Proc. CPC'06 12th Workshop on Compilers for Parallel Computers, A Coruna, Spain, Jan. 2006, pp. 60–72

Classification and generation of schedules for VLIW processors

Christoph Kessler and Andrzej Bednarski

PELAB, Department of Computer Science
S-58183 Linköping, Sweden
{chrke,andbe}@ida.liu.se
http://www.ida.liu.se/~chrke

Abstract. We identify and analyze different classes of schedules for VLIW processors. The classes are induced by various common techniques for generating or enumerating them, such as integer linear programming or list scheduling with backtracking. In particular, we study the relationship between VLIW schedules and their equivalent linearized forms (which may be used, e.g., with superscalar processors), and we identify classes of VLIW schedules that can be created from a linearized form using an in-order VLIW compaction heuristic, which is just the static equivalent of the dynamic instruction dispatch algorithm of in-order issue superscalar processors.
We also show that, in certain situations, certain schedules generally cannot be constructed by incremental scheduling algorithms that are based on topological sorting of the data dependence graph.
We summarize our findings as a hierarchy of classes of VLIW schedules.
These results can sharpen the interpretation of the term "optimality" used with various methods for optimal VLIW scheduling, and may help to identify classes that can be safely ignored when searching for a time-optimal schedule.

Key words: Instruction-level parallelism, instruction scheduling, code generation, code compaction, integer linear programming, VLIW architecture.

1 Introduction

Instruction scheduling is the task of mapping each instruction of a program to a point (or set of points) of time when it is to be executed, such that constraints implied by data dependences and limited resource availability are preserved. For RISC and superscalar processors with dynamic instruction dispatch, it is sufficient if the schedule is given as a linear sequence of instructions, such that the information about the issue time and the functional unit can be inferred by simulating the dispatcher's behavior. The goal is usually to minimize the execution time while avoiding severe constraints on register allocation. Alternative goals for scheduling can be minimizing the number of registers (or temporary memory locations) used, or the energy consumed.

The problem of finding a *time-optimal schedule*, *i.e.*, a schedule that takes a minimum number of clock cycles to execute, is NP-complete for almost any nontrivial target

Table 1. Some differences in the assembler-level programming model of RISC, superscalar, and VLIW architectures (adapted from [27] and [12]).

	RISC/embedded	Superscalar	VLIW
scheduling / compaction	mostly static	dynamic (hardware dispatcher)	static
instruction stream	linear; sequence of seq. instructions	linear; sequence of seq. instructions	parallel; sequence of long instruction words
issue order	in-order	implicit; in-order or out-of-order	explicit in-order
exposure of parallelism	—	implicit (absence of hazards)	explicit (by compiler)
exposure of data dependence	—	via register names	— (hardcoded by compiler)
exposure of structural hazards	compiler or hardware	hardware dispatcher	programmer/compiler
representation of NOPs	mostly explicit	implicit where NOPs follow from data dep. or resource usage, otherwise explicit	explicit (in uncompressed format)

architecture [1, 7, 15, 24, 25] except for certain combinations of very simple target processor architectures and tree-shaped dependency structures [2, 5, 6, 11, 16, 21, 23, 26].

In many cases, simple heuristics such as greedy list scheduling work well enough to obtain a schedule with decent performance. In some cases, however, the user is willing to afford spending a significant amount of time in optimizing the code, such as in the final compilation of time-critical parts in application programs for DSPs.

For the general case of DAG-structured dependences, various algorithms for time-optimal local instruction scheduling have been proposed, based on integer linear programming [13, 18, 30, 32], branch-and-bound [8, 14, 31], and constraint logic programming [3]. Also, dynamic programming can been used for time-optimal scheduling of basic blocks up to a certain size [29, 20].

In this paper, we consider in-order issue superscalar processors and VLIW processors (with a single general-purpose register file where each register is equally accessible to each instruction). An extension to clustered VLIW processors (with multiple register files, each one connected to a specific subset of the functional units) is possible, but requires to simultaneously consider the partitioning of data and operations across the clusters. For simplicity, we omit this scenario here and refer instead to our general dynamic programming method for integrated code generation for clustered VLIW processors described elsewhere [20].

