UC Irvine
ICS Technical Reports

Title
An efficient global resource constrained technique for exploiting instruction level parallelism

Permalink
https://escholarship.org/uc/item/9rc3p8gn

Authors
Nicolau, Alexandru
Novack, Steven

Publication Date
1992-01-21

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Notice: This Material
may be protected
by Copyright Law
(Title 17 U.S.C.)

Z
699
C3
no. 92-08

An Efficient Global Resource Constrained Technique for Exploiting Instruction Level Parallelism

Alexandru Nicolau and Steven Novack
Department of Information and Computer Science
University of California
Irvine, CA 92717

Technical Report Number 92-08

January 21, 1992

Abstract

A new Global Resource-constrained Percolation (GRiP) scheduling technique is presented for exploiting instruction level parallelism. Other techniques that have been proposed either have been prohibitively expensive in terms of computation or have limited parallelism. The GRiP technique has been implemented and simulation results are presented.

Keywords: Instruction-level parallelism, Percolation Scheduling, Software Pipelining, Resource-constraints, VLIW.

1 Introduction

In this paper we present an efficient resource constrained method for exploiting instruction level parallelism that is based on a new Global Resource-constrained Percolation (GRiP) scheduling technique. Though we present GRiP in the context of Percolation Scheduling and Perfect Pipelining, it could also be used for other instruction level parallelization techniques, such as Trace Scheduling[Fi81] and Enhanced Pipelined Percolation Scheduling[EbNa89]. GRiP scheduling is motivated by the belief that resource constraints should be an integral part of scheduling and should be based on global information about a program.

Globally considering resource constraints during the scheduling process is important for three reasons. First, in the realm of software pipelining, resource constraints can be used to determine how many iterations to allow into the pipelined loop body. Without global resource constraints information, the number of iterations must be limited by some arbitrary (and possibly narrow-sighted) criteria. To understand the importance of this, consider pipelining a vectorizable loop with 5 operations for a VLIW machine with 4 functional units. If resource constraints are separated from the pipelining process, there is no a priori limit on the parallelism exposed and thus no natural convergence of the software pipelining. In such situations, unconstrained pipelining techniques (e.g. [AiNi88c] and [Eb87]) typically limit the parallelism at the throughput level to the equivalent of one sequential iteration per pipelined iteration (i.e. one iteration per cycle). In this context the best possible schedule for our example would execute 5 operations every 2 instructions (4 operations in one instruction and the remaining operation in the other). If resource constraints are incorporated like in Modulo scheduling [GrLa86, RaGl82], convergence is less arbitrary, but no guarantee of good utilization can be provided since the scheduler takes a local (1 or 2 iterations) view of the code. By allowing the resources to be filled as part of a global scheduling process, 4 iterations would be let into the final pipelined loop body, thus resulting in a schedule that would execute at the peak capacity of the machine of 4 operations per instruction.

The second reason for incorporating resource constraints into the scheduling process is to allow for intelligent decisions about when to perform speculative scheduling. Speculatively scheduling an operation from a path that may or may not be chosen can vastly improve performance if the path is actually taken; however, in the presence of resource constraints, it may also prevent the scheduling of another more useful operation. Aggressive speculative motions of operations in the absence of resource information can yield significant *slowdowns* in some instances as possibly useless operations compete with useful operations for scarce resources. Having knowledge about what resources are available while scheduling allows for more sophisticated decisions about whether or not to speculatively schedule an operation. For instance, when a large number of resources are currently available, it would be worthwhile to allow the speculative scheduling of operations; on the other hand, with only a few resources, it might be better to prohibit it until all non-speculative operations have been scheduled. Of course the efficacy of these decisions could be improved with accurate knowledge of path execution probabilities. Currently, neither speculative scheduling heuristics nor branch prediction are incorporated into our new GRiP technique; however, both could be added in a straight forward manner since the heuristic part of our GRiP technique is completely abstracted away from the actual transformations in accordance with the hierarchical nature of Percolation Scheduling. Without speculative scheduling heuristics, GRiP always allows speculative scheduling.

The third reason for integrating resource constraints into the scheduling process has to do with computational efficiency. Scheduling without resource constraints requires operations to move as far as data dependencies allow. Resource constraints usually limit the distance that operations will move since operations are frequently prevented from moving due to a lack of available resources before they would have been stopped by data dependencies.

The remainder of this paper is organized as follows. Section 2 provides a brief overview of Percolation Scheduling and Perfect Pipelining. Section 3 describes the GRiP technique. Section 4

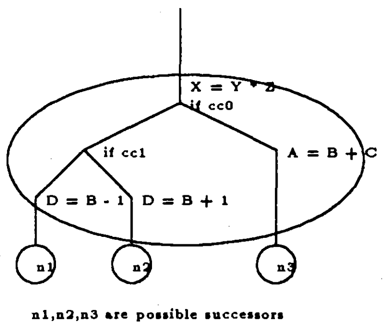


Figure 1: VLIW Instruction

presents simulation results.

2 Percolation Scheduling and Perfect Pipelining

This section provides an overview of Percolation Scheduling (PS) and Perfect Pipelining (PP) which provide the parallelization tools upon which our implementation of GRiP is built. For a thorough discussion of Percolation Scheduling, refer to [AiNi88c, Ni85a]. For more complete coverage of Perfect Pipelining, see [AiNi88c].

