Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction,
SIGPLAN Notices Vol. 19, No. 6, June 1984

Parallel Processing: A Smart Compiler and a Dumb Machine

Joseph A. Fisher, John R. Ellis,
John C. Ruttenberg, and Alexandru Nicolau

Department of Computer Science, Yale University
New Haven, CT 06520

Abstract

Multiprocessors and vector machines, the only successful parallel architectures, have coarse-grained parallelism that is hard for compilers to take advantage of. We've developed a new fine-grained parallel architecture and a compiler that together offer order-of-magnitude speedups for ordinary scientific code.

Introduction

Compilers have traditionally played second fiddle to hardware projects in parallel processing. Parallel architectures have been built to be hand coded, and attempts at compiler writing were mere afterthoughts. These attempts have been unsurprisingly unsuccessful.

The two most common types of parallel architectures built to date have been vector machines and multiprocessors. Compiling (or simply hand coding) for either requires matching an overview of the coarse structure of the application to that of the hardware. It's conceivable that hand coders and compilers might someday be good at this; but so far they haven't been, and there's no reason for optimism. There has been a general failure at culling large amounts of parallelism from ordinary applications.

So instead of building an architecture first and a compiler second, we have simultaneously developed a compiler and an architecture intended for scientific computing. Using a technique called **trace scheduling**, the Bulldog compiler finds large amounts of parallelism in ordinary scientific code. Taking advantage of this parallelism requires a new architecture, which we call VLIW (Very Long Instruction Word).

The Bulldog compiler is finished, and it compiles ordinary scientific programs into highly parallel machine code for a large class of VLIWs, achieving order-of-magnitude speedups over traditional architectures. We think VLIW architectures are practical in the very near future, and we're building a VLIW machine, the ELI (Enormously Long Instructions) to prove it.

In this paper we'll describe some of the compilation techniques used by the Bulldog compiler. The ELI project and the details of Bulldog are described elsewhere [4, 6, 7, 15, 17].

VLIW Architectures

Highly parallel machines that actually have been built fall into two broad classes: multiprocessors and vector machines. Both classes provide coarse-grained parallelism which is hard for a compiler to use.

With multiprocessors, a compiler must minimize communication and synchronization while trying to keep all the processors busy, avoiding the delays when one processor must wait for another. This forces a compiler to look for large sections of relatively independent control and data; compilers have only been able to do this for programs consisting of simple data-independent inner loops.

With vector machines, a compiler must find large aggregates in the program that can be fetched and operated upon simultaneously using relatively simple operators. This requires finding a high degree of regularity in the data and control, and compilers haven't been able to do that either for very many programs.

Instead of coarse-grained parallelism inaccessible to a compiler, VLIWs provide fine-grained parallelism that a trace-scheduling compiler can easily use. In a VLIW machine, every resource is completely and independently controlled, by which we mean:

Timing control. Every single action takes an amount of time predictable by the compiler. The time may vary according to the operation.

Flow control. There is a single thread of control, a single instruction stream, that initiates each fine-grained operation; many such operations can be initiated each cycle.

Communications control. All communications are completely choreographed by the compiler and are under explicit control of the compiled program. The source, destination, resources, and time of a data transfer are all known to the compiler. There is no sense of packets containing destination addresses or of hardware scheduling of transfers.

Such fine-grained control of a highly parallel machine requires very large instructions, hence the name Very Long Instruction Word architecture.

Figure 1 shows a picture of a hypothetical VLIW machine. It has 16 **clusters** connected by simple data buses. Each cluster is a reduced instruction set processor that has local registers, instruction memory, optional data memory, a few functional units implementing integer and/or floating scalar operations, and a partial crossbar connecting these elements within the cluster.

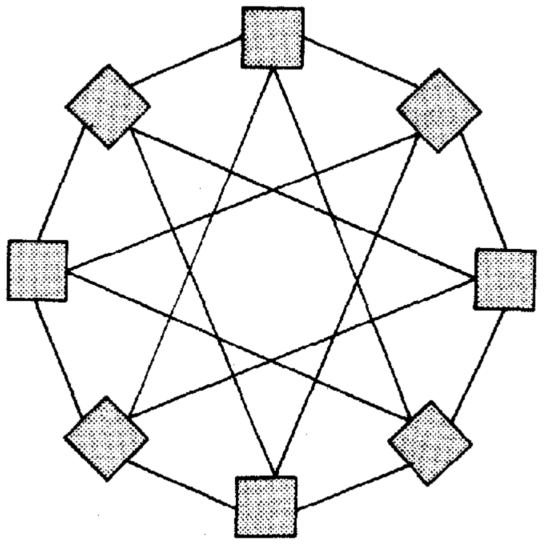


Figure 1: A hypothetical VLIW; each box is a separate cluster

All the clusters run in lockstep and are controlled by a single instruction stream. An instruction specifies the action of every element of every cluster independently; for example, one instruction may initiate a floating add in cluster 1, a floating subtract in cluster 2, an integer multiply in cluster 3, a register transfer between clusters 1 and 4, etc. Consequently, instructions will be very large (at least several hundred bits).

VLIW machines are far too large to have a single crossbar connecting all their elements. Instead, the clusters are connected by buses for transferring scalar values. It may well take several hops to move a value between distant clusters.

VLIWs need not have the regularity implied by the picture. The interconnections between the clusters, the type and number of elements within the clusters, and the connections between cluster elements can (and probably will) be asymmetric.

Before the advent of trace scheduling, it wasn't practical to build VLIW machines because no mere mortal could program them by hand. It is just barely possible to program horizontally microcoded machines and wide processors such as the FPS-164 and the MARS-432, but the amount of effort involved is tremendous. Programming a VLIW with 16 or more times the number of functional units is out of the question. Without a compiler for a high-level language, VLIWs would be useless.

Compilers for VLIWs

At first blush compiling high level languages for VLIWs might appear to be an impossible task, given that they are programmed at such a low, detailed level. But in fact the Bulldog compiler isn't that much different from a traditional optimizing compiler.

