Compiler and Architectural Techniques for Improving the Effectiveness of VLIW Compilation

J.P. Grossman

Abstract

Effective VLIW compilation requires optimizing across basic block boundaries. In this mildly opinionated paper we survey a variety of techniques which allow the compiler to do so. We focus on ***trace scheduling****,* ***speculative execution****. and* ***software pipelining****. These techniques are effective, but cannot in general optimally schedule instructions for all traces of execution. To address this problem, we present* ***delayed issue****, a novel instruction issuing mechanism which allows a VLIW machine to perform out-of-order execution at a fraction of the cost of typical out-of-order issue hardware.*

1 Introduction

VLIW architectures offer high performance at a much lower cost than dynamic out-of-order superscalar processors. By allowing the compiler to directly schedule machine resource usage, the need for expensive instruction issue logic is obviated. Furthermore, while the enormous complexity of superscalar issue logic limits the number of instructions that can be issued simultaneously, VLIW machines can be built with a large number of functional units (e.g. up to 28 on the Multiflow TRACE [Colwell88]), allowing a much higher degree of instruction-level parallelism (ILP).

In computer architecture there tends to be a "conservation of complexity" effect whereby simpler hardware demands more complicated compilers. This is certainly the case with VLIW; sophisticated compilers are required in order to achieve the maximum possible performance. In particular, in order to make effective use of a large number of functional units it is necessary to perform optimizations across basic block boundaries, as the amount of parallelism available within basic blocks tends to be quite limited ([Smith89], [Wall91]).

In this paper we survey some common compiler and architectural techniques for increasing program ILP and making more effective use of the available hardware resources. We begin by discussing **Trace Scheduling** ([Fisher81], [Fisher83]) in section 2. In trace scheduling, compilation proceeds by selecting a likely path of execution (called a trace). This trace is compacted by performing code motion without regard to basic block boundaries. To preserve correct execution, off-trace compensation code must be inserted when instructions are moved past a split or a join. When the trace is fully optimized, a new trace is selected; this process repeats until all traces of execution have been compacted.

A limitation of trace scheduling is that it cannot in general migrate instructions above conditional branches. However, such code motions are possible on machines that provide support for **speculative execution** [Smith90]. In section 3 we describe various architectural mechanisms that have been proposed to support speculative execution. We review the results of [Chang95] in which three different speculative execution models are compared.

Another limitation of trace scheduling is that it does not handle loops very well; in [Fisher81] no attempt is made to migrate instructions across the conditional loopback branch. A much more effective technique for compiling loops is **Software Pipelining** ([Rau81], [Touzeau84], [Lam88]) in which multiple iterations of a single loop are overlapped in software. In section 4 we review a number of different software pipelining algorithms as well as some architectural mechanisms which have been proposed and/or implemented.

While trace scheduling, speculative execution and software pipelining are effective techniques, they require that all parallelism be explicitly discovered by the compiler. In many cases it is impossible or impractical for the compiler to optimize all possible traces and loops. Modern processors address this problem with superscalar, dynamic out-of-order issue logic which allows instructions to be issued in parallel whether or not they were scheduled together by the compiler. However, this requires complex hardware support.

In section 5 we present **Delayed Issue**, a novel technique which allows instructions to be executed out-of-order without the hardware complexity of dynamic out-of-order issue. Instructions are inserted into per-functional unit delay queues using delays specified by the compiler. Instructions within a queue are issued in order; out of order execution results from different instructions being inserted into the queues at various delays. In addition to improving performance, delayed issue reduces code expansion due to trace scheduling and software pipelining.

2 Trace Scheduling and Related Work

Trace scheduling was introduced by Joseph Fisher in [Fisher81]. Originally conceived as an algorithm for compacting microcode, it is a very general technique that can produce highly optimized code for any VLIW, scalar or superscaler architecture. The motivation behind trace scheduling is to be able to optimize across basic block boundaries. We will begin our discussion of trace scheduling by reviewing **list scheduling**, a technique for scheduling instructions within a single basic block. We then show how this algorithm is extended to schedule traces that span multiple blocks, we explain how loops are handled, and we outline some related work.

2.1 List Scheduling

The instructions within a basic block can be represented as a directed acyclic graph called the **dependency graph**. An edge exists from instruction I to instruction J whenever J has a dependence on I. In this case we write $I < J$. The problem of finding an optimal schedule is NP-complete, but list scheduling is a simple heuristic which seems to perform within a few percent of optimal in practice [Fisher81].

List scheduling works by placing the instructions in a linear order $I_1, I_2, \ldots, I_n$ and then scheduling them one at a time in that order. Each instruction is scheduled at the earliest possible time taking into account both its dependencies and its resource requirements in relation to the already-scheduled instructions. The linear order is chosen as follows: define a *path* to be a sequence of instructions, each of which depends on the previous one. The *height* $h(I)$ of instruction I is the length of the longest path starting at I. The order is then the reverse order of the heights, that is, the order is chosen so that $h(I_1) = h(I_2) = \ldots = h(I_n)$.

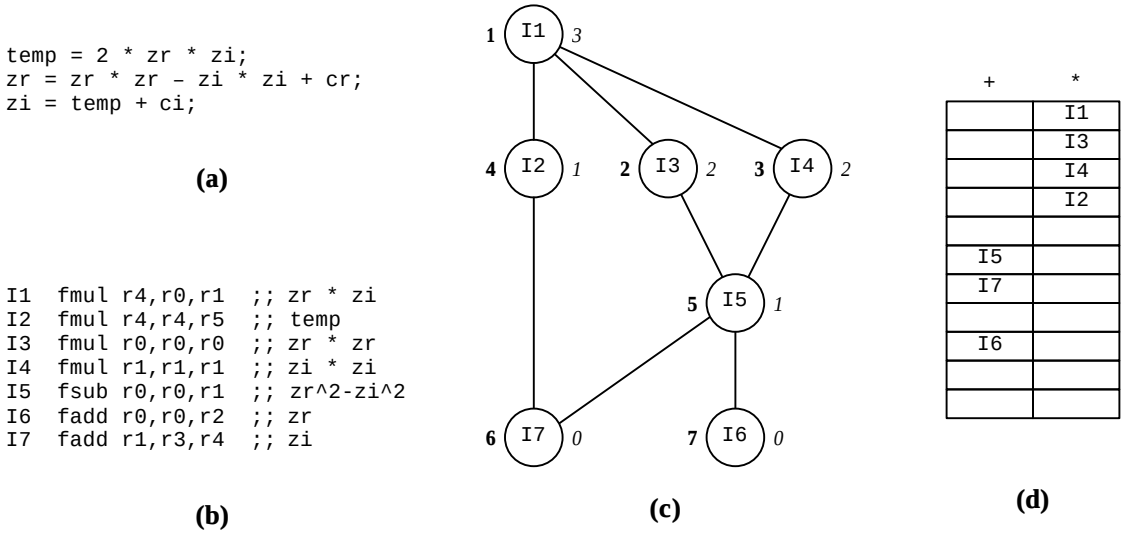


Figure 1: (a) Small basic block. **(b)** Assembly code. **(c)** DAG. Solid lines represent flow dependencies, dashed lines represent anti-dependencies. Node heights are shown in italics, scheduling order is shown in bold. **(d)** Generated schedule assuming a machine with one floating point add and one floating point multiply, each with a three cycle latency.

Figure 1 shows an example of list scheduling being applied to the instructions of a basic block. Figures 1a and 1b show the C code and equivalent assembly for one iteration of a mandelbrot set calculation. Figure 1c shows the corresponding directed acyclic graph, along with the height and scheduling order for each node. Figure 1d shows the schedule that would be generated for a simple machine having a single floating point multiply and a single floating point add.