Percolation Scheduling is a system for performing parallelizing transformations on the program graphs of the VLIW[Fi83] computation model. A ***program graph*** is a directed graph wherein each node is an instruction and edges represent control flow. An ***instruction*** is a set of "conventional" operations[1] (possibly involving multiple conditional jumps). The terms *instruction* and *node* are used synonymously throughout this paper. For simplicity of exposition, we assume that all operations are completed within a single cycle. An extension to PS that allows for multi-cycle operations is presented in [Po91]. The execution semantics of VLIW instructions are as follows:

1. Operands for all operations are fetched.
2. Results of all operations are computed but not stored. The "result" of a conditional is to select a branch in the tree.
3. Values are stored in this step. There are two variants: Plain VLIW[Fi83] and IBM VLIW[Eb88]. Plain VLIW instructions store all results computed in the instruction, irrespective of the selected path. IBM VLIW instructions store only those results that were computed along the path selected by the conditionals. Given these execution semantics, it is possible to graphically represent any single IBM VLIW instruction as a tree with operations associated with the various paths through the tree (see Figure 1 for an example). We use the IBM VLIW model, but the discussion presented herein is equally valid for the Plain VLIW model as well.
4. Next instruction is chosen to be the instruction reached by following the selected branches through the conditional jumps in the current instruction.

[1]e.g. A = B op C, load/store A ADDR, jump-cond C DEST, etc

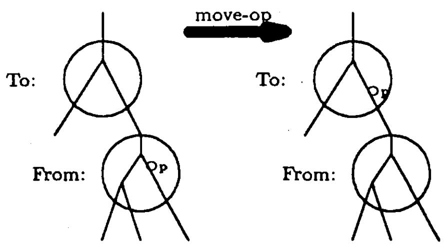


Figure 2: move-op(From,To,Op,Path)

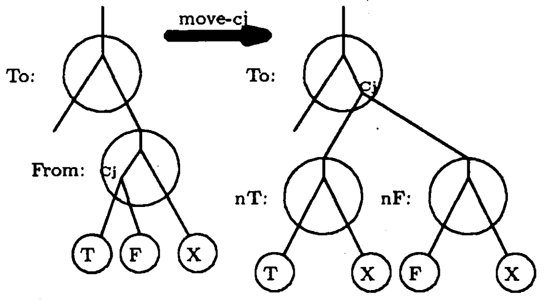


Figure 3: move-cj(From,To,Op,Path)

Core Transformations

At the heart of PS is a pair of semantic preserving transformations called move-op and move-cj. These transformations move an operation or a conditional jump up one instruction in the program graph. By starting with a program wherein each instruction contains a single operation, move-op and move-cj can be used to "percolate" operations as high as data dependencies allow, thus resulting in a maximally parallelized version of the original program, subject to data dependencies and incremental (adjacent instruction) transformations. The move-op and move-cj transformations are shown in Figures 2 and 3, respectively.

Both *move-op(From,To,Op,Path)* and *move-cj(From,To,Op,Path)* will fail if *Op* has a true data dependence on an operation in *To* that is on *Path*. An operation might also fail to move if there is a *write-live conflict* or a *move-past-read conflict*. A *write-live conflict* exists when trying to move an operation *Op* from a node *From* to a node *To* if *Op* writes to a register that is live at the entry to *From*, but that is not killed by *Op*. A *move-past-read conflict* exists if *Op* writes to a value that is read in *From*. Due to the execution semantics of VLIW instructions, it is legal for an operation to write to a register that is read in the same instruction;[2] however, if the operation that does the write moves up it will destroy the value stored in that register. *Write-live conflicts* and *Move-past-read conflicts* can be removed via a process known as *renaming*. Assume that we are attempting to move an operation *Op* from the node *From* to the node *To*. If there is a free register, say *R*, available, then all existing *move-past-read* and *write-live* conflicts can be removed by replacing all syntactic copies of *Op* in *From* with the copy $Def(Op) \leftarrow R$ and by changing $Def(Op)$ to *R* before attaching

[2]The reason this is allowed is that anti-dependencies only imply correct execution, not strict precedence.

```
function migrate(n,op)
    foreach s ∈ successors(n) do
        migrate(s,op)
    end foreach
    foreach s ∈ successors(n) do
        move-op-or-cj(s,n,op,continuation(s,n,op))
    end foreach
end function
```

Figure 4: The migrate transformation

Op to *To*.[3] Note that copy operations produced by renaming do not generate new values and do not prevent code motion. Assume that we are moving $A \leftarrow B$ op C to *To* and that in *To* along *Path*, there is a copy $B \leftarrow X$. In order to allow the move, we simply change the use of B into a use of X: $A \leftarrow X$ op C.

The next layer above the core transformations, the *migrate* function, is used for moving operations as high as possible on a specified subgraph of the program. *Migrate(n,op)* moves *op* as high as possible on the subgraph dominated by n and is defined in Figure 4.

Perfect Pipelining

Software pipelining is a technique for exploiting data parallelism by overlapping the execution of successive iterations of a loop. PS (and other techniques) can be used to overlap n iterations of a loop by simply unwinding the loop n times and moving operations as high as possible toward the loop entry. Consider a loop containing the operations A,B,C where each operation depends on the preceding one and A also has a loop-carried dependency on itself. The table in Figure 5 shows how 4 iterations could be overlapped for such a loop. Simple software pipelining is achieved by retaining the back edge from node 6 to node 1 and provides a speedup of 2 in the loop body.

