Software Pipelining: An Effective Scheduling Technique for VLIW Machines

Monica Lam

Department of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Abstract

This paper shows that software pipelining is an effective and viable scheduling technique for VLIW processors. In software pipelining, iterations of a loop in the source program are continuously initiated at constant intervals, before the preceding iterations complete. The advantage of software pipelining is that optimal performance can be achieved with compact object code.

This paper extends previous results of software pipelining in two ways: First, this paper shows that by using an improved algorithm, near-optimal performance can be obtained without specialized hardware. Second, we propose a *hierarchical reduction* scheme whereby entire control constructs are reduced to an object similar to an operation in a basic block. With this scheme, all innermost loops, including those containing conditional statements, can be software pipelined. It also diminishes the start-up cost of loops with small number of iterations. Hierarchical reduction complements the software pipelining technique, permitting a consistent performance improvement be obtained.

The techniques proposed have been validated by an implementation of a compiler for Warp, a systolic array consisting of 10 VLIW processors. This compiler has been used for developing a large number of applications in the areas of image, signal and scientific processing.

The research was supported in part by Defense Advanced Research Projects Agency (DOD) monitored by the Space and Naval Warfare Systems Command under Contract N00039-87-C-0251, and in part by the Office of Naval Research under Contracts N00014-87-K-0385 and N00014-87-K-0533.

Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation
Atlanta, Georgia, June 22–24, 1988

1. Introduction

A VLIW (very long instruction word) machine [5, 11] is similar to a horizontally microcoded machine in that the data path consists of multiple, possibly pipelined, functional units, each of which can be independently controlled through dedicated fields in a "very long" instruction. The distinctive feature of VLIW architectures is that these long instructions are the machine instructions. There is no additional layer of interpretation in which machine instructions are expanded into micro-instructions. While complex resource or field conflicts often exist between functionally independent operations in a horizontal microcode engine, a VLIW machine generally has an orthogonal instruction set and a higher degree of parallelism. The key to generating efficient code for the VLIW machine is global code compaction, that is, the compaction of code across basic blocks [12]. In fact, the VLIW architecture is developed from the study of the global code compaction technique, trace scheduling [10].

The thesis of this paper is that software pipelining [24, 25, 30] is a viable alternative technique for scheduling VLIW processors. In software pipelining, iterations of a loop in a source program are continuously initiated at constant intervals without having to wait for preceding iterations to complete. That is, multiple iterations, in different stages of their computations, are in progress simultaneously. The steady state of this pipeline constitutes the loop body of the object code. The advantage of software pipelining is that optimal performance can be achieved with compact object code.

A drawback of software pipelining is its complexity; the problem of finding an optimal schedule is NP-complete. (This can be shown by transforming the problem of resource constrained scheduling problem [14] to the software pipelining problem). There have been two approaches in response to the complexity of this problem: (1) change the architecture, and thus the characteristics of the constraints, so that the problem becomes tractable, and (2) use heuristics. The first approach is used in the polycyclic [25] and Cydrome's Cydra

architecture; a specialized crossbar is used to make optimizing loops without data dependencies between iterations tractable. However, this hardware feature is expensive; and, when inter-iteration dependency is present in a loop, exhaustive search on the strongly connected components of the data flow graph is still necessary [16]. The second approach is used in the FPS-164 compiler [30]. Software pipelining is applied to a restricted set of loops, namely those containing a single Fortran statement. In other words, at most one inter-iteration data dependency relationship can be present in the flow graph. The results were that near-optimal results can be obtained cheaply without the specialized hardware.

This paper shows that software pipelining is a practical, efficient, and *general* technique for scheduling the parallelism in a VLIW machine. We have extended previous results of software pipelining in two ways. First, we show that near-optimal results can often be obtained for loops containing both intra- and inter-iteration data dependency, using software heuristics. We have improved the scheduling heuristics and introduced a new optimization called modulo variable expansion. The latter implements part of the functionality of the specialized hardware proposed in the polycyclic machine, thus allowing us to achieve similar performance.

Second, this paper proposes a *hierarchical reduction* scheme whereby entire control constructs are reduced to an object similar to an operation in a basic block. Scheduling techniques previously defined for basic blocks can be applied across basic blocks. The significance of hierarchical reduction is threefold: First, conditional statements no longer constitute a barrier to code motion, code in innermost loops containing conditional statements can be compacted. Second, and more importantly, software pipelining can be applied to arbitrarily complex loops, including those containing conditional statements. Third, hierarchical reduction diminishes the penalty of short loops: scalar code can be scheduled with the prolog and epilog of a pipelined loop. We can even software pipeline the second level loop as well. The overall result is that a consistent speed up is obtained whenever parallelism is available across loop iterations.

Software pipelining, as addressed here, is the problem of scheduling the operations within an iteration, such that the iterations can be pipelined to yield optimal throughput. Software pipelining has also been studied under different contexts. The software pipelining algorithms proposed by Su et al. [27, 28], and Aiken and Nicolau [1], assume that the schedules for the iterations are given and cannot be changed. Ebcioglu proposed a software pipelining algorithm to generate code for a hypothetical machine with infinitely many hardware resources [7]. Lastly, Weiss and Smith compared the results of using loop unrolling and software pipelining to generate scalar code for the Cray-1S architecture [31]. However, their software pipelining algorithm only overlaps the computation from at most two iterations. The unfavorable results obtained for software pipelining can be attributed to the particular algorithm rather than the software pipelining approach.

The techniques described in this paper have been validated by the implementation of a compiler for the Warp machine. Warp [4] is a high-performance, programmable systolic array developed by Carnegie Mellon and General Electric, our industrial partner. The Warp array is a linear array of VLIW processors, each capable of a peak computation rate of 10 million floating-point operations per second (10 MFLOPS). A Warp array typically consists of ten processors, or cells, and thus has an aggregate bandwidth of 100 MFLOPS.