2.2 Trace Scheduling

A general program can be described by a directed graph whose nodes are the basic blocks (called the **program flow graph**), as shown in figure 2. A **trace** is an acyclic path through this graph (also shown in figure 2). The goal of trace scheduling is to be able to optimize an entire trace in the same way that list scheduling can optimize a single basic block. The difficulty is that if list scheduling is naively applied to the trace, instructions can move past splits (conditional branches) and joins (instructions with off-trace predecessors). This results in extra or missing instructions on traces other than the one being optimized; as a result these other traces will generally no longer execute correctly.

A key observation is that we can solve this problem by adding off-trace compensation code which restores the original semantics of program execution. To see how this works, first note that it suffices to describe the compensation code that must be created when two adjacent instructions are transposed, as any instruction reordering can be described as a finite sequence of transpositions. Suppose, then, that we have a sequence of three consecutive instructions A, B, C in the trace and we wish to transpose A and B. There are four cases to consider based on the instruction B: (a) B is neither a split nor a join, (b) B is a split, (c) B is a join, (d) B is both a split and a join. The compensation code for these four cases is shown in figure 3 in the form of graph transformations.

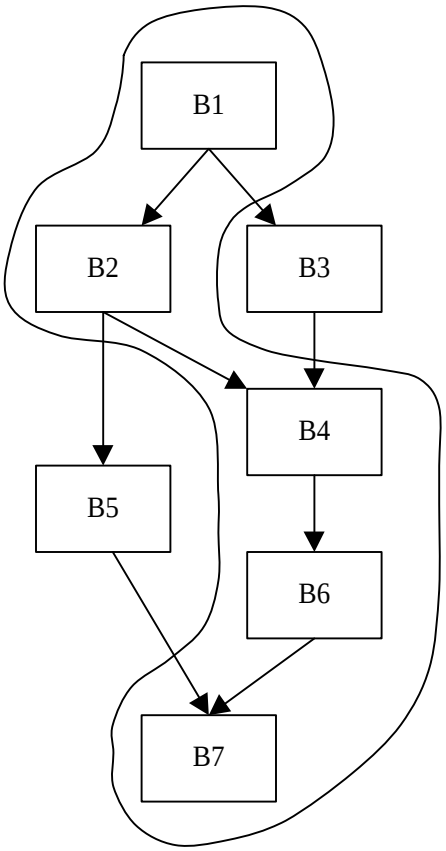


Figure 2: Directed graph of basic blocks. The dashed outline shows one trace which passes through B1, B2, B4, B6, B7.

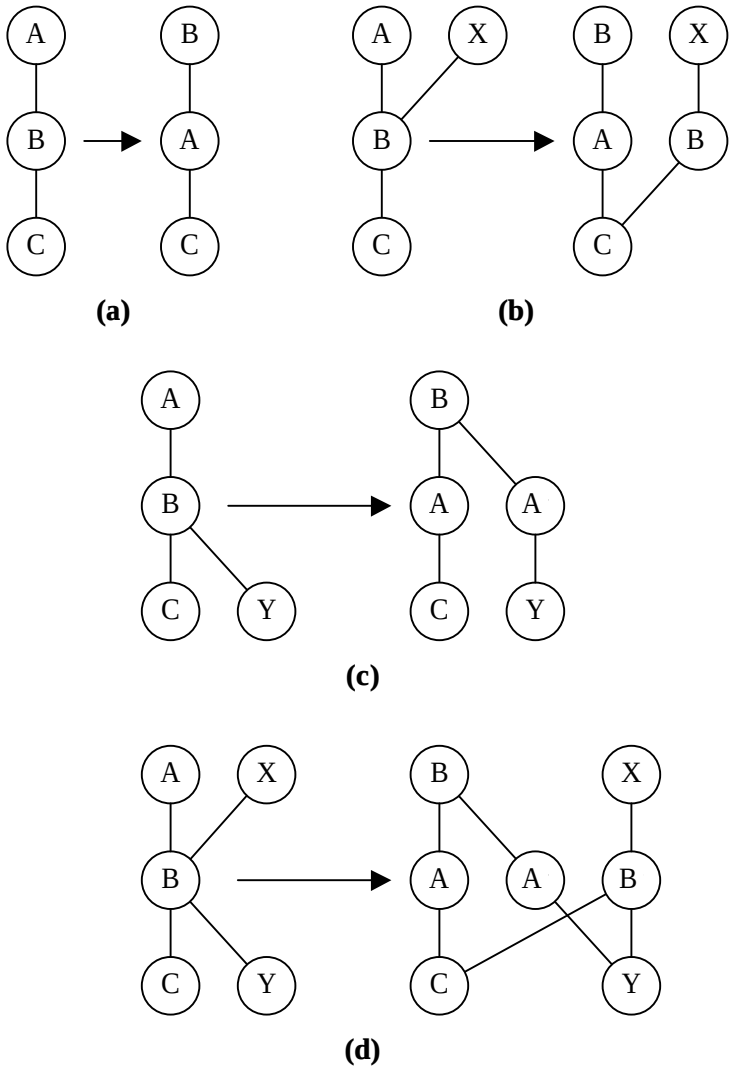


Figure 3: Compensation code is added to preserve program semantics when instructions A, B are reordered. X, Y are off-trace instructions.

In all cases the compensation code consists of copies of A and B, shown as A'and B'. Whether or not A is a split or a join is immaterial. If A is a join, the instruction which jumps to A is simply modified to jump to B after the transformation is applied. If A is a split, then in general A and B cannot be reordered (this is discussed further in section 3), but if they can then the transformations of figure 3 can be applied without modification. Finally, it is not a problem if multiple off-trace instructions jump to B. This situation can be handled either by generating multiple copies of the compensation code, or by redirecting these jumps through a single new intermediate jump to B.

2.3 The Trace Scheduling Algorithm for Acyclic Graphs

The algorithm presented in [Fisher81] for trace scheduling programs with acyclic flow graphs is as follows. First, the most likely trace is selected, that is, the trace which at compile time is predicted to have the highest probability of being executed. The dependency graph for the entire trace is generated. Dependency edges are added from each conditional branch to all subsequent instructions

which cannot safely be moved above the branch. In [Fisher81] it is deemed unsafe to move an instruction *I* above a branch *B* if and only if *I* writes to some register which is read by some off-trace successor of *B*. In section 3 we will see that this condition alone is insufficient in a general setting.

Once this augmented dependency graph has been created, list scheduling is used to compact the entire trace. When the trace is scheduled, "bookkeeping" is performed to insert compensation code wherever necessary to preserve program semantics (although it would seem to make more sense to perform scheduling and bookkeeping simultaneously in order to reduce complexity, the two operations are presented as separate steps in [Fisher81]). Finally, the entire process is repeated until all traces have been compacted.

2.4 Trace Scheduling Code Containing Loops

In [Fisher81] a hierarchical approach is used to trace schedule loops. It is assumed that the program flow graph is **reducible** [Hecht77], which means that any two loops are either disjoint or nested. Under this assumption it is possible to arrange the loops in a sequence L_1, L_2, ... , L_n such that if i < j then either L_i and L_j are disjoint or L_i is contained in L_j (figure 4a). The loops are then compacted in this order using trace scheduling; after each loop is compacted it is replaced by a single node, called a *loop representative*, which represents the aggregate resource usage of the loop.

To schedule a loop using trace scheduling, the back-edge is removed in the loop's flow graph. By assumption on the ordering of the loops, all loops nested in the loop being scheduled have already been replaced by single nodes (figure 4b). Hence, once the back-edge is removed the flow graph is acyclic, and the loop may be compacted using the algorithm of section 2.3. When all the loops have been compacted and replaced by loop representatives the resulting program flow graph is acyclic (figure 4c) and can therefore be compacted using trace scheduling.