Perfect Pipelining can be used to improve pipelined schedules by changing the structure of a loop and, given enough resources, guarantees a speedup equal to that achievable after infinite (full) unwinding of the loop, without actual unwinding, subject only to the technique used for parallelization and true data dependencies. Imagine the loop in Figure 5 unwound an infinite number of times. The cba pattern in the middle continuously repeats after iteration 2. We can exploit this fact by making this repeated pattern the new loop body and everything before and after it, the pre-loop and post-loop code, respectively (see Figure 6). By doing this, we achieve a speedup of 3 in the loop body. "Simple" pipelining on this example using any fixed unwinding of iterations yields

[3] *Def(Op)* stands for the register that *Op* defines.

	Iteration			
Node	0	1	2	3
1	a			
2	b	a		
3	c	b	a	
4		c	b	a
5			c	b
6				c

Figure 5: Overlapping loop iterations

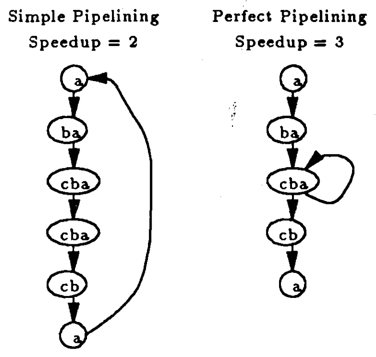


Figure 6: Pipelining Example

a speedup that is strictly less than 3. Perfect Pipelining achieves the effect of infinite unwinding on all paths through the loop, even in the presence of conditional jumps.

Other techniques that have been proposed for performing software pipelining are presented in [RaGl82], [GrLa86], and [EbNa89].

3 GRiP Scheduling

This section describes a new GRiP technique for exploiting instruction level parallelism that can be used in conjunction with Perfect Pipelining. GRiP was inspired by a technique based on scheduling *unifiable operations*[4] [EbNi89] that was also proposed for use with Perfect Pipelining[AiNi90]. Unifiable Operations (Unifiable-ops) scheduling ensures that operations can be scheduled as early as data dependencies and resource constraints allow; however, while very effective in principle, it is likely to be too expensive for practical applications. A discussion of the reasons for this is provided in section 3.1. GRiP Scheduling is a computationally efficient approximation of the Unifiable-ops method that in the worst case may artificially restrict code motion, but that in practice allows operations to be scheduled as early as data dependencies and resource constraints allow.

In order to clarify the motivation for the GRiP technique, we begin this section with a brief description of the Unifiable-ops technique and a discussion of why it is so expensive, especially for Perfect Pipelining. Following this we describe the GRiP technique, how it avoids the problems of Unifiable-ops Scheduling, and the price paid for the added efficiency.

3.1 Why the Unifiable-ops Technique is Expensive

Unifiable-ops scheduling as presented in [EbNi89] consists of three steps. First, a heuristic is used to rank the importance of all operations in the program.[5] Second, data dependency information is used to compute a set of *Unifiable-ops* for each node in the program graph. The *Unifiable-ops* set at a node n consists of all the operations that can immediately be moved to n by some sequence of PS transformations. Stated differently, the *Unifiable Operations* set at a node n is the set of all operations on the subgraph dominated by n that are not on the same data dependency chain as any operation currently in n. Implicit in this definition is that the Unifiable-ops sets need to be updated *incrementally* as operations move.[6] The third step consists of making a top down traversal of the program, filling the resources of each node with the best operations available in the Unifiable-ops set of that node. Figure 7 shows pseudocode for a Unifiable-ops scheduler and Figure 8 shows an example of scheduling with the Unifiable-ops technique. For simplicity we use examples without conditional jumps; however, both PS and PP in general, and Unifiable-ops scheduling and GRiP scheduling in particular, can deal with conditionals.

[4] Unifiable operations are occasionally referred to as Available operations in the literature.

[5] Actually, the operation ranking could be changed dynamically during the scheduling process, but for the purposes of this discussion, we can think of it as fixed.

[6] If global updating were required, the scheme would obviously be impractical.

```
procedure schedule( n )
    while (resources remain) and (Unifiable-ops(n) $\neq \emptyset$) do
        op ← choose-op(Unifiable-ops(n))
        migrate(n, op)
    end while
end procedure
```