The remainder of this paper is organized as follows: Section 2 introduces basic terminology. Section 3 describes the various classes of VLIW schedules. Section 4 dis-

cusses the limitations imposed on scheduling by instruction selection decisions, and Section 5 concludes.

2 Background

2.1 A general model of instruction-level parallel target processors

We assume that we are given an in-order issue superscalar or a VLIW processor with f *resources* $U_1, ..., U_f$, which may be functional units or other limited resources such as internal buses.

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, $\omega > 1$.

In order to model instruction issue explicitly, we provide ω separate *instruction issue units* u_i, $1 \leq i \leq \omega$, which can take up at most one instruction per time unit (clock cycle) each. For $1 \leq i \leq \omega$, let I_i denote the set of instructions that can be issued to unit u_i. Let $I \subset I_1 \times ... \times I_\omega$ denote the set of all (statically legal) instruction words. Then, $\mathcal{S}_k \subset I^k$ denotes the set of all (statically legal) schedules whose length is k instruction words, and

$$\mathcal{S} = \bigcup_{k=1}^{\infty} \mathcal{S}_k$$

the set of all (statically legal) schedules. Usually, for all these subset relations holds inequality, because some combinations of instructions within a long instruction word or in a subsequence of long instruction words may be illegal as they need to find data in specific places or would subscribe to the same resource at the same time.

For a VLIW processor, the contents of all u_i at any time corresponds directly to a legal long instruction word. In the case of a superscalar processor, it corresponds to the instructions issued at that time as resulting from the dynamic instruction dispatcher's interpretation of a given linear instruction stream.

Beyond an issue unit time slot at the issue time, an instruction usually needs one or several resources, at issue time or later, for its execution. For each instruction y issued at time (clock cycle) t, the resources required for its execution at time t, $t + 1$, ... can be specified by a *reservation table* [10], a bit matrix o_y with $o_y(i, j) = 1$ iff resource i is occupied by y at time $t + j$. Let $O_i = \max_y\{j : o_y(i, j) = 1\}$ denote the latest occupation of resource i for any instruction. For simplicity, we assume that an issue unit is occupied for one time slot per instruction issued.

An instruction $y \in V$ may read and/or write registers and/or memory locations in certain clock cycles during its execution. For each instruction y, information on the clock cycles (relative to issue time) for reading operands and for writing a result (the latter is usually referred to as the *latency* of y) is given, for instance, in a formal architecture description used in a retargetable code generator. For any data dependence from an instruction y to an instruction y', there is thus a well-defined *delay* $\ell(y, y')$ of clock cycles that must pass between the issue time of y and the earliest possible issue time of y'. Hence, $\ell(y, y')$ annotates the edge (y, y') in the data dependence graph. For

instance, in the case of a data flow dependence $y\delta y'$, the result of instruction y issued at time t is available for instructions y' issued at time $t + \ell(y, y')$ or later. A somewhat more detailed modeling of latency behavior for VLIW processors is given *e.g.* by Rau *et al.* [28].

2.2 Scheduling

We are given a target-level basic block (*i.e.*, instruction selection has already been performed). For basic blocks, the data dependences among the instructions form a directed acyclic graph (DAG) $G = (V, E)$. In the following, let $n = |V|$ denote the number of nodes (instructions) in the DAG.

A *target schedule* (or just *schedule* for short) is a mapping s from the set of time slots in all issue units, $\{1, ..., \omega\} \times \mathbb{N}$, to instructions such that $s_{i,j}$ denotes the instruction issued to unit u_i at time slot j. Where no instruction is issued to u_i at time slot j, $s_{i,j}$ is defined as $\Box$, meaning that u_i is idle. If an instruction $s_{i,j}$ produces a value that is used by an instruction $s_{i',j'}$, it must hold $j' \geq j + \ell(s_{i,j})$. Finally, the resource reservations by the instructions in s must not overlap. The accumulated reservations for all instructions scheduled in s can be described by a *resource usage map* or global reservation table [10], a boolean matrix RU_s where $RU_s(i, j) = 1$ iff resource i is occupied at time j.

