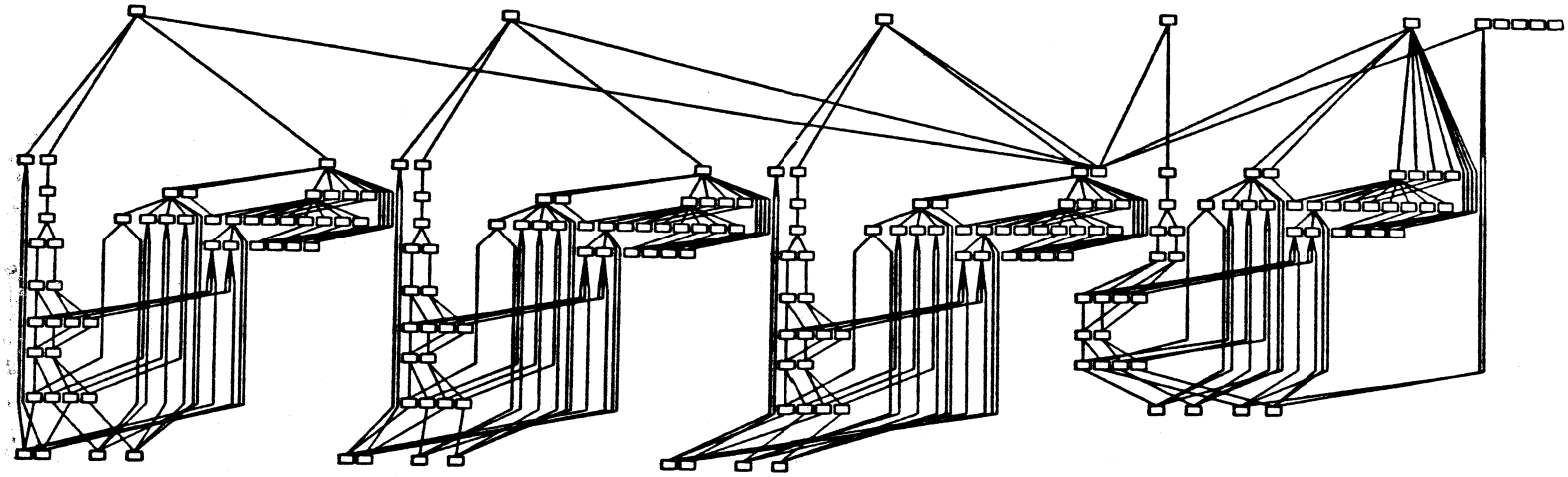

FFT (unrolled 4)

Very Long Instruction Word Architectures and the ELI-512

Joseph A. Fisher

Research Report YALEU/DCS/RR-253
Revised April 1983

YALE UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE

Abstract

By compiling ordinary scientific applications programs with a radical technique called trace scheduling, we are generating code for a parallel machine that will run these programs faster than an equivalent sequential machine—we expect 10 to 30 times faster.

Trace scheduling generates code for machines called Very Long Instruction Word architectures. In Very Long Instruction Word machines, many statically scheduled, tightly coupled, fine-grained operations execute in parallel within a single instruction stream. VLIWs are more parallel extensions of several current architectures.

These current architectures have never cracked a fundamental barrier. The speedup they get from parallelism is never more than a factor of 2 to 3. Not that we couldn't build more parallel machines of this type; but until trace scheduling we didn't know how to generate code for them. Trace scheduling finds sufficient parallelism in ordinary code to justify thinking about a highly parallel VLIW.

At Yale we are actually building one. Our machine, the ELI-512, has a horizontal instruction word of over 500 bits and will do 10 to 30 RISC-level operations per cycle [Patterson 82]. ELI stands for Enormously Longword Instructions; 512 is the size of the instruction word we hope to achieve. (The current design has a 1200-bit instruction word.)

Once it became clear that we could actually compile code for a VLIW machine, some new questions appeared, and answers are presented in this paper. How do we put enough tests in each cycle without making the machine too big? How do we put enough memory references in each cycle without making the machine too slow?

•

What Is a VLIW?

Everyone wants to use cheap hardware in parallel to speed up computation. One obvious approach would be to take your favorite Reduced Instruction Set Computer, let it be capable of executing 10 to 30 RISC-level operations per cycle controlled by a very long instruction word. (In fact, call it a VLIW.) A VLIW looks like very parallel horizontal microcode.

More formally, VLIW architectures have the following properties:

There is one central control unit issuing a single long instruction per cycle.

Each long instruction consists of many tightly coupled independent operations.

Each operation requires a small, statically predictable number of cycles to execute.

Operations can be pipelined.

These properties distinguish VLIWs from multiprocessors (with large asynchronous tasks) and dataflow machines (without a single flow of control, and without the tight coupling). VLIWs have none of the required regularity of a vector processor, or true array processor.

Many machines approximately like this have been built, but they have all hit a very low ceiling in the degree of parallelism they provide. Besides horizontal microcode engines, these machines include the CDC 6600 and its many successors, such as the scalar portion of the CRAY-1; the IBM Stretch and 360/91; and the Stanford MIPS [Hennessy 82]. It's not surprising that they didn't offer very much parallelism. Experiments and experience indicated that only a factor of 2 to 3 speedup from parallelism was available within basic blocks. (A basic block of code has no jumps in except at the beginning and no jumps out except at the end.) No one knew how to find parallelism beyond conditional jumps, and evidently no one was even looking. It seemed obvious that you couldn't put operations

from different basic blocks into the same instruction. There was no way to tell beforehand about the flow of control. How would you know whether you wanted them to be executed together?