Figure 7: Unifiable-ops Scheduler

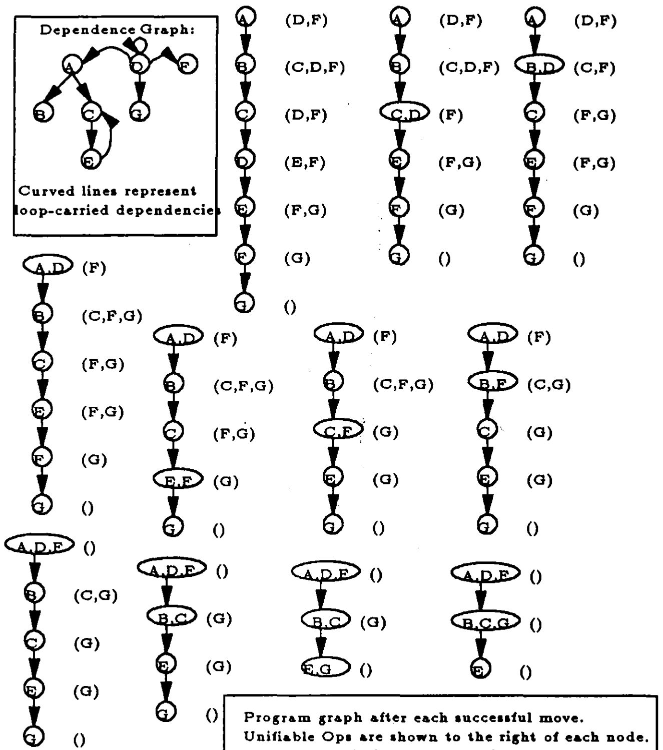

Dependence Graph:
Curved lines represent loop-carried dependencies
Program graph after each successful move.
Unifiable Ops are shown to the right of each node.
Scheduling priority is alphabetical order.

Figure 8: Scheduling with the Unifiable-ops technique

Unifiable-ops Scheduling is expensive for three reasons. First, computing and maintaining the Unifiable-ops sets is expensive. Second, by scheduling operations in a top-down fashion, and since no compaction occurs below the node currently being scheduled, the number of nodes traversed by each operation is maximized. Furthermore, this inefficiency could not easily be removed since changing the order of compaction would make the incremental updating of Unifiable-ops sets impossible, and thus the Unifiable-ops method completely impractical.

The third efficiency problem occurs only with Perfect Pipelining and occurs because Unifiable-ops scheduling always moves operations as far as data dependencies and resource constraints allow. In the presence of loop-carried dependencies (LCD's), this requires the use of a costly post-processing phase to "push" operations down into a convergent pattern. As shown in [AiNi88b], it is insufficient to limit code motion based solely on data dependencies and resource constraints in loops that have LCD's. The reason for this is that in the presence of LCD's it is possible for operations to migrate "too far". This fact is perhaps best illustrated by an example. Consider the data dependence graph shown in Figure 8. By scheduling operations based purely on data dependencies we get the schedule for 4 iterations shown in Figure 9. Notice the gaps formed between the operations in iteration 2 and which increase further in iteration 3. As we increase the number of iterations, the size of these gaps steadily increases. A direct consequence of this is that no row will be repeated and therefore that Perfect Pipelining does not naturally converge. Obviously, in order to achieve convergence, the gaps need to be limited to a fixed size. It is shown in [AiNi88b] that the optimal size for gaps varies depending on the specific dependence characteristics of a loop, but that for most loops, a gap size of zero is optimal.

In the context of Unifiable-ops scheduling, convergence of Perfect Pipelining would be ensured by a post-processing phase that would reverse all moves that had caused gaps. Handling gaps in this fashion would be prohibitively expensive since in order to undo the move of an operation x, we would have to undo any transformations that depended on x, including other code motions, dead-code elimination, renaming, node splitting, etc.

It might seem possible for Unifiable-ops scheduling to handle gaps by preventing them from ever forming; however, gaps are an inherent part of Unifiable-ops scheduling. Before an iteration, i, is scheduled, all operations from iteration i succeed all operations from iteration $i-1$ in the program graph. When the Unifiable-ops technique schedules an operation, *op*, from iteration i, *op* moves as high as data dependencies and resource constraints allow, thus producing a (possibly temporary) gap between the instruction containing *op* and the next instruction containing an operation from iteration i. This sort of gap would be created after most code motions and it would not be known until after the entire iteration had been scheduled whether or not any permanent gaps had formed (the last operation scheduled might have filled the last gap).

	Iteration			
Node	0	1	2	3
1	adf			
2	bcg	adf		
3	e	bg	adf	
4		c	bg	adf
5		e		bg
6			c	
8			e	
9				c
10				e

