Building a Retargetable Local Instruction Scheduler

Vicki Allan, Department of Computer Science, Utah State University, Logan Utah, 84322, (801) 750-2022, allanv@vicki.cs.usu.edu

Steven J. Beaty, (3612 Chipperfield Court, Fort Collins, Colorado, 80525, (970) 204-0098, beaty@lance.colostate.edu, http://www.lance.colostate.edu/˜beaty

Bogong Su, Dept. of Computer Science, William Patterson College of New Jersey, Wayne, NJ 07470, (201) 595-2960, bsusi@cunyvm.cuny.edu

Philip H. Sweany, Computer Science Department, Michigan Technological University, 1400 Townsend Drive, Houghton MI 49931-1295, (906) 487-3392, sweany@cs.mtu.edu

Abstract

Historically, instruction schedulers have been developed in an ad hoc manner. This paper explores using one scheduler for a number of different architectures and the ramifications of this. In order to achieve this generality, a machine description that encompasses a rich set of architectural features and a scheduler than can accommodate these descriptions are needed. Using the techniques described here, an efficient local instruction scheduler that generates excellent code for instruction-level parallel architectures can be built.

Keywords

Local Instruction Scheduling, Retargetable, Construction

I. INTRODUCTION

Computer manufacturers are continually striving to make faster computers by a combination of faster circuitry and increasing the amount of simultaneous computation (parallelism) in their architectures. One popular method of increasing the degree of simultaneous computation is inclusion of instruction-level parallelism (ILP.) ILP computers exploit the implicit parallelism that most programs contain [54]. They overlap the execution of operations[1] that do not depend on one another. For example, a memory address can be found while the value to store there is computed. Today, typical ILP processors have a memory address, an integer, and several floating point computational units. Future processors will have more of each of these. Some ILP processors reorder operations based on the hardware knowing they depend on each other (usually called *Superscalar* processors). Some (usually called *(V)LIW* processors) do not reorder the instruction stream but instead rely on the instructions themselves to express parallelism. Newer processors have hardware support for speculative execution. Unlike most traditional multiprocessors that usually require explicit parallelism, ILP exploitation does not require users to rewrite programs to use the potential parallelism.

While high-performance architectures have included some ILP for at least 25 years [54], recent computer designs have exploited ILP to a larger degree. This trend shows no sign of reversing. Effective use of ILP hardware requires that the instruction stream be ordered such that, whenever possible, multiple low-level operations can be in execution simultaneously. This

[1]We define an *operation* as an atomic computational function, such as an add, multiply, or memory access. An *instruction* is an abstract representation of the operations that can be issued during a single machine cycle. An instruction might contain more than one operation, and the operation(s) may not be performed in the order they are presented in the instruction.

ordering of machine operations to effectively use an ILP architecture's parallelism is typically called *instruction scheduling* (IS.) This paper describes techniques useful for building a machine-independent instruction scheduler. The development of a machine-independent scheduler is motivated by a desire to re-use the scheduler for multiple ILP architectures. *List scheduling* (LS) [42], a well-defined instruction scheduling technique, provides a basis for such a machine-independent scheduler. LS is well understood and can be easily abstracted to build an excellent scheduler independent of the ILP architecture. The ability to build a scheduler as described here, that can be used for a variety of architectures given a small amount of machine-dependent resource timing information, sharply reduces the time necessary to develop a quality compiler for a new ILP architecture.

IS is typically divided into two categories:

1. *local* IS orders operations only within the context of a single *basic block*[2],
2. *hyperblock* IS schedules loop-free sections of code, and
3. *global* IS considers more than one basic block when ordering operations.

It is commonly held that to best exploit the considerable ILP found in most programs, global scheduling is necessary. Yet, this paper's focus centers around techniques for building a local scheduler. There are several reasons for this. First, while a local scheduler is not sufficient for generation of excellent ILP code, it is necessary as many global scheduling and software pipelining techniques rely on a local scheduler. For example, trace scheduling [29], dominator-path scheduling [63], and superblock scheduling [44] use a local scheduler to schedule multiple adjacent basic blocks as though they were a single block. So do the SP techniques of URCR [60] and URPR [61]. While global scheduling techniques are well-documented, practical discussions of local schedulers are notable in their absence. This paper strives to remedy that disparity.

The remainder of this paper addresses several practical issues involved in the construction of a machine-independent list-scheduling instruction scheduler. Section II describes the machine model which is important for our methods. Section III discusses the list scheduling algorithm in detail. Section IV looks at several practical considerations that can lead to a more effective and efficient list scheduler. Section V shows how our DDD model easily adapts to scheduling instructions for a wide variety of ILP architectures, and Section VI looks at some additional

[2]A basic block is a single entrance, single exit sequence of operations that can have a branch only at the bottom.

concerns brought on by the desire to use a local scheduler within a global scheduling framework.

II. Machine Model

Before we can talk meaningfully about machine-independent instruction scheduling techniques, we need to identify the class of architectures for which such scheduling techniques are designed. We wish to make the model as general as possible to include as many architectures as possible. Our model focuses on machine resource usage as the primary issue in both retargetability and instruction scheduling. While this resource information takes many forms throughout the different phases of a compiler, for instruction scheduling's purposes, the resource information is included in the DDDs used in list scheduling. The architectures are assumed to to operate synchronously. Beyond that, they may have arbitrarily wide instruction formats, pipelined functional execution, permanent and transient storage elements with arbitrary (discrete) setup and hold times, and branches with arbitrary (discrete) branch delays. This encompasses a broad range of architectures, both non-ILP and ILP. While IS is only useful for ILP architectures, there are no inherent limitations for using it for non-ILP machines. It will not benefit the overall runtime of a program in a non-ILP machine, but it will not hurt it either. If IS can be made fast enough from a user standpoint, then a single compiler can be used for many architectures without doing any customization. Indeed, we have produced compilers based on this technology for non-ILP architectures.

A. Data Dependence DAGs

Instruction scheduling involves the placement of machine operations into machine instructions. A data dependence DAG (DDD) is often used to describe the necessary operations and their order. The nodes in a DDD contain the operations, and the edges denote a partial order on the nodes. This partial order is used to guarantee program semantics. The edges of a DDD do not constrain the order nodes are scheduled, only the order they appear in the final schedule.

As our architectural model is based upon resource usage, we rely heavily on the formalism of data dependence analysis. There are three basic types of data dependence as described by Padua et al. [50]:

- Flow Dependence — also called true dependence or data dependence. An operation m_2 is flow dependent on operation m_1 if m_1 executes before m_2 and m_1 writes to some memory

location read by m_2.

- Anti-Dependence — also called false dependence. An operation m_2 is anti-dependent on operation m_1 if m_1 executes before m_2 and m_2 writes to some memory location read by m_1, thereby destroying the value needed by m_1.
- Output Dependence. An operation m_2 is output dependent on operation m_1 if m_1 executes before m_2 and m_2 and m_1 both write to the same location.

In [66], Vegdahl uses both minimum and maximum times on the edges in a DDD to express complicated timings between nodes. In this paper, $\Delta(e) = (min, max)$ will be used to denote the timing associated with an edge in a DDD. This allows the description of a rich set of architectural features, and importantly, allows the description of many different kinds of architectures in one representation. Therefore, a scheduler using this representation can be made generic and will work for any number of different machines. For instruction scheduling the following are easily expressed with edges that have non-infinite maximum timing:

- multi-stage pipes, either homogeneous or heterogeneous (e.g., a single pipe that does both multiplications and additions),
- transient resources such as the latent register designation on I860 [37] pipe operations, and
- other operations extending beyond one clock cycle including delayed branches.

Resources that latch their values (such as general-purpose registers) are modeled with the maximum time set to an infinite value. Processor scheduling, semaphores and other inter-process[or] communication can be modeled using non-infinite maximum timings.

Using $\Delta(e)$, the range of instructions where each operation can be placed can also be calculated. This range will be termed $\Theta(op) = (min, max)$, meaning op can be scheduled in any instruction $I\iota\{\iota \mid min \leq \iota \leq max\}$. This is termed the *absolute timing* [4] for op.

The absolute timing calculation (see Section IV-B) provides an easy method to check for timing errors. For example, if a node's earliest time becomes later than its latest time, a timing error is present in the current schedule. This checking provides a means for detecting errors either in the order of packing nodes from the DDD or in the DDD itself.

To show how the resource dependence information is incorporated into a DDD, consider the code fragment shown in Figure 1. It assumes an assembler format for a hypothetical computer

```
1)      add     r4, r2, r3
2)      mult    r3, r5, r6
3)      add     r7, r1, r8
4)      sub     r8, r4, r2
```

Fig. 1. Sequential Assembly Language Example

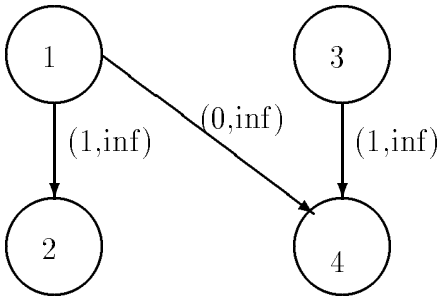


Fig. 2. DDD for Figure 1

that is a simple 3-address assembly language of the form

op source1, source2, destination

To build a DDD, we need to know the data dependence times and relationships for the resources (all registers in this case) involved in the example. Let these times be $(1, \infty)$ for the registers' flow dependence time and $(0, \infty)$ for the registers' anti-dependence time. Figure 2 shows the DDD for the code fragment depicted in Figure 1.

III. List Scheduling

List scheduling derives its name from the fact that a list of data-ready nodes (those with no unscheduled predecessors) is maintained. the input to the list scheduler is typically a DDD, representing dependence among the nodes that must be maintained in order to guarantee original program semantics. During list scheduling, the DDD node to be scheduled next is heuristically selected from the data-ready set (DRS). While this does not guarantee optimal results (local scheduling is, after all, $NP - complete$ [22]), it gives excellent results in practice [42]. In short, list scheduling performs a topological sort of the DDD in an attempt to schedule each DDD node in the shortest possible sequence of instructions subject to

1. the graph's dependence constraints, and
2. the machine's resource limitations.

We should note that list scheduling is not the only instruction scheduling technique available. It is however, both a very general scheduling technique and a very popular one, and thus we will concentrate on list scheduling in this paper. This section gives the list scheduling algorithm (Figure 3) and looks at popular heuristics for choosing the next node to schedule from the DRS.

A. Heuristics

Because list scheduling uses heuristics to prune areas of the search space that appear uninteresting, the heuristics must be chosen with great care so unsearched spaces are truly uninteresting. The choice of giving one operation higher priority than another can have great influence on the final schedule. This is particularly true in the presence of multi-cycle operations. If an operation with a long latency is scheduled late, it may have a large negative influence on the overall schedule length by serializing the code. In architectures where more of the hardware features are visible in order to achieve greater performance, this is counter-productive.

A heuristic often cited [2], [42], [62], [72] as one necessary for efficacious list scheduling is that of *critical path*. A critical path in a DDD is defined to be a longest path from any of the roots to any of the leaves [34]. It is easy to find a critical path in a DDD if all the heights of the nodes are known. Computing the height of any node is simple

- if the node is a leaf, its height is zero,
- else its height is the largest height of its successors, plus one.

To find a critical path, find a largest root node, and follow a highest successor node until reaching a leaf. The path followed will be a critical path.

This definition is correct for unweighted DDDs, that is, those whose edges are of unit length. With the introduction of weights on the edges of the DDD, the definition must be slightly modified. A *schedule-critical path* is one with the greatest sum of the edge weights from all the roots to all the leaves. We define *schedule height* as

- if the node is a leaf, its schedule height is zero,
- else its schedule height is the largest schedule height of its successors, plus the length of the edge ($\Delta(e)_{min}$) to that successor.