A traditional compiler parses the source program into an intermediate code, optimizes that intermediate code, and then translates the intermediate code into machine code. Usually, the translation to machine code is done one basic block at a time, perhaps after registers have been globally allocated.

It wouldn't be hard to construct a basic-block code generator for VLIWs. Several such code generators were written for machines with limited fine-grain parallelism such as the the FPS-164, the CDC machines, and the scalar portion of the Cray [18]. Part of the problem is equivalent to that of statically scheduling a set of interdependent jobs with different resource requirements on a fixed set of processors; this problem has been studied for years and there are many practical solutions [5].

But basic blocks have severely limited parallelism; experiments showed early on that one could expect at most a two- or three-times speedup by executing basic blocks in parallel [9, 19]. A basic block-based code generator couldn't hope to keep a VLIW with 16 or 32 processors busy. So no one ever built a VLIW.

Later experiments [14] showed, however, that if one ignored the artificial constraints imposed by basic blocks, ordinary scientific programs contained large amounts of parallelism—factors of 90 on average. If only a compiler could find it, such parallelism is more than enough to keep a VLIW busy.

Trace Scheduling

Trace scheduling finds much of that factor-of-90 parallelism by giving more than one basic block at a time to the code generator. To generate machine code, the compiler repeatedly traces out a path of many basic blocks in the intermediate-code flow graph and hands that entire path to the code generator. These paths, or **traces**, contain much more parallelism than basic blocks. The code generator treats the trace of blocks almost as if it were a single, very large basic block.

The compiler picks a trace, generates code for it, picks another trace, generates code for it, and so on until the entire flow graph has been translated to machine code. Estimates of execution frequency guide the compiler in picking traces; the blocks most likely to be executed comprise the first trace, those next likely to be executed comprise the second trace, and so on. Figure 2 shows a simple program and the traces selected from it.

The current compiler uses loop nesting and programmer-supplied hints to make reasonable guesses about block execution frequency; this method appears to work fairly well without too much help from the programmer. One could easily imagine an automatic profiler that would supply execution counts based on sample runs of the program, though it's doubtful that it would do much better than the current method of guessing.

For various reasons, a trace never extends past a loop boundary. That is, a trace can include only blocks from

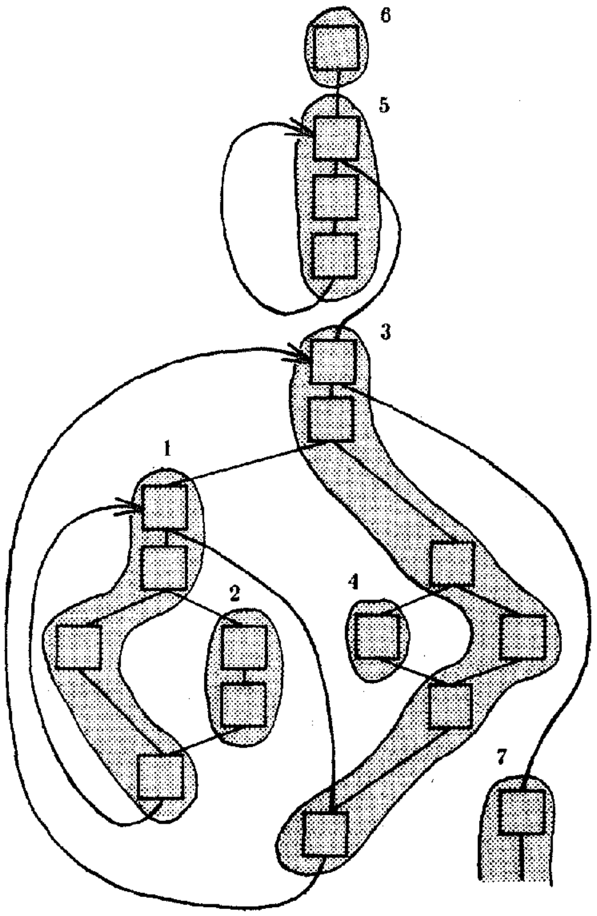


Figure 2: A flow graph with the traces selected from it

the same loop, but no blocks from containing or contained loops.

To further increase the parallelism of traces, the compiler unrolls the bodies of inner loops as many as 32 times immediately after parsing the source program into intermediate code. For example, a loop such as:

```
i := 1
LOOP {
    IF i > n THEN EXIT
    body
    i := i + 1
    }
```

unrolled three times would look like:

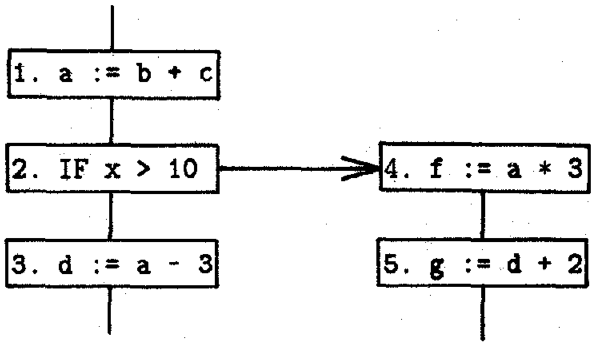
```
i := 1
LOOP {
    IF i > n THEN EXIT
    body
    i := i + 1

    IF i > n THEN EXIT
    body
    i := i + 1

    IF i > n THEN EXIT
    body
    i := i + 1
    }
```

This unrolling produces much longer traces, increasing the potential parallelism available to the code generator. (Later we'll see other uses for unrolling.)

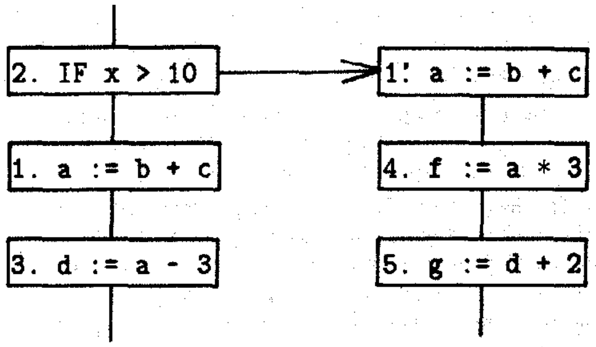
To get parallel code the code generator must substantially reorder the trace's intermediate-code operations, filling machine instructions with operations that come from widely separated places in the program; time-critical operations are usually scheduled early, while non-critical operations are often delayed. In a basic-block code generator of a traditional compiler, this reordering is relatively easy [1, 18].

