectcopy init	12	16	11	26	61552	12286
vectcopy loop	14	47	15	58	149899	29035
matrixcopy loop	18	18372	194	1782	5172442	722012
vectsum loop	12	11	10	9	36715	8896
vectsum unrolled	17	1537	58	3143	1316684	198571
matrixsum loop	17	10527	154	2898	3502106	564058
dotproduct loop	17	3495	77	4360	2382094	345256
codebk_srch bb33	17	431	41	306	319648	64948
codebk_srch bb29	13	7	12	0	17221	6433
codebk_srch bb26	11	11	13	144	73761	19275
vecsum_c bb6	20	9749	154	5920	3744583	499740
vec_max bb8	13	79	35	0	99504	37254
fni_4 (FFTW) bb6	15	454	59	0	227561	75396
fni_4 (FFTW) bb8	15	432	57	0	225610	72242
fni_4 (FFTW) bb9	17	5106	197	18	1398073	338964
fni_4 (FFTW) bb10	13	110	32	0	102747	35685
fni_4 (FFTW) bb11	15	483	59	0	234767	76753
fni_4 (FFTW) bb12	17	4885	198	16	1385802	338096
codebk_srch bb12	17	4822	142	1545	2214194	376714
codebk_srch bb22	12	8	14	0	14038	8822
codebk_srch bb24	12	2	9	3	6454	2214
fir_vselp bb10	19	9413	131	3463	4003244	565860
fir bb10	14	5106	161	2683	2331577	367212

Table 2: Optimization time and space for various basic blocks taken from DSP benchmark programs. Column #merged indicates how many partial solutions could be covered by the same ESNode, which gives a rough impression of the amount of compression achieved. Column #Impr shows how often an actual improvement in time is obtained by *update_space* in the branch if $k' < j$ (see Fig. 3).

run along the level axis. An alternative could be a postprocessing of a solution to remove unnecessary moves.

The structuring and traversal order of the solution space allows us to optimize the memory consumption of the optimization algorithm. As soon as we leave the level (h, k, l) we can safely remove all nodes stored at that level, because they will never be looked up again. This is achieved by the fact that the residence potential *RPot* is a monotonically growing function.

As a particularly interesting example we consider the family of DAGs consisting of parallel Load operations, as shown in Figure 5 for the case of 8 Loads, which was taken from Leupers' book [10, Chap. 4] that describes a heuristic solution to the same optimization problem for the same architecture. We obtained the optimal solution (see Table 7) in 15 minutes and could thereby prove that the solution reported by Leupers was indeed optimal. The results allow to compare the effect of exploiting symmetry, as described in Section 4.6. For our measurements we used a 1500 MHz PC.

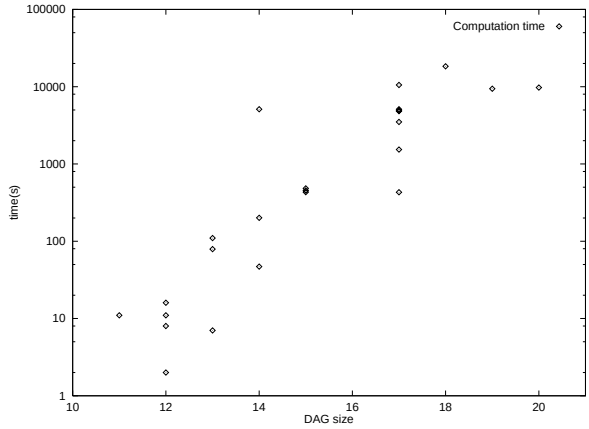

Figure 9: Optimization time requirements

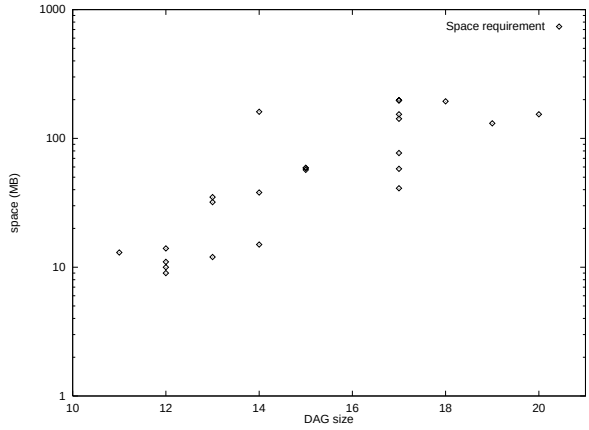

Figure 10: Optimization space requirements

Table 2 shows the behavior of our algorithm on a collection of basic blocks taken from handwritten example programs and DSP benchmarks (TI comp_bench, FFTW). It appears that the algorithm is practical for basic blocks up to size 20. The time and space requirements of these examples are visualized in Figures 9 and 10.

7. FUTURE WORK

Spilling to memory modules is currently not considered, as we assume that the register classes have enough capacity to hold all values of interest. However, this is no longer true for small residence classes, as e.g. in the case of the Hitachi SH3DSP. Our algorithm is, in principle, able to generate optimal spill code and take this code already into account for determining an optimal schedule. On the other hand, taking spill code into consideration may considerably increase the space requirements. We plan to develop further methods for the elimination of uninteresting alternatives. Note that our algorithm automatically considers spilling to *other register classes* already now.