The *reference time* $\rho(s)$ of a target schedule s is defined as follows: Let t denote the most recent clock cycle where an instruction (including explicit NOPs) is issued in s to some functional unit. If there is any fillable slot left on an issue unit at time t, we define $\rho(s) = t$, otherwise $\rho(s) = t + 1$. Incremental scheduling methods, such as the one described in the next section, may use the reference time as an earliest time slot for adding more instructions to a schedule.

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

$$\tau(s) = \max_{i,j}\{j + \ell(s_{i,j}) : \; s_{i,j} \neq \Box\}.$$

A target schedule s is *time-optimal* if it takes no more time than any other target schedule for the DAG.

This notion of time-optimality requires a precise characterization of the solution space of all target schedules that are to be considered by the optimizer. In Section 3 we will define several classes of schedules according to how they can be linearized and re-compacted into explicitly parallel form by various compaction strategies, and discuss their properties. This classification will allow us to a-priori reduce the search space, and it will also illuminate the general relationship between VLIW schedules and linear schedules for (in-order-issue) superscalar processors.

3 Classes of VLIW schedules

In a target schedule s, consider any (non-NOP) instruction $s_{i,j}$ issued on an issue unit u_i at a time $j \in \{0, ..., \rho(s)\}$. Let e denote the earliest issue time of $s_{i,j}$ as permitted by data dependences; thus, $e = 0$ if $s_{i,j}$ does not depend on any other instruction.

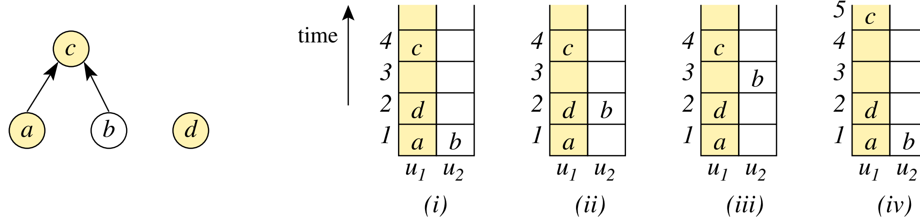

Fig. 1. A target-level DAG with four instructions, and four example schedules: (i) greedy, strongly linearizable, (ii) not greedy, strongly linearizable, (iii) not greedy, not strongly linearizable, (iv) dawdling. Instructions a, c and d with latency 3 are to be issued to unit u_1, and b to unit u_2 with $\ell(b) = 1$. We assume no resource conflicts.

Obviously, $e \leq j$. We say that $s_{i,j}$ is *tightly scheduled* in s if, for each time slot $k \in \{e, e+1, ..., j-1\}$, there were a resource conflict with some instruction issued earlier than time j if $s_{i,j}$ were issued (to u_i) already at time k.

Note that this definition implicitly assumes forward scheduling. A corresponding construction for backward scheduling is straightforward.

3.1 Greedy schedules

An important subclass of target schedules are the so-called greedy schedules where each instruction is issued as early as data dependences and available resources allow, given the placement of instructions issued earlier.

A target schedule s is called *greedy* if all (non-NOP) instructions $s_{i,j} \neq \square$ in s are tightly scheduled. Figure 1 (i) shows an example of a greedy schedule.

Any target schedule s can be converted into a greedy schedule without increasing the execution time $\tau(s)$ if, in ascending order of issue time, instructions in s are moved backwards in time to the earliest possible issue time that does not violate resource or latency constraints [8].

3.2 Strongly linearizable schedules and in-order compaction

A schedule s is called *strongly linearizable* if for all times $j \in \{0, ..., \rho(s)\}$ where at least one (non-NOP) instruction is issued, at least one of these instructions $s_{i,j} \neq \square$, $i \in \{1, ..., \omega\}$, is tightly scheduled.

The definition implies that every greedy schedule is strongly linearizable. The target schedules in Figure 1 (i) and (ii) are strongly linearizable. The schedule in Figure 1 (ii) is not greedy because instruction b is not tightly scheduled. The schedule in Figure 1 (iii) is not strongly linearizable because instruction b, which is the only instruction issued at time 2, is not tightly scheduled.