Figure 9: Pipelined Schedule with Gaps

3.2 Removing the Inefficiencies with GRiP Scheduling

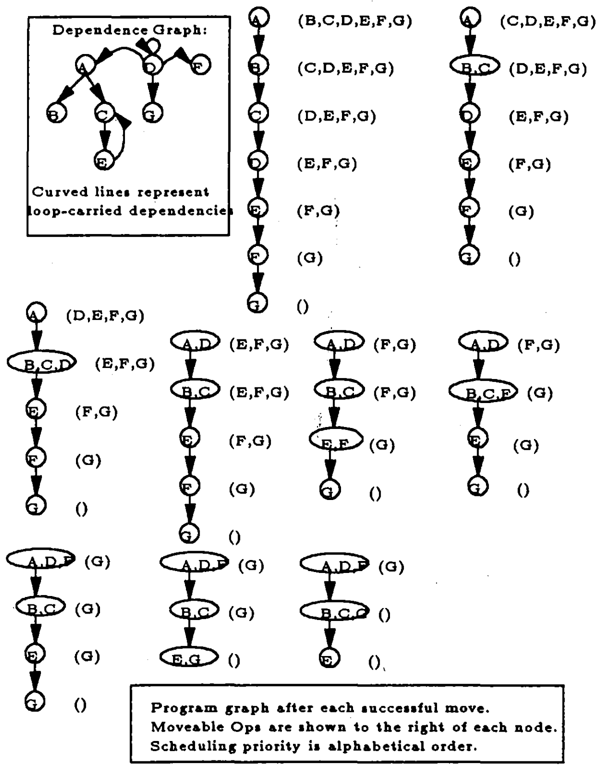
The GRiP method is based on the same principle as the Unifiable-ops method, namely, that when scheduling a node, resources should be filled by migrating a set of operations in an order determined by global information. The difference between the two is in which operations are contained within this set. Recall that in the Unifiable-ops method, only operations that will succeed in moving to a node n are allowed to move while scheduling n. When scheduling a node n using the GRiP method, *all* operations that are on the subgraph dominated by n and that can move subject to true data dependencies and resource constraints are allowed to. In other words, while scheduling n, compaction can occur on the entire subgraph dominated by n. Specifically, the GRiP method consists of three steps. First, a heuristic is used to rank the importance of all operations in the program. Second, a set of Moveable Operations (Moveable-ops) is computed for each node in the program. Initially, the *Moveable-ops* set at a node n contains all operations on the subgraph dominated by n. As scheduling progresses, operations become *unmoveable* and are removed from the Moveable-ops set. An operation becomes *unmoveable* if it has moved into or above the node currently being scheduled or if it is prevented from moving by a strict data dependency on an operation that is itself unmoveable. The third step consists of scheduling each node in a top-down traversal of the program. A node n is scheduled by attempting to migrate to n, in ranked order, all operations in the Moveable-ops set of n until no further operations can be moved to n. When GRiP is used for Perfect Pipelining, the loop body is unwound a fixed number of times before scheduling and the operation ordering heuristic requires that all operations from iteration i have higher priority than all operations from iteration $j > i$.[7] Figure 10 shows the pseudocode for a GRiP scheduler and Figure 11 shows an example of GRiP Scheduling.

The only difference in terms of scheduling ability between the GRiP method and Unifiable-ops

[7] Actually, the unwinding can be done incrementally.

```
procedure schedule( n )
    while (resources remain) and (Moveable-ops(n) $\neq \emptyset$) do
        op ← choose-op(Moveable-ops(n))
        migrate(n, op)
    end while
end procedure
```

Figure 10: GRiP Scheduler



Dependence Graph:
Curved lines represent loop-carried dependencies
Program graph after each successful move.
Moveable Ops are shown to the right of each node.
Scheduling priority is alphabetical order.

Figure 11: GRiP Scheduling

scheduling, and the reason that Unifiable-ops scheduling is able to *guarantee* that all operations travel as far as possible, is that *resource barriers* can form using GRiP scheduling, but can not form using the Unifiable-ops technique.

Definition. *Consider the program graph fragment $A \rightarrow B \rightarrow C$. If there is an operation op in C that is prevented from moving into B only because B is full and op would be moveable from B to A, and A is not full, then B is a* **resource barrier**.

Unifiable-ops scheduling ensures that operations travel as far as possible by moving only those operations that will succeed in migrating all the way to the node being scheduled. Scheduling in this fashion ensures that no node below the one currently being scheduled can become a resource barrier, thus ensuring that the current code motion does not interfere with any future code motions or scheduling decisions. On the other hand, the GRiP method does allow operations to move into and remain in nodes below the one currently being scheduled and therefore may suffer from resource barriers.

At first glance it might seem that resource barriers are a serious problem since they limit code motion, and if they are only temporary, can potentially cause operations to be scheduled out of the order implied by the scheduling heuristic; however, in practice, with a reasonable scheduling heuristic (e.g. one based on list scheduling), resource barriers are not likely to be a problem. Section 4 provides empirical evidence supporting this conjecture. A discussion of why resource barriers are not likely to be a problem is presented below.

To see how temporary resource barriers can cause operations to be scheduled out of order, consider the case of a temporary resource barrier, B, that prevents *op* from moving above it. It might be the case that some operation, say x, either in B or in an instruction preceding B has a lower priority than *op*. Since *op* is stopped by B, x will eventually get a chance to move. If at some later time B ceases to be a resource barrier (i.e. some operation moves out of B), then *op* will again be allowed to move; however, if x, which has lower priority than *op*, succeeded in moving, then it might have used a resource that should have been used by *op*.

The only way that B can cease to be a resource barrier, is if some operation y in B moves up. This can only happen if either y or some operation z that y depends on has not yet been scheduled. In this case at least one of y or z has priority less than *op*, and therefore must have started out before *op* in the original code. For a reasonable scheduling heuristic this is unlikely to happen since important operations tend to occur textually before less important ones. In any case, temporary resource barriers can be prevented from ever forming by inserting empty instructions at the beginning of the program and allowing operations to move into them in ranked order. Scheduling in this manner would ensure that any resource barrier B would be permanent since all operations in or preceding B would have already been scheduled. While eliminating temporary gaps in this fashion is easily implementable, taking such a definite approach is probably not necessary.

Permanent resource barriers are not as easily prevented as temporary ones, but nevertheless, are not likely to significantly degrade performance. Assume that B is a permanent resource barrier that prevents *op* from moving above it and that temporary resource barriers do not form.[8] In this case, all of the operations in B have higher priority than *op*. If *op* is so important as to significantly degrade performance by not moving above B, then it can be argued that the scheduling heuristic more than the resource barrier itself is at fault. In any case, the kind of data dependence graph structures that might produce this sort of situation, are not likely to occur very often in real programs.