In Figure 4, node 1 would have a height of 1 and a schedule height of 6. If node 1 does not

Algorithm List Scheduling

Input:

Operation sequence OP = $op_1, op_2, \ldots, op_s$

Data Dependence DAG, DDD

Resource function, U

Output:

Instruction sequence I = $I_1, I_2, ..., I_k$

Algorithm:

Find the priority of each operation Priority(op_i)

C = 0

Scheduled = $\emptyset$

UnScheduled = OP

DRS = $\emptyset$

WHILE $\exists op$ such that $op \in UnScheduled$ DO

DRS = op_i such that $\forall j\ op_i \succeq op_j \Rightarrow op_j \in Scheduled$

C = C + 1

I_c = $\emptyset$

FOREACH x, $x \in$DRS, in ascending order of Priority(x) DO

IF resource compatible for all K: (as defined by

$\sum U(x, R_k) + U(x, R_k) \preceq 1$)

I_c = I_c + x

Scheduled = Scheduled + x

UnScheduled = UnScheduled - x

$\forall y$ such that $x \preceq y$

IF $\forall z$ such that $z \prec y \Rightarrow z \in Scheduled$

DRS = DRS + y

DRS = DRS - x

end IF

end FOREACH

end WHILE

Fig. 3. List Scheduling Algorithm

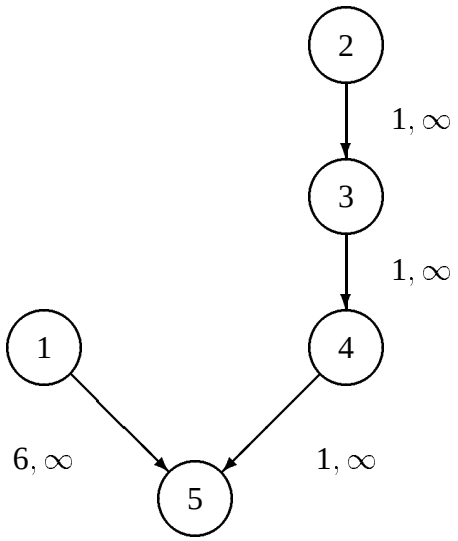


Fig. 4. Critical path comparison

have any machine-resource conflicts with the other nodes, it should be scheduled before any of the others so that it executes in parallel with the others. Basing the choice of operations upon schedule-critical path instead of critical path would accomplish this.[3]

A.1 Specifics

Certainly, a vast number of heuristics are available to reduce the search space of list scheduling. The search space is condensed by choosing one node in the DRS that appears most promising for generating a short, valid schedule. In order to get a short schedule, the schedule-critical path is the most important heuristic. This is because the schedule-critical path defines the lower bound for the length of the schedule. All other nodes might or might not have an impact on the final length; those on a critical path will.[4]

An example where scheduling based on schedule critical path does not produce a valid schedule is shown in Figure 5. Nodes 1, 3, 4, 5, and 6 form a critical path for this DDD. If the resource usage for node 2 conflicts with all the nodes on a critical path, this DDD will not be properly

[3]Critical path and schedule-critical path different only if $\Delta(e)_{min} \neq 1$.

[4]Note that there may be multiple critical paths, i.e. more than one longest path from the sources to the sinks.

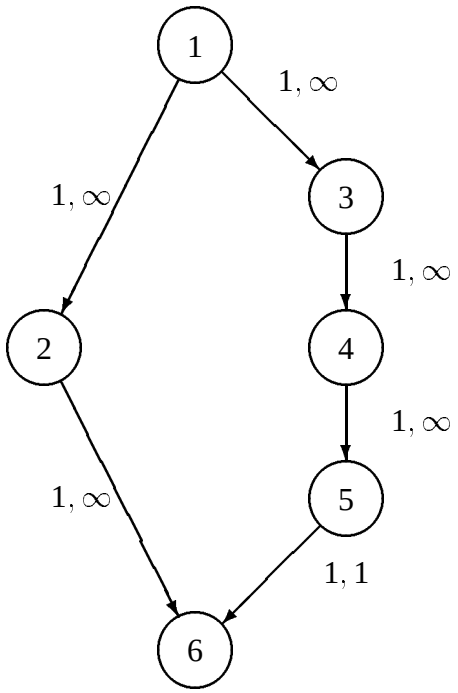


Fig. 5. A DDD where the critical path heuristic fails

scheduled. Nodes 1, 3, 4, and 5 will be placed first. Node 2 is scheduled following node 5, causing a timing violating from 5 to 6. If node 2 had been placed before node 5, the DDD could be scheduled.

Another heuristic often used to produce valid schedules is that of raising the priority of nodes having restricted timing on successor edges. The logic behind this is to first place those nodes that are more "difficult" to schedule. Nodes with unrestricted timing only depend upon being data ready for placement.[5] This is another heuristic that does not help in creating a valid schedule in this DDD.

In Figure 6, the critical path and the restricted successor heuristics form competing, erroneous heuristics. Assume node 2's resources conflict with those of nodes 3, 4, and 5. If the critical path heuristic is given the most weight, followed by the restricted successor, nodes will be scheduled in the following order: 3, 4, 5, and 1. Node 2 will not be able to be placed due to its conflicts with node 3, 4, and 5. If the heuristic importance is switched, nodes will be scheduled in the

[5]As mentioned before, list scheduling uses the data ready condition as its foremost priority heuristic.

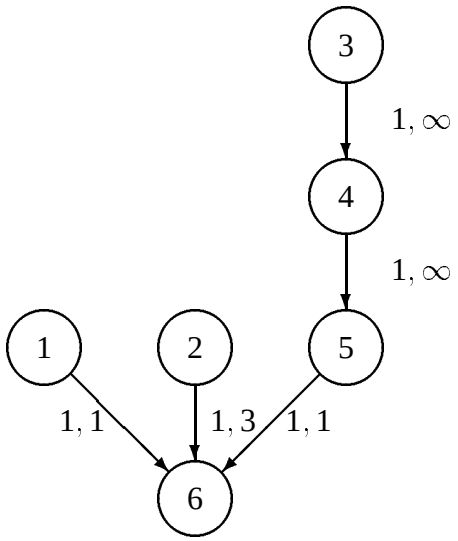


Fig. 6. A DDD where the heuristics compete.

following order: 1, 3, 4, and 5. Again node 2 is unable to be scheduled. Only if node 2 is scheduled somewhere between nodes 3, 4, 5, and 6 will a valid schedule be achieved.

Developing a set of heuristics that attempts to produce a valid schedule for a valid DDD can be challenging. Further, the most useful heuristics for assuring validity during list scheduling vary from architecture to architecture. This results from architecture-dependent features: one may have a restrictive branch delay while another may have a synchronous pipe that does not latch its output. Differing machine features make the generation and reuse of heuristics difficult when a machine-independent scheduler is desired.

A.2 Enumeration

We have investigated many heuristics in achieving valid schedules for a variety of architectures. We found that heuristics useful for one architecture are not for another. Typically, considerable testing goes into choosing heuristics for a particular target, but some heuristics, such as critical path are almost always used. Allan and Mueller describe a *descriminative polynomial selection* [5] mechanism to to combine multiple heuristic factors into one heuristic ranking. Here is a list of heuristics that have been useful for some architecture we have targeted:

1. `height`.
2. `schedule_height`.
3. `on_critical_path`.
4. `on_schedule_critical_path`.
5. `lexical_order` – ordering of nodes from source. Fisher [30] shows that program lexical order is not a good metric for list scheduling priority, our experience agrees with this. This can be used for non-ILP architectures for a default ordering.
6. `branch_node` – the node is a branch node, especially useful in the presence of delayed, restricted branching mechanisms. This has been used to increase the chance a branch node will be placed before other operations.
7. `schedule_spread` – the number of instructions an operation can be placed. The greater the spread, the more flexibility for placement.
8. `resource_usage_of_this_type` – the amount of use of this node's resource in this DDD. The more contention for resources, the earlier a node should be placed in order to free the resource as soon as possible for reuse.
9. `used_and_defined_resources` – as above, nodes that use more resources than others should be scheduled so they do not interfere with others needing those resources.
10. `least_recently_used_resource` – a method of forming round-robin reference to resources.
11. `field_usage_of_this_type` – as with resources, try to minimize field conflicts.
12. `fields_used` – as supra.
13. `least_recently_used_field` – as supra.
14. `successors` – the more successors a node has, the earlier it should be scheduled, allowing its successors to become data ready as early as possible. This exposes more parallelism to the scheduler.
15. `restricted_successors` – the more restricted successors a node has, the earlier it should be scheduled so timing is more flexible within the DDD. Once timing becomes increasingly limited, restricted successors become harder to place.
16. `total_restricted_successors` – total of $\Delta(e)$ for all restricted successors.

17. `shortest_restricted_successor` – restricted successors having a smaller $\Delta(e)$ reduce flexibility, and therefore the possibility for valid scheduling, diminishes.
18. `distance_from_succs` – a measure of how restricted the edges to the successors are.
19. `average_restricted_successor` – the average of $\Delta(e)$ for all the restricted successors.
20. identical heuristics for predecessors as 14 to 19 for successors.

All of these have proven useful in different circumstances for a given DDD.

A.3 Update Interval

Once the factors that will make up the heuristic evaluation have been chosen, we still need to address the issue of when to update the priority weightings for the DDD nodes. Two possibilities exist:

1. calculating the weights once, before the list scheduling algorithm begins (denoted *static* weighting), and
2. calculating after each node is placed in an instruction (denoted *dynamic* weighting).

Certainly, the first method requires the least computation time. It also gives a good estimate of the overall priorities present in the DDD. Its difficulty is that a DDD does not remain static throughout the scheduling process. As operations are placed into instructions, they are removed from the DDD, changing the shape and makeup. This is not reflected in the priorities if they are calculated only once. Of the heuristics listed above, 1, 3, 5, 6, 7, 8, 11, and 12 are not dynamic values and so if they are the only one used, there is no need to compute the priorities dynamically.

One important heuristic that can change during scheduling is critical path. If nodes are scheduled from a critical path, chances are favorable that the path will become shorter than another remaining in the DDD. A simple example is shown in Figure 8. Originally, nodes 1, 2, 3, 4, and 8 are on a critical path. After nodes 1 and 2 are scheduled, nodes 5, 6, 7, and 8 constitute a critical path. Another particularly important heuristic is that of schedule range ($\Theta(op)$). For example, node 3 will originally have $\Theta(3) = (1, \infty)$. After the placement of node 2, node 3 will have $\Theta(3) = (n, n)$ where n is one greater than the scheduled value of $\Theta(2)$. This is a much tighter bound on the range of node 3 and should be reflected in its priority.

The decision as to when to generate the priorities on the nodes is one that must be considered carefully when producing a list scheduler. Empirical results usually drive the decision; if the

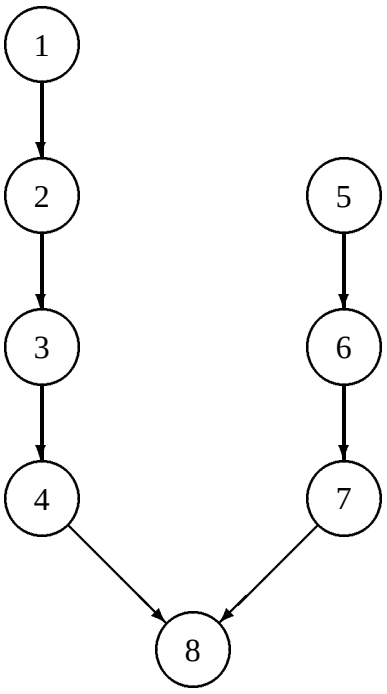


Fig. 7. Critical path competition

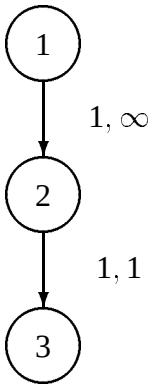


Fig. 8. Need for dynamic schedule range calculation

static method works well, no reason exists to use the dynamic method.

B. DAGs and Orders

Given that DDDs are, by definition, DAGs, we can use the properties of DAGs to provide insight into instruction scheduling. DAG edges provide a *partial order* on the nodes such that an edge between nodes specifies when nodes can "execute" relative to each other. Knuth [38] defines a partial order on a set S as a relation between the objects of S, denoted with "$\preceq$," satisfying the following properties for any objects x, y, and z (not necessarily distinct) in S:

1. If $x \preceq y$ and $y \preceq z$, then $x \preceq z$. (Transitivity.)
2. If $x \preceq y$ and $y \preceq x$, then $x = y$. (Antisymmetry.)
3. If $x \preceq x$. (Reflexivity.)