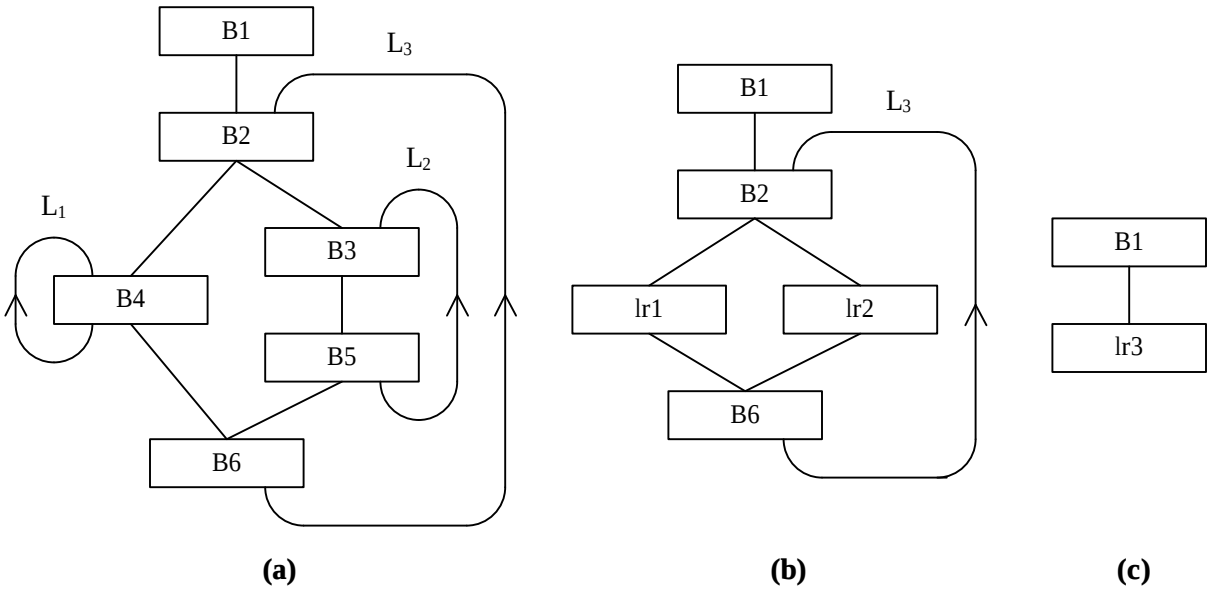


Figure 4: (a) Reducible flow graph. **(b)** Graph after L_1 and L_2 are compacted. **(c)** After L_3 is compacted

There is one caveat when loops are scheduled in this manner. Ordinarily the trace scheduler will allow two operations to be scheduled simultaneously if they are **resource compatible**, that is, if no single resource is required by both operations. However, if one of the operations is a loop representative, then this condition must be extended to require that the other operation be loop-invariant with respect to the loop, since it will be executed once for each loop iteration. In no case are two loop representatives considered to be resource compatible.

2.5 Suppression of Compensation Code

One of the drawbacks of trace scheduling is that the bookkeeping phase can cause exponential code growth ([Isoda93], [Lah83], [Linn83]). A restricted version of trace scheduling is presented in [Freuden94] which reduces code growth by avoiding and suppressing compensation code. Compensation code is avoided in two ways: by disallowing downward code motion past splits (figure 3c,d), and by disallowing upward code motion past joins with fan-in greater than four. In [Freuden94] it was found that these restrictions have small impact on program performance but reduce code growth considerably. Compensation code is suppressed by finding and eliminating redundant duplicates of instructions in the final program schedule. The algorithm used to accomplish this is based on work by Ellis in [Ellis85].

2.6 Superblock Scheduling

In [Hwu93] the superblock is proposed as a mechanism for facilitating trace scheduling compilation. A superblock is simply a trace with all joins removed via tail duplication (figure 5). It is argued that the primary benefit of superblocks is avoidance of the "complex bookkeeping" involved with moving code past joins. This argument is not very compelling in light of the simplicity of the transformations of figure 3. Moreover, superblock formation can in a sense be viewed as "worst case bookkeeping" in which the maximum amount of compensation code due to the transformation of figure 3b is generated at the outset. A more convincing argument for the use of superblocks is that they simplify the application of certain optimizations such as copy propagation [Hwu93].

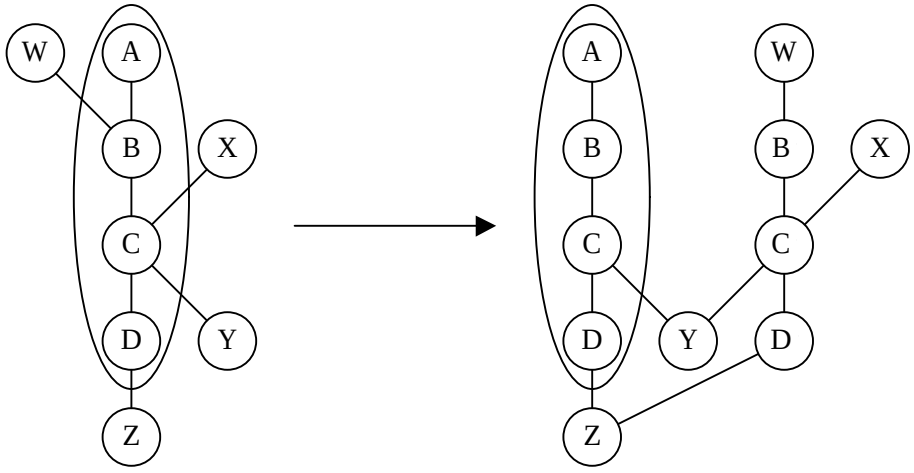


Figure 5: Tail duplication.

2.7 Predicated Execution

Trace scheduling can be viewed as a method for dealing with the difficulties presented by branches. An alternate approach is to attempt to eliminate branches altogether. **Predicated Execution** ([Hsu86], [Mahlke92a], [Mahlke95]) is a mechanism which permits some branches to be removed by allowing execution of instructions to be conditional on the values of hardware defined predicates. For example, the branch in an if-then-else structure can be replaced with an instruction which defines a predicate. The entire clause then becomes a single basic block in which instructions in the *then* clause are conditional on the predicate being true, and the instructions in the *else* clause are conditional on the predicate being false. Predicated execution can simplify compilation and avoid branch mispredict penalties as the expense of some wasted instruction bandwidth (since predicated instructions are still fetched whether or not they are executed). In [Mahlke92a], [Mahlke95] and [Hara96] it is shown that the presence of predication mechanisms can significantly improve performance.

3 Speculative Execution

The success of trace scheduling depends heavily on its ability to move instructions past splits and joins in order to discover increased levels of parallelism. While the algorithm allows code to move freely past joins and downwards past splits, it is not in general safe to move instructions upwards past a conditional branch as this can destroy the intended semantics of the program. Three specific restrictions on upward motion of an instruction *I* past a branch *B* are well known ([Chang91], [Huang94], [Chang95], [Smith90]):

1. *I* must not cause an exception.
2. *I* must not overwrite the value of a register that is needed by some other successor of *B.*
3. *I* must not alter system memory.

On conventional architectures, *I* can only be moved upwards past *B* if all three of the above conditions are met. However, it is possible to provide hardware support for **speculative execution** which allows these restrictions to be relaxed. When an instruction is moved upwards past a branch it is speculatively executed, and the hardware guarantees that the instruction will produce no visible side effects before the branch is resolved. If the branch is taken, the instruction is squashed. If the branch is not taken, the instruction is allowed to commit.

In [Chang95] a comparison is made between three different architectural models with varying degrees of support for speculative execution. The models are as follows:

Restricted model. No hardware support is provided. Code motion is subject to all three restrictions.