Any strongly linearizable schedule s can be emitted for an in-order issue superscalar processor without insertion of explicit NOP instructions to control the placement of instructions in s. We can represent each strongly linearizable schedule s by a sequence

Input: Linear schedule (instruction sequence) $S = \langle y_1, ..., y_n \rangle$
Output: Schedule s as in-order compaction of S
Method:
current $\leftarrow$ 0;
RU $\leftarrow$ **new** empty global reservation table;
s $\leftarrow$ **new** empty schedule;
for $i = 1, ..., n$ **do**
    while y_i not yet scheduled **do**
        if y_i can be scheduled at time *current*
            then $s_{i,current} \leftarrow y_i$; commit reservations to *RU*; **break**;
            else *current* $\leftarrow$ *current* + 1;
return s

Fig. 2. Pseudocode for in-order compaction.

$\bar{s}$ containing the instructions of s such that the dynamic instruction dispatcher of an in-order issue superscalar processor, when exposed to the linear instruction stream $\bar{s}$, will schedule instructions precisely as specified in s. For instance, a linearized version of the schedule in Figure 1 (ii) is $\langle a, d, b, c \rangle$. We can compute a *linearized version* $\bar{s}$ from s by concatenating all instructions in s in increasing order of issue time, where we locally reorder the instructions with the same issue time j such that a tightly scheduled one of them appears first in $\bar{s}$. A similar construction allows to construct linearized schedules $\bar{s}$ for EPIC/VLIW processors and reconstruct the original schedule s from $\bar{s}$ if the in-order issue policy is applied instruction by instruction.

In the reverse direction, we can reconstruct a strongly linearizable schedule s from a linearized form $\bar{s}$ by a method that we call *in-order compaction*, where instructions are placed on issue units in the order they appear in $\bar{s}$, as early as possible by resource requirements and data dependence, but always in nondecreasing order of issue time. In other words, there is a nondecreasing "current" issue time t such that all instruction words issued at time 1,...,$t-1$ are already *closed*, *i.e.*, no instruction can be placed there even if there should be a free slot. The instruction word at time t is currently *open* for further insertion of instructions, and the instruction words for time $t+1$ and later are still unused. The "current" issue time t pointing to the open instruction word is incremented whenever the next instruction cannot be issued at time t (because issue units or required resources are occupied or required operand data are not ready yet), such that the instruction word at time $t+1$ is opened and the instruction word at time t is closed. Proceeding in this way, the "current" issue time t is just the reference time $\rho(s)$ of the schedule s being constructed (see Figure 2).

As an example, in Figure 1 (iii), we cannot reconstruct the original schedule s from the sequence $\langle a, d, b, c \rangle$ by in-order compaction, as b would be issued in the (after having issued d at time 1) still free slot $s_{2,1}$, instead of the original slot $s_{2,2}$.

Generally there may exist several possible linearizations for a strongly linearizable schedule s, but their in-order compactions will all result in the same schedule s again.

In-order compaction fits well to scheduling methods that are based on topological sorting of the data dependence graph, because such schedules can be constructed (and optimized) *incrementally*; this is, for instance, exploited in the dynamic programming

algorithms in OPTIMIST [20]. In the context of in-order compaction of strongly linearizable schedules it is thus well-defined to speak about *appending* an instruction y to a schedule s (namely, assigning it an issue slot as early as possible but not earlier than $\rho(s)$, resulting in a new schedule) and about a *prefix* s' of a schedule s (namely the in-order compaction of the prefix $\bar{s'}$ of a suitable linearization $\bar{s}$ of s).

Finally, we note the following inclusion relationship:

Theorem 1. *For each instruction sequence S with greedy compaction s, there exists a (usually different) instruction sequence S' such that the in-order compaction of S' is equal to s.*

Proof. $S' = \bar{s}$, *i.e.*, S' is obtained by linearizing s as defined above.

3.3 Weakly linearizable schedules

In a non-strongly-linearizable schedule $s \in \mathcal{S}$ there is a time $t \in \{1, ..., \rho(s)\}$ such that no instruction issued in s at time t is tightly scheduled. The class of *weakly linearizable schedules* comprises the strongly and nonstrongly linearizable schedules.