If $x \preceq y$ and $x \neq y$ then we write $x \prec y$. $\preceq$ is termed "precedes or is equal to", and $\prec$ is termed "precedes." For the relation $\prec$, transitivity is also defined.

Given a DAG, scheduling involves finding a *total order* consistent with the partial order that the DAG represents. A total order is a partial order $a_1, a_2, \ldots, a_n$ such that whenever $a_j \prec a_k$ we have $j < k$ [38]. A algorithm to generate such a total order called a *topological sort*. Of course, there are potentially many different topological sorts for the partial order represented by a DAG.

C. Enumerating Orders

As scheduling a DAG requires performing a topological sort, we might well ask how many different topological sorts are possible for a given DAG. The answer is, in part, that the number of different topological sorts for DAG is primarily determined by the number of edges in the DAG. In a completely-connected DAG D (one with an edge between every pair of nodes on neighboring levels[6]), with nodes N and edges E, the number of different possible orderings is

$$D(N,E) \prod_{l=0}^{l<=levels} (\text{number of nodes in } l)!$$

This formula can be derived by observing that all nodes at level l must be placed before any nodes in level $l+1$. The number of different orderings at any level is the number of permutations for the nodes at that level. This results in a lower bound for the number of orderings for a DAG.

[6]The *level* of a node in a DAG is defined to be length of the longest path from the roots to the node.

The upper bound may be calculated by using a completely unconnected graph. The function is then simply

$$D(N, E)\ N!$$

This is the number of permutations of all the nodes. This represents the ultimate in flexibility. As the number of orderings increase, the number of different final schedules increase, allowing a scheduler more opportunities to create good schedules. A difficulty with reducing the number of edges is the resultant increase in the size of the search space. To make use of the increased flexibility, a powerful search technique must be used.

Recent work by Brightwell and Winkler [16] has shown that determining the actual number of total orders in a DAG, given a partial ordering, is #P–complete. That is, the problem is at least as hard as finding all the Hamiltonian circuits existing in a graph [32]. #P–complete enumeration problems are thought to be "harder" than their corresponding NP–complete existence problems. For example, if P=NP, and it could be shown in polynomial time that an arbitrary graph *contains* a Hamiltonian circuit, it is not apparent that this would provide a polynomial time method of knowing *how many* Hamiltonian circuits exist. The fact that instruction scheduling is NP–complete tells us that no known algorithm can guarantee an optimal schedule without performing an exhaustive search. Given that determining the number of topological sorts possible for a given DAG is #P–complete, we cannot realistically expect to find an optimal solution and thus, heuristics are essential to perform list scheduling.

D. Complexity

List scheduling has a complexity of $O(n^2)$ [42], [33]. This is because it operates on a precedence graph; general precedence graphs have $O(n^2)$ edges [19]. Landskov et al. give another technique for viewing list scheduling's complexity: consider the worst-case DDD, one where no data dependencies (edges) exist between nodes and all nodes have resource conflicts with each other. To schedule any node, one must examine all the nodes that remain in the graph for their current priority to choose the most important one. One must also examine already-placed nodes within the node's range to check for interference with the chosen node; all of them will interfere. Therefore

1. nodes are checked $\frac{(n-1)n}{2}$ times to decide which to schedule next, and

2. chosen nodes are then checked for conflict $\frac{(n-1)n}{2}$ times against the nodes already placed.

This bounds the problem by $n^2 - n$ or $O(n^2)$ (note either bounds the problem by $\frac{n^2}{2}$ or $O(n^2)$.)

Given this complexity bound, can an optimal schedule be found in polynomial time? If list scheduling could generate all possible schedules in polynomial time, the shortest could certainly be chosen in polynomial time. The question is therefore transformed into: Can all schedules be generated in polynomial time? The answer, as indicated in Section III-C is that theory tells us it is highly doubtful.

Consider the generation of the data ready set from which list scheduling chooses operations to be scheduled. To add a member to the DRS, all of the node's predecessors must already have been scheduled (and in some fashion, removed from the graph.) If a node has any unscheduled predecessors, it cannot be added. This operation of finding which nodes to add to the DRS is an example of producing a topological sort of a precedence graph, that has complexity of $O(n^2)$ (topological sorting has $O(\max(nodes, edges))$, and precedence graphs can have $O(n^2)$ edges.) As this is also the complexity of the entire list scheduling technique, list scheduling must be as hard as producing a topological sort of a general precedence graph. As above, calculating conflicts adds to the complexity, but does not change the order. If both resource and encoding conflicts must be checked, complexity can become as great as $3n^2$, still $O(n^2)$. If topological sorting was not required to properly schedule a graph, a method with less complexity might be possible. In Section III-C, generating the number of total orders consistent with a partial order is discussed. An upper bound of $O(n!)$ is given and any process to enumerate the total orderings is said to be #P–complete.

The impact of recognizing that list scheduling is topological sorting has several results, namely it:

1. Shows that it cannot generate a known optimal schedule in polynomial time.
2. Gives a method for viewing list scheduling, i.e. seeing it as topological sorting.
3. Produces a method for analyzing the algorithm.
4. Demonstrates that for valid input, valid output is *possible*.

Another interesting point is observed: list scheduling's building a data ready set, and thereby performing a topological sort, is a heuristic used to create valid schedules. The implicit heuristic is: scheduling nodes with no unscheduled predecessors results in valid total orderings more often

than scheduling nodes with unscheduled predecessors.

IV. Practical Considerations

Given our basic DDD model and the list scheduling techniques described in Section III, there are several additional features that contribute to an effective local scheduler. One of the foremost concerns deals with which "direction" scheduling proceeds. We have assumed to this point that DDDs would be scheduled top-down, from the sources to the sinks. It is certainly possible to scheduled bottom up, from sinks to sources. Section IV-A discusses scheduling direction considerations.

The computation of absolute times for the nodes needs to be efficient in order to have an efficient scheduler. Section IV-B discusses our method.

Another concern when scheduling is the possibility that scheduling will fail. This is a concern because we are using a heuristic method (as we must) and "difficult" timing may lead to scheduling failures. This is especially true when considering edges that represent restrictive timing (non-infinite max times). Several techniques have been suggested to decrease the risk of scheduling failure. Three such techniques, *check and schedule*, *foresighted scheduling*, and *lookahead scheduling* are discussed here.

Finally we consider the often antagonistic relationship between register assignment and instruction scheduling, Both register assignment and instruction scheduling are necessary components of a compiler for ILP architectures. Whichever is executed first during compilation, however, will adversely effect the efficiency of the other. Section IV-F investigates this problem and suggests several practical solutions.

A. Scheduling Direction

The direction a scheduler traverses a DDD can have a large influence on the final schedule length. Thus far, forward traversals have been discussed. To schedule in the backwards direction, no changes to the algorithms thus far enumerated are required. The change occurs exclusively in the data structure representing the DDD. Here, all the sources become sinks and vice versa, all the predecessors edges become successor edges and vice versa, and all the operations are placed in instructions beginning at the end of the schedule. Upon completion of scheduling, the instruction list is reversed to reproduce the original semantic ordering of the source.

Allan and Mueller [4] note that one direction will succeed in producing a valid schedule in a given architecture much more often than the other. Bias is predicted on the presence of restricted timing within the DDD. Reasons for this bias include:

1. Presence of restricted branch delays. If branches in an architecture have (n, m) (or more likely (n, n)) timing to the end of the DDD, there exist few (or one) instructions in which they may be placed. Reverse traversal will tend to place this type of operation in the correct instruction early in the scheduling process, increasing the chance for a valid schedule. Increasing the branch node's scheduling priority is another way of achieving this end.
2. Presence of restricted machine pipe stages. As above with branch delays, (n, m) pipe stages can cause failures when a pipe operation is not instantiated at the proper time. Traversal direction is dependent on whether

(a) the inputs of the pipe are latched,
(b) the outputs of the pipe are latched,
(c) both are, or
(d) neither are.

3. Use of transient condition code registers within the DDD.

It is also possible for the direction to have an effect on the length of the final schedule without considering the impact of restricted edges. The reason is simple: direction has an impact on the order nodes are chosen to be placed. This is because the formation of the data-ready sets differs between the two directions. It may be worthwhile to attempt both and choose the shorter. If one direction fails to produce a valid schedule, the other direction certainly should be tried.

B. Computing Absolute Timing

In the context of building a list scheduler, it is convenient to associate both relative times and absolute times with each DDD node. While relative times (decorated on the DDD edges) indicate the timing relationship between two DDD nodes, the absolute time of a node indicates the possible range of instructions into which that DDD node might be scheduled.

An operation is said to be *data ready* if all of its predecessor operations have been scheduled. Similarly, an operation is said to be *timing ready* if it is data ready and the minimum edge time, between each of the node's predecessors and the node, has elapsed.

Assigning an absolute time interval to each node in the DDD provides a measure of urgency

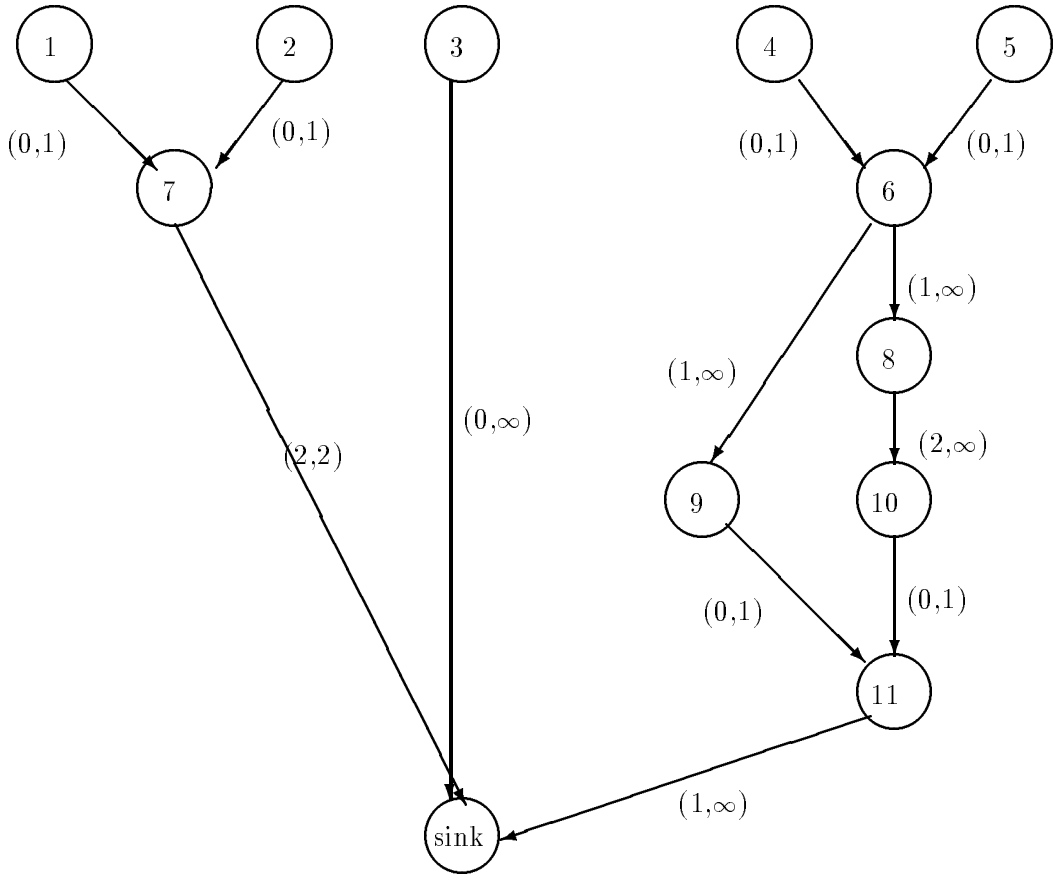


Fig. 9. Data Dependency DAG. edges are annotated with (min,max) pairs.

and yields a dynamic priority. The absolute time of an operation is a range of instruction indices in which the operation can be placed. For example in Figure 9, once O_6 is placed in instruction I_p, relative timing information indicates successor O_9 is ready to be placed in any of the next 1 through ∞ instructions. However, because of the timing along the path $O_6 \rightarrow O_8 \rightarrow O_{10} \rightarrow O_{11}$, placing O_9 before instruction I_{p+2}, will result in an eventual timing error.