Occasionally people have built much more parallel VLIW machines for special purposes. But these have been hand-coded. Hand-coding long-instruction-word machines is a horrible task, as anyone who's written horizontal microcode will tell you. The code arrangements are unintuitive and nearly impossible to follow. Special-purpose processors can get away with hand coding because they need only a very few lines of code. The Floating Point Systems AP-120b can offer speedup by a factor of 5 or 6 in a few special-purpose applications for which code has been handwritten at enormous cost. But this code does not generalize, and most users get only the standard 2 or 3—and then only after great labor and on small programs.

We're talking about an order of magnitude more parallelism; obviously we can forget about hand coding. But where does the parallelism come from?

Not from basic blocks. Experiments showed that the parallelism within basic blocks is very limited [Tjaden 70, Foster 72]. But a radically new global compaction technique called trace scheduling can find large degrees of parallelism beyond basic-block boundaries. Trace scheduling doesn't work on some code, but it will work on most general scientific code. And it works in a way that makes it possible to build a compiler that generates highly parallel code. Experiments done with trace scheduling in mind verify the existence of huge amounts of parallelism beyond basic blocks [Nicolau 81]. Nicolau repeats an earlier experiment done in a different context that found the same parallelism but dismissed it; trace scheduling was then unknown and immense amounts of hardware would have been needed to take advantage of the parallelism [Riseman 72].

Why Not Vector Machines?

Vector machines seem to offer much more parallelism than the factor of 2 or 3 that current VLIWs offer. Although vector machines have their place, we don't believe they have much chance of success on general-purpose scientific code. They are crucifyingly difficult to program, and they speed up only inner loops, not the rest of the code.

To program a vector machine, the compiler or hand coder must make the data structures in the code fit nearly exactly the regular structure built into the hardware. That's hard to do in first place, and just as hard to change. One tweak, and the low-level code has to be rewritten by a very smart and dedicated programmer who knows the hardware and often the subtleties of the application area. Often the rewriting is unsuccessful; it's back to the drawing boards again. Many people hope that highly vectorized code can be produced from ordinary scalar code by a very intellegent compiler [Padua 80]. We believe that vectorizing will produce sufficient parallelism in only a small percentage of programs.

And vectorizing works only on inner loops; the rest of the code gets no speedup whatsoever. Even if 90% of the code were in inner loops, the other 10% would run at the same speed as on a sequential machine. Even if you could get the 90% to run in zero time, the other 10% would limit the speedup to a factor of 10.

Trace Scheduling

The VLIW compiler we have built uses a recent global compaction technique called trace scheduling [Fisher 81]. This technique was originally developed for microcode compaction, compaction being the process of generating very long instructions from some sequential source.

Horizontal microcode is like VLIW architectures in its style of parallelism. It differs in

having idiosyncratic operations and less parallel hardware. Other techniques besides trace scheduling have been developed for microcode compaction [Tokoro 78, Dasgupta 79, Jacobs 82]. They differ from trace scheduling in taking already compacted basic blocks and searching for parallelism in individual code motions between blocks. That might work for horizontal microcode but it probably won't work for VLIWs. VLIWs have much more parallelism than horizontal microcode, and these techniques require too expensive a search to exploit it.

Trace scheduling replaces block-by-block compaction of code with the compaction of long streams of code, possibly thousands of instructions long. Here's the trick: You do a little bit of preprocessing. Then you schedule the long streams of code as if they were basic blocks. Then you undo the bad effects of pretending that they were basic blocks. What you get out of this is the ability to use well-known, very efficient scheduling techniques on the whole stream. These techniques previously seemed confined to basic blocks.

To sketch briefly, we start with loop-free code that has no back edges. Given a reducable flow graph, we can find loop-free innermost code [Aho 77]. Part (a) of figure 1 shows a small flow graph without back edges. Dynamic information—jump predictions —is used at compile time to select streams with the highest probability of execution. Those streams we call "traces." We pick our first trace from the most frequently executed code. In part (b) of figure 1, a trace has been selected from the flow graph.

Preprocessing prevents the scheduler from making absolutely illegal code motions between blocks, ones that would clobber the values of live variables off the trace. This is done by adding new, special edges to the data precedence graph built for the trace. The new edges are drawn between the test operations that conditionally jump to where the variable is live and the operations that might clobber the variable. The edges are added to the data precedence graph and look just like all the other edges. The scheduler, none the wiser, is then permitted to behave just as if it were scheduling a single basic block. It pays no attention whatsoever to block boundaries.

After scheduling is complete, the scheduler has made many code motions that will not correctly preserve jumps from the stream to the outside world (or rejoins back). So a postprocessor inserts new code at the stream exits and entrances to recover the correct machine state outside the stream. Without this ability, available parallelism would be unduly constrained by the need to preserve jump boundaries. In part (c) of the figure, the trace has been isolated and in part (d) the new, uncompacted code appears at the code splits and rejoins.

Then we look for our second trace. Again we look at the most frequently executed code, which by now includes not only the source code beyond the first trace but also any new code that we generated to recover splits and rejoins. We compact the second trace the same way, possibly producing recovery code. (In our actual implementation, we have been pleasantly surprised at the small amount of recovery code that gets generated.) Eventually, this process works its way out to code with little probability of execution, and if need be more mundane compaction methods are used so as not to produce new code.