Non-strongly linearizable schedules s, such as those in Figure 1 (iii) and (iv), can only be linearized if explicit NOP instructions are issued in the linearized version to occupy certain units and thereby delay the placement of instructions that are subsequently issued on that unit. In Figure 1 (iii), we could reconstruct the schedule, *e.g.*, from the sequence $\langle a, d, \texttt{NOP1}, b, c \rangle$ where `NOP1` denotes an explicit NOP instruction on unit 1.

In principle, all schedules in $\mathcal{S}$ are weakly linearizable, as an arbitrary number of explicit NOPs could be used to guide in-order compaction to produce the desired result.

However, some non-strongly linearizable schedules such as that in Figure 1 (iii) may be superior to greedy or strongly linearizable schedules if register need or energy consumption are the main optimization criteria.

3.4 Non-dawdling schedules

Now we will identify a finite-sized subset of $\mathcal{S}$ that avoids obviously useless cycles and may require only a limited number of NOPs to guide in-order compaction.

We call a schedule $s \in \mathcal{S}$ *dawdling* if there is a time slot $t \in \{1, ..., \tau(s)\}$ such that (a) no instruction in s is issued at time t, and (b) no instruction in s is running at time t, *i.e.*, has been issued earlier than t, occupies some resource at time t, and delivers its result at the end of t or later (see Figure 1 (iv) for an example). In other words, the computation makes no progress at time t, and time slot t could therefore be removed completely without violating any data dependences or resource constraints. Hence, dawdling schedules are never time-optimal. They are never energy-optimal either, as, according to standard energy models, such a useless cycle still consumes a certain base energy and does not contribute to any energy-decreasing effect (such as closing up in time instructions using the same resource) [19]. By repeated removal of such useless time slots, each dawdling schedule could eventually be transformed into a non-dawdling schedule. It is obvious that a dawdling schedule always requires explicit

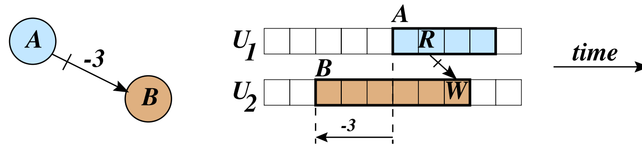

Fig. 3. Example of a negative latency along an antidependence edge in the basic block's data dependence graph (adapted from [28]). Here, instruction A reads in its second cycle (R) a register that the longer-running instruction B overwrites (W) in its last cycle (disjoint resources are assumed). Instruction B could thus be issued up to 3 cycles *earlier* than A without violating the antidependence, *i.e.*, the latency from A to B is -3. However, scheduling methods based on topological sorting of the DAG and greedy or in-order compaction are, in general, unable to issue a dependent instruction earlier than its DAG predecessor.

NOPs for linearization, *i.e.*, is not strongly linearizable. There are also non-dawdling schedules (such as that in Figure 1 (iii)) that are not strongly linearizable.

Even for a fixed basic block of n instructions, there are infinitely many dawdling schedules, as arbitrarily many non-productive cycles could be inserted. There are, however, only finitely many non-dawdling schedules, as there are only n instructions and, in each clock cycle, at least one of them progresses in a non-dawdling schedule. Hence, both from a time optimization and energy optimization point of view, dawdling schedules can be safely ignored.

An extension of OPTIMIST to scan all (topsort-generatable) non-dawdling schedules is possible by considering, when appending another instruction to a schedule, all possibilities for issuing NOP instructions to any issue unit up to the longest latency of running instructions, thus without introducing a useless cycle.

3.5 Schedules generatable by topological sorting

For certain architectures and combinations of dependent instructions, *negative latencies* could occur. For instance, if there were pre-assigned registers, there may be antidependences with negative latencies. This means that an optimal schedule may actually have to issue an instruction earlier than its predecessor in the dependence graph (see Figure 3 for an example).

In VLIW scheduling, such cases cannot be modeled in a straightforward way if methods based on topological sorting of the dependence graph and in-order compaction are used, because these rely on the causal correspondence between the direction of DAG edges and the relative issue order of instructions.