General model. The hardware provides a set of non-trapping instructions. This allows restriction (1) to be relaxed, but restrictions (2) and (3) are still enforced. An example of an architecture which falls into this category is the Multiflow TRACE which provides a set of non-trapping load instructions [Colwell87].

Boosting model. Full hardware support is provided for speculative execution. Exceptions caused by speculatively executing instructions are suppressed, a "shadow register file" holds results of speculative instructions until the branch is resolved, and similarly a "shadow store buffer" holds values speculatively written to memory until the branch is resolved. All three restrictions are relaxed, and arbitrary upward code motion past a single branch instruction is permitted. This model is based on the TORCH architecture proposed in [Smith90].

The simulation results of [Chang95] show that the performance of the general model is significantly better than the performance of the restricted model. However, the boosting model provides little additional performance advantage. There are two reasons why these results are not surprising. First, load operations tend to be at the head of critical paths while store operations tend to be at the tails, so the benefits of being able to perform early loads are much greater than the benefits of performing early stores. Thus, we would expect a set of non-trapping loads to provide a much more noticeable gain in performance than a shadow store buffer mechanism for performing speculative stores. Second, it is often possible to work around restriction (2) in software via register renaming, loop induction variable expansion, and accumulator expansion.

These results are a rare treat; usually in computer architecture it is the case that avoiding complexity implies sacrificing performance. In this case, however, we see that the relatively inexpensive general model performs almost as well as the much more complicated model providing support for fully speculative execution.

3.1 Recovering Exceptions

In both the general and the boosting models, exceptions that would normally occur must be suppressed when an instruction is speculatively executed. Often it is desirable to recover these exceptions once the branch has been resolved. In [Mahlke92b] it is proposed that *sentinels* be used to explicitly check for exceptions. When an trapping instruction is moved upwards past a conditional branch, it is divided into two parts: a non-trapping version of the same instruction which is migrated, and a sentinel instruction that remains at the original location. If the non-trapping instruction causes an exception condition, an *exception tag* is set in the destination register of the instruction. When and if the sentinel instruction is later executed, it explicitly checks the register tag to see if an exception should be raised.

An alternate mechanism is described in [Smith90] which does not required extra instructions and has the advantage of supporting precise interrupts. If an exception occurs while executing a speculative instruction, the exception is postponed until the branch is resolved. If the branch resolves in the direction of the speculative instruction, then the shadow structures are invalidated and the speculative instructions are re-executed non-speculatively. When the exception causing instruction is re-executed, a sequential interrupt occurs at the appropriate time.

```
for (i = 0 ; i < N ; i++)
{
        y[i] = x[i] * x[i] + c;
}
```

(a) Simple loop

```
L0:     LOAD  a2,[a0]; PADD a0,a0,1
        FMUL  a3,a2,a2
        FADD  a4,a3,c
        STORE a4,[a1]; PADD a1,a1,1; LOOP L0
```

(b) Assembly code

memory	addition			multiplication			integer/pointer
LOAD							PADD
				FMUL			
					FMUL		
						FMUL	
	FADD						
		FADD					
			FADD				
STORE							PADD

(c) Scoreboard of hardware resource utilization

memory	addition			multiplication			integer/pointer
LOAD	*FADD*		**FADD**		***FMUL***		PADD
STORE		*FADD*		FMUL		***FMUL***	**PADD**
LOAD	***FADD***		*FADD*		FMUL		**PADD**
STORE		***FADD***		**FMUL**		FMUL	*PADD*
LOAD	FADD		***FADD***		**FMUL**		*PADD*
STORE		FADD		*FMUL*		**FMUL**	***PADD***
LOAD	**FADD**		FADD		*FMUL*		***PADD***
STORE		**FADD**		***FMUL***		*FMUL*	PADD

(d) Scoreboard for pipelined loop – different fonts correspond to different iterations

Figure 6: Simple loop before and after software pipelining

4 Software Pipelining

Another shortcoming of trace scheduling is that it handles loops quite poorly. The strength of trace scheduling is its ability to optimize across branches, yet in [Fisher81] no attempt is made to optimize across the loop-back branch of a loop. One can imagine trying to address this problem by unrolling the loop a number of times and then applying trace scheduling to the unrolled loop [Freuden94]. Clearly as the loop is unrolled more times, better schedules will be obtained using trace scheduling. In the limit, one might expect a periodic structure to develop that could be used as the kernel for a more compact loop. Since trace scheduling moves instructions across branches, each iteration of this kernel will contain instructions from multiple iterations of the original loop.

The central idea of software pipelining is to abandon trace scheduling and instead search directly for such a compact loop kernel. The code is scheduled such that new iterations are initiated at a fixed interval, called the *initiation interval*, which in the literature has been given the rather unfortunate symbol II (we despise this notation, but we will use it). As an example, consider the simple loop shown in figure 6a. Assuming a VLIW architecture with a memory unit, a floating point addition unit, a floating point multiplication unit, an integer arithmetic unit, and a hardware looping construct, the equivalent assembly code is shown in figure 3b. If floating point addition and multiplication take 3 cycles, integer operations take 1 cycle, and the data resides in the cache so that memory references take 1 cycle, then figure 3c depicts a scoreboard of the hardware resource utilization, and we see that the II is 8. We can pipeline the loop to achieve an II of 2, as shown in figure 3d.

The goal of any software pipelining algorithm is to maximize performance by minimizing II. There have been a plethora of algorithms proposed and evaluated in the literature. A full list is well beyond the scope of this paper; see [Allan95] for a comprehensive review. We will instead focus on the algorithm presented in [Lam88], which we develop in the next few sections. We then briefly describe some interesting improvements to this algorithm as well as some architectural support for software pipelining.

4.1 Problem Statement

The loop to be pipelined can be represented as a directed graph whose nodes are operations and whose edges are dependencies between them. Dependencies can be either *intra-iteration* or *inter-iteration*, since the execution of an instruction can depend on other instructions from the same or previous iterations. If the instruction v must execute d cycles after instruction u from the p^{th} previous iteration, then we assign to the edge (u, v) the label $\langle p, d \rangle$. Figure 7 shows the graph corresponding to the loop of figure 6.

The problem of finding a schedule for a given II is the problem of assigning each instruction u to a time slot $s(u)$ subject to two contraints:

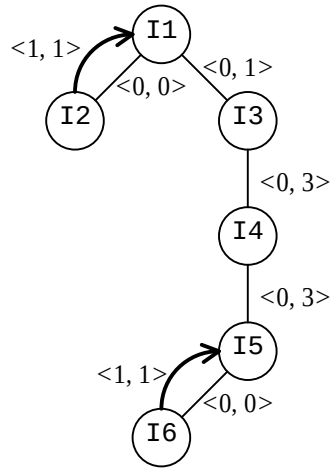
Resource constraints. Since a new iteration is initiated every II cycles, in the steady state instructions in time slots k, $k + II$, $k + 2 \cdot II$, ... will execute simultaneously for each k. Thus, for each k this set of instructions must be resource compatible.

Precedence constraints. If the edge (u, v) has label $\langle p, d \rangle$, then $s(v) - s(u) = d - p \cdot II$.

4.2 Selection of II

Computation of the minimum possible II is NP-complete [Lam88], so a heuristic is required. The approach in [Lam88] is to first find a lower bound for II, then attempt to find a schedule for this value of II. If this fails, then II is increased by one and the process repeats until a schedule is found. Such a linear search is preferred over a binary search as schedulability is not monotonic, that is, it is possible for a schedule to exist with II = n but not II = n+1 [Lam89].

```
I1  LOAD  a2, [a0]
I2  PADD  a0, a0, 1
I3  FMUL  a3, a2, a2
I4  FADD  a4, a3, c
I5  STORE a4, [a1]
I6  PADD  a1, a1, 1
```



(a) **(b)**

Figure 7: (a) Instructions in loop body. **(b)** Dependency graph. Inter-iteration dependencies in bold.