Trace scheduling provides a natural solution for loops. Hand coders use software pipelining to increase parallelism, rewriting a loop so as to do pieces of several consecutive iterations simultaneously. Trace scheduling can be trivially extended to do software pipelining on any loop. We simply unroll the loop for many iterations. The unrolled loop is a stream, all the intermediate loop tests are now conditional jumps, and the stream gets compacted as above.

While this method of handling loops may be somewhat less space-efficient than is theoretically necessary, it can handle arbitrary flow of control within each old loop iteration, a major advantage in attempting to compile real code. Figure 2 which is generally analogous to figure 1, shows how loops are handled.

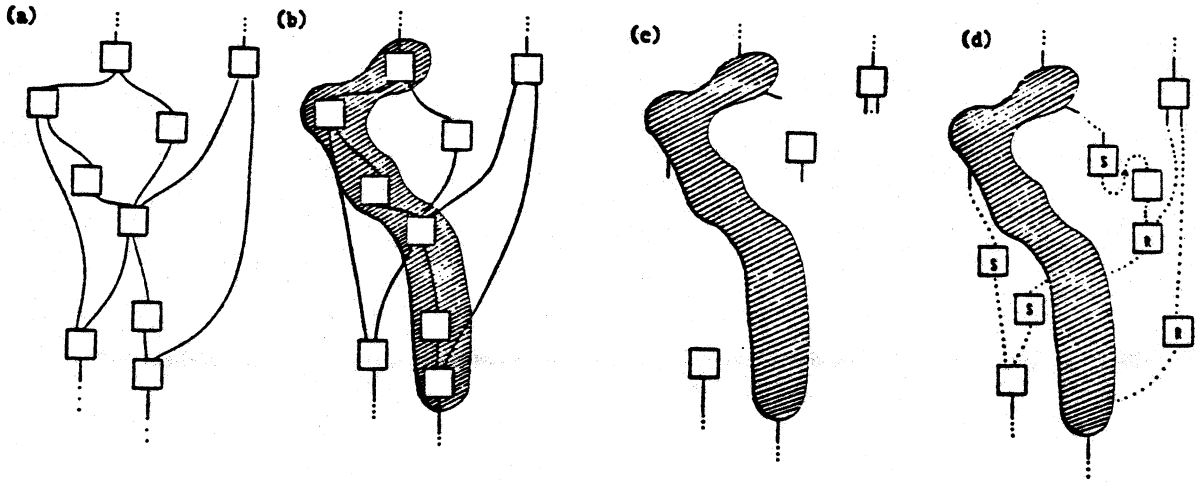

Figure 1: Trace Scheduling Loop-Free Code.

(a) A flow graph with each block representing a basic block of code.
(b) A trace picked from the flow graph.
(c) The trace has been scheduled but not rejoined to the rest of the code.
(d) The sections of unscheduled code that allow rejoining.

Bulldog, a Trace-Scheduling Compiler

We have implemented a trace-scheduling compiler in compiled Maclisp on a DEC-2060. We call it Bulldog to suggest its tenacity (and prevent people from thinking it was written at Harvard). Bulldog has 5 major modules, as outlined in figure 3.

Our first code generator is for an idealized VLIW machine that takes a single cycle to execute each of its RISC-level operations (not too drastic an idealization) and does unlimited memory accesses per cycle (entirely too drastic an idealization). We are using the code generator to help debug the other modules of the compiler and to measure available parallelism. Average operations packed per intruction is a spurious measure of speedup. Instead we divide the number of parallel cycles the code took to execute by the number of sequential cycles in running the uncompacted code.

By comparison with the idealized code, real ELI code will contain many incidental small operations. Whether that implies the same speedup, or less, or more, is subject to

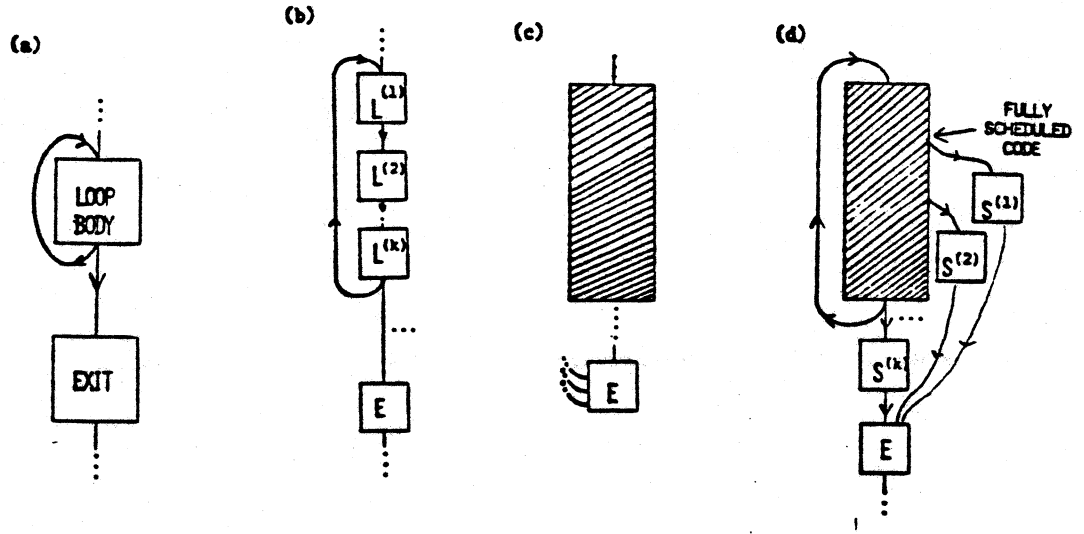

Figure 2: Trace Scheduling Loops.