Likewise, superscalar processors could exploit this additional scheduling slack only if out-of-order issue is possible.

This case demonstrates a general drawback of all methods based on topological sorting of the data dependence graph, such as list scheduling [9], Vegdahl's approach [29] and OPTIMIST [20], or Chou and Chung's branch-and-bound search for superscalar processors [8].

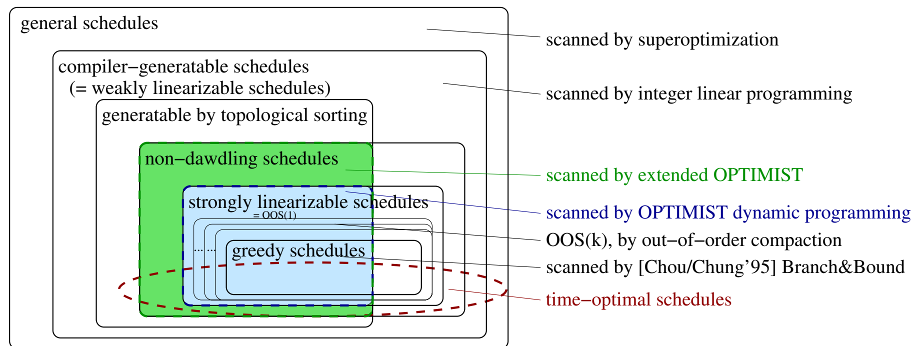

Fig. 4. Hierarchy of classes of VLIW schedules for a basic block.

Workarounds to deal with the negative latency problem may though be possible. A local and very limited workaround could consist in defining superinstructions for frequently occurring types of antidependence chains that are discovered as a whole and treated specially. Another possibility is to build schedules from sequences by greedy or in-order compaction as before but to allow *non-causal instruction sequences* as input, where a successor node may appear in the sequence before its predecessor. Non-causal sequences cannot be generated by topological sorting; instead, all permutations would be eligible. However, this may lead to deadlocks, namely when a free slot is needed at some position in the "past" that has already been occupied. In contrast, causal sequences in connection with monotonic compaction algorithms such as greedy or in-order compaction are deadlock-free, as the schedule can always "grow" in the direction of the time axis. A third possibility to exploit negative latencies could be to keep causal instruction sequences but consider out-of-order compaction. This, however, may delete the direct correspondence between compacted schedule and linearized forms, such as the notion of a schedule prefix as defined above, unless additional restrictions are introduced.

A more elegant solution to this problem of negative latencies can be provided by *integer linear programming*, which does not at all involve the notion of causality because the declarative formulation of an ILP instance as a system of inequalities is naturally acausal. (Internally, ILP solvers may certainly consider the integer variables in some order, which however is generally not under the control of the user.)

3.6 Summary: Hierarchy of classes

Figure 4 surveys the inclusion relationships between the various classes that we have derived in this section. In the cases described in the previous subsection, the set of time-optimal (dashed) schedules may actually not include any topsort-generatable schedule (shaded area) at all. In any case, it will be sufficient to only consider greedy schedules if a time-optimal schedule is sought, and non-dawdling schedules if looking for an energy-optimal schedule.

4 Interaction with instruction selection and superoptimization

Instruction selection maps IR nodes to semantically equivalent target instructions. In compilers, this is usually realized by pattern matching, where the compiler writer specifies, for each target processor instruction, one or several small graphs consisting of IR node and edge templates, often in the form of a tree. Such a pattern *matches* (or *covers*) a subgraph of the IR if an isomorphic mapping can be found. Certainly, there are many possible partitionings of the IR into coverable subgraphs, and for each subgraph there may exist multiple possible coverings. A valid instruction selection is one where each IR node is covered exactly once.

Certainly, better code for the corresponding source-level basic block might be found if different instructions were selected in the instruction selection phase. This is one of the reasons why integrated methods that consider instruction selection, scheduling and maybe other code generation problems such as partitioning or register allocation simultaneously. We refer to previous work [20] for more details.