The two constraints of the previous section can be used to obtain lower bounds for II. If n copies of a given resource are available, then the resource constraint requires that the utilization of this resource on each cycle be at most n. Since in the steady state each instruction executes once every II cycles, it follows that the total utilization of this resource by the entire loop must be at most $n \cdot$II, giving the first lower bound for II. Next, for each cycle c in the graph, we can sum the precedence constraints over the edges of c. Since c is a cycle the left side of the inequality is zero and we are left with $d(c) - p(c) \cdot \text{II} = 0$. Thus, $max \lceil d(c) / p(c) \rceil$ is a second lower bound for II, where the maximum is taken over all cycles in the dependency graph.

4.3 Scheduling Without Conditional Branches

Assume for the time being that the loop contains no conditional branches; we will deal with this case in section 4.6. For a given II, scheduling proceeds as follows. First, the strongly connected components (SCC) of the graph are found [Tarjan72] (these are the maximal components for which there is a directed path from every node to every other node). Each SCC is scheduled individually using list scheduling, where the *intra-iteration* edges of the SCC are used to produce a topological ordering of the nodes. The original graph is then reduced by representing each SCC as a single node with resource usage determined by the schedule for that SCC. The resulting reduced graph is acyclic (if it contained a cycle then the entire cycle would be a SCC), so it admits a topological sorting which is used for a final list scheduling. If at any stage a node cannot be scheduled in II consecutive time slots due to resource constraints then the algorithm aborts and the entire process is restarted with the next II.

The above algorithm is arbitrary in many respects. It represents an attempt to adapt a successful scheduling technique (list scheduling) to cyclic graphs. To first order, the justification for the algorithm given in [Lam89] is that "it works", producing optimal schedules for many loops. The following much simpler alternate algorithm is presented in [Rau94]. First, a total ordering is imposed on all instructions in the loop which is used to define instruction priorities (we omit the details of this ordering). The scheduling function then repeatedly selects the highest priority unscheduled instruction and schedules it. If it is impossible to schedule the instruction without conflicts, then rather than abort as in [Lam88], the instruction is forcibly scheduled and displaces (unschedules) whichever instructions it conflicts with. Thus, scheduling decisions are 'soft', and a given instruction may be scheduled and unscheduled multiple times. The algorithm aborts and increases II when the total number of scheduled instructions (counting repeat schedulings) exceeds a specified budget. This algorithm was also found to give near-optimal performance.

4.4 Modulo Variable Expansion

Suppose that when the instructions are scheduled, there is a value which is defined at some point and used n time steps later. This would seem to impose the restriction II = n, as otherwise the next instance of the instruction defining the value II cycles later would overwrite the value before it could be used. We can solve this problem by using k different registers to store the variable, where on iteration m we use the $(m \bmod k)^{th}$ register. In this case, the value is not overwritten for $k \cdot$II timesteps, so as long as $k \cdot \text{II} = n$ the program will execute correctly.

The integer n is called the *lifetime* of the value. In order to use a different register for k consecutive iterations we need to unroll the loop k times (since a single copy of an instruction

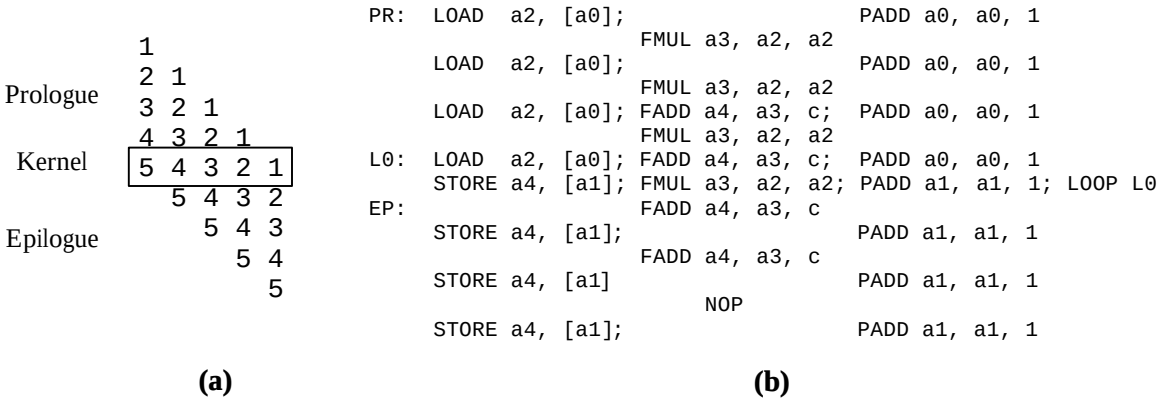


Figure 8: Prologue and Epilogue

cannot access different registers on different cycles without hardware support). The minimum degree of unrolling required is therefore $U = max \lceil n/\mathrm{II} \rceil$, where the maximum is taken over all value lifetimes n. In [Lam88] it is asserted that the number of registers used to store a value must evenly divide U, but this is false. For example, if $U = 5$, $II = 3$ and $n = 4$ for some value, then we can use 3 registers a, b, c to store the value in the repeating sequence $a, b, c, a, b, a, b, c, a, b, \ldots$

4.5 Prologue and Epilogue

If the II cycle kernel which is generated contains m overlapping iterations, then we need a *prologue* to initiate m-1 iterations before entering the kernel and an *epilogue* to complete m-1 iterations after exiting the kernel. This is depicted graphically in figure 8a, and in figure 8b we show the prologue, kernel and epilogue for the loop of figure 6. Note that this results in considerable code expansion; the original loop contains 6 instructions, whereas the pipelined loop together with the prologue and epilogue contains 24 instructions, a 300% increase in code size.

4.6 Hierarchical Reduction

In [Lam88] a single mechanism is proposed to handle, with minor differences, both loops and conditionals. As with trace scheduling, the idea is to schedule these program constructs separately, then represent them as a single node having the aggregate resource usage and scheduling constraints of the construct. The loop is reduced in this bottom-up manner until only one node remains (the final schedule for the loop).

To schedule conditional statements, the two branches are first scheduled independently. The entire conditional is then represented as a single node whose resource usage at each time step is the union of the resource usages of the two branches. The length of the node in the maximum of the lengths of the branches. After the entire loop is scheduled, the conditional branch is reinserted, and any instructions scheduled in parallel with the conditional node are duplicated in both branches.

Loops are handled similarly. The difference is that while instructions should be allowed to be scheduled in parallel with the prologue and epilogue of the reduced loop, they should not be allowed to overlap the loop kernel. Thus, all resources in the kernel are marked as used.

4.7 Related Work

An enormous literature exists on the subject of software pipelining. The following sections outline some of the more interesting ideas which are based on the algorithm in [Lam88]. The majority of these seek to improve the performance of software pipelining in the presence of conditional branches.

4.7.1 Fractional II

The lower bounds on II described in section 4.2 are not in general integers. Thus, even when it is reported that a schedule is found with the "optimal" II, better performance may be possible if II is somehow allowed to be a fractional value. This can in effect be accomplished by unrolling the loop a number of times before applying software pipelining. If the loop is unrolled k times and an initiation interval of II' is found for the unrolled loop, then in a sense we have II = II'/k as k iterations are initiated every II' cycles. In [Lavery95] it is shown that this technique can produce surprisingly large speedups, particularly when the theoretical lower bounds on II are less than 1.