(a) A loop body (which might contain arbitrary flow of control) and its exit code.
(b) The loop body has been unwound k times.
(c) Traces are picked through the unwound loop and it is scheduled.
(d) The newly scheduled loop is rejoined to the rest of the code.

debate. These incidental operations may slow down the sequential code more than the parallel, making the speedup due to parallelism all the greater. Only time will tell.

The front end we are currently using generates our RISC-level intermediate code, N-address code or NADDR. The input is a local Lisp-sugared FORTRAN, C, or Pascal level language called Tiny-Lisp. It was something we built quickly to give us maximal flexibility. We have an easy time writing simple code for it, we didn't have to write a parser, and we can fiddle with the compiler easily, which has proved to be quite useful. A FORTRAN '77 subset compiler into NADDR is written and being debugged, and we will consider other languages after that. Our RISC-level NADDR is very easy to generate code for and to apply standard compiler optimizations to.

We have two more code generators being written right now. A full ELI-512 generator is quite far along—a subset of it is now being interfaced to the trace picker and fixup code. We are also writing a FPS-164 code generator. The FPS-164 is the successor to the Floating Point Systems AP-120b, probably the largest-selling machine ever to have horizontal microcode as its only language. There is a FORTRAN compiler for the FPS-164, but our experience has been that it finds little of even the small amount of parallelism available on

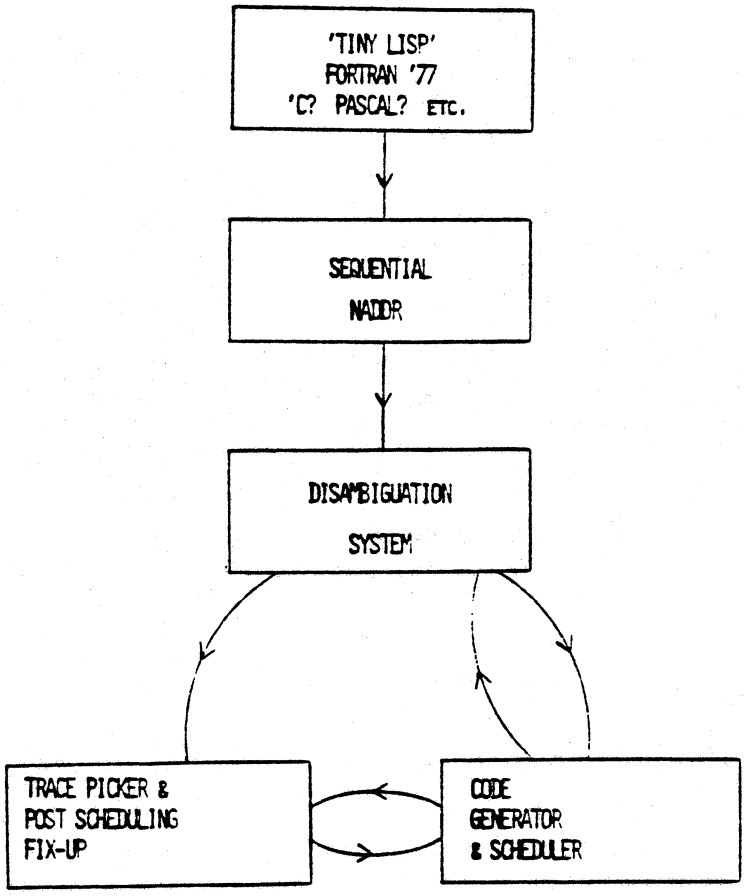

Figure 3: The Structure of the Bulldog Compiler.

that machine. A compiler that competes with hand code would really change the potential usability of that machine (it's very difficult to hand code) and would demonstrate the versatility of trace scheduling.

Memory Anti-Aliasing on Bulldog

Trace scheduling makes it necessary to do massive numbers of code motions in order to fill instructions with operations that come from widely separated places in the program. Code motions are restricted by data precedence. For example, suppose our program has the steps:

(1) $Z := A + X$
(2) $A := Y + W$

Our code motions must not cause (2) to be scheduled earlier than (1). So the trace scheduler builds a data-precedence edge before scheduling.

But what happens when A is an array reference?

(1) $Z := A[\ expr1\ ] + X$
(2) $A[\ expr2\ ] := Y + W$

Whether (2) may be done earlier than (1) is ambiguous. If *expr1* can be guaranteed to be different from *expr2*, then the code motion is legal; otherwise not. Answering this question is the problem of anti-aliasing memory references. With other forms of indirection, such as chasing down pointers, anti-aliasing has little hope of success. But when indirect references are to array elements, we can usually tell they are different at compile time. Indirect references in inner loops of scientific code are almost always to array elements.

The system implemented in the Bulldog compiler attempts to solve the equation *expr1* = *expr2*. It uses reaching definitions [Aho 77] to narrow the range of each variable in the expressions. We can assume that the variables are integers and use a diophantine equation solver to determine whether they could be the same. Range analysis can be quite sophisticated. In the implemented system, definitions are propagated as far as possible, and equations are solved in terms of simplest variables possible. We do not yet use branch conditions to narrow the range of values a variable could take, but we will.