Even if we admit backtracking of instruction selection to explore potentially better conditions for instruction scheduling, compiler back-ends can generally only consider those target schedules that can be derived from covering the IR with formally specified patterns for instructions in the given, formally specified instruction set. Such a collection of patterns can only yield a conservative approximation to the —in general, undecidable— complete set of codes with the same semantics as the corresponding IR-level DAG. There may thus exist a more involved composition of instructions that computes the same semantic function as defined by the IR DAG in even shorter time, but there is no corresponding covering of the DAG. Finding such an advanced code by exhaustively enumerating arbitrary program fragments and semiautomatically testing or formally verifying them to see whether their behavior is as desired, is known as *superoptimization* [17, 22].

5 Conclusion

We have characterized various classes of VLIW schedules and identified some limitations of scheduling methods that are based on topological sorting.

For optimization purposes, it is sufficient to only consider a finite subset of the overall (infinite) set $\mathcal{S}$ of VLIW schedules. For finding a time-optimal schedule, it is safe to only consider greedy schedules (or any enclosing superclass). Actually, the OPTIMIST optimizer, which is based on topological sorting, scans the somewhat larger class of strongly linearizable schedules, mainly for technical reasons. For finding an energy-optimal schedule, we can limit our search space to non-dawdling schedules.

We can also conclude that scheduling methods based on integer linear programming are, under certain circumstances, superior to methods based on topological sorting of the dependence graph.

Scheduling depends on other decisions made earlier or simultaneously in the code generation process. Alternative choices in instruction selection, partitioning, or register allocation may have considerable influence on scheduling. In [20] and [4] we described integrated code generation methods that solve these problems simultaneously.

Still, instruction selection will be limited by the amount of formally specified semantic equivalences that were supplied by the compiler designer.

Acknowledgements: This research was partially funded by Linköping university, project Ceniit 01.06, and SSF RISE.

References

1. A.V. Aho, S.C. Johnson, and Jeffrey D. Ullman. Code generation for expressions with common subexpressions. *J. ACM*, 24(1), January 1977.
2. Denis Barthou, Franco Gasperoni, and Uwe Schwiegelshohn. Allocating communication channels to parallel tasks. In *Environments and Tools for Parallel Scientific Computing*, volume 6 of *Advances in Parallel Computing*, pages 275–291. 1993.
3. Steven Bashford and Rainer Leupers. Phase-coupled mapping of data flow graphs to irregular data paths. In *DAES*, pages 1–50, 1999.
4. Andrzej Bednarski and Christoph Kessler. Integer linear programming versus dynamic programming for optimal integrated VLIW code generation. In *Proc. 12th Int. Workshop on Compilers for Parallel Computers (CPC'06)*, January 2006.
5. David Bernstein and Izidor Gertner. Scheduling expressions on a pipelined processor with a maximal delay on one cycle. *ACM Trans. Program. Lang. Syst.*, 11(1):57–67, January 1989.
6. David Bernstein, Jeffrey M. Jaffe, Ron Y. Pinter, and Michael Rodeh. Optimal scheduling of arithmetic operations in parallel with memory access. Technical Report 88.136, IBM Israel Scientific Center, Technion City, Haifa (Israel), 1985.
7. David Bernstein, Michael Rodeh, and Izidor Gertner. On the complexity of scheduling problems for parallel/pipelined machines. *IEEE Trans. Comput.*, 38(9):1308–1314, September 1989.
8. Hong-Chich Chou and Chung-Ping Chung. An Optimal Instruction Scheduler for Superscalar Processors. *IEEE Trans. Parallel and Distrib. Syst.*, 6(3):303–313, 1995.
9. E.G. Coffman Jr. (Ed.). *Computer and Job/Shop Scheduling Theory*. John Wiley & Sons, 1976.
10. Edward S. Davidson, Leonard E. Shar, A. Thampy Thomas, and Janak H. Patel. Effective control for pipelined computers. In *Proc. Spring COMPCON75 Digest of Papers*, pages 181–184. IEEE Computer Society Press, February 1975.
11. Christine Eisenbeis, Franco Gasperoni, and Uwe Schwiegelshohn. Allocating registers in multiple instruction-issuing processors. In *Proc. Int. Conf. Parallel Architectures and Compilation Techniques (PACT'95)*, 1995. Extended version available as technical report No. 2628 of INRIA Roquencourt, France, July 1995.
12. Joseph A. Fisher, Paolo Faraboschi, and Cliff Young. *Embedded Computing: A VLIW approach to architecture, compilers, and tools*. Morgan Kaufmann / Elsevier, 2005.
13. C. H. Gebotys and M. I. Elmasry. *Optimal VLSI Architectural Synthesis*. Kluwer, 1992.
14. Silvina Hanono and Srinivas Devadas. Instruction scheduling, resource allocation, and scheduling in the AVIV Retargetable Code Generator. In *Proc. Design Automation Conf.* ACM Press, 1998.
15. John Hennessy and Thomas Gross. Postpass Code Optimization of Pipeline Constraints. *ACM Trans. Program. Lang. Syst.*, 5(3):422–448, July 1983.
16. T. C. Hu. Parallel sequencing and assembly line problems. *Operations Research*, 9(11), 1961.