4.7.2 Predicated Execution and Enhanced Modulo Scheduling

One of the objections to hierarchical reduction of conditional constructs is its rigidity. Since conditionals are scheduled separately without any regard to the resource requirements of the rest of the loop, the resulting reduced nodes may have complex resource usages that create artificial conflicts and may prevent a schedule from being found. In [Warter93a] another approach is suggested wherein conditional branches are replaced by predicated execution, as described in section 2.7. The advantage of this method is that it allows the clause instructions to be scheduled at the same time as the rest of the loop. This provides additional flexibility and increases the probability of finding a schedule. The disadvantage is that the resource requirements of the conditional construct are now the *sum* rather than the union of the requirements of the individual clauses. This decreases the likelihood of finding a schedule and may even raise the theoretical lower bound on II. Somewhat surprisingly perhaps, in [Warter93a] it is found that predicated execution provides a 25-50% improvement, with larger simulated improvements on architectures with higher instruction issue rates.

In [Warter92] this work is extended by applying *reverse-if conversion* [Warter93b] to the resulting schedule. This is a transformation which converts predicated code to unpredicated code with conditional branches. Scheduling in this manner provides two advantages over simple predicated execution. First, since separate streams of execution are generated, resource usage is again given by a union rather than a sum. Second, hardware support for predicated execution is not required.

4.7.3 Multiple Initiation Intervals

A disadvantage of the techniques discussed thus far is that II is resource-constrained from below by the most resource-intensive path of execution through the loop, even if this path is rarely followed during program execution. In [Warter95] this problem is addressed by constructing multiple-II kernels. The scheduling procedure is reminiscent of trace scheduling; the most likely trace of execution is chosen and scheduled separately with the smallest possible II. The next trace is scheduled on top of this trace, filling in holes where possible and extending beyond the end to produce a schedule with a slightly larger II. Predicated execution is used where necessary to ensure

that instructions are executed only in the traces to which they belong. The result of this procedure is a multiple-II kernel in which the most likely trace is highly optimized at the expense of less likely traces.

An extremely aggressive multiple-II algorithm which provides simultaneously optimal performance on *all* traces is presented in [Stoodley96a] and [Stoodley96b]. The **All Paths Pipelining** (APP) algorithm schedules each trace separately. The (potentially large number of) kernels are then combined using a fairly sophisticated bookkeeping/pattern matching algorithm. The resulting schedule yields optimal performance for consecutive iterations that follow the same trace, and near optimal performance when a transition is made from one trace to another. No hardware support of any kind is required. The primary disadvantage of APP is the potential for exponential code explosion due to the large number of traces, but for many loops the amount of code expansion is found to be surprisingly small.

4.8 Architectural Support for Software Pipelining

Hardware support for software pipelining is typically used to eliminate some or all of the code expansion normally associated with this technique. One way to accomplish this is to provide support for modulo variable expansion so that the same effect can be achieved without multiple copies of the loop code. Specifically, a mechanism must be provided for accessing different pieces of data using the same register name so that a single copy of the code can be used for each loop iteration. In [Su90] a horrible architectural structure is proposed in which data is physically shifted between registers arranged in an array. In [Ugurdag93] an equally distasteful mechanism is described in which functional units are connected to a set of register files through large reconfigurable crossbars. These are two unfortunate examples of what can happen when designers fail to realize that "It's the wires, stupid" [Knight97].

A more practical solution found in both the Cydra 5 and the (ridiculously named) Itanium is **register rotation** ([Dehnert89], [Doshi99]). A certain set of registers are specified as rotating registers. When an instruction references one of these registers, a hardware index is added to the register number specified in the instruction (modulo the number of rotating registers). The index is incremented at the start of each new iteration. Thus, the same instruction will refer to different registers on each cycle.

Both the Cydra 5 and the Itanium also provide a predicated execution mechanism that allows the prologue and epilogue to be eliminated. The instructions in the loop kernel are predicated on a set of *shifting predicates*, p_0, p_1, ... , p_n. Instructions in the kernel which form part of the k^{th} overlapped iteration are predicated on p_k. When the loop is first entered, p_0 is set to **true** and p_1, ... , p_n are set to **false**. Thus, only instructions which are part of the first iteration will be executed. At the start of each new iteration, the predicates are shifted up by one index, so that with each loop-back branch a new iteration starts while the previous iterations continue, until a steady state is reached. Finally, at the end of the loop p_0 is set to **false** so that no new iterations are started. With each loop-back the predicates shift up so that one more iteration terminates, until all the p_i are **false**.

5 Delayed Issue

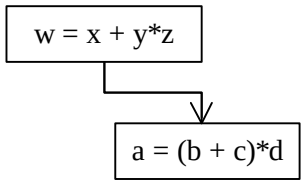
Trace scheduling and software pipelining are both static techniques in that they require compile-time decisions to be made regarding instruction ordering. However, in many cases the compiler may not be able to optimally schedule instructions as:

- It is difficult to optimize across basic block boundaries when the program flow graph has high fan-in (e.g. a subroutine) or high fan-out (e.g. an indexed or indirect jump)
- Some optimizations may be suppressed for simplicity [Stoodley96b] or to avoid unwanted code expansion [Freuden94]

In these instances, program execution can benefit from out-of-order execution which has the effect of dynamically compacting the instruction stream. This is an effective and well-known mechanism that exists in many commercial processors, but it is also costly. If the architecture maintains a window of M instructions waiting to be issued and is capable of generating N results per cycle, then the hardware complexity is at least O(MN) as each result may be an operand for any of the M instructions. Since part of the design philosophy of VLIW is to keep the hardware simple, it becomes desirable to find cost effective alternatives that can achieve similar performance with much less overhead. In this section we present **delayed issue**, a novel instruction issue mechanism for VLIW processors which enables out-of-order execution without the complexity of dynamic out-of-order issue hardware. A more detailed presentation of delayed issue can be found in [Grossman00].

5.1 Implementation

Consider the program flow across two basic blocks shown in figure 9a, and suppose that for some reason the compiler is unable to optimize across the boundary between then. Ignoring branches, four assembly instructions are generated (figure 9b) which, due to the dependencies (t1, t2), cannot be compacted. Assuming a pipelined architecture that performs floating point addition and multiplication in three cycles, these instructions will take 10 cycles to execute when issued in-order (figure 9c) and 7 cycles when issued out-of-order (figure 9d).



(a) Program flow

```
1     FMUL t1,y,z
2     FADD w,x,t1
3     FADD t2,b,c
4     FMUL a,t2,d
```

(b) Instruction sequence

```
1 _ _ 2 3 _ _ 4 _ _
```

(c) In order execution: 10 cycles

```
1 3 _ 2 4 _ _
```

(d) Out of order execution: 7 cycles

Figure 9: In order versus out of order execution

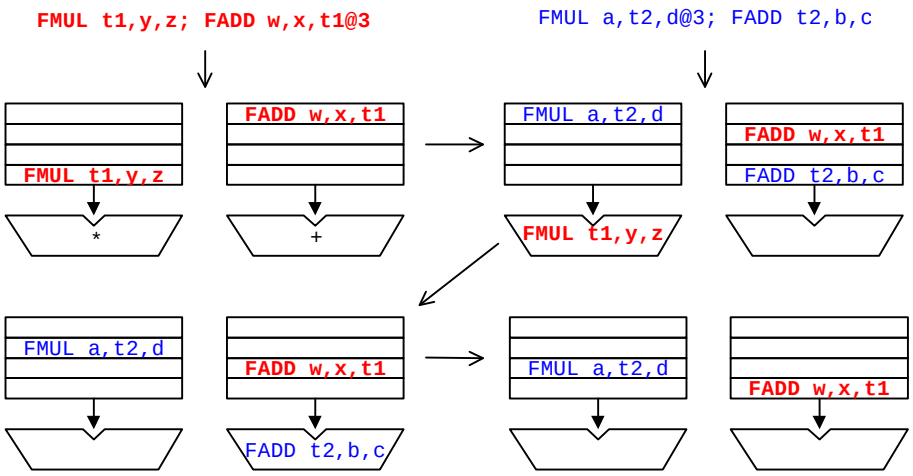


Figure 10: Delayed Issue: delay queues for adder and multiplier

A key observation is that while the compiler may not know which instructions *can* be issued, it is often able to determine which ones *can't* be issued. For example, the compiler can easily figure out that the second instruction in figure 9b cannot be issued until three cycles after the first one due to the data dependency (t1). A delayed issue mechanism allows the compiler to communicate this information to the hardware by specifying these delays explicitly as part of the instruction. Specifically, each operation in a VLIW instruction is given an integer delay; operations with non-zero delays are held back for the specified number of cycles. Using explicit delays, we can rewrite the code sequence of figure 9b as follows:

```
1    FMUL t1,y,z;    FADD w,x,t1@3
2    FMUL a,t2,d@3; FADD t2,b,c
```