By tolerating the possibility of the degraded performance that could, in principle, be caused by resource barriers, GRiP is able to overcome all three of the inefficiencies of the Unifiable-ops method. First, expensive Unifiable-ops sets are replaced by trivially maintainable Moveable-ops sets. Second, since scheduling a node n using Moveable-ops implicitly allows (at least partial) compaction of the entire subgraph dominated by n, travel distances are no longer maximized. Finally, and perhaps most importantly, when used for Perfect Pipelining, GRiP is able prevent the formation of permanent gaps.

3.3 Gap Prevention

To ensure convergence of Perfect Pipelining, the GRiP method incorporates a gap prediction and prevention facility that allows limited size, *temporary*, gaps to form during PS, but that guarantees that permanent gaps will not form.[9] Furthermore, the temporary gaps that GRiP does allow to form are not caused by LCD's, but rather are necessary to allow for code motion. Consider two operations x and y from the same iteration where y has a true data dependence on x and is located in an instruction immediately following the one that contains x. Assume that by moving x up one instruction that a gap would be formed that would be filled by moving y up one instruction. If we prohibit the formation of all, even temporary, gaps, we would incorrectly prohibit both x and y from moving.

GRiP scheduling prevents permanent gaps from forming by ensuring that only temporary gaps of the type mentioned above are ever created during scheduling. GRiP accomplishes this by using a localized test to determine whether or not a permanent gap *might* be formed by moving an operation *op* up one node. If a permanent gap is possible, GRiP suspends the movement of *op* until it can guarantee that moving it would not cause a permanent gap. Since all operations below the node currently being scheduled are allowed to move (subject to data dependencies and resource constraints), then, unless moving *op* actually would cause a permanent gap, other operations from the same iteration as *op* will eventually move up far enough for the localized test to determine that moving *op* will not cause a permanent gap. By ensuring that no operation that precedes a suspended operation moves and that no operation is allowed to pass a suspended operation, GRIP guarantees that operations are moved in their ranked order.[10]

The following *Gapless-move* test is used to determine whether or not a permanent gap could be

[8] This is a reasonable assumption since the formation of temporary resource barriers can be prevented.

[9] Since the optimal size of gaps for most loops is zero (and for the sake of simplicity) we deal with gaps by attempting to remove them entirely. The solution presented here could be parameterized to allow for gaps of varying size.

[10] Unless resource barriers cause operations to be scheduled out of order, as discussed earlier.

created by moving an operation up one node in the program graph. Notice that this test will allow the formation of temporary gaps of the type mentioned above.

Gapless-move(From,To,Op)[11] if one of the following is true:

1. *Op* is the only operation scheduled at *From*. In this case, *From* will be deleted if *Op* moves out of it.

2. More than one operation from the same iteration as *Op* is scheduled at *From*. In this case, if *Op* moves out of *From*, then *From* will still contain an operation from *Op*'s iteration.

3. *Op* is the last operation in *Op*'s iteration. In this case, no operation from *Op*'s iteration exists after *From*, so no gap can be formed by moving *Op*.

4. *From* has a successor node S that contains an operation X from the same iteration as *Op* such that X would be moveable from S to *From* given that *Op* succeeded in moving to *To* and such that *Gapless-move(S,From,X)* is true. In this case, if *Op* moves out of *From*, then there is another operation from *Op*'s iteration that is able to move into *From* without causing a permanent gap. This fact is proved below in theorem 1.

Notice that conditions 1 through 3 imply that no gap would be created by moving *Op* from *From* to *To*. Condition 4 allows a temporary gap of size 1 to be created, but only if it can certainly be filled by some sequence of PS transformations. To satisfy condition 4, the *Gapless-move* test need only find a single operation from the current iteration that could fill the gap allowed by condition 4. There is no guarantee that the gap will be filled with the operation found by the *Gapless-move* test; however, since all operations from the current iteration have higher priority than all operations from any succeeding iterations, the gap will definitely be filled by *some* operation from the current iteration. Typically, the *Gapless-move* test will be satisfied by conditions 1 through 3. Hence, any search required by condition 4 is likely to be very localized.

Theorem 1. *Gapless-move(From,To,Op) implies that a gap created at From by moving Op to To can be filled by some sequence of PS transformations.*

Proof: Induction on length of path from *From* to last node containing an operation from the same iteration as *Op*. If a gap is formed by moving *Op* from *From* to *To*, then *Gapless-move(From,To,Op)* succeeded due to condition 4 which implies that for some S and X, *Gapless-move(S,From,X)* is true and X is moveable from S to From. Basis: If *Gapless-move(S,From,X)* succeeded due to conditions 1, 2, or 3, then the gap is closed by moving X from S to *From*. Step: If *Gapless-move(S,From,X)* succeeded due to condition 4, then the gap at *From* is replaced by a gap at S by moving X from S to *From*. By induction hypothesis, the gap at S can be filled by some sequence of PS transformations.□

The formation of permanent gaps is prevented using the *Gapless-moves* test in conjunction with the following scheduling rules:

[11] Read: a move of the operation *Op* from the node *From* to the node *To* will not cause a permanent gap

1. Allow an operation *Op* to move from a node *From* to a node *To* only if *Gapless-move(From,To,Op)* is true. If *Op* is not allowed to move, mark it as being ***suspended.*** Suspended operations will not be allowed to attempt another move until the possibility of satisfying the *Gapless-moves* test exists.