By doing one basic block at a time, a traditional code generator is assured that all jumps into the block from the outside are to the block's first instruction, and that there is at most one conditional jump in the block, which must be at the end. But looking at figure 2, one immediately notices that traces consisting of many blocks will have more than one conditional jump and that there will be jumps from outside the trace into the middle of the trace. This complicates the task of reordering considerably; in addition to the normal data-precedence rules for basic block operations, the compiler must also worry about jumps off the trace and jumps into the trace.

Let's first consider reordering in the presence of conditional jumps. Suppose that we have the following fragment of a flow graph:

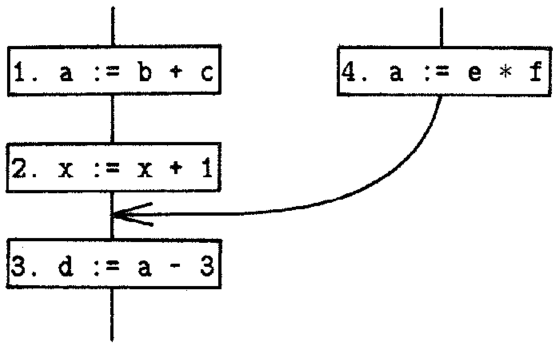


and that the current trace consists of operations 1, 2, and 3. Suppose that the code generator decides that operation 1 is not time-critical and should be moved below the conditional jump 2. If it moves 1 below the jump, then operation 4, which reads the variable **a** written by 1, will get the wrong value of **a**. So if 1 is moved below 2, the compiler will have to make a copy of 1, 1', on the off-trace edge of the jump:

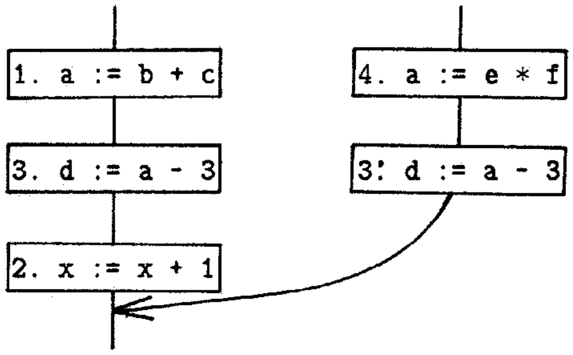


Conversely, suppose that the code generator decides that 3 is time-critical and would like to move it before the jump. Because 3 writes the variable **d**, and 5 reads the previous value of **d**, moving 3 above the jump would be incorrect, since 5 would then get the wrong value of d. If the value of d were not used on the off-trace edge of the jump, then moving 3 above the jump would be permissible.

What about jumps from blocks outside the trace into the middle of the trace? For example, assume that in the following fragment the current trace consists of operations 1, 2, and 3:



Suppose 3 was time critical and the code generator wanted to move it before 2, above the spot where 4 jumps to the trace. By itself, this motion is incorrect, because 4 writes variable a and 3 reads it; 3 would no longer get the correct value of a from 4. The solution is to make a copy of 3, 3', on the incoming edge right below 4; in this way, no matter which path is executed, d will still get the same value.



The general rules for code motions relative to jumps and rejoins within a trace are:

> If a trace operation moves below a conditional jump, a copy of it must be placed on the off-trace edge of the jump.
>
> A trace operation that writes a variable can't move above a conditional jump if the variable is live on the off-trace edge of the jump.
>
> If a trace operation moves above a rejoin to the trace, then a copy of it must be placed on the off-trace rejoining edge.

In these examples we've only considered simple operations moving past jumps and rejoins to the trace, but conditional jumps as well may move past other jumps and rejoins. The same rules apply, though there are some minor complications in copying conditional jumps.

After generating machine code for a trace, the copies of intermediate code operations resulting from the code motions are inserted into the flow graph. They will be selected and compiled as part of later traces. One might think that excessive code motion would cause an explosion in copied operations, resulting in a very large object program, or perhaps that trace scheduling doesn't even terminate. In fact, it does terminate [15], and experiments show that the amount of copying is quite acceptable [4].

Memory Reference Disambiguation

Indirect memory references arising from pointer dereferencing and array indexing pose special problems for a trace-scheduling compiler. Long traces contain many such indirect references, and in order to take advantage of the potential parallelism in the trace, the code generator must be able to reorder the references as it does other operations in the trace. To see why, consider this fragment of a trace:

```
1. v[i] := e1
2. x    := v[i]
3. v[j] := e2
4. y    := v[j]
```

Without knowing anything about the indices i and j, a compiler must assume that i could equal j, and thus that operation 3 must be executed after both 1 and 2; under this assumption, there is no available parallelism in the fragment. But if the compiler knew somehow that I and J were never equal, then 1 and 3 could be performed in parallel and 2 and 4 in parallel, a doubling in speed. Analogous situations arise from dereferencing pointers.

To achieve the most parallelism, the compiler must disambiguate as many memory references as possible, determining whether they could possibly be to the same memory location. Disambiguating pointer dereferences is tough; there are few obvious clues in the program to help the compiler determine whether two pointers might point at the same object. But in our target domain of scientific code, the inner loops consist almost entirely of array references, and it usually isn't hard to disambiguate such references.

The disambiguator is a separate module of the Bulldog compiler. The code generator asks the disambiguator questions of the form, "Can these two vector references possibly refer to the same memory location?" The disambiguator answers yes, no, or I-don't-know. The I-don't-know answers are the ones that restrict parallelism.