We have implemented anti-aliasing. Unfortunately, it is missing a few of its abilities—very few, but enough to keep it from being fully effective. In this case the truism really holds: The chain is only as strong as its weakest link. Our current fix is to let the programmer add anti-aliasing assertions to the source. The system tells the programmer what things it wishes it had known, so the programmer has no trouble adding the right assertions. So far we have had to let the programmer make only modest anti-aliasing assertions, which require only a few straightforward modifications to the compiler. (The modifications are on our queue.)

There is not enough address space on the DEC -20 to run extremely large programs through Bulldog. We expected this, and the next step in the project is to move to a machine with more address space. We have speeded up small programs such as matrix multiplies and convolutions by as much as a factor of 60; simple manipulations of large bodies of data will be speeded up even more than that when we can run them on more data. Larger, more complicated programs speed up in the range of 5 to 10. Good, but not as good as we want. Examining the results by hand makes it clear that when anti-aliasing has all its abilities and we're running on a virtual address machine, speedup will be considerable.

A Machine To Run Trace-Scheduled Code

The ELI-512 has 16 clusters, each containing an ALU and some storage. The clusters are arranged circularly, with each communicating to its nearest neighbors and some communicating with farther removed clusters. Figures 4 and 5 sketch the ELI and its clusters.

The ELI uses its 500+ bit instruction word to initiate all of the following in each instruction cycle:

> 16 ALU operations. 8 will be 32-bit integer operations, and 8 will be done using 64-bit ALUs with a varied repertoire, including pipelined floating-point calculations.
>
> 8 pipelined memory references—more about these later.
>
> 32 register accesses.
>
> Very many data movements, including operand selects for the above operations.
>
> A multiway conditional jump based on several independent tests—more about these later too.

(With this much happening at once, only a maniac would want to code the ELI by hand.) To carry out these operations, the ELI has 8 M-clusters and 8 F-clusters. Each M-

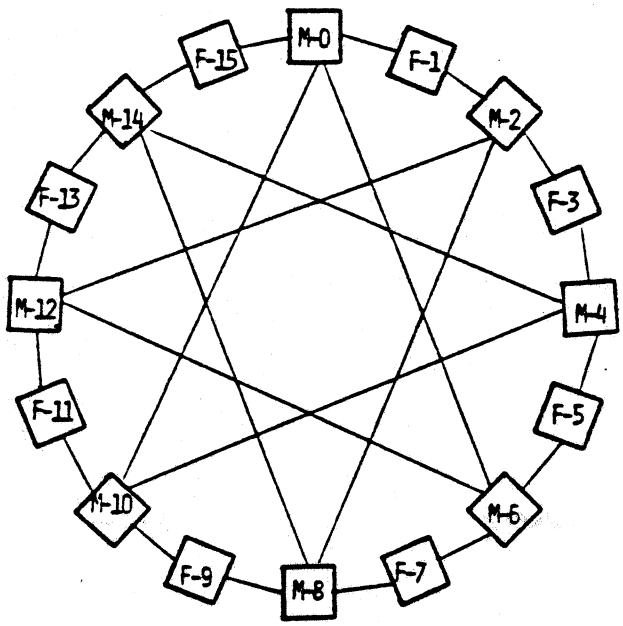

Figure 4: The Global Interconnection Scheme of the ELI-512.

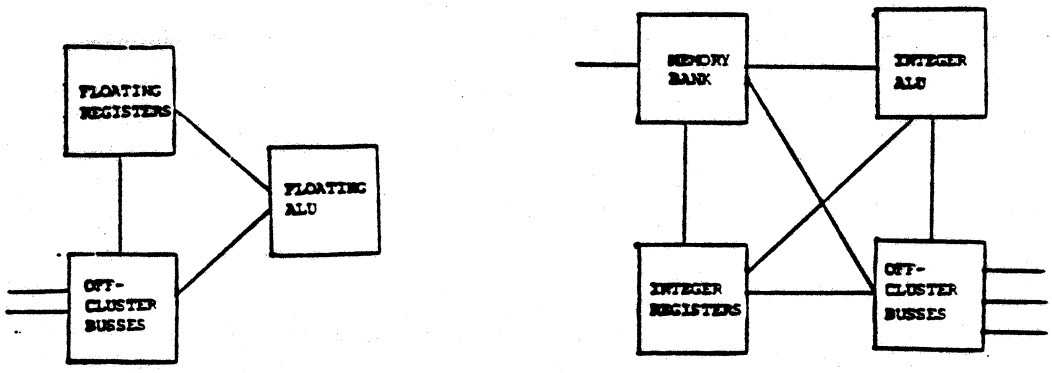

Figure 5: Typical M- and F-Cluster Block Diagrams.

cluster has within it:

A local memory module (of so far undetermined size).

An integer ALU which is likely to spend most of its time doing address calculations. The exact repertoires may vary from cluster to cluster, and won't be fixed until we tune the architecture using actual code.

A multiport integer register bank.

A limited cluster crossbar, with 8 or fewer participants. Some of the participants will be off-cluster busses. Some of the crossbar connections will not be made.

And each F-cluster has within it:

A floating point ALU. The repertoires of the ALUs will vary from cluster to cluster and won't be fixed until we tune the architecture.