Absolute timing utilizes the relative timing information and the location of previously scheduled nodes. If O_m has absolute timing (min_m, max_m), O_m may only be placed at an instruction with indices between min_m and max_m inclusive. Though min_m always has a finite value, max_m is often initially infinite, due to infinite relative times. As predecessors are placed, the absolute time interval shrinks as min_m increases and max_m decreases. When O_m is placed at I_k, (min_m, max_m) is changed to (k, k). If a node has absolute timing such that $min_m > max_m$, the algorithm fails as the node cannot be placed; no placement can satisfy the conflicting absolute

$set\_time(node)$

/* node has just had a change in absolute time.

set_time propagates those changes to surrounding nodes. */

{ for each successor, $succ$, of $node$

if the following statements alter *succ*'s absolute time

$min_{succ} = Max(min_{succ}, min_{node} + min_{node \rightarrow succ})$

$max_{succ} = Min(max_{succ}, max_{node} + max_{node \rightarrow succ})$

then call $set\_time(succ)$

for each predecessor, $pred$, of $node$

if the following statements alter *pred*'s absolute time

$min_{pred} = Max(min_{pred}, min_{node} - max_{pred \rightarrow node})$

$max_{pred} = Min(max_{pred}, max_{node} - min_{pred \rightarrow node})$

then call $set\_time(pred)$

}

Fig. 10. Absolute Timing Algorithm. Min and Max refer to functions returning the minimum and maximum of the arguments, respectively.

timing requirements.

The recursive algorithm that computes absolute time is shown in Figure 10. Given that there is an edge from a to b, min_a refers to the minimum absolute time of the node and $min_{a \rightarrow b}$ refers to the relative minimum timing from a to b. If the algorithm initially computes conflicting absolute timing, the graph is unfeasible regardless of the method used to schedule it.

A new absolute timing pair for node m is computed having the largest possible range that satisfies the current absolute timing and the edge timing. Then, the old absolute timing and the new absolute timing are combined by using the intersection of the two ranges. The maximum of the minimum times and the minimum of the maximum times provides this overlap in range. If the ranges do not overlap, the new minimum will be greater than the new maximum. This indicates that both timings cannot be satisfied, and the algorithm fails. The absolute timing algorithm is first used to produce initial absolute timing. As each operation is placed in an instruction during

scheduling, the absolute timing algorithm propagates the effects of the placement by restricting timing assignments.

Note that relative times are assigned by the code generator, whereas the absolute times are assigned solely as an aid to scheduling. Relative times are determined from a small amount of contextual information; absolute times are assigned after examining the entire graph and are updated as operations are placed.

C. Check and Schedule

Su's Check and Schedule (CAS) [62] starts with the timing analysis in a given DDD, and modifies the DDD to avoid scheduling failures due to incorrect timings. Su calls this timing computation *extended timing.*

CAS continues with timing conflict checking and a resource conflict checking to examine the DDD, and will abandon the scheduling and provide checking information if the given DDD proves erroneous. There are two reasons for schedule failure: failure caused by unfortunate scheduling decisions, termed scheduling failure, and failure inherent in the DDD termed infeasibility.

As the last phase of the overall procedure, the CAS algorithm will determine the priorities of operations and schedule those with the highest priority after resource checking is conducted. An operation will be scheduled provided it passes this checking.

Extended timing is the inferred timing between every node in a DDD. The extended timing between node i and node j in a given DDD is defined to be the final strict timing when all the nodes that have direct or indirect timings with node i and node j are considered. Namely, if node j has more than one scheduling interval, the intersection of them will be taken as its extended timing, denoted by (ext_min, ext_max). Extended timing is important in the identification of infeasible DDD's as any pair of nodes in which ext_min $\leq$ ext_max implies the DDD is infeasible.

Figure 11 gives the CAS algorithm.

D. Foresighted Scheduling

In basic LS, the instructions of the schedule are packed in sequence, with the last operation placed in instruction I_i before the first operation is placed in instruction I_{i+1}. Operations are selected for placement in order of priority. LS under timing constraints differs from traditional

Algorithm Check and Schedule

Input:

DDD and resource vectors of each node.

Output:

The scheduled instruction sequence

Algorithm:

1. Calculate extended timings of all pairs of nodes in DDD and replace relative timings with extended timings.
2. Check timing conflict of all pairs of nodes. If any timing conflict, terminate CAS and output corresponding information.
3. Check resource conflict.
4. cc(current cycle) = 0.
5. WHILE (there exists any node unscheduled) DO
   - 5.1 Calculate priority for all data-ready nodes.
   - 5.2 Pick up node n that has highest priority.
   - 5.3 S = Foresight (n, Constrained, cc) to check whether all successor nodes can be scheduled.
   - 5.4 IF S
     - place n into I_{cc}

     ELSE
     - IF max(n) $>$ cc
       - place n into data-ready set, DRS(cc+1)

       ELSE
       - return failure

   end WHILE
6. cc = cc + 1
7. IF not all nodes scheduled
   - goto 5.

Fig. 11. Check and Schedule Algorithm

techniques in that the single data ready list is replaced by a timing ordered list of data-ready sets [4]. When an operation becomes data ready, it is placed on the DRS associated with its absolute minimum time. For example, when O_a is placed in instruction I_r and there is an edge $a \rightarrow b$, b is placed in the DRS Set_s if all predecessors of b have been scheduled and its absolute minimum time is s. When instruction I_s is formed, only operations from Set_s are considered. Any operations in Set_s that are not placed in I_s whose absolute maximum time exceeds s are added to Set_{s+1}. If there exists an operation in Set_s not placed in I_s that has an absolute max time of s, it cannot be moved to the next timing ordered set. Scheduling fails as the operation is not scheduled in the allowable interval. LS under timing constraints is also different in that scheduling can fail as resource constraints prohibit a node from executing within its timing constraints.

References [4] suggest several methods for computing priority to minimize the risk of failure. The smaller the range for a node, the higher the priority. However, the basic problem with traditional LS techniques is that priority usually provides a weak measure of the effects of decisions on the final schedule. Foresighted LS attempts to make better decisions by testing the immediate effects of a potential scheduling choice on the schedulability of remaining constrained (finite range) nodes.

Though this practice is expensive, the cost of foresighted scheduling can be reduced by the reuse of information. The constrained sets differ by a few operations due to the fact that once an operation enters the constrained set, it normally remains constrained until it is placed. The approach used by Wijaya and Allan is to make local modifications to the previous foresight schedule rather than to start over each time [71].

E. Lookahead

As noted before, the edges in a DDD only limit the ordering in the final schedule, not the order the schedule is created. So long as the partial order is preserved, the order of placing the nodes is irrelevant. The absolute timing algorithm specifies the range in the final schedule where an operation can be placed. Because the foresight routine examines instructions in this range for node placement, if foresight succeeds in finding a valid place for an operation, then that placement will be valid in the final schedule. An alternative view is that not only *can* a node be placed where foresight predicts, it *should* be placed there. A method termed *lookahead* [8]

was developed to place operations instead of just testing for the possibility of placement. The original motivation for lookahead was to increase both the speed and the chances of creating valid schedules for a stochastic scheduling method [7]; it was then noticed it could speed up generic LS as well.

Several minor changes to LS with lookahead need to be noted. First, the definition of data ready does not change; i.e., it is still those nodes in the graph that have no unscheduled predecessors. The computation of these nodes might be different. It is no longer enough to remove nodes from the DRS when they are placed by LS; one must also add and remove nodes in the DRS based on those lookahead places. Lookahead can remove any or all the nodes on the DRS; it can also make nodes further down in the graph data ready by placing all their predecessors. The scheduler must also ignore all the nodes that are placed by lookahead during later stages of the scheduling process. Both of these conditions are handled in the compiler by the addition of a flag in the nodes that state whether or not the node has been placed, either by LS or by lookahead. It is also important for lookahead to check nodes in a breadth-first manner so that no cycles develop during the procedure.

A decision must be made as to whether to pack only the nodes with $\Theta(op) = (a, a)$ (equivalently $\Delta(e) = (n, n)$), or additionally to pack the nodes with $\Theta(op) = (a, b)\{a, b \mid a < b < \infty\}$. In the first case, no choice exists as to when to pack the nodes, they must be placed in instruction $\iota + n$. The second case contains more flexibility and requires the analysis of a tradeoff. Having lookahead place them will result in a larger chance of generating a valid schedule, similar to the improvement that foresight has to plain LS. However, if lookahead does not immediately place the $\Theta(op) = (a, b)\{a, b \mid a < b < \infty\}$ nodes, the scheduler may be able to produce a more compact final sequence.

This tradeoff varies with the flexibility in the operation's schedule range in the current DDD, and in the architecture, making it difficult to analyze the tradeoff universally. For example, if the current DDD is wide (displaying a lot of parallelism), constrained operations might need to be placed immediately so that other parallel operations do not consume all needed resources in $\Theta(op)$. For machines with a large amount of available parallelism, final placement should probably be deferred, allowing the most flexibility for the scheduler. Placement decisions made between the time of finite constraint and final packing are less likely to have a deleterious effect

as there is more “room” in each instruction for operations in these types of architectures. A heuristic based on the range for any node could be tuned on a per-machine basis to control the amount of lookahead.

Note that any failure to place a restricted node with lookahead would result in a failure later in the scheduling process, thereby reducing the time spent scheduling an infeasible schedule. Lookahead also schedules nodes without having to topologically sort them. By doing so it reduces the number of nodes LS must deal with and thereby increases the speed of scheduling [9].

Also note that naîve lookahead places nodes in the final schedule non-heuristically. That is, there is no order in examining the constrained nodes based on node weights built into lookahead. While this expedites the process, lookahead could be extended to deal directly with differing priorities in the constrained node set so that the final schedule length is optimized.

It is important to understand that using lookahead with LS does not guarantee that no timing failures will occur; it only lessons the chances of encountering such failures. There still is the possibility that valid DDDs exist that cannot be scheduled due to poor choices made by the node priority heuristics. This is an inherent problem when only searching a small subspace of the possible solutions.

F. Register Assignment and Instruction Scheduling

The optimization of keeping program values in registers as much as possible consists of two (potentially) distinct problems: *register allocation* which determines those program values that will be placed into a register resource, and *register assignment* which maps those program values to be allocated to a register to the available machine register set.

While Sethi has shown optimal register assignment computationally intractable [58], good heuristics exist that produce near-optimal results in reasonable time. One popular method, developed by Chaitan, [17], [18] initially assumes an infinite number of available “symbolic” registers, allocates each scalar value to a distinct symbolic register, and later maps the symbolic registers to the finite target architecture register set using a graph coloring heuristic. We assume a graph-coloring register assignment scheme because of its proven effectiveness and because its basis in well-founded mathematical principles allows easy retargetability from one architecture to another. We use Briggs’ [15] methods that enhance the effectiveness of this method of register

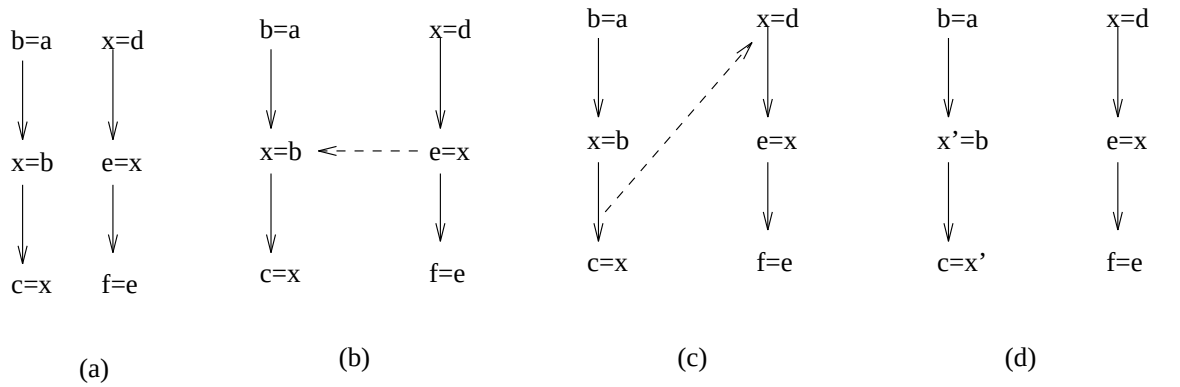


Fig. 12. (a) DDD without anti-dependency edges (b) One way of inserting anti-dependency edges (c) Another way of inserting anti-dependency edges (d) Renaming to eliminating anti-dependency edges

assignment.