How does the disambiguator disambiguate two vector references v[i] and v[j]? Using the conventional flow analysis of reaching definitions, the disambiguator derives symbolic expressions e_i and e_j for the indices i and j in terms of the induction variables and loop invariants of the loops enclosing the two references. It then compares the two expressions symbolically to see if they could possibly be equal; that is, it sees if there are any integer-valued solutions to the equation $e_i - e_j = 0$.

For example, suppose that for the following code the code generator asked about the two vector references v[j] and v[k]:

```
m := e1
FOR i := 1 to n DO {
     j := i + m
     v[j] := e2
     k := j + 1
     x := v[k]
     }
```

The disambiguator derives the expressions i+m for index j and i+m+1 for index k. The two indices are equal if and only if (i+m)-(i+m+1)=0. The disambiguator simplifies that to -1=0 and concludes that j could not possibly equal k; therefore, v[j] and v[k] refer to different memory locations.

Often the equation doesn't simplify so neatly; for example, what about $4I+2J+1=0$? Finding solutions to integer-valued equations is a well known problem in number theory, and for linear equations the question is easy to answer; luckily almost all derivations of vector indices in scientific code are linear. (There are no integer solutions to $4I+2J+1=0$.)

But what about the copying of operations resulting from code motions during trace scheduling? The program is continually changing due to these copies being inserted in the flow graph, and it might seem that the flow analysis information must be incrementally recomputed after each trace. Fortunately this is not the case, and a static reaching analysis is sufficient [15]; intuitively, this is because the trace-scheduling algorithm preserves, in a loose sense, the reaching definitions of copies.

Our experience so far has been that this simple method of disambiguation completely disambiguates most memory references in most scientific programs. But this isn't good enough—if only two references in an inner loop were not disambiguated, actual parallelism could decrease by half or even more. Unfortunately, we've found that to make the disambiguator more sophisticated would not only be difficult to implement, but it would also make compilation unacceptably more expensive. And we had several example loops, including the inner loop of Fast Fourier Transform, that could be easily disambiguated by hand, but for which we had no practical automatic techniques. So no matter what level of disambiguator functionality we settled on, it still wouldn't be able to handle all the time-critical inner loops of scientific programs.

Some way was needed for the programmer to tell the compiler that two memory references are indeed to different locations. We've implemented an assertion facility by which the programmer can tell the compiler key facts about the program; if the compiler can't automatically distinguish two memory references, it consults the programmer-supplied assertions.

For example, suppose that the compiler can't disambiguate the references in this code fragment:

```
x          := v[i]
v[j+k+i]  := y
```

The programmer can add an assertion:

```
ASSERT j+k > 0
x          := v[i]
v[j+k+i]  := y
```

that the compiler uses to deduce that the two vector references are to different locations.

How does the programmer know where assertions are needed? The compiler tells him. Whenever it encounters two references it can't distinguish, it prints out information identifying them and the simplified symbolic expression representing the difference of the vector indices. For the above example, it would print out the question:

```
j+k = 0?
```

So far, it has always been immediately clear to the programmer what assertions are needed to completely disambiguate the program. Typically, only one or two assertions are required for any one program; the compiler performs all the drudge work of applying the assertions to disambiguate individual memory references.

The Global Memory Bottleneck

Many designs of parallel architectures fail because of lack of memory bandwidth. They have small, fast, local memories clustered around the computing elements, with large aggregate data stored in a larger, slower, shared global memory. For programs that manipulate large aggregates, especially for scientific programs, the global memory is a severe bottleneck; it can't fetch and store elements of the aggregate data fast enough to keep the computing elements busy. Put another way, it is easy to build a dual-ported memory, but very hard (and expensive) to build an 8- or 16-ported memory.

Most fast machines use a cache combined with interleaved memory banks to provide higher bandwidth. For example, by putting even addresses into one bank and odd addresses into another, the bandwidth doubles, since the two banks operate in parallel. But this design doesn't scale up easily, because there is still a single central controller that accepts memory requests and distributes them to the individual banks. Servicing two requests at a time is easy; servicing 8 or 16 at a time becomes a nightmare.

We solved the memory bottleneck problem as we solved other problems, using a combination of new architecture and smart software. We noticed that in scientific programs most of the memory references result from small inner loops enumerating through the elements of large arrays. Further, the central memory controller isn't really needed for those accesses, since the particular bank of each access could be predicted at compile time. If computing elements could access individual banks without going through the central controller, the memory bottleneck would be alleviated.

Unfortunately, even in scientific code it is not always possible to compute the banks of memory references at compile time. Even if the architecture supports direct reference to banks, it must still support general references for which the bank is not known statically.

In the ELI architecture, each memory bank has a **frontdoor** and a **backdoor.** The frontdoor provides direct access for memory references known at compile time to be in the bank. The backdoors of all the banks are connected to a more traditional central memory controller; a memory reference whose bank is unknown at compile time must be made through the controller. If the compiler can statically determine the bank of a memory reference, it will generate code to reference the bank directly through the frontdoor; otherwise, it will generate a slower backdoor reference.

To determine the bank of a memory reference, the Bulldog compiler uses techniques very similar to memory disambiguation. Flow analysis is used to derive a symbolic expression for the index of a memory reference; the modulo of that index relative to the number of banks yields the bank. If the compiler can't uniquely determine the bank, the programmer can help by adding assertions.

The compiler also has to apply some source transformations. For example, consider the following implementation of vector addition:

```
FOR i := 1 TO n DO
    a[i] := b[i] + c[i]
```

Suppose we know that our machine has 8-way interleaving of memory. By unrolling the body of the loop 8 times:

```
FOR i := 1 TO n BY 8 DO
    a[i+0] := b[i+0] + c[i+0]
    a[i+1] := b[i+1] + c[i+1]
    a[i+2] := b[i+2] + c[i+2]
    a[i+3] := b[i+3] + c[i+3]
    a[i+4] := b[i+4] + c[i+4]
    a[i+5] := b[i+5] + c[i+5]
    a[i+6] := b[i+6] + c[i+6]
    a[i+7] := b[i+7] + c[i+7]
```

it isn't hard to determine at compile time the bank of each memory access within the loop, given the starting address of the vectors. In general, the compiler needs to unroll such loops some multiple of the number of banks.