A multiport floating register bank.

A limited cluster crossbar, with 8 or fewer participants. Some of the participants will be off-cluster busses. Some of the crossbar connections will not be made.

Do not be deceived by occasional regularities in the structure. They are there to make the hardware easier to build. The compiler doesn't know about them, and it doesn't attempt to make any use of them. When we start running scientific code through the compiler, we will undoubtedly further tune the architecture. We will want to remove as many busses as we can, and many of the regularities may disappear.

Current plans are to construct the prototype ELI from 100K ECL logic, though we may opt for Shottkey TTL.

Problems

Nobody's ever wanted to build a 512-bit-wide instruction word machine before. As soon as we started considering it, we discovered that there are two big problems. How do you put enough tests in each instruction without making the machine too big? How do you put enough memory references in each instruction without making the machine too slow?

Comparing VLIWs with vector machines illustrates the problems to be solved. VLIWs put fine-grained, tightly coupled, but logically unrelated operations in single instructions. Vector machines do many fine-grained, tightly coupled, logically related operations at once to the elements of a vector. Vector machines can do many parallel operations between tests; VLIWs cannot. Vector machines can structure memory references to entire arrays or slices of arrays; VLIWs cannot. We've argued, of course, that vector machines fail on general scientific code for other reasons. How do we get their virtues without their vices?

VLIWs Need Clever Jump Mechanisms

Short basic blocks implied a lack of local parallelism. They also imply a low ratio of operations to tests. If we are going to pack a great many operations into each cycle, we had better be prepared to make more than one test per cycle. Note that this is not a problem for today's statically scheduled operation machines, which don't pack enough operations in each instruction to hit this ratio.

Clearly we need a mechanism for jumping to one of several places indicated by the results of several tests. But not just any multiway jump mechanism will do. Many horizontally microcodable machines allow several tests to be specified in each microinstruction. But the mechanisms for doing this are too inflexible to be of significant use here. They do not allow for multiple independent tests, but rather offer a hardwired selection of tests that may be done at the same time. Some machines allow some specific set of bits to alter the next address calculation, allowing a 2^n-way jump. This is used, for example, to implement an opcode decode, or some other hardware case statement.

Another approach that won't suffice for us is found in VAX 11/780 microcode [Patterson 79]. There, any one of several fixed sets of tests can be specified in a given instruction. A mask can be used to select any subset of those tests, which are logically ANDed into the jump address. Unfortunately, the probability that two given conditional tests appear in the same set in the repertoire is very low. In compacting it is extremely unlikely that one can place exactly the tests one wants to in a single instruction, or even a large subset of them. Instead, combinations are hardwired in advance. One would guess that the combinations represent a convenient grouping for some given application program, in the case of the VAX, presumably the VAX instruction set emulator.

The most convenient support the architecture could possibly provide would be a 2^n-way jump based on the results of testing n independent conditions. This is not as unrealistic as it sounds; such a mechanism was considered in the course of developing trace scheduling [Fisher 80], and seemed quite practical. It turned out, however, to be more general than we needed.

After we had actually implemented trace scheduling, a surprising fact emerged: What trace scheduling requires is a mechanism for jumping to any of n+1 locations as the result of n independent tests. The tests should be any combination of n from the repertoire of tests available on the machine.

The jump works just the way a COND statement works in Lisp. For simplicity, we will pretend that a test's failing means failing and wanting to stay on the trace and that succeeding means succeeding and wanting to jump off the trace. A statement to express the multiway jump might appear as:

```
(cond   (test1          label1)
        (test2          label2)
        . . .
        (testk          labelk)
        . . .
        (testn          labeln)
        (succeed        label-fall-through) )
```

If the first test, *test1*, fails, it wants to stay on the trace and the second test, *test2*, is made. If that fails, it too wants to stay on the trace. If a test, *testk*, succeeds, then it wants to jump off the trace to *labelk;* no on-trace tests after *testk* will be made. If all the tests fail, we finally get to the last address in the instruction and fall through.

We find that n+1 target labels suffice; 2^n aren't needed. We sort all the tests for one instruction in the order in which they appeared in the trace. Then when a test succeeds we are glad that we already executed the tests that came earlier in the source order but we don't much care about the ones that came later since we're now off the trace.

It is not hard to build a n+1-way jump mechanism. Figure 6 shows that all we need is a priority encoder and n test multiplexers. The wide instruction word selects each of the n tests with n j-bit (where j is log the number of tests) fields. The first of the n tests (in sequence order) that wants to jump off the trace in effect selects the next instruction.