Each Warp cell has its own sequencer and program memory. Its data path consists of a floating-point multiplier, a floating-point adder, an integer ALU, three register files (one for each arithmetic unit), a 512-word queue for each of the two inter-cell data communication channels, and a 32 Kword data memory. All these components are connected through a crossbar, and can be programmed to operate concurrently via wide instructions of over 200 bits. The multiplier and adder are both 5-stage pipelined; together with the 2 cycle delay through the register file, multiplications and additions take 7 cycles to complete.

The machine is programmed using a language called W2. In W2, conventional Pascal-like control constructs are used to specify the cell programs, and asynchronous computation primitives are used to specify inter-cell communication. The Warp machine and the W2 compiler have been used extensively for about two years, in many applications such as low-level vision for robot vehicle navigation, image and signal processing, and scientific computing [2, 3]. Our previous papers presented an overview of the compiler and described an array level optimization that supports efficient fine-grain parallelism among cells [15, 20]. This paper describes the scheduling techniques used to generate code for the parallel and pipelined functional units in each cell.

This paper consists of three parts: Part I describes the software pipelining algorithm for loops containing straight-line loop bodies, focusing on the extensions and improvements. Part II describes the hierarchical reduction approach, and shows how software pipelining can be applied to all loops. Part III contains an evaluation and a comparison with the trace scheduling technique.

2. Simple loops

The concept of software pipelining can be illustrated by the following example: Suppose we wish to add a constant to a vector of data. Assuming that the addition is one-stage pipelined, the most compact sequence of instructions for a single iteration is:

```
1   Read
2   Add
3
4   Write
```

Different iterations can proceed in parallel to take advantage of the parallelism in the data path. In this example, an iteration can be initiated every cycle, and this optimal throughput can be obtained with the following piece of code:

```
1   Read
2   Add    Read
3          Add    Read
4 L:Write         Add    Read   CJump L
5          Write         Add
6
7                        Write
```

Instructions 1 to 3 are called the *prolog*: a new iteration is initiated every instruction cycle and execute concurrently with all previously initiated iterations. The *steady state* is reached in cycle 4, and this state is repeated until all iterations have been initiated. In the steady state, four iterations are in progress at the same time, with one iteration starting up and one finishing off every cycle. (The operation **CJump L** branches back to label **L** unless all iterations have been initiated.) On leaving the steady state, the iterations currently in progress are completed in the *epilog*, instructions 5 through 7. The software pipelined loop in this example executes at the optimal throughput rate of one iteration per instruction cycle, which is four times the speed of the original program. The potential gain of the technique is even greater for data paths with higher degrees of pipelining and parallelism. In the case of the Warp cell, software pipelining speeds up this loop by nine times.

2.1. Problem statement

Software pipelining is unique in that pipeline stages in the functional units of the data path are not emptied at iteration boundaries; the pipelines are filled and drained only on entering and exiting the loop. The significance is that optimal throughput is possible with this approach.

The objective of software pipelining is to minimize the interval at which iterations are initiated; the *initiation interval* [25] determines the throughput for the loop. The basic units of scheduling are minimally indivisible sequences of micro-instructions. In the example above, since the result of the addition must be written precisely two cycles after the computation is initiated, the add and the write operations are grouped as one indivisible sequence. While the sequence is indivisible, it can overlap with the execution of other sequences. The minimally indivisible sequences that make up the computation of an iteration are modeled as nodes in a graph. Data dependencies between these sequences are mapped onto precedence constraints between the corresponding nodes; associated with each node is a resource reservation table indicating the resources used in each time step of the sequence. To ensure a compact steady state, two more constraints are imposed: the initiation interval between all consecutive iterations must be the same, and the schedule for each individual iteration must be identical. In other words, the problem is to schedule the operations within an iteration, such that the same schedule can be pipelined with the shortest, constant initiation interval.

The scheduling constraints in software pipelining are defined in terms of the initiation interval:

1. *Resource constraints.* If iterations in a software pipelined loop are initiated every sth cycle, then every sth instruction in the schedule of an iteration is executed simultaneously, one from a different iteration. The total resource requirement of every sth instructions thus cannot exceed the available resources. A modulo resource reservation table can be used to represent the resource usage of the steady state by mapping the resource usage of time t to that of time $t \bmod s$ in the modulo resource reservation table.

2. *Precedence constraints.* Consider the following example:

```
FOR i := 1 TO 100 DO
BEGIN
    a := a + 1.0;
END
```

The value of **a** must first be accessed before the store operation, and the store operation must complete before the data is accessed in the second iteration. We model the dependency relationship by giving each edge in the graph two attributes: a minimum iteration difference and a delay. When we say that the minimum iteration difference on an edge (u,v) is p and the delay is d, that means node v must execute d cycles after node u from the pth previous iteration. Let $\sigma : V \rightarrow N$ be the schedule function of a node, then

$$\sigma(v) - (\sigma(u) - s \cdot p) \geq d, \quad \text{or} \quad \sigma(v) - \sigma(u) \geq d - s \cdot p,$$

where s is the initiation interval. Since a node cannot depend on a value from a future iteration, the minimum iteration difference is always nonnegative. The iteration difference for an intra-iteration dependency is 0, meaning that the node v must follow node u in the same iteration. As illustrated by the example, inter-iteration data dependencies may introduce cycles into the precedence constraint graph.

2.2. Scheduling algorithm

The definition of scheduling constraints in terms of the initiation interval makes finding an approximate solution to this NP-complete problem difficult. Since computing the minimum initiation interval is NP-complete, an approach is to first schedule the code using heuristics, then determine the initiation interval permitted by the schedule. However, since the scheduling constraints are defined in terms of the initiation interval, if the initiation interval is not known at scheduling time, the schedule produced is unlikely to permit a good initiation interval.