While determination of data dependencies follows traditional data flow analysis, anti-dependencies are not as straightforward. An anti-dependency exists when a register is reused. Hence, anti-dependency edges are tied to register assignment. Suppose x is defined and used twice in a section of code as in Figure 12a, in which the code has been augmented with data dependency edges. We can force one live track to complete before the other by inserting anti-dependency edges as shown in Figure 12b or 12c [3]. Notice that the length of the longest dependence chain is four in Figure 12b and is six in Figure 12c.[7] If renaming is allowed, as in Figure 12d, the longest dependence chain is three. Actually the problem is much more pervasive. Since every variable must be mapped to a physical register (assuming a load/store architecture), there are anti-dependencies between uses of the same register.

Register assignment can be done before or after scheduling, the decision of which to do first being a common phase ordering problem. When scheduling is done first, anti-dependencies are ignored in this phase as infinite registers are assumed. The succeeding register assignment phase typically has many more conflict edges to consider because of the greater overlap of live tracks. Hence, there is a higher probability that there will be insufficient registers. When spill code is introduced, it is not efficiently integrated into the surrounding parallel code without a subsequent scheduling pass. The likelihood of needing spill code, requiring two scheduling passes, convince some that register assignment should be done before scheduling. The problem

[7] If we assume that uses of a register precede stores within a machine instruction, the nodes connected by an anti-dependence can be performed simultaneously. Thus, the dependence chains are three and five, respectively.

is that many register assignment algorithms attempt to use the minimal number of registers. In a sequential environment this is advantageous, but in a parallel environment, the fewer the registers the greater the anti-dependency edges that limit parallelism. There is considerable research in combining these phases [36], [23], [35], [15], [39], [52], [55], [51], [11].

Given that neither early nor late register assignment provides a good solution to the phase ordering of register assignment and instruction scheduling, several reseachers have proposed phase-coupling techinques.

Bradlee [13] discusses a method of mixing the two phases in which initial passes of an instruction scheduler get estimates of the schedule cost given a certain number of registers. The scheduler is run locally with a very limited number of registers and then again with the maximum number of registers in the machine. These values are then used to allocate a certain number of registers for each basic block.

As a method to merge register assignment and instruction scheduling, Freudenberger [31] combines register allocation and assignment within trace scheduling. This combination is motivated in part by the assumption that trace scheduling, being an aggressive global scheduler, would be crippled by early register assignment. In addition, Freudenberger assumes that the register pressure added by an unchecked trace scheduler would make it virtually impossible for late register assignment to avoid spilling. Thus, he treats registers as resources and his scheduler will only “assign” a value to a register during scheduling when a free register is available.

Pinter’s work [51] realizes that early register assignment is too conservative due to adding extraneous anti-dependences. To avoid this, her algorithm creates a parallelizable interference graph. To generate such a graph, the schedule graph, or DDD, is analyzed and all true dependences are found. Any machine dependences (resource conflicts) are then added to this graph. The graph’s complement is then constructed and referred to as the false dependence graph. The union of the register interference graph and a false dependence graph is created. This new graph is the parallel interference graph, and represents all the true interference that exists between values. It can be colored to give a register assignment that does not retard any available ILP. Since it is likely that this graph is not colorable with the available registers, Pinter carefully chooses which edges to remove in order to avoid creating anti-dependences that might retard the final schedule.

Brasier [14] describes *Combining Register Assignment Interference Graphs* (CRAIG), a framework that combines early register assignment, and late register assignment with (possibly) an additional pass to maximize parallelism while reducing spill costs. CRAIG mediates the "tug-of-war" between register assignment and instruction scheduling by providing a mechanism to decrease anti-dependencies (thus increasing scheduling freedom) even when spill code needs to be added.

CRAIG incorporates this "mediation" as a schedule cost considering both schedule efficiency and register pressure. Schedule cost is a heuristic designed to meet the goals of the specific code generator, and will reflect the level of register spill code that will be "tolerated" in late register assignment. One possible schedule cost heuristic is that the cost will be considered too high if any register spills are necessary in late register assignment. Less restrictive heuristics would allow spilling if the expected schedule benefit is high enough. CRAIG initially schedules before register assignment is performed. Whenever this initial schedule cost is deemed too high, CRAIG goes back to the original linear code and attempts *early* register assignment. The intuition is that the original code will have a less busy interference graph and will therefore have a lower cost due to register pressure. If this schedule cost is still too high, CRAIG accepts this schedule based on the assumption that it is the best that we can do under the circumstances. If, however, the schedule cost is not too high it is likely that anti-dependences have been added, and thus, the schedule can be improved. CRAIG will attempt to reclaim some of this lost efficiency by removing as many of these anti-dependences as possible, up to the point where the schedule cost is too high.

By adding edges found exclusively in the late register assignment interference graph, CRAIG creates interference between those values the scheduler forced to be in different registers. If they are mapped to the same register in the early register assignment interference graph, then CRAIG has identified and removed an anti-dependence that potentially inhibits a more efficient schedule.

V. ILP Architectures

Given the flexibility provided by our DDD model in which labels are annotated with *min,max* times, we can build an instruction scheduler for a broad class of ILP architectures. For purposes of our discussion we shall consider four categories of ILP architecture; *Long-Instruction-Word*

(LIW), *Superscalar*, *Very-Long-Instruction-Word* (VLIW), and *Pipelined*. Of course these categories are far from distinct. Many ILP architectures include pipelines. Still, pipelines add some additional complexity to our machine model, so we choose to treat them separately.

In the discussions to follow, we show how the computations necessary to compute the "function" (whatever it might be) defined by the code fragment of Figure 1 can be overlapped in the different ILP computers.

Long-Instruction-Word (LIW) computers increase ILP by providing a wider instruction word, allowing more operations to be specified in each instruction. LIW architectures typically include pipelined functional units, instruction parallelism to a relatively large degree, and complex timing. LIW machines are often built not as general-purpose computing engines, but rather to maximize performance for some special time-critical application. Such application-specific LIW computers are popular for a wide range of applications including signal processing, image processing, graphics and flight simulation. Using available functional units (such as adders, multipliers, and address generators), and multiple memory resources, a designer can build an LIW architecture tuned for a specific application, allowing increased computer power with little resource waste. Examples of LIW computers include Pixar's Chap [43], the ESIG-1000 from Evans and Sutherland [28], and Intel's i860 [37]. For our example, consider a simple LIW computer that includes two identical ALUs, each of which can do any of a multiply, an add, or a subtract in one machine cycle. Each ALU would be represented by a separate instruction field and so since we have four operations to perform and we can overlap two in any instruction, we might be able to execute the above fragment in two machine cycles. IS would look at the vertical code of Figure 1 and recognize that flow dependences exist between operation 1) and 2) and between 3) and 4). Operations 1) and 2) have a flow dependence because 1) sets r3's value to that read by 2). Similarly 3) sets r8's value to that read by 4), leading to the dependence between 3) and 4). Thus, IS cannot overlap either 1) and 2) or 3) and 4). Given these constraints it is easy to schedule the four necessary operations into two LIW instructions. Our hypothetical LIW code for this example is shown in Figure 13 where the & character is meant to specify that the two operations are to be included in a single instruction.

In contrast to LIW architectures that explicitly specify instruction-level parallelism in the instruction word, superscalar machines allow exploitation of instruction-level parallelism by al-

```
add     r4, r2, r3      &       add     r7, r1, r8
mult    r3, r5, r6      &       sub     r8, r4, r2
```

Fig. 13. LIW Assembly Language for Figure 1

lowing multiple operations to be executed during a single machine cycle. Unlike LIW machines, a superscalar computer includes hardware to check for flow dependences between adjacent instructions. A superscalar of degree n can issue n operations in a single cycle. To find n independent operations, the superscalar architecture maintains a *window* of the next w instructions of the sequential instruction stream. If, during execution, n independent operations can be found within the next w unexecuted operations, they are issued in the next cycle. If, however, data dependence limits the operations ready to execute to be less than n stalls and dead time result while operations are forced to wait for the results of prior operations before they can be executed. For simple superscalar architectures n and w might be equal. More sophisticated models allow $w > n$, but allowing out-of-order execution within the instruction stream does add complexity to the architecture. Current superscalar architectures typically set $w = n$.

Consider a simple superscalar in which both n and w are 2. If the instruction sequence presented by the compiler is the original one shown in Figure 1, our mythical superscalar architecture would require three operations to execute the fragment. Operation 1) would be executed in the first machine cycle. But since a data conflict exists between 1) and 2), operation 2) would have to wait for the first machine cycle to complete. Then in the second machine cycle 2) and 3) could both be executed, as no dependence exists between them. In the third machine cycle, operation 4) could be executed.

If IS rearranges the initial code, we can easily come up with an order that allows the superscalar architecture to execute the fragment in two machine cycles. Consider reordering the instruction stream to present the operations in Figure 14. Operations 1) and 3) are independent and, thus, the superscalar machine could execute them in parallel in the first machine cycle. Similarly, the second machine cycle could perform operations 2) and 4). Notice that, in some sense, we have made equivalent use of the same level of parallel hardware in the LIW and superscalar examples, and both cases require instruction scheduling to make best use of the available parallelism.

Very-Long-Instruction-Word (VLIW) computers differ from LIW in kind as well as size.

```
1)      add     r4, r2, r3
3)      add     r7, r1, r8
2)      mult    r3, r5, r6
4)      sub     r8, r4, r2
```

Fig. 14. Superscalar Assembly Language

Their long instruction word (on the order of 1000-2000 bits for current machines) is a result of replicated simple processors whose instruction words are concatenated into a single instruction word. Thus, VLIW architectures not only generally provide more ILP, but the functional units are much more homogeneous than is generally the case for LIW computers. The Multiflow TRACE series of computers [20] are VLIW computers, as are computers being built at IBM, described in [24]. Like LIW machines, the VLIW model of computation assumes that the compiler is solely responsible for determining which operations can be performed in parallel. In contrast, the superscalar model assumes co-operation between the compiler and the hardware in which each has responsibilities for extracting parallelism. For our example code fragment, the VLIW "code" would be equivalent to the LIW reordering shown above. In fact, while VLIW and LIW computers can be quite different, for the purposes of our discussion of instruction scheduling, we shall treat them identically.

Pipelines are not so much a defining characteristic of one class of computers as they are a hardware technique used to support ILP. As such they are found in all ILP architectures. In pipelined execution, operations are overlapped by allowing several different operations to be at different stages of computation during the same cycle.