But how do we actually produce the next address? We could place n+1 candidates for the post of next address in full in each instruction. But even on the ELI using so many instruction bits for that would seem like overkill. The idea of using the select bits as part

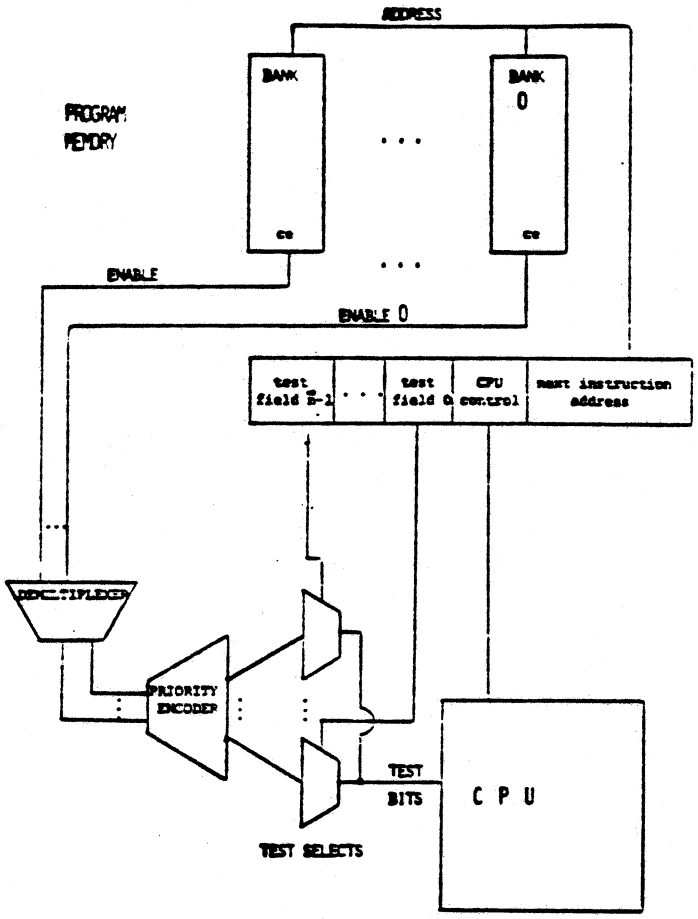

Figure 6: Implementing n+1-way Independent Jumps.

of the next address (as in the restricted multiway jumps referred to above) seems right, if we can overcome a small packing problem.

For example, if n=3, and if the current instruction has a next instruction address field of, say, 00000011, then we have the following situation:

Test Field	Condition Selected	Address If Test Is First To Succeed
0	*test1*	00 00000011
1	*test2*	01 00000011
2	*test3*	10 00000011
3	*succeed*	11 00000011

In this scheme we do not increment a program counter to get the next address, though that could be fitted in if there were some advantage in speed for a particular hardware implementation.

What happens when we pack fewer than n tests in a cycle? From the example above, with n=3, it might seem that we need to spend four program memory locations for each set of target addresses. But what if we have straightline code, or want to do only one or two tests in some cycles (as we surely will)? Do we have to waste the unused slots? We can avoid wasting slots if we include a test that always succeeds and another that always fails.

With these tests, we can cause two instructions which each want to pack one test to share an address slice, or we can pack an instruction that does two tests with an instruction that wants to do no test. For example, take two instructions, *instr1*, which wants to do *test1*, and *instr2*, which wants to do *test2*. We can arrange to have them both jump to a label in the address slice 00000011 (and thus both have 00000011 as their next-address field) as follows:

After *instr1*, we jump to either 00 00000011 or to 01 00000011. As a result, the next address field of *instr1* is 00000011, and the test fields are filled in as below.

Test Field	Condition Selected	Address If Test Is First To Succeed
0	*test1*	00 00000011
1	*succeed*	01 00000011
2	don't care	won't happen
3	*succeed*	won't happen

After *instr2*, we jump to either 10 00000011 or to 11 00000011. So it looks like:

Test Field	Condition Selected	Address If Test Is First To Succeed
0	*fail*	won't happen
1	*fail*	won't happen
2	*test2*	10 00000011
3	*succeed*	11 00000011

Since we don't have an incremented program counter and can rearrange addresses at will, these allocations can be done in a straightforward matter in a postpass program. A little space may be wasted at code rejoins, but not much.

Our previous work on 2^n-way jumps applies also to n+1-way jumps and contains a more complete explanation of these ideas [Fisher 80].

The Jump Mechanism on the ELI-512

The ELI-512 will have an n+1-way jump mechanism like the one described above. During the time when we are tuning the machine design and the compiler, we will determine how many tests are appropriate; it seems likely that n will be 3, 4, or 5. We have two instruction-fetch mechanisms under consideration.

Delayed branches are an old microcode instruction-fetch trick that works particularly well here for [Gross 82, Patterson 82]. In a delayed-branch mechanism the address of instruction M+k is determined by the result of a test in instruction m.The k instructions between m and m+k are done whether the test succeeds or fails. Using trace scheduling, we know which way most jumps go; so we can fill the gap with instructions that will probably be in the execution stream. The current compiler handles delayed jumps as a matter of course, but we've taken no measurements on whether or how much they slow down the code.

The alternative is to fetch an entire slice at once. We would have n+1 banks of instruction memory and would fetch all the next candidate words, using the next-instruction address of an instruction as soon as it is selected. Then, when the tests have settled down, the bits coming out of the priority encoder can multiplex from among the n+1 choices. The large words on the ELI may make this technique difficult.

VLIW Compilers Must Predict Memory Banks

With so many operations packed in each cycle, many of them will have to be memory references. But (as in any parallel processing system) we cannot simply issue many references each cycle; two things will go wrong. Getting the addresses through some kind of global arbitration system will take a long time. And the probability of bank conflict will approach 1, requiring us to freeze the entire machine most cycles.

But here we can rely (as usual) on a combination of smart compiler and static code. We ask our compiler to look at the code and try to predict what bank the reference is in. When we can predict the bank, we can use a dedicated address register to refer to it directly. To make several references each cycle, we access each of the memory banks' address registers individually. No arbitration is necessary, since the paths of the addresses will never cross.