To resolve this circularity, the FPS compiler uses an iterative approach [30]: first establish a lower and an upper bound on the initiation interval; then use binary search to find the smallest initiation interval for which a schedule can be found. (The length of a locally compacted iteration can serve as an upper bound; the calculation of a lower bound is described below). We also use an iterative approach, but we use a linear search instead. The rationale is as follows: Although the probability that a schedule can be found generally increases with the value of the initiation interval, schedulability is not monotonic [21]. Especially since empirical results show that in the case of Warp, a schedule meeting the lower bound can often be found, sequential search is preferred.

A lower bound on the initiation interval can be calculated from the scheduling constraints as follows:

1. *Resource constraints.* If an iteration is initiated every s cycles, then the total number of resource units available in s cycles must at least cover the resource requirement of one iteration. Therefore, the bound on the initiation interval due to resource considerations is the maximum ratio between the total number of times each resource is used and the number of available units per instruction.
2. *Precedence constraints.* Cycles in precedence constraints impose delays between operations from different iterations that are represented by the same node in the graph. The initiation interval must be large enough for such delays to be observed. We define the delay and minimum iteration difference of a path to be the sum of the minimum delays and minimum iteration differences of the edges in the path, respectively. Let s be the initiation interval, and c be a cycle in the graph. Since

   $$\sigma(v)-\sigma(u)\geq d(e)-s\cdot p(e)$$

   we get:

   $$d(c)-s\cdot p(c)\leq 0.$$

   We note that if $p(c)=0$, then $d(c)$ is necessarily less than 0 by the definition of a legal computation. Therefore, the bound on the initiation interval due to precedence considerations is

   $$\max_c \frac{\lceil d(c) \rceil}{p(c)}, \quad \forall \text{ cycle } c \text{ whose } p(c)\neq 0.$$

2.2.1. Scheduling acyclic graphs

The algorithm we use to schedule acyclic graphs for a target initiation interval is the same as that used in the FPS compiler, which itself is derived from the list scheduling algorithm used in basic block scheduling [9]. List scheduling is a non-backtracking algorithm, nodes are scheduled in a topological ordering, and are placed in the earliest possible time slot that satisfies all scheduling constraints with the partial schedule constructed so far. The software pipelining algorithm differs from list scheduling in that the modulo resource reservation table is used in determining if there is a resource conflict. Also, by the definition of modulo resource usage, if we cannot schedule a node in s consecutive time slots due to resource conflicts, it will not fit in any slot with the current schedule. When this happens, the attempt to find a schedule for the given initiation interval is aborted and the scheduling process is repeated with a greater interval value.

2.2.2. Scheduling cyclic graphs

In adapting the scheduling algorithm for acyclic graphs to cyclic graphs, we face the following difficulties: a topological sort of the nodes does not exist in a cyclic graph; precedence constraints with nodes already scheduled cannot be satisfied by examining only those edges incident on those nodes; and the maximum height of a node, used as the priority function in list scheduling, is ill-defined. Experimentation with different schemes helped identify two desirable properties that a non-backtracking algorithm should have:

1. Partial schedules constructed at each point of the scheduling process should not violate any of the precedence constraints in the graph. In other words, were there no resource conflicts with the remaining nodes, each partial schedule should be a partial solution.
2. The heuristics must be sensitive to the initiation interval. An increased initiation interval value relaxes the scheduling constraints, and the scheduling algorithm must take advantage of this opportunity. It would be futile if the scheduling algorithm simply retries the same schedule that failed.

These properties are exhibited by the heuristics in the scheduling algorithm for acyclic graphs. A scheduling algorithm for cyclic graphs that satisfies these properties is presented below.

The following preprocessing step is first performed: find the strongly connected components in the graph [29], and compute the closure of the precedence constraints in each connected component by solving the all-points longest path problem for each component [6, 13]. This information is used in the iterative scheduling step. To avoid the cost of recomputing this information for each value of the initiation interval, we compute this information only once in the preprocessing step, using a symbolic value to stand for the initiation interval [21].

As in the case of acyclic graphs, the main scheduling step is iterative. For each target initiation interval, the connected components are first scheduled individually. The original graph is then reduced by representing each connected component as a single vertex: the resource usage of the vertex represents the aggregate resource usage of its components, and edges connecting nodes from different connected components are represented by edges between the corresponding vertices. This reduced graph is acyclic, and the acyclic graph scheduling algorithm can then be applied.

The scheduling algorithm for connected components also follows the framework of list scheduling. The nodes in a connected component are scheduled in a topological ordering by considering only the *intra-iteration* edges in the graph. By the definition of a connected component, assigning a schedule to a node limits the schedule of all other nodes in the component, both from below and above. We define the *precedence constrained range* of a node for a given partial schedule as the legal time range in which the node can be scheduled, without violating the precedence constraints of the graph. As each node is scheduled, we use the precomputed longest path information to update the precedence constrained range of each remaining node, substituting the symbolic initiation interval value with the actual value. A node is scheduled in the earliest possible time slot within its constrained range. If a node cannot be scheduled within the precedence constrained range, the scheduling attempt is considered unsuccessful. This algorithm possesses the first property described above: precedence constraints are satisfied by all partial schedules.

As nodes are scheduled in a topological ordering of the intra-iteration edges, the precedence constrained range of a node is bounded from above only by inter-iteration edges. As the initiation interval increases, so does the upper bound of the range in which a node is scheduled. Together with the strategy of scheduling a node as early as possible, the range in which a node can be scheduled increases as the initiation interval increases, and so does the likelihood of success. The scheduling problem approaches that of an acyclic graph as the value of the initiation interval increases. Therefore, the approach also satisfies the second property: the algorithm takes advantage of increased initiation interval values.