A pipelined machine of degree m requires m cycles to complete an operation, but the cycle time can be much shorter ($1/m$) than that of non-pipelined architectures. The parallelism is realized because, in any cycle, up to m different, independent operations can be in executing in different stages of the pipeline(s) and a final result is produced. As with LIW and superscalar architectures, IS is necessary to generate efficient code for pipelined architectures, but, as with superscalar computers, the parallelism is implicit rather than explicitly in the assembly language as occurs for LIW architectures. Consider a pipelined machine of order 3 that requires 3 machine cycles to complete any ALU operation but can start a new ALU operation each machine cycle,

as long as the new operation is not dependent upon data that is as yet unavailable from the pipe. Without operation reordering, our fragment of code for the pipelined machine would be:

```
1)      add     r4, r2, r3
        nop
        nop
2)      mult    r3, r5, r6
3)      add     r7, r1, r8
        nop
        nop
4)      sub     r8, r4, r2
```

The *nops* are necessary because operation 2) cannot be started until operation 1) completes (three machine cycles after it starts), and similarly operation 4) must wait on operation 3). By reordering the instructions, however, we can obtain the following:

```
1)      add     r4, r2, r3
3)      add     r7, r1, r8
        nop
2)      mult    r3, r5, r6
4)      sub     r8, r4, r2
```

This new arrangement saves us 3 cycles for this trivial example. So, again IS scheduling is useful in generating excellent code. Of course, pipelining has been a common feature of architectures for a long time, and is included in many modern high-performance computers. When considering real architectures a compiler needs to be able to combine scheduling of pipelines with whatever other features provide for ILP.

Now let us consider how the DDD edge timing model allows us to build a scheduler for each of these types of architectures. The basic plan is to use the different data dependence timings for a computer's memory resources to determine the edge timings, as is demonstrated in Figure 2. However, we will need to add a few special edge timings to enable the DDDs to accurately reflect some ILP features. Most of these additions will deal with pipeline concerns.

First, we need to discuss how the DDD timing model applies to general LIW, VLIW, and

superscalar architectures. Actually some may question whether static instruction scheduling is even required for superscalar processors. One might assume that for a superscalar architecture, the compiler need not do instruction scheduling at all, but rather leave all of the scheduling to the hardware. This might be a valid argument if a superscalar architecture could not only reorder the instruction stream (something few commercial superscalar machines can do now) but could consider the entire (remaining) instruction stream while performing such reordering. Since this is not feasible, it is well accepted that for superscalar architectures, an instruction-scheduling compiler is required for efficient code. In fact, the best way for a compiler to schedule operations for a superscalar computer is to assume that the machine is, in effect, an LIW architecture and to simulate the parallelism that can be found at run-time by construction of a wide instruction that represents the multiple consecutive simple operations that will be issued at run time.

The main difference among the LIW, superscalar, and VLIW models, as far as building DDDs for IS is concerned deals with whether the hardware merely accepts the schedule provided by the compiler (LIW, VLIW models) or whether the hardware must determine which operations can simultaneously begin execution (superscalar.)

One difference between the LIW (and VLIW) model of computation and the superscalar model is the potential need for NOP instructions. In LIW machines, the computer starts one wide instruction after another. If the machine must wait for a previously-started operation to finish before continuing, a NOP must be included. In the superscalar model, the machine will automatically wait until the operands of the next operation are ready before continuing so the NOPs are not necessary in the assembly code generated. While this difference may be significant in architectural considerations it has little impact upon the instruction scheduler, as in order to accurately determine the cycles required for a schedule, the scheduler can assume NOPs are included even in the superscalar model. Thus, NOPs were included in the pipelined example above even though nothing definite was said about whether the underlying computation model was LIW or superscalar.

Another difference between LIW and superscalar computing does have an effect on the scheduler, however. When the compiler alone determines the order operations are issued (LIW model), the instruction scheduler has more freedom in reordering operations. When scheduling for a superscalar, an additional restriction must be placed on the scheduler to ensure semantically

correct code. This restriction deals with multi-cycle operation latencies and typically arises with pipelines. Consider the code fragment for an architecture where a multiply requires three cycles to complete, while an add requires a single cycle:

```
1)      add     r2, r4, r5
2)      mult    r3, r1, r4
3)      add     r5, r4, r6
```

Our instruction scheduler would know that the multiply is not going to change the value of r4 until 3 cycles after the start of the multiply instruction. Thus, we would like to reverse the order of the first two instructions to give the following code:

```
2)      mult    r3, r1, r4
1)      add     r2, r4, r5
        nop
3)      add     r5, r4, r6
```

This is a reasonable thing to do in the LIW model, as operation 1) would use the original value of r4 (the value before the multiply) because r4 would not have been changed yet. Notice how a superscalar computer would execute this sequence, however. It would recognize the write-after-read hazard (anti-dependence) between operations 2) and 1) in the scheduled sequence and thus delay two cycles until the new value of r4 was complete. This would lead not only to an unnecessary delay, but to incorrect code as well. Therefore, when modeling a superscalar architecture, additional DDD edges must be inserted to ensure that this type of reordering does not occur. Figure 15 shows both LIW and superscalar model DDDs for this example code fragment. Note that the edge between Nodes 1 and 2 in the superscalar model will somewhat limit the potential parallelism, but it will ensure correct program semantics for a superscalar architecture. On the other hand, the LIW model requires an additional node to represent that actual definition of r4 due to the multiply operation of Node 1, but includes no edge between Nodes 1 and 2.

Pipelines add additional complexity for our timing model that is worthy of mention. Namely, some architectures include *implicitly* advanced n-stage pipelines where once a computation starts the result is written n cycles later. Alternatively, some ILP architectures support *explicitly*

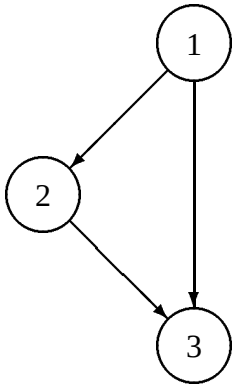


Fig. 15. Superscalar DDD

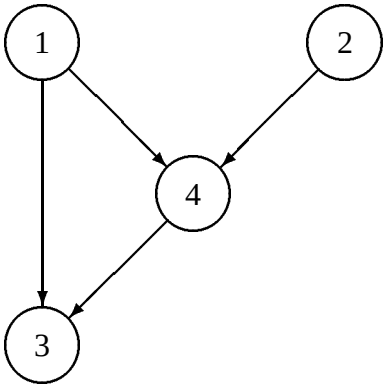


Fig. 16. LIW DDD

advanced pipelines where n explicit instructions are necessary to "push" a computation through a n-stage pipeline. The Intel i860 is such an architecture. Actually both types of pipelines are easily modeled, the implicit pipeline with a single DDD node and the explicit with n individual nodes with appropriate edges between them.

VI. Using a Local Scheduler in Global Scheduling

This paper's main thrust focuses on building a retargetable local instruction scheduler. We need to at least consider some global scheduling methods, however, since many global methods rely in part on a local scheduler. This use of a local scheduler as part of global scheduling techniques

adds additional complexity to the local scheduler. We consider three types of global scheduling that rely on local schedulers, namely 1) *software pipelining* which reorders operations within a loop, overlapping execution of several iteration's operations within a single loop execution, 2) global scheduling using *meta-blocks*, and 3) global scheduling techniques based upon list scheduling which attempt to overcome the difficulties of small basic blocks in branch-intensive code. Considering this additional complexity, it is best to build an extremely flexible local scheduler so that it can be properly adapted to each of its many different uses.

A. Software Pipelining

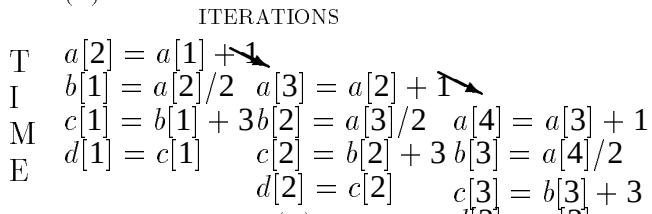
Scheduling code in a loop is complicated by the fact that, to achieve best results, operations from different iterations must be scheduled together. This overlapping of loop iterations into a single loop schedule is called *software pipelining*. The dependence information for operations in a loop is complicated by the fact that operations have dependences with operations from various iterations. If we let one node represent an operation from *all* iterations, the dependence must not only specify *min* and *max* times, but also specify *dif*, which is the difference in the iterations from which the operations come. To characterize the dependence, a dependence edge, $a \rightarrow b$, is annotated with a (*dif, min, max*) dependence tuple. The *dif* value indicates the minimum and maximum number of iterations the dependence spans, termed the *iteration difference*. If we use the convention that a^m is the copy of a from iteration m, then ($a \rightarrow b$, *dif*, *min*, *max*) indicates there is a dependence between a^m and $b^{m+dif}, \forall m$. The minimum and maximum delays intuitively represent the number of instructions that an operation takes to complete. More precisely, for a given value of *min*, if a^m is placed in instruction t (I_t) then b^{m+dif} can be placed no earlier than I_{t+min} and no later than I_{t+max}.

Dependence edges are categorized as follows. A *loop independent* edge represents a must follow relationship among operations of the *same* iteration; *dif* is zero. A *loop carried* edge shows a relationship between the operations of different iterations. Loop carried dependences may turn traditional DDDs into cyclic graphs [74].

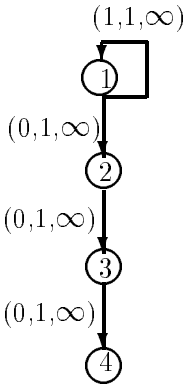
The idea behind software pipelining is that the body of a loop can be reformed so the new body of the loop represents operations from multiple iterations. When one iteration of the original loop can start before previous iterations finish executing, more parallelism is potentially unveiled. Numerous systems completely unroll the body of the loop before scheduling to take advantage

```
for (i=1;i<=n;i++)
 O1: a[i + 1] = a[i] + 1
 O2: b[i] = a[i + 1] / 2
 O3: c[i] = b[i] + 3
 O4: d[i]=c[i]
```

(a)



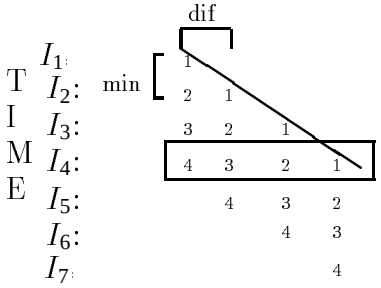
(b)



(c)

```
a[2] = a[1]+1
b[1] = a[2]/2
c[1]=b[1]/3
a[3]=a[2]+1
b[2]=a[3]/2
a[4]=a[3]+1
for (i=1;i<=n-3;i++)
   d[i]=c[i]
   c[i+1]=b[i+1]+3
   b[i+2]=a[i+3]/2
   a[i+4]=a[i+3]+1
(postlude)
```

(d)



(e)

of parallelism between iterations. Software pipelining achieves an effect similar to unlimited loop unrolling.

Since adjacent iterations are overlapped in time, dependences between various operations must be identified. To see the effect of the dependences in Figure VI-Aa, it is often helpful to unroll a few iterations as in Figure VI-Ab. Figure VI-Ac shows the DDD of the loop body. In this example, all dependences are true dependences. The edges $1 \rightarrow 2$, $2 \rightarrow 3$, and $3 \rightarrow 4$ are loop independent while the edge $1 \rightarrow 1$ is a loop carried dependence. If each iteration of the loop in Figure VI-Aa is scheduled without overlap, four instructions are required for each iteration as no two operations can be done in parallel (due to the dependences). However, if we consider operations from several iterations, there is a dramatic improvement.

A.1 Scheduling

Software pipelining loop code takes two forms. The first type of algorithm, termed *kernel-recognition*, schedules operations from various iterations and checks to see when a repeating kernel has been found. List scheduling (based on a topological ordering using the loop independent arcs) is used to schedule operations from various iterations. Often the priority is based on the elapsed time since last execution; an operation from an earlier iteration is given preference over an operation from a later iteration. Other times data ready operations are prohibited from executing even if no other operation can use the resource. This takes the form of restricting the span of iterations represented in the instruction or simply delaying execution of nodes in a non-critical dependence cycle. This encourages the formation of a cyclic pattern as one operation is not allowed to execute at a faster rate than others. Algorithms of this type include [1], [48], [26], [25], [6], [53], [67]. Since the scheduler is quite restricted, a flexible local scheduler may not be too helpful.