More sophisticated compiler techniques are used when loops aren't as well behaved. For example, if the starting index of a loop is not a constant but a variable, a memory reference in the loop body could easily be in different banks for different executions of the loop. But by adding a special pre-loop, the compiler can guarantee that all the references in the loop body are to known banks. The pre-loop executes a copy of the loop body until the index reaches a known value modulo the number of banks, at which point control transfers to the main loop.

For example, given the following loop:

```
FOR i := m to n DO
    a[i] := b[i] + c[i]
```

the compiler (assuming 8 banks) would transform that into:

```
FOR i := m to n DO
    IF 0 = i MOD 8 THEN
        temp := i
        BREAK
    a[i] := b[i] + c[i]
FOR i := temp to n DO
    ASSERT 0 = i MOD 8
    a[i+0] := b[i+0] + c[i+0]
    a[i+1] := b[i+1] + c[i+1]
    a[i+2] := b[i+2] + c[i+2]
    a[i+3] := b[i+3] + c[i+3]
    a[i+4] := b[i+4] + c[i+4]
    a[i+5] := b[i+5] + c[i+5]
    a[i+6] := b[i+6] + c[i+6]
    a[i+7] := b[i+7] + c[i+7]
```

Code Generation

Generating machine code from intermediate basic blocks for a traditional architecture is well understood—compilers do it every day. The two main problems are register allocation and instruction selection. A compiler must decide whether to keep particular values in memory or in registers. It must also map intermediate operations onto one or more machine instructions, which may be difficult if the machine has a rich instruction set.

The problems faced by a VLIW compiler generating code for a large trace are somewhat different and more complex.

Foremost, a VLIW compiler must worry about packing many machine operations into a single, large, parallel machine instruction. A traditional code generator merely outputs a stream of machine instructions, one or more per intermediate operation, that are appended together to form the object code. But a VLIW code generator must juggle the machine operations to get as many as possible to fit into each parallel machine instruction.

Because VLIWs are essentially reduced-instruction-set processors, there is no problem in selecting machine operators for intermediate code operations, since the intermediate code operations closely correspond to the machine level. But unlike a traditional machine, a VLIW offers many hardware functional units implementing the same operator, and the compiler must choose which one to use for a particular intermediate operation. Because of the long data paths between distant elements, the code generator must try to cluster operations to minimize data movement between elements. This problem is called **operation placement**.

For example, a VLIW machine may have 16 memory banks and 32 different functional units implementing the integer-add operation. To minimize data movement, the compiler must try to perform the vector indexing calculations on integer ALUs near the memory bank containing the vector elements.

Data routing is the problem of choosing data paths (buses and registers) to move data between elements of the machine. Between a source and destination there might be several paths, and the compiler must pick one that will least conflict with other activities. The move might take several hops between the source and destination, and the compiler must allocate a register after each hop to temporarily hold the value.

Finally, register allocation is tougher with a VLIW, since it could have at least as many register banks as functional units. The compiler must not only decide when to move a value into a register from memory but also which banks will hold the value. Sometimes it's advantageous to copy a value into several banks so that it can be used by many functional units simultaneously.

Obviously, operation placement, data routing, and register allocation are all interdependent. Compilers for existing horizontally-microcoded machines haven't had to deal with these problems because the target architectures offer little choice: An operation can be done in only one or two functional units, there are only one or two paths between any two points in the machine, and a functional unit is serviced by only one or two register banks.

We've built two code generators for the Bulldog compiler, one that uses a sophisticated strategy and one that uses a much simpler strategy but handles a more realistic range of machine models. The two code generators differ primarily in their approach to operator placement and register allocation.

The code generators get a trace of basic blocks as input and produce parallel machine code as output, treating the trace as if it were one very large basic block. Like many traditional code generators, our code generators convert the intermediate operations into a directed acyclic graph. The nodes of the DAG represent operations, and there is an edge between two nodes if one node uses the value produced by the other. They then form a **schedule** of machine instructions by traversing the nodes in some topological order, choosing machine operations for intermediate operators and filling the instructions of the schedule with the machine operations chosen. To prevent illegal code motions past jumps and to force undisambiguated memory references to be evaluated in the correct order, new edges are introduced to prevent one node from being evaluated before another.

The Operation-Scheduling Code Generator

Of the two code generators, the operation scheduler [17] uses the more sophisticated strategy. Operation placement, data routing, and register allocation are all delayed as long as possible, and the decisions about a particular intermediate operation are not made until the very point when the operation is placed on the schedule of machine instructions.

The parameterized machine model used by the current operation scheduler is limited in one important sense: Every functional unit has only a single feasible register bank to use for its result. This means that its register bank choices are in some cases fully constrained by the choice of functional units. But this is a restriction of the current implementation, not of the general technique.

To generate code for a trace, the operation scheduler forms an expression DAG. It then enumerates the nodes of the DAG (operations) in a topological order, placing the operations on the schedule of machine instructions. As each operation is considered, the code generator chooses a functional unit, data paths to deliver the operands to the functional unit, and a register bank to hold the result, and it finds cycles on the schedule where these actions can be placed.

To make these choices, the operation scheduler first calculates an earliest cycle that an operation could be scheduled based on the availability of operands. For each operand, a list is kept giving all the cycles and locations the operand is available. An operation can be started only after all the operands become available. (The reordering constraints of trace scheduling and disambiguation also affect the earliest cycle.)