2.3. Modulo variable expansion

The idea of modulo variable expansion can be illustrated by the following code fragment, where a value is written into a register and used two cycles later:

```
Def (R1)
op
Use (R1)
```

If the same register is used by all iterations, then the write operation of an iteration cannot execute before the read operation in the preceding iteration. Therefore, the optimal throughput is limited to one iteration every two cycles. This code can be sped up by using different registers in alternating iterations:

```
  Def (R1)
  op        Def (R2)
L: Use (R1) op        Def (R1)
            Use (R2) op        Def (R2)  CJump L
                     Use (R1) op
                              Use (R2)
```

We call this optimization of allocating multiple registers to a variable in the loop *modulo variable expansion*. This optimization is a variation of the variable expansion technique used in vectorizing compilers [18]. The variable expansion transformation identifies those variables that are redefined at the beginning of every iteration of a loop, and expands the variable into a higher dimension variable, so that each iteration can refer to a different location. Consequently, the use of the variable in different iterations is thus independent, and the loop can be vectorized. Modulo variable expansion takes advantage of the flexibility of VLIW machines in scalar computation, and reduces the number of locations allocated to a variable by reusing the same location in non-overlapping iterations. The small set of values can even reside in register files, cutting down on both the memory traffic and the latency of the computation.

Without modulo variable expansion, the length of the steady state of a pipelined loop is simply the initiation interval. When modulo variable expansion is applied, code sequences for consecutive iterations differ in the registers used, thus lengthening the steady state. If there are n repeating code sequences, the steady state needs to be *unrolled* n times.

The algorithm of modulo variable expansion is as follows. First, we identify those variables that are redefined at the beginning of every iteration. Next, we pretend that every iteration of the loop has a dedicated register location for each qualified variable, and remove all *inter-iteration* precedence constraints between operations on these variables. Scheduling then proceeds as normal. The resulting schedule is then used to determine the actual number of registers that must be allocated to each variable. The lifetime of a register variable is defined as the duration between the first assignment into the variable and its last use. If the lifetime of a variable is l, and an iteration is initiated every s cycles, then at least $\lceil \frac{l}{s} \rceil$ number of values must be kept alive concurrently, in that many locations.

If each variable v_i is allocated its minimum number of locations, q_i, the degree of unrolling is given by the lowest common multiple of $\{q_i\}$. Even for small values of q_i, the least common multiple can be quite large and can lead to an

intolerable increase in code size. The code size can be reduced by trading off register space. We observe that the minimum degree of unrolling, u, to implement the same schedule is simply $\max_i q_i$. This minimum degree of unrolling can be achieved by setting the number of registers allocated to variable v_i to be the smallest factor of u that is no smaller than q_i, i.e.,

$$\min n, \text{ where } n \geq q_i \text{ and } u \bmod n = 0.$$

The increase in register space is much more tolerable than the increase in code size of the first scheme for a machine like Warp.

Since we cannot determine the number of registers allocated to each variable until all uses of registers have been scheduled, we cannot determine if the register requirement of a partial schedule can be satisfied. Moreover, once given a schedule, it is very difficult to reduce its register requirement. Indivisible micro-operation sequences make it hard to insert code in a software pipelined loop to spill excess register data into memory.

In practice, we can assume that the target machine has a large number of registers; otherwise, the resulting data memory bottleneck would render the use of any global compaction techniques meaningless. The Warp machine has two 31-word register files for the floating-point units, and one 64-word register for the ALU. Empirical results show that they are large enough for almost all the user programs developed [21]. Register shortage is a problem for a small fraction of the programs; however, these programs invariably have loops that contain a large number of independent operations per iteration. In other words, these programs are amenable to other simpler scheduling techniques that only exploit parallelism within an iteration. Thus, when register allocation becomes a problem, software pipelining is not as crucial. The best approach is therefore to use software pipelining aggressively, by assuming that there are enough registers. When we run out of registers, we then resort to simple techniques that serializes the execution of loop iterations. Simpler scheduling techniques are more amenable to register spilling techniques.

2.4. Code size

The code size increase due to software pipelining is reasonable considering the speed up that can be achieved. If the number of iterations is known at compile time, the code size of a pipelined loop is within three times the code size for one iteration of the loop [21]. If the number of iterations is not known at compile time, then additional code must be generated to handle the cases when there are so few iterations in the loop that the steady state is never reached, and when there are no more iterations to initiate in the middle of an unrolled steady state.

To handle these cases, we generate two loops: a pipelined version to execute most of the iterations, and an unpipelined version to handle the rest. Let k be the number of iterations started in the prolog of the pipelined loop, u be the degree of unrolling, and n be the number of iterations to be executed. Before the loop is executed, the values of n and k are compared. If $n<k$, then all n iterations are executed using the unpipelined code. Otherwise, we execute $n-k \bmod u$ iterations using the unpipelined code, and the rest on the pipelined loop. At most $\lceil \frac{l}{s} \rceil$ iterations are executed in the unpipelined mode, where l is the length of an iteration and s is the initiation interval. Using this scheme, the total code size is at most four times the size of the unpipelined loop.

A more important metric than the total code length is the length of the innermost loop. An increase in code length by a factor of four typically does not pose a problem for the machine storage system. However, in machines with instruction buffers and caches, it is most important that the steady state of a pipelined loop fits into the buffers or caches. Although software pipelining increases the total code size, the steady state of the loop is typically much shorter than the length of an unpipelined loop. Thus, we can conclude that the increase in code size due to software pipelining is not an issue.

3. Hierarchical reduction

The motivation for the *hierarchical reduction* technique is to make software pipelining applicable to all innermost loops, including those containing conditional statements. The proposed approach schedules the program hierarchically, starting with the innermost control constructs. As each construct is scheduled, the entire construct is reduced to a simple node representing all the scheduling constraints of its components with other constructs. This node can then be scheduled just like a simple node within the surrounding control construct. The scheduling process is complete when the entire program is reduced to a single node.