2. After a successful move, check for ***suspended*** operations. If there are any, mark them as being not suspended and continue moving operations ***in their ranked order*** – now that these operations have been unsuspended, they may have higher precedence than the operation currently being migrated.

3. Only operations below the lowest suspended operation are allowed to move.

Theorem 1 ensures that rule 1 will prevent permanent gaps from forming. The following theorem states that even with gap prevention, operations are scheduled in ranked order. Theorem 2 is predicated on the assumption that resource barriers do not form. The reason for this is that, as discussed earlier and regardless of whether gap prevention is employed or not, resource barriers can cause operations to be scheduled out of order.

Theorem 2. *If resource barriers do not form, then by following the above rules, if an operation op is prevented from moving to a node n because n is full, then all operations that were previously moved to n have higher priority than op.*[12]

Proof: Assume that op is prevented from moving to n because n is full but that op has a higher priority than some operation, say x, that moved into n (and did not move out). If resource barriers do not form and if there are no suspended operations, then all operations are scheduled in strictly decreasing order of priority, being stopped only by data dependencies and resource constraints. Therefore, x must have been moved to n while op was suspended. This implies that at some point in time when op was suspended, either (a) x was in the same node as op, (b) x was in a node that preceded the node that contained op, or (c) x was in a node that succeeded the node that contained op. Rule 3 states that for cases (a) and (b), x would not have been allowed to move, which leaves case (c). Rule 2 states that in the presence of suspended operations an operation may move at most one step, therefore, x can not move above op while op is suspended. This is a contradiction of the initial assumption.□

Figure 12 shows the pseudocode used for implementing *Gapless-moves* in the GRiP method. Figure 13 shows the final gapless schedule for our running example (see Figure 9) produced by GRiP scheduling with *Gapless-moves.* Notice that convergence of Perfect Pipelining is achieved by making nodes 4 and 5 the new loop body.

[12] An operation that started in n, but that did not move out might have a lower priority than op.

```
procedure schedule( n )
    while (resources remain) and (Moveable-ops(n) $\neq \emptyset$) do
        op $\leftarrow$ choose-op(Moveable-ops(n)) follows gap prevention rules
        migrate(n, op)
        if (something moved) and (ops are suspended) then
            unsuspend all ops
        end if
    end while
end procedure


function migrate(n,op)
    foreach s $\in$ successors(n) do
        migrate(s,op)
    end foreach
    if something moved and ops are suspended then
        return
    end if
    foreach s $\in$ successors(n) do
        if Gapless-move(s,n,op) then
            move-op-or-cj(s,n,op,continuation(s,n,op))
        else
            mark op as suspended
        end if
    end foreach
end function
```

Figure 12: GRiP Scheduler with Gap Prevention

	Iteration			
Node	0	1	2	3
1	adf			
2	bcg	adf		
3	e	bg	adf	
4		c	b	
5		e	g	adf
6			c	b
8			e	g
9				c
10				e

Figure 13: Final Gapless Schedule

3.4 Scheduling Heuristics

Any operation ordering heuristic may be used with GRiP scheduling. A reasonable heuristic would be based on list scheduling and would try to schedule the critical paths first, perhaps weighting priorities based on branch probabilities. The heuristic used for creating the schedules discussed in the next section is based on the following relation:

Operation A has *higher priority* than operation B if one of the following are true:

1. The longest data dependence chain rooted at A is longer than the longest data dependence chain rooted at B.

2. The longest data dependence chains of A and B are equal, but A has more dependents in the data dependence graph than B.

When used for Perfect Pipelining, we add the stipulation that all operations from iteration i have higher priority than all operations from iteration $j > i$.

4 Results

We have implemented the GRiP technique described in the preceding section and have integrated it into an ongoing UCI VLIW compiler project. This compiler also supports an "unconstrained" software pipelining technique called POST. POST works in two phases. First, GRiP scheduling is applied *with infinite resources* to obtain a pipelined loop. Second, POST applies resource constraints by breaking apart nodes that contain too many operations and allowing further percolation to fill any nodes that have become underutilized as a result of the breaking. This method of applying resource constraints as a post-processing phase of scheduling is described in [Po91].

Table 1 provides simulation results comparing the speedups obtained by the GRiP method and the POST method. In all cases GRiP performs no worse than POST and for many loops, performs

	2 FU's		4 FU's		8 FU's	
Loop	GRiP	POST	GRiP	POST	GRiP	POST
LL1	2.0	2.0	4.0	3.5	7.9	7.0
LL2	2.0	1.9	3.8	3.6	7.3	6.9
LL3	2.0	1.8	4.0	3.0	8.0	4.5
LL4	2.0	2.0	4.3	3.9	8.4	5.9
LL5	2.0	2.2	4.4	3.7	5.5	5.5
LL6	2.0	1.8	3.6	2.8	3.6	3.3
LL7	2.0	1.9	4.0	3.9	7.9	7.6
LL8	2.0	1.9	3.4	3.1	4.3	4.0
LL9	2.0	2.0	4.0	3.9	7.9	7.7
LL10	2.0	2.0	4.0	2.9	7.1	3.6
LL11	2.3	2.3	4.5	4.5	8.9	8.9
LL12	2.0	1.8	4.0	3.0	8.0	4.5
LL13	2.1	1.9	3.0	2.7	3.0	3.0
LL14	1.9	1.9	3.7	3.2	4.8	4.5
Mean	2.0	2.0	3.9	3.4	6.6	5.5
WHM	2.0	1.9	3.9	3.3	5.6	4.8

Table 1: Observed Speed-up

much better. Notice that for 2 and 4 functional units, GRiP results are essentially optimal. At 8 functional units, GRiP fills resources as best as possible given the limits on parallelism imposed by the loops themselves.