The operand availability lists are also used to compute a search list of likely functional units for an operation. Functional units closest to the operands are considered first, and distant units are considered last. That is, the list is ordered by the longest data path of any operand to the functional unit.

```
proc SearchForBinding( operation )
    incr cycle from EarliestCycle( operation ) do
        for each fu in FunctionalUnitSearchList( operation ) do
            if fu is available at cycle and the operands
                can be fetched to fu's inputs by cycle
                and there is a register for the result at
                cycle
            then
                Schedule operation to take place in fu
                at cycle.
                Schedule the data movements for the
                operands and the result.
                return
```

Figure 3: Algorithm for binding intermediate operations

```
proc FindDataPath( start-bank, end-bank, start-cycle,
due-cycle )
    if start-cycle > due-cycle then
        return false
    if start-bank = end-bank then
        return true
    incr cycle from start-cycle to due-cycle do
        for next-bank in SP[ start-bank, end-bank ] do
            if the data-path from start-bank to next-
                bank is
                available in cycle
            then if FindDataPath( next-bank, end-
                bank, cycle + 1,
                                    due-cycle )
            then
                return true
    return false
```

Figure 4: Algorithm for finding data paths

Figure 3 sketches the algorithm that binds intermediate operations to particular functional units, data paths, and register banks and schedules the machine operation.

Starting with the earliest cycle an operation could be scheduled, each functional unit in the search list is considered in turn. If the functional unit is not in use that cycle, if the operands can be moved to the inputs of the functional unit by that cycle, and if a register is available to hold the result, then the operation is scheduled on that functional unit in that cycle. Otherwise, the next cycle is considered, and the entire search process repeated.

The dynamic method for finding data paths relies on a short-path table indexed by register banks. For every pair of register banks Ri and Rj, SP[Ri, Rj] gives a list of register banks immediately adjacent to Ri that are on short paths from Ri to Rj.

The search for a data path is performed by the recursive procedure shown in figure 4. **FindDataPath** returns true if it can find a path between register banks **start-bank** and **end-bank**. The parameter **start-cycle** is the first cycle that the value in question is available in the starting register bank, and **due-cycle** is the last allowable cycle the value may be delivered to the ending register bank. To find a path between **start-bank** and **end-bank**, the procedure looks for a path from start-bank to an adjacent register bank **next-bank**; if it finds one, it then recursively looks for a path from **next-bank** to **end-bank**.

It's possible that the order in which operations are considered may affect the parallelism of the machine code. In general, there are many topological orderings of a DAG, and the code generator must choose one. We've experimented with several ordering heuristics, including height in the DAG (maximum distance to an exit) and estimated execution counts. The preliminary results have been mixed; there haven't been great differences between the heuristics.

The List-scheduling Code Generator

The list-scheduling code generator [4] is the simpler of the two code generators. Only a sketch of the algorithm will be given here.

The code generator uses a parameterized machine model capable of describing a large class of realistic VLIW architectures. The elements of the model are register banks, functional units (memory banks, adders, multipliers, etc.), and the connections between them. Arbitrary topologies of elements can be constructed. A shortest-path table is computed from the machine description giving the time delay and shortest path between any two elements.

Specified for each register bank are the number of registers and the number of input and output ports. Specified for each functional unit are the operations implemented by that unit, the time delay of the operations, and how frequently pipelined operations can be initiated. Associated with every register bank and functional unit are sets of resources required to perform the operations of that element; similarly, every point-to-point connection between elements has an associated set of resources required to move data across the connection. These resource sets let us describe conflicts due to hardware limitations, e.g. that only one of two buses may be used in any cycle, or that a memory bank can initiate at most two reads or writes every three cycles.

Generating code for a trace consists of three main phases: representing the trace as a directed acyclic graph, functional unit assignment, and list scheduling. The assignment phase picks functional units for each of the intermediate operations, and the list-scheduling phase then enumerates the nodes in a topological order, packing them into machine instructions.

The assignment phase is analogous to the register allocation of traditional compilers, and in fact was inspired by the top-down-greedy register-allocation algorithm [3]. Traditional register allocation tries to assign a limited set of registers to the operations of the DAG, minimizing the movement of data between registers and memory. Analogously, the assignment phase allocates functional units to intermediate operations, minimizing the costly movements of data between distant functional units.

The assignment algorithm simplistically assumes that the functional units are the only limited resource and that there will never be any bus or register-port conflict when moving values between functional units. Using a recursive procedure **Assign**, the code generator attempts to pick a good functional unit for each node (intermediate operation) in the DAG, making a guess as to which cycle

```
for each node with no successors (readers) do
    Assign( node, empty-set )

proc Assign( node, estimated-destinations )
    /* Assigns a functional unit to node. estimated-
       destinations is a guess as to the set of functional
       units where the value produced by node might be
       used. */

    if node is already assigned then
        return
    for each operand of node do
        Assign( operand, LikelyFUs( node, estimated-
        destinations ) )

    Pick one of LikelyFUs( node, estimated-destinations )
    and assign it to node. Estimate the the earliest cycle
    in which it can be scheduled and record the func-
    tional unit as being busy during that cycle.

proc LikelyFUs( node, estimated-destinations )
    /* Returns a set of functional units that could com-
       pute node and move its value to estimated-
       destinations as early as possible. */

    Consider each functional unit capable of computing
    node. For each unit, estimate the earliest cycle that
    the values of the operands could be moved to the
    unit, the operation computed, and its value moved to
    the closest of estimated-destinations. Return the
    functional units having the earliest such cycles.
```

Figure 5: The functional-unit assignment algorithm

the operation will be scheduled. The measure of goodness of an assignment is how early the operation can be scheduled on the assigned functional unit and its produced value moved to the functional units of the operations reading the value.

Assign, shown in figure 5, recursively propagates from the exits to the entrances of the DAG estimates of where an operation can be best computed. When it reaches the entrance nodes, it then works its way back to the exit nodes, making final assignments of functional units to operations.

Once functional units have been assigned to operations of the DAG, the list-scheduling phase emits actual machine code by enumerating the nodes in a topological order and filling in the schedule of machine instructions. The instructions are formed in order: first cycle 0, then cycle 1, then cycle 2, etc. To form the next instruction, the list scheduler considers all nodes that are **data ready**, i.e. nodes all of whose predecessors have already been scheduled. It fills the instruction with as many of the data-ready operations as possible using first-fit; when no more can be squeezed into the current instruction, it is emitted and a new instruction started.