The hierarchical reduction technique is derived from the scheduling scheme previously proposed by Wood [32]. In Wood's approach, scheduled constructs are modeled as black boxes taking unit time. Operations outside the construct can move around it but cannot execute concurrently with it. Here, the resource utilization and precedence constraints of the reduced construct are visible, permitting it to be scheduled in parallel with other operations. This is essential to software pipelining loops with conditional statements effectively.

3.1. Conditional statements

The procedure for scheduling conditional statements is as follows: The THEN and ELSE branches of a conditional statement are first scheduled independently. The entire conditional statement is then reduced to a single node whose scheduling constraints represent the union of the scheduling constraints of the two branches. The length of the new node is the maximum of that of the two branches; the value of each entry in the resource reservation table is the maximum of the corresponding entries in the tables of the two branches. Precedence constraints between operations inside the branches and those outside must now be replaced by constraints between the node representing the entire construct and those outside. The attributes of the constraints remain the same.

The node representing the conditional construct can be treated like any other simple node within the surrounding construct. A schedule that satisfies the union of the constraints of both branches must also satisfy those of either branch. At code emission time, two sets of code, corresponding to the two branches, are generated. Any code scheduled in parallel with the conditional statement is duplicated in both branches. Although the two branches are padded to the same length at code scheduling time, it is not necessary that the lengths of the emitted code for the two branches be identical. If a machine instruction does not contain any operations for a particular branch, then the instruction simply can be omitted for that branch. The simple representation of conditional statements as straight-line sequences in the scheduling process makes it easy to overlap conditional statements with any other control constructs.

The above strategy is optimized for handling short conditional statements in innermost loops executing on highly parallel hardware. The assumption is that there are more unused than used resources in an unpipelined schedule, and that it is more profitable to satisfy the union of the scheduling constraints of both branches all the time, so as not to reduce the opportunity for parallelism among operations outside the conditional statement. For those cases that violate this assumption, we can simply mark all resources in the node representing the conditional statements as used. By omitting empty machine instructions at code emission time, the short conditional branch will remain short. Although this scheme disallows overlap between the conditional statement and all other operations, all other forms of code motion around the construct can still take place.

3.2. Loops

The prolog and epilog of a software pipelined loop can be overlapped with other operations outside the loop. This optimization can again be achieved by reducing a looping construct to a simple node that represents the resource and precedence constraints for the entire loop. Only one iteration of the steady state is represented. The steady state of the loop, however, should not be overlapped with other operations. To prevent this, all resources in the steady state are marked as consumed.

3.3. Global code motions

Once conditional statements and loops are represented as straight-line sequences of code, scheduling techniques, formerly applicable only to basic blocks, such as software pipelining and list scheduling, can be applied to compound control constructs. Global code motions automatically take place as the enclosing construct is scheduled according to the objective of the scheduling technique.

The significance of hierarchical reduction is to permit a consistent performance improvement be obtained for all programs, not just those programs that have long innermost loops with straight-line loop bodies. More precisely, hierarchical reduction has three major effects:

1. The most important benefit of hierarchical reduction is that software pipelining can be applied to loops with conditional statements. This allows software pipelining to be applied to all innermost loops. Overlapping different iterations in a loop is an important source of parallelism.
2. Hierarchical reduction is also important in compacting long loop bodies containing conditional statements. In these long loop bodies, there is often much parallelism to be exploited within an iteration. Operations outside a conditional statement can move around and into the branches of the statement. Even branches of different conditional statements can be overlapped.
3. Hierarchical reduction also minimizes the penalty of short vectors, or loops with small number of iterations. The prolog and epilog of a loop can be overlapped with scalar operations outside the loop; the epilog of a loop can be overlapped with the prolog of the next loop; and, lastly, software pipelining can be applied even to an outer loop.

4. Evaluation

To provide and overall picture of the compiler's performance, we first give the statistics collected from a large sample of user programs. These performance figures show the effect of the software pipelining and hierarchical reduction techniques on complete programs. To provide more detailed information on the performance of software pipelining, we also include the performance of Livermore loops on a single Warp cell.

4.1. Performance of users' programs

The compiler for the Warp machine has been in use for about two years, and a large number of programs in robot navigation, low-level vision, and signal processing and scientific computing have been developed [2, 3]. Of these, a sample of 72 programs have been collected and analyzed [21]. Table 4-1 lists the performance of some representative programs, and the performance of all 72 programs is graphically presented in Figure 4-1.

Task All images are 512×512	Time (ms)	Mflops
100×100 matrix multiplication	25	79.4
512×512 complex FFT (1 dimension)	164	71.9
3×3 convolution	70	65.7
Hough transform	2113	42.2
Local selective averaging	406	39.2
Shortest path Warshall's algorithm (350 nodes, 10 iterations)	104	24.3
Roberts operator	192	15.2

Table 4-1: Performance on Warp array

All application programs in the experiment have compile-time loop bounds, and their execution speeds were determined statically by assuming that half of the data dependent branches in conditional statement were taken. All the programs in the sample are homogeneous code, that is, the same cell program is executed by all cells. Except for a short setup time at the beginning, these programs never stall on input or output. Therefore, the computation rate for each cell is simply one-tenth of the reported rate for the array.

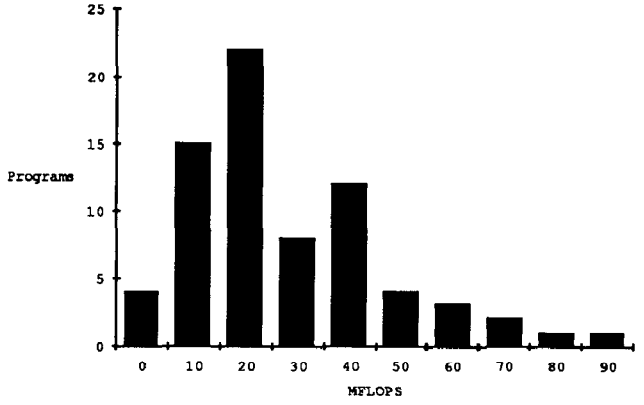

Figure 4-1: Performance of 72 users' programs

To study the significance of software pipelining and hierarchical reduction, we compare the performance obtained against that obtained by only compacting individual basic blocks. The speed up is shown in Figure 4-2. The average factor of increase in speed is three. Programs are classified according to whether they contain conditional statements. 42 of the 72 programs contain conditional statements. We observe that programs containing conditional statements are sped up more. The reason is that conditional statements break up the computation into small basic blocks, making code motions across basic blocks even more important.