However, the second method of software pipelining, termed *modulo* scheduling is greatly benefited by a flexible local scheduler. Modulo determines a target II and then schedules one copy of the loop body so that it will form a legal schedule if successive copies of the loop body (representing successive iterations) are offset by II instructions. If an operation cannot be placed, the target II is incremented and the process begins again. Algorithms of this type include [41], [40], [73], [21], [36], [44], [56], [55], [54], [57], [64], [69]. A flexible local scheduler is extremely important in being able to create a schedule that is legal in a cyclic sense. Scheduling

mistakes are compounded by either having to repeat the whole process with a larger II or having to unschedule nodes that cause problems. Either situation is expensive.

B. Scheduling Meta-Blocks

A considerable body of research has shown that exploitation of significant amount of ILP requires global scheduling techniques [65], [47], [68]. Several global scheduling techniques make use of a local scheduler, however, to reorder code for meta-blocks. A meta-block is a group of blocks that a global scheduler treats as a single block. Trace scheduling [29], [27] is an example of such a global scheduler. It attempts to optimize the most frequently executed paths of a control flow graph (CFG), possible at the expense of the less frequently-executed paths. Blocks that are included in the frequently-executed paths are called *on-trace* and those in the less frequently-traveled paths are called *off-trace.* The basic idea is to use a local scheduling algorithm to move code between blocks in an on-trace path to reduce the number of instructions within that trace. Other global scheduling algorithms that rely on a local scheduler to re-order code in meta-blocks include Bernstein's Global Scheduling [10], Hwu's Sentinel Scheduling [45], and Sweany's Dominator-Path Scheduling [63]. Such use of a local scheduler on DDDs that contain control flow complicates the local scheduling algorithm.

The problem is that local schedulers, including list scheduling, are designed based upon the premise that each operation within the DDD will be executed the same number of times, namely once for each time the basic block (from which the DDD came) is executed. When scheduling meta-blocks, this assumption is violated. The local instruction scheduler may move operations throughout the combined meta-block. A traditional local scheduler does so without regard for the potential execution frequency of the locale in which an operation is to be finally placed. Thus, the scheduler may move an operation from a block that has a relatively low execution frequency to one that has a higher frequency. This could result in a schedule that takes more instruction cycles to execute than that generated by an instruction scheduler which does not allow for inter-block code movement.

In an attempt to overcome this deficiency of traditional list scheduling, Bourke [12] has defined Frequency-Based List Scheduling (FBLS) which considers differing execution frequencies within meta-blocks while scheduling. FBLS amends the basic list scheduling algorithm by revising only the operation placement policy in an attempt to reduce the instruction cycles required

to execute a schedule for a meta-block. To do this, it modifies list scheduling so that once a DDD node has been chosen for scheduling, a two phase approach for operator placement is used. The first phase attempts to schedule an operation only within an already existing instruction[8] with which the operation can execute in parallel. If the scheduler cannot locate such an existing instruction, a second phase is used. The second phase creates a new instruction in the "best" portion of the meta-block in which all the dependences of the selected operation are met. The "best" portion is defined as the area in the meta-block that has the lowest execution frequency that overlaps with the possible range of the operation to be placed. This scheme presupposes a mechanism to partition the DDD for a meta-block into sections of differing execution frequencies and the further ability to place an instruction in whichever such partition is desired.

It should be noted that one consequence of FBLS is some loss of scheduling flexibility *within* the meta-block itself. The main rationale for most global schedulers is that a larger scheduling context will lead to better schedules. That is why many methods rely on meta-blocks. By partitioning meta-blocks, we lose some (but not all) of that flexibility. The tradeoffs between inefficiency due to ignoring frequency information and inefficiency due to loss of scheduling flexibility have not yet been thoroughly investigated.

C. Branch Intensive Code

Since many application programs are branch intensive and basic blocks are small, there is a need to extend the scheduling beyond basic blocks to achieve better performance [46]. Parallelizing code involving branches is the goal of two widely known techniques termed *predicated* execution and *speculative* execution.

Predicated execution effectively removes conditional jumps so the code can be scheduled like a basic block. To effect predicated execution, the results of a branch condition are stored in a predicate register, P. An instruction in the true branch such as *a = b + c* is replaced by a predicated instruction *a = b + c if P_t* that specifies that the operation will actually be completed only if the predicate P is true. An instruction on the false branch such as *d=e*f* would be replaced by *d=e*f if P_f*. Thus, if the predicate does not have the required truth value, the operation will never change the state of the machine. The resources to execute the

[8]This existing instruction must exist in the scheduling range for the selected operation.

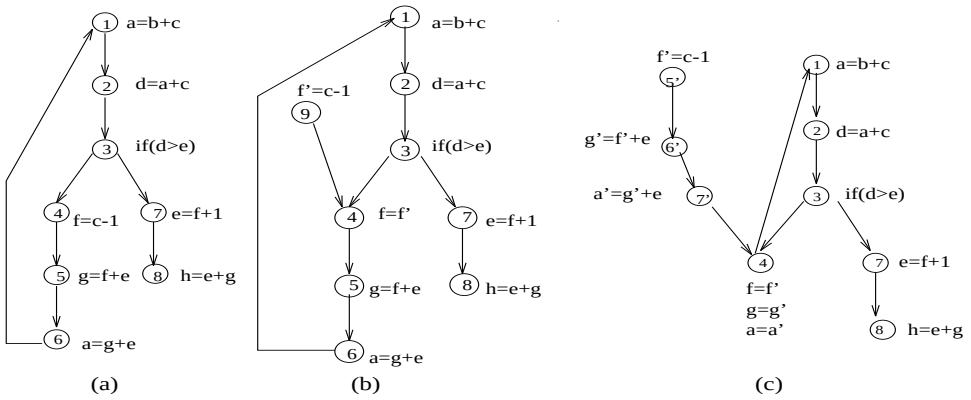


Fig. 17. (a) Original data dependence graph.(b) After renaming $f = c - 1$ (c) Dependence graph after renaming and forward substituting $g = f + e$ and $a = g + e$.

operation may or may not be consumed, depending on the architecture. Mahlke et al. propose a parallel architecture that supports predicated execution. The predicated operations may execute concurrently with the statement that computes the predicate [44], [57]. This is made possible because (for this architecture) the operation is executed regardless of the value of the predicate, but is only allowed to change the result value if the predicate value is satisfied. Since the write-back is performed in a later part of the execution cycle than the computation of the value, this is feasible. By this process, code containing branches is converted into straight-line, branch-free code, making scheduling much simpler. Warter et al. [70] propose a reverse *if*-conversion process to convert the predicated representation back to the control flow graph representation in order to facilitate architectures without predicated execution support. A flexible scheduler that keeps predicated code close together (when dispersing it creates no advantage) reduces live tracks and also simplifies reverse if-conversion.

Speculative execution refers to the execution of operations *before* it is known that they will be useful. It is similar to predicated execution except instead of allowing the operation to be performed at the same time step as the predicate, the operation can be performed many time steps before its usefulness is known. Renaming and forward substitution are used by to move operations past predicates [26], [49]. Renaming is a technique that replaces a statement such as $x = g(y)$ with the pair of statements $x' = g(y)$ and $x = x'$. Since the new variable (x') is used only in the copy operation, the assignment is free to move out of the predicate. Figure 17(b) shows how renaming is used to move the operation $f = c - 1$ past the predicate. A new variable

f' is created and is assigned the result of the expression $c - 1$. In order to preserve the original semantics of the program, the value assigned to f' is copied back into f. Since f' is used only by the copy operation, it can move past the predicate. Forward substitution refers to replacing the use of a variable with the expression that computes the variable. Figure 17(c) shows the resultant graph formed after renaming and forward substituting $g = f + e$ and $a = g + e$.

Predicated and speculative execution allow operations between various basic blocks to be overlapped, and hence permit decreased schedule lengths. However, excessive speculative or predicated operations may result in the generation of inefficient code. Speculated operations, if not controlled, may slow down non-speculative code. Also, predicated execution always executes operations from both the branches irrespective of which branch is finally taken. This may decrease the performance as the union of the branch requirements much be accommodated. Deciding how to combine the advantages of each technique is the topic of [59].

VII. Conclusions

Given the increasing importance that instruction-level parallelism plays in modern architectures, the need to perform instruction scheduling in compilers for ILP architectures, and the decreasing time-to-market for new computers, it is imperative that machine-independent instruction scheduling software be available. This paper describes techniques useful for building such an instruction scheduler that is easily retargeted to a broad class of ILP architectures by use of a small amount of machine-dependent resource information.

The foundation of the scheduling techniques described here is the DDD arc timing model that allows representation of a wide range of architectural features. Given such a DDD structure, list scheduling is relatively easy to implement. Section V showed how our model can be used to generate efficient code for a variety of ILP architectural types. Of course, list scheduling is a heuristic approach to scheduling DDDs as the task of finding optimal schedules is NP-complete. Like any heuristic-driven approach, the choice of ordering heuristics is paramount. We discuss 20 heuristics used in an existing retargetable scheduler.

In addition to choosing heuristics carefully, there are several additional practical considerations to be addressed when building an instruction scheduler. The fact that scheduling direction (top-down or bottom-up) can make a significant difference in scheduling efficiency suggests that performing both and saving the best schedule is a viable option. Given that scheduling can fail

for a DDD, several techniques for reducing failures and improving scheduling time should be considered. Check and schedule, foresighted scheduling and lookahead scheduling are three such techniques described here.

Ordering the compiler phases of register assignment and instruction scheduling presents further concerns. No matter whether register assignment or instruction scheduling is performed first, the other suffers. We discuss this problem and suggest practical solutions that either combine register assignment and instruction scheduling or lessen the deleterious effects one has on the other.

Finally, when building a scheduler, one must consider whether that local scheduler will be used as part of a global scheduling technique. If so, the designer should consider frequency-based list scheduling for best results.

In short, this paper describes a list scheduling framework and several important practical details that, taken together, will allow implementation of an efficient local instruction scheduler.

REFERENCES

[1] AIKEN, A. *Compaction-Based Parallelization*. PhD thesis, Department of Computer Science, Cornell University, Ithaca, NY, 1988.

[2] ALLAN, V. *A Critical Analysis of the Global Optimization Problem for Horizontal Microcode*. PhD thesis, Computer Science Department, Colorado State University, Fort Collins, Colorado, 1986.

[3] ALLAN, V. Data Dependency Graph Bracing. In *Proceedings of the 21st Microprogramming Workshop (MICRO-21)* (San Diego, CA, December 1988), pp. 91–93.

[4] ALLAN, V., AND MUELLER, R. Compaction with General Synchronous Timing. *IEEE Transactions on Software Engineering 14*, 5 (May 1988), 595–599.

[5] ALLAN, V., AND MUELLER, R. Microcode compaction with general synchronous timing. *IEEE Transactions on Software Engineering (Special Section on Microprogramming) 14*, 5 (May 1988), 595–599.

[6] ALLAN, V., RAJAGOPALAN, M., AND LEE, R. Software Pipelining: Petri Net Pacemaker. In *Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism* (Orlando, FL, January 20-22 1993).

[7] BEATY, S. *Instruction Scheduling Using Genetic Algorithms*. PhD thesis, Mechanical Engineering Department, Colorado State University, Fort Collins, Colorado, 1991.

[8] BEATY, S. J. Lookahead scheduling. In *Proceedings of the 25th Annual International Symposium on Microarchitecture (Micro-25)* (Portland, Oregon, December 1992), pp. 256–259.

[9] BEATY, S. J. List scheduling: Alone, with foresight, and with lookahead. In *Conference on Massively Parallel Computing Systems (MPCS): the Challenges of General-Purpose and Special-Purpose Computing* (May 1994).

[10] BERNSTEIN, D., AND RODEH, M. Global instruction scheduling for superscalar machines. In *Conference on Programming Language Design and Implementation* (Toronto, June 1991), SIGPLAN '91, pp. 241–255.

[11] BERSON, D., GUPTA, R., AND SOFFA, M. Resource Spackling: A Framework for Integrating Register Allocation in Local and Global Schedulers. In *PACT 94* (Montreal Canada, August 23-26 1994), p. to appear.

[12] BOURKE, III, M. J. Frequency-based list scheduling: Extending list scheduling to consider execution frequency. Master's thesis, Computer Science Department, Michigan Technological University, Houghton, MI, 1993.