During assignment and list-scheduling the code generator is often faced with a choice of several nodes. For example, at each step in list-scheduling there are many data-ready nodes, only some of which will fit into the current instruction. In such cases, the code generator orders the nodes by height (maximum distance to an exit of the

DAG), on the assumption that the nodes of greatest height are the most time-critical and should take priority.

The destination register bank and register for a value produced by an operation are chosen on the fly when scheduling the operation. The list scheduler looks for an available register bank on the shortest path between the functional unit producing the value and the functional units that will be using the value.

Data movements between distant register banks are also scheduled on the fly during list scheduling. As soon as a value-producing operation is scheduled, the list scheduler looks at all the operations reading the value. If any are more than one register bank away, the list scheduler inserts special copy nodes into the DAG between the producing node and the distant reading nodes that will move the value to the distant functional units. These copy nodes will be scheduled just like normal operations, getting the values to the reading functional units as early as free hardware resources will allow.

List scheduling Versus Operation Scheduling

How do the two code generators compare?

We haven't yet run extensive experiments, but preliminary results indicate that on simple machine models there is little difference in the quality of the object code produced. Why little difference? It would seem that the exhaustive branch-and-bound search methods of the operation scheduler would surely do better than the simple heuristics of the list scheduler. But the operation scheduler offers only a simplified machine model with few interdependencies; the critical resource in the model appears to be functional units, not data paths or register bank-access. Since the list scheduler assumes that the only critical resources are functional units, it's not surprising that there is no difference between the two code generators on the simplified model.

It's likely that with complicated machine models having limited data paths and complex topologies, an operation scheduler would generate better code than the list scheduler. But expanding the branch-and-bound search of the operation scheduler to efficiently handle more realistic machine models might make the operation scheduler more complicated.

As for compilation time, right now the list scheduler is slightly faster. Both code generators take time linearly proportional to the size of the input trace. But the list scheduler time is linearly proportional to the size of the machine model, whereas the branch-and-bound search of operation scheduling takes time exponentially proportional to the complexity of the machine model. We're not sure how severely this exponential factor might slow down operation scheduling with complex machine models.

The complexity of the implementations are roughly comparable (about 7000 lines of code), always an important consideration for practical compilers. Again, the operation scheduler might become significantly more complicated when it is expanded to handle more realistic machine models.

Looking ahead, perhaps a combination of the two code generators might provide the best solution. The functional unit assignment algorithm of the list scheduler could be used to heuristically guide the branch-and-bound search of the operation scheduler.

Preliminary Results

We're currently running extensive experiments measuring the performance of our compiler. As test data we're collecting a quite respectable library of scientific Fortran routines. We're running all of the routines through the compiler and measuring their performance on four machine models.

The **ideal machine** has infinite resources: infinite registers and functional units, no communications penalty, and 1-cycle operations. Performance of the ideal machine shows how much parallelism trace scheduling and disambiguation can find; parallelism is measured by taking the ratio of sequential operations to ideal machine instructions.

The **simple ELI**, used by both the operation-scheduling and the list-scheduling code generators, is an 8-cluster machine similar to figure 1, each cluster having 4 functional units connected by a complete crossbar to a multi-ported register bank. The simple ELI is more realistic than the ideal machine, but is still impractical.

The **realistic ELI** is an 8-cluster machine, each cluster having partial crossbars and multiple, practically ported register banks. It is very close to the actual ELI currently being designed. Only the list scheduler can generate code for the realistic ELI.

The **pipelined-sequential machine** is a traditional machine "built with the same technology" as the realistic ELI. But it has one cluster for which only one pipelined operation can be initiated every cycle, and the register bank allows only 2 reads and 1 write every cycle. This model resembles a CDC 6600 or the scalar portion of the Cray.

Comparing the simple and realistic ELI models with the pipelined-sequential model will tell us how much of the extra parallelism offered by the ELI models is actually being used by the compiler.

Our library currently consists of:

- Simple matrix operations (multiply, transpose, add, reduce)
- Generating prime numbers
- Convolution
- FFT
- LU decomposition
- Tridiagonal solvers (3 versions)
- Quicksort
- LINPAK inner loops (LINPAK is a widely used Fortran package for solving linear systems)

Soon we'll be adding the following routines taken from Forsythe, Malcom, and Moler [8]:

- Spline interpolation
- Integration using adaptive quadrature
- Initial value ODE, RKF45
- Solving non-linear equations
- One-dimensional optimization
- Singular value decomposition

Complete results for running all of these programs on all four models will be presented at the conference and in the theses soon to appear [4, 17]. We've compiled all of the first group for the ideal machine and a few for the simple ELI and the realistic ELI.

So far we've only had time to analyze Fast Fourier Transform in any depth. On the ideal machine, we've got a speedup of 47 (that's the ratio of sequential ideal instructions to parallel ideal instructions). On the simple 8-cluster ELI we've got a speedup of 7.5 (the ratio of pipelined-sequential instructions to simple ELI instructions). The input vector size was 512.

Of the other programs, we've run many of them through the compiler and generated correct code, but we haven't analyzed and tuned them yet for performance (disambiguating memory references, unrolling and reorganizing loops for maximum parallelism, etc.). However, their performance on the ideal machine shows how much parallelism the trace scheduling and disambiguation are finding. From the results shown in figure 6 you can see that the Bulldog compiler is finding quite a bit of parallelism in ordinary scientific code. We're confident that on most of the routines we'll get realistic ELI performance similar to FFT's performance.

Program	Speedup
FFT	47
Tridiagonal solver	9
LU decomposition	12
LINPAK inner loops	11
Prime number generation	13
Matrix multiply	25*
Convolution	25*

*The parallelism found by the compiler in these simple programs is limited only by the size of the input data.

Figure 6: Ideal machine speedup (sequential instructions/parallel instructions)

Previous Work

Early parallelism experiments [9, 19] indicated that there was very little available parallelism in ordinary programs. The pessimism of those experiments combined with the difficulty of hand-coding VLIWs focused research on multiprocessors, vector machines, and data flow machines and away from VLIWs.