Large DAGs require heuristic pruning of the solution space to cope with the combinatorial complexity. This could, for instance,

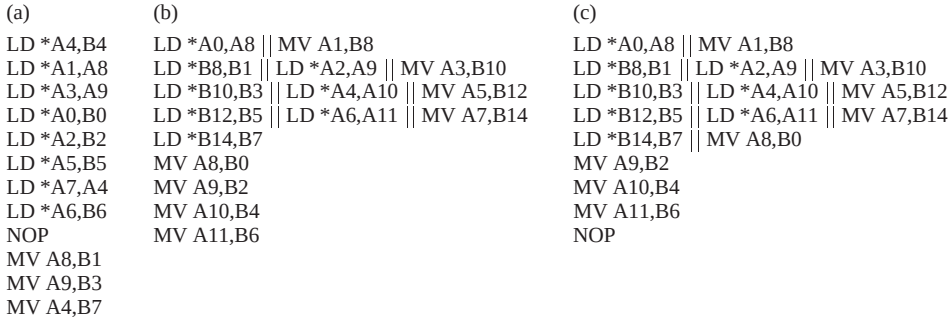

Figure 7: a) schedule generated by TI-C compiler (12 cycles) [10], b) heuristically optimized schedule [10] (9 cycles), c) optimal schedule generated by OPTIMIST (9 cycles)

be achieved by limiting the number of ESNodes per cell of the three-dimensional solution space.

We did not yet really exploit the option of working with a lattice of residence classes that would result from a more general definition of residence classes based on the versatility relation. This is an issue of future research.

8. RELATED WORK

Aho and Johnson [1] use a linear-time dynamic programming algorithm to determine an optimal schedule of expression *trees* for a single-issue, unit-latency processor with homogeneous register set and multiple addressing modes, fetching operands either from registers or directly from memory.

Vegdahl [14] proposes a dynamic programming algorithm for time-optimal scheduling that uses a similar compression strategy as described in Section 2.3 for combining all partial schedules of the same subset of nodes. In contrast to our algorithm, he first constructs the entire selection DAG, which is not leveled in his approach, and then applies a shortest path algorithm. In contrast, we take the time and space requirements of the partial schedules into account immediately when constructing the corresponding selection node. Hence, we need to construct only those parts of the selection DAG that could still lead to an optimal schedule. Instruction selection and residences are not considered in [14].

The split-node DAG technique used in AVIV [5] modifies the DAG such that it contains explicitly the possible variations for generating code. This also includes nodes for transfer instructions. AVIV uses branch-and-bound as basic optimization mechanism with aggressive heuristic pruning. The framework is retargetable by specifying the target architecture in the ISDL specification language.

Chou and Chung [2] enumerate all possible target-schedules to find an optimal one. They propose methods to prune the enumeration tree based on structural properties of the DAG such as a symmetry relation. Their algorithm is suitable for basic blocks with up to 30 instructions, but instruction selection and residences are not considered.

Leupers [10, Chap. 4] uses a phase-decoupled heuristic for generating code for clustered VLIW architectures. The mutual interdependence between the partitioning phase (i.e., fixing a residence class for every value) and the scheduling phase is heuristically solved by an iterative process based on simulated annealing.

Wilson et al. [15] apply integer linear programming to integrated code generation for non-pipelined architectures.

Kästner [7] developed a phase coupled optimizer generator that reads in a processor specification described in *Target Description Language* (TDL) and generates a phase coupled optimizer which is specified as an integer linear program that takes restrictions and features of the target processor into account. An exact and optimal solution is produced, or a heuristic based, if the time limit is exceeded. In this framework, the full phase integration is not possible for larger basic blocks, as the time complexity is too high.

9. REFERENCES

[1] A. Aho and S. Johnson. Optimal Code Generation for Expression Trees. *J. ACM*, 23(3):488–501, July 1976.

[2] H.-C. Chou and C.-P. Chung. An Optimal Instruction Scheduler for Superscalar Processors. *IEEE Transactions on Parallel and Distributed Systems*, 6(3):303–313, 1995.

[3] C. W. Fraser and D. R. Hanson. *A Retargetable C Compiler: Design and Implementation*. Benjamin Cummings Publishing Co., 1995.

[4] R. Freiburghouse. Register Allocation via Usage Counts. *Comm. ACM*, 17(11), 1974.

[5] S. Hanono and S. Devadas. Instruction Selection, Resource Allocation, and Scheduling in the AVIV Retargetable Code Generator. In *Design Automation Conference*, pages 510–515, 1998.

[6] Hitachi Ltd. Hitachi SuperH RISC engine SH7729. Hardware Manual ADE-602-157 Rev. 1.0, Sept. 1999.

[7] D. Kästner. PROPAN: A Retargetable System for Postpass Optimisations and Analyses. In *ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*, June 2000.

[8] C. Kessler and A. Bednarski. A Dynamic Programming Approach to Optimal Integrated Code Generation. In *ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*, June 2001.

[9] C. W. Keßler. Scheduling Expression DAGs for Minimal Register Need. *Computer Languages*, 24(1):33–53, Sept. 1998.

[10] R. Leupers. *Code Optimization Techniques for Embedded Processors*. Kluwer, 2000.

[11] K. Mehlhorn and S. Näher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, 2000.

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

[13] Texas Instruments. *TMS320C6000 CPU and Instruction Set Reference Guide*, October 2000.

[14] S. R. Vegdahl. A Dynamic-Programming Technique for Compacting Loops. In *Proc. 25th Annual IEEE/ACM Int. Symp. Microarchitecture*, pages 180–188. IEEE Computer Society Press, 1992.

[15] T. Wilson, G. Grewal, B. Halley, and D. Banerji. An integrated approach to retargetable code generation. In *Proc. International Symposium on High-Level Synthesis*, pages 70–75, May 1994.