[13] BRADLEE, D., EGGERS, S., AND HENRY, R. Integrating register allocation and instruction scheduling for RISCs. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Santa Clara, California, April 1991).

[14] BRASIER, T. S. Frigg: A new approach to combining register assignment and instruction scheduling. Master's thesis, Computer Science Department, Michigan Technological University, Houghton, MI, 1994.

[15] BRIGGS, P., COOPER, K., KENNEDY, K., AND TORCZON, L. Coloring heuristics for register allocation. In *Conference on Programming Language Design and Implementation, SIGPLAN* (Portland Oregon, June 1989), pp. 275–284.

[16] BRIGHTWELL, G., AND WINKLER, P. Counting linear extensions is #p-complete. DIMACS Technical Report 90-49, Bellcore, 445 South Street, Morristown, New Jersey, 07960, July 1990.

[17] CHAITIN, G. Register allocation and spilling via graph coloring. In *Proceedings of the ACM SIGPLAN 82 Symposium on Compiler Construction* (June 1982), pp. 201–207.

[18] CHAITIN, G., AUSLANDER, M., CHANDRA, A., COCKE, J., HOPKINS, M., AND MARKSTEIN, P. Register allocation via coloring. *Computer Languages 6* (1981).

[19] COFFMAN, E. *Computer and Job-Shop Scheduling Theory*. Jon Wiley & Sons, New York, 1976.

[20] COLWELL, R., NIX, R., O'DONNELL, J., PAPWORTH, D., AND RODMAN, P. A VLIW architecture for a trace scheduling compiler. *IEEE Transactions on Computers 37*, 8 (August 1988).

[21] DEHNERT, J., HSU, P. Y.-T., AND BRATT, J. Overlapped Loop Support in the Cydra-5. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Boston, MA, April 1989), pp. 26–38.

[22] DEWITT, D. *A Machine-Independent Approach to the Production of Optimal Horizontal Microcode*. PhD thesis, Department of Computer and Communication Sciences, University of Michigan, Ann Arbor, MI, 1976.

[23] D.G., B., EGGERS, S., AND HENRY, R. Integrating Register Allocation and Instruction Scheduling for RISCs. In *International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS-V* (1991), pp. 122–131.

[24] EBCIOGLU, K., AND NICOLAU, A. A global resource-constrained parallelization technique. In *Proceedings of the Third International Conference on Supercomputing* (1989), pp. 154–163.

[25] EBCIOĞLU, K. A Compilation Technique for Software Pipelining of Loops With Conditional Jumps. In *Proceedings of the 20th Microprogramming Workshop (MICRO-20)* (Colorado Springs, CO, December 1987), pp. 69–79.

[26] EBCIOĞLU, K., AND NAKATANI, T. A New Compilation Technique for Parallelizing Loops with Unpredictable Branches on a VLIW Architecture. In *Languages and Compilers for Parallel Computing*, D. Gelernter, Ed. MIT Press, Cambridge,MA, 1990, pp. 213–229.

[27] ELLIS, J. R. *Bulldog: A Compiler for VLIW Architectures*. The MIT Press, 1985. PhD thesis, Yale, 1984.

[28] EVANS AND SUTHERLAND. *The Breadth of Visual Simulation Technology*, 1989.

[29] FISHER, J. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers C-30*, 7 (July 1981), 478–490.

[30] FISHER, J. *The Optimization of Horizontal Microcode Within and Beyond Basic Blocks: An Application of Processor Scheduling*. PhD thesis, Courant Institute of Mathematical Sciences, New York University, New York, NY, October 1979.

[31] FREUDENBERGER, S. M., AND RUTTENBERG, J. C. Phase ordering of register allocation and instruction scheduling. In *Code Generation - Concepts, Tools, Techniques: Proceedings of the International Workshop on Code Generation* (London, May 1992), R. Giegerich and S. L. Graham, Eds., Springer-Verlag, pp. 146–172.

[32] GAREY, M., AND JOHNSON, D. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, San Francisco, CA, 1979.

[33] GASPERONI, F. Compilation techniques for VLIW architectures. Tech. Rep. 435, New York University, Courant Institute of Mathematical Sciences, New York University, March 1989.

[34] GIBBONS, A. *Algorithmic Graph Theory*. Cambridge University Press, Cambridge, England, 1985.

[35] GOODMAND, J., AND HSU, H.-C. Code Scheduling and Register Allocation in Large Basic Blocks. In *Proceedings of the 1988 International Conference on Supercomputing* (July 1988), pp. 442–452.

[36] HUFF, R. A. Lifetime-Sensitive Modulo Scheduling. In *Conference Record of SIGPLAN Programming Language and Design Implementation* (June 1993).

[37] INTEL. *i860 64-bit Microprocessor Programmer's Reference Manual*, 1990.

[38] KNUTH, D. *The Art of Computer Programming*, second ed., vol. I: Fundamental Algorithms. Addison-Wesley, Reading, MA, 1973.

[39] KURLANDER, S. M., AND FISCHER, C. N. Zero-cost range splitting. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation* (Orlando,FL, June 1994), pp. 257–265.

[40] LAM, M. *A Systolic Array Optimizing Compiler*. PhD thesis, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1987.

[41] LAM, M. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation* (Atlanta, GA, June 1988), pp. 318–328.

[42] LANDSKOV, D., DAVIDSON, S., SHRIVER, B., AND MALLETT, P. Local microcode compaction techniques. *ACM Computing Surveys 12*, 3 (September 1980), 261–294.

[43] LEVINTHAL, A., AND PORTER, T. Chap: A SIMD graphics processor. *Computer Graphics 18*, 3 (1984).

[44] MAHLKE, S., LIN, D., CHEN, W., HANK, R., AND BRINGMANN, R. Effective Compiler Support for Predicated Execution Using the Hyperblock. In *Proceedings of the 25th Annual International Symposium on Microarchitecture (MICRO-25)* (Portland, OR, December 1-4 1992), pp. 45–54.

[45] MAHLKE, S. A., CHEN, W. Y., MEI W. HWU, W., RAU, B. R., AND SCHLANSKER, M. S. Sentinel scheduling for VLIW and superscalar processors. In *asplos5* (Boston, MA, oct 1992), vol. 27, pp. 238–247.

[46] NICOLAU, A. *Parallelism, Memory Anti-aliasing, and Correctness Issues for a Trace Scheduling Compiler*. PhD thesis, Department of Computer Science, Yale University, New Haven, Conn, December 1984.

[47] NICOLAU, A., AND FISHER, J. Measuring the parallelism available for very long instruction word architectures. *IEEE Transactions on Computers 33*, 11 (Nov 1984), 968–976.

[48] NICOLAU, A., AND POTASMAN, R. An Environment for the Development of Microcode for Pipelined Architectures. In *Proceedings of the 23rd Symposium and Workshop on Microprogramming and Microarchitecture* (Orlando, Florida, November 1990), pp. 69–79.

[49] NICOLAU, A., POTASMAN, R., AND WANG, H. Register Allocation, Renaming and Their Impact on Fine Grain Parallelism. In *Proceedings of the 4th International Workshop on Languages and Compilers for Parallel Processing* (August 1991).

[50] PADUA, D., KUCK, D., AND LAWRIE, D. High Speed Multiprocessors and Compilation Techniques. *IEEE Transactions on Computers C-29*, 9 (Sept 1980), 763–776.

[51] PINTER, S. Register Allocation with Instruction Scheduling. In *SIGPLAN '93 Conference on Programming Language Design and Implementation* (Albuquerque, NW, June 23-25 1993), pp. 248–257.

[52] PINTER, S. S., AND PINTER, R. Y. Program Optimization and Parallelization Using Idioms. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages* (January 1991), pp. 79–92.

[53] RAJAGOPALAN, M., AND ALLAN, V. H. Specification of Software Pipelining using Petri Nets. *International Journal on Parallel Processing 3*, 22 (1994), 279–307.

[54] RAU, B. R., AND FISHER, J. A. Instruction-Level Parallel Processing: History, Overview, and Perspective. *The Journal of Supercomputing 7* (1993), 9–50.

[55] RAU, B. R., LEE, M., TIRUMALAI, P., AND SCHLANSKER, M. S. Register Allocation for Modulo Scheduled Loops: Strategies, Algorithms, and Heuristics. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation"* (San Francisco, CA, June 1992).

[56] RAU, B. R., SCHLANSKER, M. S., AND TIRUMALAI, P. Code Generation Schema for Modulo Scheduled Loops. In *Proceedings of Micro-25, The 25th Annual International Symposium on Microarchitecture* (December 1992).

[57] RAU, B. R., YEN, D. W. L., YEN, W., AND TOWLE, R. A. The Cydra 5 Departmental Supercomputer: Design Philosophies, Decisions, and Trade-offs. *The IEEE* Computer (January 1989), 12–25.

[58] SETHI, R. Complete register allocation problems. *SIAM Journal of Computing 4*, 3 (1975), 226–248.

[59] SRINIVAS, M., NICOLAU, A., AND ALLAN, V. H. An Approach to Combine Predicated/Speculative Execution for Programs with Unpredictable Branches. In *PACT 94* (Montreal Canada, August 23-26 1994).

[60] SU, B., AND DING, S. Some experiments in global microcode compaction. In *Proceedings of the 18th Microprogramming Workshop (MICRO-18)* (Asilomar, CA, Nov 1985), pp. 175–180.

[61] SU, B., DING, S., AND XIA, J. URPR – an extension of URCR for software pipelining. In *Proceedings of the 19th Microprogramming Workshop (MICRO-19)* (New York, NY, December 1986), pp. 94–103.

[62] SU, B., DING, S., AND XIA, J. Microcode compaction with timing constraints. In *Proceedings of the 20th Microprogramming Workshop (MICRO-20)* (Colorado Springs, CO, December 1987).

[63] SWEANY, P., AND BEATY, S. Dominator-path scheduling — a global scheduling method. In *Proceedings of the 25th International Symposium on Microarchitecture (MICRO-25)* (Portland, OR, December 1992), pp. 260–263.

[64] TIRUMALAI, P., LEE, M., AND SCHLANSKER, M. Parallelization of Loops with exits on pipelined architectures. In *Proceedings of SuperComputing '90* (November 1990), pp. 200–212.

[65] TJADEN, G., AND FLYNN, M. Detection and parallel execution of independent instructions. *IEEE Transactions on Computers C-19*, 10 (Oct 1970), 889–895.

[66] VEGDAHL, S. *Local Code Generation and Compaction in Optimizing Microcode Compilers*. PhD thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, 1982.

[67] VEGDAHL, S. A Dynamic-Programming Technique for Compacting Loops. In *Proceedings of Micro-25, The 25th Annual International Symposium on Microarchitecture* (December 1992).

[68] WALL, D. W. Limits of instruction-level parallelism. In *asplos4* (Santa Clara, CA, April 1991), pp. 176–188.

[69] WARTER, N., HAAB, G., AND BOCKHAUS, J. Enhanced Modulo Scheduling for Loops with Conditional Branches. In *Proceedings of the 25th Annual International Symposium on Microarchitecture (MICRO-25)* (Portland, OR, December 1-4 1992), pp. 170–179.

[70] WARTER, N., MAHLKE, S. A., MEI W. HWU, W., AND RAU, B. R. Reverse If-Conversion. In *PLDI* (Albuquerque, NM, June 1993), pp. 290–299.

[71] WIJAYA, P., AND ALLAN, V. Incremental Foresighted Local Compaction. In *Proceedings of the 22th Microprogramming Workshop (MICRO-22)* (Dublin, Ireland, August 1989).

[72] WOOD, G. On the packing of micro-operations into micro-instruction words. In *Proceedings of the 9th Microprogramming Workshop (MICRO-9)* (December 1978).

[73] ZAKY, A. M. *Efficient Static Scheduling of Loops on Synchronous Multiprocessors*. PhD thesis, Department of Computer and Information Science, Ohio State University, Columbus, OH, 1989.

[74] ZIMA, H., AND CHAPMAN, B. *Supercompilers for Parallel and Vector Computers*. ACM Press, Cambridge, MA, 1991.