Data flow machines are still a gleam in the researcher's eye. Maybe they'll eventually provide thousand-fold parallelism, but there are still too many unsolved problems. Meanwhile, the ELI project has demonstrated a practical hardware and software architecture that offers mere ten-fold speedups right now.

There has been little success in compiling programs for multiprocessors. For example, the Cm* project [11] was hamstrung by the difficulty in distributing programs among the multiple processors.

The major effort in automated code production for vector machines and multiprocessors was undertaken at the University of Illinois. Kuck and his group developed a system, Parafrase, whose main goal is to generate code for fast, highly parallel machines [16]. Parafrase relies on extensive global data-dependence analysis and no global flow information. A memory-reference disambiguation mechanism eliminates superfluous dependency edges in the data-dependency graph due to ambiguous array references [2]. Using a large library of source transformations, Parafrase attempts to fit the available parallelism to the target architecture. Because the architectures cannot use fine-grained, operation-level parallelism, the disambiguator and the transformations operate at a coarse, all-or-nothing level, ignoring anything that cannot fit the mold. As a result, Parafrase ignores large amounts of parallelism existing in ordinary programs.

Many of the ideas of our project were originally motivated by the research on the compilation of high level languages into horizontal microcode. Fisher's thesis introduced the concept of trace scheduling and discussed heuristics for using list scheduling to generate horizontal microcode from vertical microcode [5]. Sites used list scheduling to optimize the code for the scalar, pipelined portion of the Cray [18]. Since then there have been many papers about variants on trace scheduling and other techniques attempting translation into horizontal microcode [12, 13].

Our research significantly differs from this microcode research. First, the VLIW architectures we've been studying are not horizontally-microcoded machines—they are reduced-instruction-set processors that hide the detailed complexity and asymmetry of microcode, while offering many times the architectural parallelism of current microcoded machines. Second, most of the microcode compilation techniques generate vertical microcode with registers, functional units, and data paths already assigned; then they try to compact the vertical code into horizontal code, perhaps reassigning registers and functional units in the process [10]. Finally, most of the microcode research is paper research with very few people actually building real compilers.

Acknowledgements

Other participants in the ELI project include John O'Donnell, Charles Marshall, Abhiram Ranade, Mark Sidell, Doug Baldwin, and Richard Kelsey.

This work was supported in part by the National Science Foundation grants #MCS 81-06181 and #MCS 83-08988 and the Office of Naval Research contract #N000014-82-K-0184.

References

[1] A. V. Aho and J. D. Ullman.
Principles of Compiler Design.
Addison-Wesley, 1977.

[2] Uptal Banerjee.
Speedup of ordinary programs.
Technical Report UIUCDS-R-79-989, University of Illinois Department of Computer Science, October 1979.

[3] William A. Barrett and John D. Couch.
Compiler Construction: Theory and Practice.
Science Research Associates, Chicago, 1979, pages 581-587.

[4] John R. Ellis.
Bulldog: A Compiler for VLIW Architectures.
PhD thesis, Yale University, July 1984.
Expected.

[5] J. A. Fisher.
The optimization of horizontal microcode within and beyond basic blocks: An application of processor scheduling with resources.
U.S. Department of Energy Report COO-3077-161, Courant Mathematics and Computing Laboratory, New York University, October 1979.

[6] Joseph A. Fisher.
Very long instruction word architectures and the ELI-512.
In *The 10th Annual International Symposium on Computer Architecture*, pages 140-150. IEEE Computer Society and Association for Computing Machinery, June 1983.

[7] Joseph A. Fisher and John J. O'Donnell.
VLIW Machines: Multiprocessors we can actually program.
In *Compcon 84*, pages 299-305. IEEE Computer Society, February 1984.

[8] George E. Forsythe, Michael A. Malcolm, and Cleve B. Moler.
Computer Methods for Mathematical Computations.
Prentice-Hall, 1977.

[9] C. C. Foster and E. M. Riseman.
Percolation of code to enhance parallel dispatching and execution.
IEEE Transactions on Computers 21(12):1411-1415, December 1972.

[10] John Hennessy and Thomas Gross.
Postpass code optimization of pipeline constraints.
ACM Transactions on Programming Languages and Systems 5(3):422-448, July 1983.

[11] Anita K. Jones and Edward F. Gehringer, editors.
The Cm* multiprocessor project: A research review.
Technical Report CMU-CS-80-131, Computer Science Department, Carnegie-Mellon University, July 1980.

[12] Association for Computing Machinery.
12th Annual Microprogramming Workshop, 1979.

[13] Association for Computing Machinery and IEEE Computer Society.
The 16th Annual Microprogramming Workshop, 1983.

[14] Alexandru Nicolau and Joseph A. Fisher.
Using an oracle to measure parallelism in single instruction stream programs.
In *14th Annual Microprogramming Workshop*, pages 171-182. ACM Special Interest Group on Microprogramming, October 1981.

[15] Alexandru Nicolau.
Parallelism, Memory Anti-aliasing and Correctness Issues for a Trace Scheduling Compiler.
PhD thesis, Yale University, June 1984.
Expected.

[16] D. A. Padua, D. J. Kuck, and D. H. Lawrie.
High speed multiprocessors and compilation techniques.
IEEE Transactions on Computers 29(9):763-776, September 1980.

[17] John C. Ruttenberg.
Delayed Binding Code Generation for a VLIW Supercomputer.
PhD thesis, Yale University, June 1984.
Expected.

[18] Richard L. Sites.
Instruction ordering for the Cray-I computer.
Technical Report CS-023, Department of Electrical Engineering and Computer Science, University of California at San Diego, July 1978.
Ellis remembers reading this six years ago. He's talked to Sites, who remembers his work on this problem quite well, but doesn't remember writing the tech report. Ellis has also talked to the secretary responsible for distributing UCSD Computer Science reports, and she claims this report really does exist. But we haven't yet received our copy.

[19] G. S. Tjaden and M. J. Flynn.
Detection and parallel execution of independent instructions.
IEEE Transactions on Computers 19(10):889-895, October 1970.