where we use the notation op@N to denote an operation which is delayed for N cycles. To implement these delays in hardware we introduce per-functional unit delay queues. When a group of instructions is decoded, the instructions are inserted into the corresponding queues using the specified delays. Instructions within a queue are executed in the order that they appear; out of order execution results from different instructions being inserted into the queues at various delays. This is depicted in figure 10, which shows the first few cycles of execution for the above instructions. The order of execution is the same as for full out-of-order issue, and the execution time is therefore the same (7 cycles).

5.2 Rationale

It may at first seem counter-intuitive that delaying instructions can expedite program execution. The reason that delayed issue works is that it allows the hardware to make better use of its instruction issue logic. In a dynamic out-of-order issue processor, this resource is replicated as many times as there are instructions waiting to be issued so that any of these instructions can be issued on any cycle. In an in-order processor, the logic is not replicated, and so if an instruction stalls due to a data dependency it ties up the resource until the dependency is resolved. On cycles in which it can be statically determined that the instruction will stall, this is a shameful waste of costly silicon. With delayed issue, this wastage is avoided; placing such instructions in delay queues keeps the issue mechanism available for instructions that can actually use it.

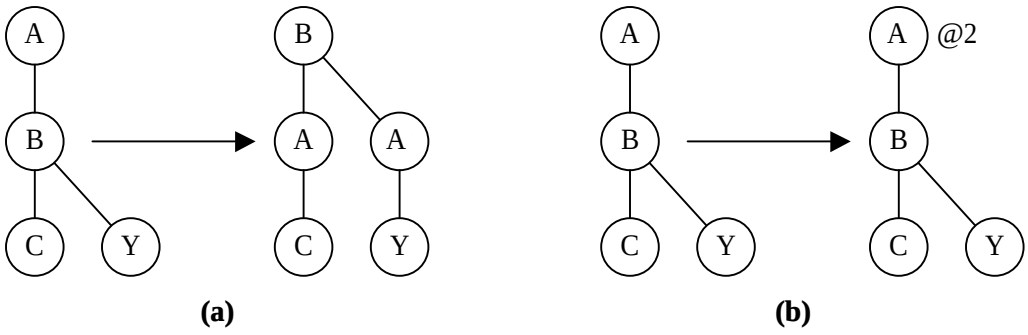


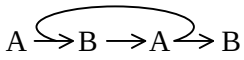
Figure 11: (a) When instructions are physically rescheduled, compensation code is required. **(b)** Using delayed issue to achieve the same order of execution with no compensation code.

5.3 Trace Scheduling

Delayed issue can be used to reduce code expansion in trace scheduling. As shown in figure 3, moving an instruction downwards past a split or a join necessitates the creation of compensation code to preserve program semantics. If instead of actually moving the instruction past the split or the join we specify a delay, then we achieve the same execution order and no compensation code is needed since other traces are unaffected by this change. An example is shown in figure 11; in both cases the order of execution is changed from A, B to B, A, but no compensation code is required when using delayed issue.

5.4 Software Pipelining

In section 4.5 the need for prologue and epilogue code was explained. Another explanation is that there are several instances of instructions A, B such that A precedes B in each iteration, but in order to pipeline the loop B must be scheduled closer to the start of the kernel than A. Without delayed issue, the only way to achieve this schedule is to actually place B earlier than A in the assembly code. We therefore need a prologue containing A to precede B the first time the loop is entered, and we need an epilogue containing B to succeed A when the loop is exited (figure 12a). Using delayed issue, however, we can place B after A in the loop and specify a delay. This has the effect of scheduling B closer to the start of the loop when the loopback branch is taken, and after the loop when the loop is exited. We can therefore eliminate the prologue and epilogue code (figure 12b). In figure 12c we have rewritten the loop in 8b using delayed issue; the number of instructions has dropped from 24 back down to 6.



(a) Prologue, Epilogue

(b) Delayed issue

```
L0:  LOAD  a2,[a0];   FMUL a3,a2,a2@1; PADD a0,a0,4
     STORE a4,[a1]@7; FADD a4,a3,c@3;  PADD a1,a1,4@7; LOOP L0
```

(c) Using delayed issue to rewrite the loop with no prologue or epilogue

Figure 12

6 Conclusion

There is a common misconception that compiling for a VLIW machine is "too difficult". Certainly it is no trivial task, but in this paper we have seen that there are a number of techniques which have been very successful in generating efficient VLIW code. Trace scheduling allows programs to be optimized across basic block boundaries, and is particularly effective when profiling information reveals commonly executed traces. Speculative execution improves the performance of trace scheduling by allowing instructions to migrate upwards past branches. Software pipelining complements trace scheduling by generating extremely efficient loop code.

Code expansion is one of the problems with both trace scheduling (due to compensation code) and software pipelining (due to prologue and epilogue code, as well as potentially exponential code growth with certain algorithms such as APP). This problem has been successfully addressed with both software and hardware techniques. The Multiflow trace scheduling compiler employs various heuristics to limit the amount of code growth. Predicated execution can be used to collapse branches into straight line code and remove duplicate instructions that exist in both conditional clauses. Rotating registers obviate the need to duplicate loop kernels to perform modulo variable expansion. Shifting predicates and delayed issue can both be used to eliminate prologue and epilogue code.

In many cases compilers are limited in their ability to perform optimizations, and program execution can benefit from run-time instruction reordering. In this paper we have presented delayed issue, which can provide the performance advantages of out-of-order execution without the high cost normally associated with such mechanisms. Since instructions can be issued only from the heads of delay queues, the issue logic is no more complex than that of a standard VLIW machine. Another benefit of delayed issue is that it addresses the code expansion problem in both trace scheduling and software pipelining.

Due to their unconventional nature, until recently VLIW architectures have been the exception rather than the norm. However, as scalar processors are reaching the limits of their potential performance, designers are beginning to focus more on VLIW as the next stage in the evolution of computer architecture. Itanium [Doshi99] and Crusoe [Klaiber00] are two examples of VLIW processors which are entering the commercial marketplace. We have seen that the compiler technology to support these processors is well established; furthermore advances in processing technology, notably the on-die integration of logic and DRAM, provide hardware support for the high instruction bandwidth requirements of VLIW. We have little doubt that in the near future VLIW will be considered standard computer architecture, and that scalar machines will take their place in history alongside punchcards, vacuum tubes, and core memory.

References

[Allan95] Vicki H. Allan, Reese B. Jones, Randall M. Lee, Stephen J. Allan, "Software Pipelining", ACM Computing Surveys, Vol. 27, No. 3, September 1995.

[Chang95] Pohua P. Chang, Nancy J. Warter, Scott A. Mahlke, William Y. Chen, Wen-mei W. Hwu, "Three Architectural Models for Compiler-Controlled Speculative Execution", IEEE Transactions on Computers, Vol. 44, No. 4, April 1995, pp. 481-494

[Colwell88] Robert P. Colwell, Robert P. Nix, John J. O'Donnell, David B. Papworth, Paul K. Rodman, "A VLIW Architecture for a Trace Scheduling Compiler", IEEE Transactions on Computers, Vol. 37, No. 8, August 1988, pp. 967-979.