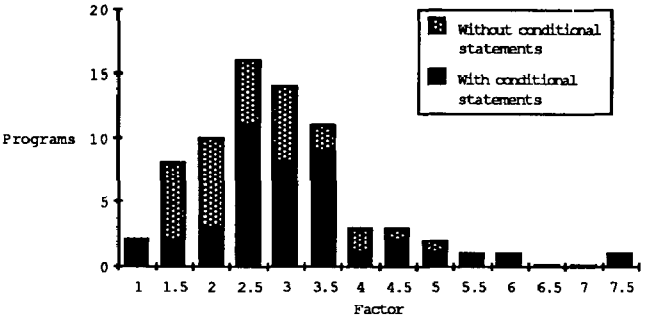

Figure 4-2: Speed up over locally compacted code

In this sample of programs, 75% of all the loops are scheduled with an initiation interval matching the theoretical lower bound. 93% of the loops containing no conditional statements or connected components are pipelined perfectly. The presence of conditional statements and connected components make calculating a tight lower bound for the initiation interval and finding an optimal schedule difficult. In particular, branches of a conditional statement are first compacted as much as possible, with no regard to the initiation interval of the loop. Software pipelining is then applied to the node representing the conditional statement, treating its operations as indivisible. This approach minimizes the length of the branches to avoid code explosion, but increases the minimum initiation interval of the loop. Of the 25% of the loops for which the achieved initiation interval is greater than the lower bound, the average efficiency is 75% [21].

4.2. Livermore loops

The performance of Livermore loops [23] on a single Warp cell is presented in Table 4-2. The Fortran programs were translated manually into the W2 syntax. (The control constructs of W2 are similar to those of Pascal.) The translation was straightforward except for kernels 15 and 16, which required the code be completely restructured. The INVERSE and SQRT functions expanded into 7 and 19 floating-point operations, respectively. The EXP function in loop 22 expanded into a calculation containing 19 conditional statements. The large numbers of conditional statements made the loop not pipelinable. In fact, the scheduler did not even attempt to pipeline this loop because the length of the loop (331 instructions) was beyond the threshold that it used to decide if pipelining was feasible. Loops 16 and 20 were also not pipelined, because the calculated lower bound on the initiation interval were within 99% of the length of the unpipelined loop.

Kernel	MFLOPS	Efficiency (lower bound)	Speed up
1	6.2	1.00	8.25
2*	2.5	0.75	3.25
3	1.4	1.00	2.71
4*	1.4	1.00	2.71
5	0.6	0.94	1.12
6*	1.4	1.00	2.86
7	7.9	1.00	6.00
8*	8.2	1.00	2.29
9	7.7	1.00	4.27
10	3.4	0.85	5.31
11	0.5	0.90	1.30
12	1.7	1.00	4.00
13	2.0	1.00	2.63
14**	2.2	1.00	3.32
15	5.9	0.85	5.50
16	0.3	1.00	1.00
17	0.8	1.00	1.20
18	7.5	0.97	3.70
19	0.7	1.00	1.24
20	0.9	0.99	1.00
21	3.0	1.00	6.00
22***	1.1	0.56	1.00
23	1.1	1.00	1.10
24	0.4	1.00	1.33
H-Mean	1.2		

* Compiler directives to disambiguate array references used

** Multiple loops were merged into one

***EXP function expanded into 19 IF statements

Table 4-2: Performance of Livermore loops

The MFLOPS rates given in the second column are for single-precision floating-point arithmetic. The third column contains lower bound figures on the efficiency of the software pipelining technique. As they were obtained by dividing the lower bound on the initiation interval by the achieved interval value, they represent a lower bound on the achieved efficiency. If a kernel contains multiple loops, the figure given is the mean calculated by weighing each loop by its execution time. The speed up factors in the fourth column are the ratios of the execution time between an unpipelined and a pipelined kernel. All branches of conditional statements were assumed to be taken half the time.

The performance of the Livermore loops is consistent with that of the users' programs. Except for kernel 22, which has an extraordinary amount of conditional branching due to the particular EXP library function, near-optimal, and often times, optimal code is obtained. The MFLOPS rates achieved for the different loops, however, vary greatly. This is due to the difference in the available parallelism within the iterations of a loop.

There are two major factors that determine the maximum achievable MFLOPS rate: data dependency and the critical resource bottleneck.

1. *Data dependency.* Consider the following loop:

```
FOR i := 0 TO n DO BEGIN
    a := a * b + c;
END;
```

Because of data dependency, each multiplication and addition must be performed serially. As additions and multiplications are seven-stage pipelined, the maximum computation rate achievable by the machine for this loop is only 0.7 MFLOPS.

2. *Critical resource bottleneck.* A program that does not contain any multiplication operations can at most sustain a 5 MFLOPS execution rate on a Warp cell, since the multiplier is idle all the time. In general, the MFLOPS achievable for a particular program is limited by the ratio of the total number of additions and multiplications in the computation to the use count of the most heavily used resource.

Inter-iteration data dependency, or recurrences, do not necessarily mean that the code is serialized. This is one important advantage that VLIW architectures have over vector machines. As long as there are other operations that can execute in parallel with the serial computation, a high computation rate can still be obtained.