Since GRiP depends on heuristics to decide how resources should be filled, the speedups shown in Table 1 do not necessarily represent the maximum potential of GRiP, but rather are intended to convey a notion of how well GRiP can perform even with the simple operation ordering defined in section 3.4.

At the source level, none of the Livermore Loops presented in Table 1 contain explicit conditional jumps; however, as a result of unwinding, multiple copies of the original loop control conditional become internal conditionals in each of these loops. Since GRiP does not currently utilize branch prediction or speculative scheduling heuristcs, only results for loops with this kind of conditional are presented. "Internalized" loop control conditionals are handled by giving priority to operations on paths internal to the loop being pipelined. GRiP is capable of compacting programs with explicit internal conditionals, but we feel that results for such loops would only be meaningful if presented in conjunction with speculative scheduling heuristics. Speculative scheduling heuristics and branch prediction could be added to GRiP as well as to techniques that do not incorporate resource constraints into the scheduling process; however, we anticipate that due to the sensitivity of speculative scheduling to resource constraints, GRiP would perform much better than any such "unconstrained" techniques. [We have preliminary results that support this conjecture that we intend to provide in the final version of this paper.]

Our compiler uses the GNU C compiler (GCC) as a front-end that produces an optimized (sequential) intermediate language as output. The GRiP scheduler converts this intermediate representation into a sequential VLIW program graph wherein each node contains a single intermediate language statement. GRiP then parallelizes this code using the technique described in the preceding section. As a result of compaction, some operations in the original code become redundant and are removed. This is the reason that some of the speed-ups in Table 1 are larger than the apparent maximum indicated by the number of functional units. Redundant operation removal is not a necessary part of PS; however, it is a useful optimization that is best performed incrementally as part of the scheduling process in order to ensure that unnecessary operations do not compete with useful operations for resources.

References

[AiNi88b] A. Aiken and A. Nicolau. "Optimal Loop Parallelization". *Proceedings SIGPLAN 88, Conference on Programming Language Design and Implementation*, Atlanta, GA, June 22-24, 1988.

[AiNi88c] A. Aiken and A. Nicolau. "Perfect Pipelining: A new loop parallelization technique". In *Proceedings of the 1988 European Symposium on Programming.* Springer Verlag Lecture Notes in Computer Science no. 300, March 1988.

[AiNi90] A. Aiken and A. Nicolau. "A Realistic Resources - Constrained Software Pipelining Algorithm". *Proceedings of the 3rd Workshop on Programming Languages and Compilers for Parallel Computing*, Irvine, CA, August 1990.

[Eb87] K.Ebcioglu. "A Compilation Technique for Software Pipelining of Loops with Conditional Jumps". *Proceedings of the 20th Annual Workshop on Microprogramming*, pp. 69-79, ACM Press, 1987.

[Eb88] K.Ebcioglu. "Some Design Ideas for a VLIW Architecture for Sequential-Natured Software". *Proceedings IFIP*, 1988.

[EbNa89] K.Ebcioglu, and T. Nakatani. "A New Compilation Technique for Parallelizing Loops with Unpredictable Branches on a VLIW Architecture". *Proceedings of the 2nd Workshop on Programming Languages and Compilers for Parallel Computing*, Urbana, IL, 1989.

[EbNi89] K.Ebcioglu, and A.Nicolau. "A *global* resource-constrained parallelization technique". *Proceedings of ACM SIGARCH ICS-89: International Conference on Supercomputing*, Crete, Greece June 2-9 1989.

[Fi81] J. A. Fisher. "Trace Scheduling: A technique for global microcode compaction". *IEEE Transactions on Computers*, No. 7,pp. 478-490, 1981.

[Fi83] J. A. Fisher. "Very Long Instruction Word architectures and the ELI-512". *Proceedings of the 10th Annual Internation Architecture Conference*, Stockholm, June 1983.

[GrLa86] T. Gross, M. S. Lam. "Compilation for high-performance systolic array". *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*, July 1986.

[Ni85a] A. Nicolau. "Uniform Parallelism Exploitation in Ordinary Programs". *Proceedings of the 1985 International Conference on Parallel Processing*, 1985.

[Po91] R. Potasman "Percolation-Based Compiling for Evaluation of Parallelism and Hardware Design Trade-Offs" PhD thesis, University of California at Irvine, 1991

[RaGl82] B. R Rau, C. D. Glaeser. "Efficient Code Generation for Horizontal Architectures: Compiler Techniques and Architectural Support". *Proceedings of the 9th Symposium on Computer Architecture*, April 1982.