[Dehnert89] James C. Dehnert, Peter Y. T. Hsu, Joseph P. Bratt, "Overlapped Loop Support in the Cydra 5", proc. ASPLOS '89, pp. 26-38.

[Doshi99] Gautam Doshi, "Understanding The IA-64 Architecture", available online at http://developer.intel.com/design/ia-64/downloads/idfisa.htm

[Ellis85] J. R. Ellis, "Bulldog: A Compiler for VLIW Architectures", Technical Report DCS/RR-364, Feb. 1985, Yale University.

[Fisher81] Joseph A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction", IEEE Transactions on Computers, Vol. C-30, No. 7, July 1981, pp. 478-450.

[Fisher83] Joseph A. Fisher, "Very Long Instruction Word Architectures and the ELI-512", Proc. ISCA '83, pp. 140-150.

[Freuden94] Stefan M. Freudenberger, Thomas R. Gross, Geoffrey P. Lowney, "Avoidance and Suppression of Compensation Code in a Trace Scheduling Compiler", ACM Transactions on Programming Languages and systems, Vol. 16, No. 4, July 1994, pp. 1156-1214.

[Grossman00] J.P. Grossman, "Cheap Out-of-Order Execution using Delayed Issue", submitted to ICCD 2000.

[Hara96] Tetsuya Hara, Hideki Ando, Chikako Nakanishi, Masao Nakaya, "Performance Comparison of ILP Machines with Cycle Time Evaluation", Proc. ISCA '96, pp. 213-224.

[Hecht77] M. S. Hecht, Flow Analysis of Computer Programs, New York: Elsevier, 1977, 232pp.

[Hsu86] P. Y. Hsu, E. S. Davidson, "Highly Concurrent Scalar Processing", Proc. ISCA '96, pp. 386-395.

[Huang94] Andrew S. Huang, Gert Slavenburg, John Paul Shen, "Speculative Disambiguation: A Compilation Technique for Dynamic Memory Disambiguation", Proc. ISCA '94, pp. 200-210.

[Hwu93] Wen-mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringman, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, Daniel M. Lavery, and the Members of the Urbana-Champaign Volunteer Fire Department, "The Superblock: An Effective Technique for VLIW and Superscalar Compilation", Journal of Supercomputing, 1993, pp. 229-248.

[Isoda83] S. Isoda, Y. Kobayashi, T. Ishida, "Global Compaction of Horizontal Microprograms Based on the Generalized Data Dependency Graph", IEEE Transactions on Computers, Vol. C-32, No. 10, Oct. 1983, pp. 922-933.

[Klaiber00] Alexander Klaiber, "The Technology Behind Crusoe Processors", technical white paper available at http://www.transmeta.com/crusoe/technology.html, January, 2000.

[Knight97] Tom Knight, "It's the wires, stupid", personal communication, 1997.

[Lah83] J. Lah, D. Atkins, "Tree Compaction of Microprograms", Proc. 16th Annual Microprogramming Workshop, Oct. 1983, pp. 23-33.

[Lam88] Monica Lam, "Software Pipelining: An Effective Scheduling Technique for VLIW Machines", Proc. SIGPLAN '88 Conference on Programming Language Design and Implementation, Atlanta, Georgia, June 22-24, 1988, pp. 318-328.

[Lam89] Monica S. Lam, A Systolic Array Optimizing Compiler, Kluwer Academic Publishers, 1989, 200pp.

[Lavery95] Daniel M. Lavery, Wen-mei W. Hwu, "Unrolling-Based Optimizations for Modulo Scheduling", Proc. 1995 Annual International Symposium on Microarchitecture.

[Linn83] J. L. Linn, "SRDAG Compaction – a Generalization of Trace Scheduling to Increase the Use of Global Context Information", Proc. 16th Annual Microprogramming Workshop, Oct. 1983, pp. 11-22.

[Mahlke92a] Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, Roger A. Bringmann, "Effective Compiler Support for Predicated Execution Using the Hyperblock", Proc. 1992 International Symposium on Microarchitecture, pp. 45-54.

[Mahlke92b] Scott A. Mahlke, William Y. Chen, Wen-mei W. Hwu, B. Ramakrishna Rau, Michael S. Schlansker, "Sentinel Scheduling for VLIW and Superscalar Processors", Proc. ASPLOS '92, pp. 238-247.

[Mahlke95] Scott A. Mahlke, Richard E. Hank, James E. McCormick, David I. August, Wen-mei W. Hwu, "A Comparison of Full and Parial Predicated Execution Support for ILP Processors", Proc. ISCA '95, pp. 138-150.

[Rau81] B. R. Rau, C. D. Glaeser, "Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing", Proc. 1981 Annual Workshop on Microprogramming, pp. 183-198.

[Rau94] B. Ramakrishna Rau, "Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops", Proc. 27th Annual International Symposium on Microarchitecture, 1994, pp. 63-74.

[Smith89] Michael D. Smith, Mike Johnson, Mark A. Horowitz, "Limits on Multiple Instruction Issue", Proc. ASPLOS '89, pp. 290-302

[Smith90] Michael D. Smith, Monica S. Lam, Mark A. Horowitz, "Boosting Beyond Static Scheduling in a Superscalar Processor", Proc. ISCA '90, pp. 344-354.

[Stoodley96a] Mark G. Stoodley, Corinna G. Lee, "Software Pipelining Loops with Conditional Branches", Proc. 1996 Annual International Symposium on Microarchitecture, pp. 262-273.

[Stoodley96b] Mark G. Stoodley, "Scheduling Loops with Conditional Statements", Master's thesis, Department of Electrical and Computer Engineering, University of Toronto, 1996.

[Su90] Bogong Su, Jian Wang, Zhizhong Tang, Wei Zhao, Yimin Wu, "A Software Pipelining Based VLIW Architecture and Optimizing Compiler", Proc. 1990 Annual International Symposium on Microarchitecture, pp. 17-27.

[Tarjan72] Robert Tarjan, "Depth-First Search and Linear Graph Algorithms", SIAM Journal of Computing, Vol. 1, No. 2, Jume 1972, pp. 146-160.

[Touzeau84] Roy F. Touzeau, "A Fortran Compiler for the FPS-164 Scientific Computer", Proc. 1984 ACM SIGPLAN Symposium on Compiler Construction, pp. 48-57.

[Ugurdag93] H. Faith Ugurdag, Christos A. Papachristou, "A VLIW Architecture Based on Shifting Register Files", Proc. 1993 Annual International Symposium on Microarchitecture, pp. 263-268.

[Wall91] David W. Wall, "Limits of Instruction-Level Parallelism", Proc. ASPLOS '91, pp. 176-188.

[Warter92] Nancy J. Warter, Grand T. Haab, Krishna Subramanian, John W. Bockhaus, "Enchanced Modulo Scheduling for Loops with Conditional Branches", Proc. 1992 Annual International Symposium on Microarchitecture.

[Warter93a] Nancy J. Warter, Daniel M. Lavery, Wen-mei W. Hwu, "The Benefit of Predicated Execution for Software Pipelining", Proc. 1993 Hawaii International Conference on System Sciences, Vol. 1, pp. 497-506.

[Warter93b] Nancy J. Warter, Scott A. Mahlke, Wen-mei W. Hwu, B. Ramakrishna Rau, "Reverse If-Conversion", Proc. 1993 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 290 – 299.

[Warter95] Nancy J. Warter-Perez, Noubar Partamian, "Modulo Scheduling with Multiple Initiation Intervals", Proc. 1995 Annual International Symposium on Microarchitecture, pp. 111-118.