5. Comparison with trace scheduling

The primary idea in trace scheduling [10] is to optimize the more frequently executed traces. The procedure is as follows: first, identify the most likely execution trace, then compact the instructions in the trace as if they belong to one big basic block. The large block size means that there is plenty of opportunity to find independent activities that can be executed in parallel. The second step is to add compensation code at the entrances and exits of the trace to restore the semantics of the original program for other traces. This process is then repeated until all traces whose probabilities of execution are above some threshold are scheduled. A straightforward scheduling technique is used for the rest of the traces.

The strength of trace scheduling is that all operations for the entire trace are scheduled together, and all legal code motions are permitted. In fact, all other forms of optimization, such as common subexpression elimination across all operations in the trace can be performed. On the other hand, major execution traces must exist for this scheduling technique to succeed. In trace scheduling, the more frequently executed traces are scheduled first. The code motions performed optimize the more frequently executed traces, at the expense of the less frequently executed ones. This may be a problem in data dependent conditional statements. Also, one major criticism of trace scheduling is the possibility of exponential code explosion [17, 19, 22, 26].

The major difference between our approach and trace scheduling is that we retain the control structure of the com-

putation. By retaining information on the control structure of the program, we can exploit the semantics of the different control constructs better, control the code motion and hence the code explosion. Another difference is that our scheduling algorithm is designed for block-structured constructs, whereas trace scheduling does not have similar restrictions. The following compares the two techniques in scheduling loop and conditional branching separately.

5.1. Loop branches

Trace scheduling is applied only to the body of a loop, that is, a major trace does not extend beyond the loop body boundary. To get enough parallelism in the trace, trace scheduling relies primarily on source code unrolling. At the end of each iteration in the original source is an exit out of the loop; the major trace is constructed by assuming that the exits off the loop are not taken. If the number of iterations is known at compile-time, then all but one exit off the loop are removed.

Software pipelining is more attractive than source code unrolling for two reasons. First, software pipelining offers the possibility of achieving optimal throughput. In unrolling, filling and draining the hardware pipelines at the beginning and the end of each iteration make optimal performance impossible. The second reason, a practical concern, is perhaps more important. In trace scheduling, the performance almost always improves as more iterations are unrolled. The degree of unrolling for a particular application often requires experimentation. As the degree of unrolling increases, so do the problem size and the final code size.

In software pipelining, the *object* code is sometimes unrolled at code emission time to implement modulo variable expansion. Therefore, the compilation time is unaffected. Furthermore, unlike source unrolling, there is an optimal degree of unrolling for each schedule, and can easily be determined when the schedule is complete.

5.2. Conditional statements

In the case of data dependent conditional statements, the premise that there is a most frequently executed trace is questionable. While it is easy to predict the outcome of a conditional branch at the end of an iteration in a loop, outcomes for all other branches are difficult to predict.

The generality of trace scheduling makes code explosion difficult to control. Some global code motions require operations scheduled in the main trace be duplicated in the less frequently executed traces. Since basic block boundaries are not visible when compacting a trace, code motions that require large amounts of copying, and may not even be significant in reducing the execution time, may be introduced.

Ellis showed that exponential code explosion can occur by reordering conditional statements that are data independent of each other [8]. Massive loop unrolling has a tendency to increase the number of possibly data independent conditional statements. Code explosion can be controlled by inserting additional constraints between branching operations. For example, Su et al. suggested restricting the motions of operations that are not on the critical path of the trace [26].

In our approach to scheduling conditional statements, the objective is to minimize the effect of conditional statements on parallel execution of other constructs. By modeling the conditional statement as one unit, we can software pipeline all innermost loops. The resources taken to execute the conditional statement may be as much as the sum of both branches. However, the amount of wasted cycles is bounded by the operations within the conditional statement.

6. Concluding remarks

This paper shows that near-optimal, and sometimes optimal, code can be generated for VLIW machines. We use a combination of two techniques: (1) software pipelining, a specialized scheduling technique for iterative constructs, and (2) hierarchical reduction, a simple, unified approach that allows multiple basic blocks to be manipulated like operations within a basic block. While software pipelining is the main reason for the speed up in programs, hierarchical reduction makes it possible to attain consistently good results on even programs containing conditional statements in innermost loops and innermost loops with small numbers of iterations. Our experience with the Warp compiler is that the generated code is comparable to, if not better than, handcrafted microcode.

The Warp processors do not have any specialized hardware support for software pipelining. Therefore, the results reported here are likely to apply to other data paths with similar degrees of parallelism. But what kind of performance can be obtained if we scale up the degree of parallelism and pipelining in the architecture? We observe that the limiting factor in the performance of Warp is the available parallelism among iterations in a loop. For those loops whose iterations are independent, scaling up the hardware is likely to give a similar factor of increase in performance. However, the speed of all other loops are limited by the cycle length in their precedence constraint graph. The control of all functional units by a central sequencer makes it difficult for VLIW architectures to exploit other forms of parallelism other than the parallelism within a loop. This suggests that there is a limit to the scalability of the VLIW architecture. Further experimentation is necessary to determine this limit.

Acknowledgments

The research reported in this paper is part of my Ph.D. thesis. I especially want to thank my thesis advisor, H. T. Kung, for his advice in the past years. I would like to thank all the members in the Warp project, and in particular, Thomas Gross for his effort in the W2 compiler. I also want to thank Jon Webb, C. H. Chang and P. S. Tseng for their help in obtaining the performance numbers.

References

1. Aiken, A. and Nicolau, A. Perfect Pipelining: A New Loop Parallelization Technique. Cornell University, Oct., 1987.

2. Annaratone, M., Bitz, F., Clune E., Kung H. T., Maulik, P., Ribas, H., Tseng, P., and Webb, J. Applications of Warp. Proc. Compcon Spring 87, San Francisco, Feb., 1987, pp. 272-275.

3. Annaratone, M., Bitz, F., Deutch, J., Hamey, L., Kung, H. T., Maulik P. C., Tseng, P., and Webb, J. A. Applications Experience on Warp. Proc. 1987 National Computer Conference, AFIPS, Chicago, June, 1987, pp. 149-158.