17. Rajeev Joshi, Greg Nelson, and Keith Randall. Denali: a goal-directed superoptimizer. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, pages 304–314, 2002.
18. Daniel Kästner. *Retargetable Postpass Optimisations by Integer Linear Programming*. PhD thesis, Universität des Saarlandes, Saarbrücken, Germany, 2000.
19. Christoph Kessler and Andrzej Bednarski. Energy-optimal integrated VLIW code generation. In Michael Gerndt and Edmund Kereku, editors, *Proc. 11th Workshop on Compilers for Parallel Computers*, pages 227–238, July 2004.
20. Christoph Kessler and Andrzej Bednarski. Optimal integrated code generation for VLIW architectures. Accepted for publication in *Concurrency and Computation: Practice and Experience*, 2005.
21. Steven M. Kurlander, Todd A. Proebsting, and Charles N. Fisher. Efficient Instruction Scheduling for Delayed-Load Architectures. *ACM Trans. Program. Lang. Syst.*, 17(5):740–776, September 1995.
22. Henry Massalin. Superoptimizer – a look at the smallest program. In *Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-II)*, pages 122–126, 1987.
23. Waleed M. Meleis and Edward D. Davidson. Optimal Local Register Allocation for a Multiple-Issue Machine. In *Proc. ACM Int. Conf. Supercomputing*, pages 107–116, 1994.
24. Rajeev Motwani, Krishna V. Palem, Vivek Sarkar, and Salem Reyen. Combining Register Allocation and Instruction Scheduling (Technical Summary). Technical Report TR 698, Courant Institute of Mathematical Sciences, New York, July 1995.
25. Krishna V. Palem and Barbara B. Simons. Scheduling time–critical instructions on RISC machines. *ACM Trans. Program. Lang. Syst.*, 15(4), September 1993.
26. Todd A. Proebsting and Charles N. Fischer. Linear–time, optimal code scheduling for delayed–load architectures. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, pages 256–267, June 1991.
27. B. R. Rau and Joseph Fisher. Instruction-level parallel processing: History, overview, and perspective. *The J. of Supercomputing*, 7(1–2):9–50, May 1993.
28. B. Ramakrishna Rau, Vinod Kathail, and Shail Aditya. Machine-description driven compilers for EPIC and VLIW processors. *Design Automation for Embedded Systems*, 4:71–118, 1999. Appeared also as technical report HPL-98-40 of HP labs, Sep. 1998.
29. Steven 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.
30. Kent Wilken, Jack Liu, and Mark Heffernan. Optimal instruction scheduling using integer programming. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, pages 121–133, 2000.
31. Cheng-I. Yang, Jia-Shung Wang, and Richard C. T. Lee. A branch-and-bound algorithm to solve the equal-execution time job scheduling problem with precedence constraints and profile. *Computers Operations Research*, 16(3):257–269, 1989.
32. L. Zhang. *SILP. Scheduling and Allocating with Integer Linear Programming*. PhD thesis, Technische Fakultät der Universität des Saarlandes, Saarbrücken (Germany), 1996.