4. Annaratone, M., Arnould, E., Gross, T., Kung, H. T., Lam, M., Menzilcioglu, O. and Webb, J. A. "The Warp Computer: Architecture, Implementation and Performance". *IEEE Transactions on Computers C-36*, 12 (December 1987).

5. Colwell, R. P., Nix, R. P., O'Donnell, J. J., Papworth, D. B., and Rodman, P. K. . A VLIW Architecture for a Trace Scheduling Compiler. Proc. Second Intl. Conf. on Architectural Support for Programming Languages and Operating Systems, Oct., 1987, pp. 180-192.

6. Dantzig, G. B., Blattner, W. O. and Rao, M. R. All Shortest Routes from a Fixed Origin in a Graph. Theory of Graphs, Rome, July, 1967, pp. 85-90.

7. Ebcioglu, Kemal. A Compilation Technique for Software Pipelining of Loops with Conditional Jumps. Proc. 20th Annual Workshop on Microprogramming, Dec., 1987.

8. Ellis, John R. *Bulldog: A Compiler for VLIW Architectures.* Ph.D. Th., Yale University, 1985.

9. Fisher, J. A. *The Optimization of Horizontal Microcode Within and Beyond Basic Blocks: An Application of Processor Scheduling with Resources.* Ph.D. Th., New York Univ., Oct. 1979.

10. Fisher, J. A. "Trace Scheduling: A Technique for Global Microcode Compaction". *IEEE Trans. on Computers C-30*, 7 (July 1981), 478-490.

11. Fisher, J. A., Ellis, J. R., Ruttenberg, J. C. and Nicolau, A. Parallel Processing: A Smart Compiler and a Dumb Machine. Proc. ACM SIGPLAN '84 Symp. on Compiler Construction, Montreal, Canada, June, 1984, pp. 37-47.

12. Fisher, J. A., Landskov, D. and Shriver, B. D. Microcode Compaction: Looking Backward and Looking Forward. Proc. 1981 National Computer Conference, 1981, pp. 95-102.

13. Floyd, R. W. "Algorithm 97: Shortest Path". *Comm. ACM 5*, 6 (1962), 345.

14. Garey, Michael R. and Johnson, David S.. *Computers and Intractability A Guide to the Theory of NP-Completeness.* Freeman, 1979.

15. Gross, T. and Lam, M. Compilation for a High-performance Systolic Array. Proc. ACM SIGPLAN 86 Symposium on Compiler Construction, June, 1986, pp. 27-38.

16. Hsu, Peter. *Highly Concurrent Scalar Processing.* Ph.D. Th., University of Illinois at Urbana-Champaign, 1986.

17. Isoda, Sadahiro, Kobayashi, Yoshizumi, and Ishida, Toru. "Global Compaction of Horizontal Microprograms Based on the Generalized Data Dependency Graph". *IEEE Trans. on Computers c-32*, 10 (October 1983), 922-933.

18. Kuck, D. J., Kuhn, R. H., Padua, D. A., Leasure, B. and Wolfe, M. Dependence Graphs and Compiler Optimizations. Proc. ACM Symposium on Principles of Programming Languages, January, 1981, pp. 207-218.

19. Lah, J. and Atkin, E. Tree Compaction of Microprograms. Proc. 16th Annual Workshop on Microprogramming, Oct., 1982, pp. 23-33.

20. Lam, Monica. Compiler Optimizations for Asynchronous Systolic Array Programs. Proc. Fifteenth Annual ACM Symposium on Principles of Programming Languages, Jan., 1988.

21. Lam, Monica. *A Systolic Array Optimizing Compiler.* Ph.D. Th., Carnegie Mellon University, May 1987.

22. Linn, Joseph L. SRDAG Compaction - A Generalization of Trace Scheduling to Increase the Use of Global Context Information. Proc. 16th Annual Workshop on Microprogramming, 1983, pp. 11-22.

23. McMahon, F. H. Lawrence Livermore National Laboratory FORTRAN Kernels: MFLOPS.

24. Patel, Janak H. and Davidson, Edward S. Improving the Throughput of a Pipeline by Insertion of Delays. Proc. 3rd Annual Symposium on Computer Architecture, Jan., 1976, pp. 159-164.

25. Rau, B. R. and Glaeser, C. D. Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing. Proc. 14th Annual Workshop on Microprogramming, Oct., 1981, pp. 183-198.

26. Su, B., Ding, S. and Jin, L. An Improvement of Trace Scheduling for Global Microcode Compaction. Proc. 17th Annual Workshop in Microprogramming, Dec., 1984, pp. 78-85.

27. Su, B., Ding, S., Wang, J. and Xia, J. GURPR – A Method for Global Software Pipelining. Proc. 20th Annual Workshop on Microprogramming, Dec., 1987, pp. 88-96.

28. Su, B., Ding, S. and Xia, J. URPR – An Extension of URCR for Software Pipeline. Proc. 19th Annual Workshop on Microprogramming, Oct., 1986, pp. 104-108.

29. Tarjan, R. E. "Depth first search and linear graph algorithms". *SIAM J. Computing 1*, 2 (1972), 146-160.

30. Touzeau, R. F. A Fortran Compiler for the FPS-164 Scientific Computer. Proc. ACM SIGPLAN '84 Symp. on Compiler Construction, June, 1984, pp. 48-57.

31. Weiss, S. and Smith, J. E. A Study of Scalar Compilation Techniques for Pipelined Supercomputers. Proc. Second Intl. Conf. on Architectural Support for Programming Languages and Operating Systems, Oct., 1987, pp. 105-109.

32. Wood, Graham. Global Optimization of Microprograms Through Modular Control Constructs. Proc. 12th Annual Workshop in Microprogramming, 1979, pp. 1-6.