When we ask the compiler which bank a reference is going to be in, what are the chances of getting an answer? In the static code we expect to run on a VLIW, very good. Scalars always have known locations. What about arrays? The same system that does anti-aliasing can attempt to see which bank a reference is in. As you'll recall, loops get unwound to increase parallelism. In fact, it's the anti-aliasing system that does the unwinding, since it knows which are induction variables and which aren't. (It renames non-induction variables that appear in successive unwound iterations to avoid unnecessary data-precedence.) By unrolling so that the array subscripts increase by a multiple of the number of banks every iteration, the anti-aliasing system often makes it possible to predict banks.

What about unpredictable references? Two kinds of unpredictable references muddy up this scheme.

First of all, we might simply have no chance to predict the bank address of a reference. For example, we might be chasing down a pointer and need access to the entire memory address space. But such accesses are assumed to be in the tiny minority; all we have to do is be sure they don't excessively slow down the local, predictable accesses. Our solution is to build a shadow memory-access system that takes addresses for any bank whatsoever and returns the values at those addresses. This requires our memory banks to be dual-ported and have lockout capabilities. The allocation of hardware resources should favor predictable access; unpredictable access can be made slower. And the predictable accesses should have priority in case of bank conflict; we can make the machine freeze when an unpredictable reference doesn't finish in time. If there are too many of these references, the machine will perform badly. But in that case conservative data-precedence would have destroyed any chance at large amounts of parallelism anyway.

The second problem is that even when we have arrays and have unrolled the loops properly, we might not be able to predict the bank location of a subscript. For example the subscript value might depend on a loop index variable with a data-dependent starting point. Unknown starting values don't ruin our chances of doing predictable references inside the loop. All we have to do is ask the compiler to set up a kind of pre-loop. The pre-loop looks like the original loop, but it exits when the unknown variable reaches some known value modulo the number of banks. Although it may itself be unwound and compacted, the pre-loop has to use the slow unpredictable addressing system on the unpredictable references. But it will execute some short number of cycles compared to the very long unwound loop. The situation given B banks is illustrated in figure 7.

(a)

```
for i := k to n do
        loop body
```

(b)

```
        i := k
Loop:   loop body
        if i >= n then goto FallThrough
        i := i + 1
        loop body
        if i >= n then goto FallThrough
        i := i + 1
        . . .
        loop body
        if i >= n then goto FallThrough
        i := i + 1
        goto Loop
FallThrough:
```

(c)

```
        i := k
Preloop: if i = 0 mod B then goto Loop
        loop body
        if i >= n then goto FallThrough
        i := i + 1
        goto Preloop

Loop:   assert i = 0 mod B
        loop body
        if i >= n then goto FallThrough
        i := i + 1
        loop body
        if i >= n then goto FallThrough
        i := i + 1
        . . .
        loop body
        if i >= n then goto FallThrough
        i := i + 1
        goto Loop
FallThrough:
```

Figure 7: Adding a Preloop.

When we have a loop variable i that begins at an unknown bank, we add a preloop to find a value of i that is a multiple of the number of banks B. Version (a) contains a source loop. In (b), the loop is unwound. In (c), we have added the preloop, which executes until i is a known value modulo the number of banks. (Note that we are using FORTRAN loop style with the test at the end.)

Memory Accessing in the ELI-512

The current design of the ELI counts on the system outlined above: bank prediction, precedence for local searches, and pre-looping. Each of the 8 M-clusters has one memory access port used for times when the bank is known. We will start one pipelined access per cycle per M-cluster (which may require us to be able to distinguish among at least 16 physical banks, 2 per module, depending upon the design of the memory). This will give us a potential data memory access bandwidth of about 400 megabytes per second if our cycle time is in the neighborhood of 150 ns. To implement pre-looping, we will have tests for addresses modulo the number of banks.

In addition, the ELI will have two access ports that address memory globally. When they are ready, the results of a global fetch are put in a local register bank. When a reference is made to the data, the system freezes if the data isn't in the registers, and all the pending global references sneak in while they have a chance. We expect this state of affairs to be quite infrequent.

What We Are Doing and Not Doing

This paper offers solutions to the problems standing in the way of using Very Long Instruction Word architectures to speed up scientific code. These problems include highly parallel code generation, multiple tests in each cycle, and multiple memory references in each cycle.

The Bulldog compiler and experiments done on real code have demonstrated that a large degree of parallelism exists in typical scientific code. Given that the existence of this parallelism makes VLIW machines desirable, we are building one: the ELI-512, a very parallel attached processor with a 500+ bit instruction word. We expect the ELI to speed up code by a factor of 10-30 over an equivalent sequential machine. We will be generating good code for the ELI before we build it. We are also writing a compiler for the FPS-164, a much less parallel but otherwise similar architecture.

Our code generators use trace scheduling for locating and specifying parallelism originating in far removed places in the code. The n+1-way jump mechanism makes it possible to schedule enough tests in each cycle without making the machine too big. Bank prediction, precedence for local searches, and pre-looping make it possible to schedule enough memory references in each cycle without making the machine too slow.

Partly to reduce the scope of the project and partly because of the the nature of VLIWs, we are limiting ourselves in various ways. ELI will be an attached processor—no I/O, no compilers, no ELI simulators, no user amenities. Rather we will choose a sane host. ELI will not be optimized for efficient context switch or procedure call. The ELI will be running compute-bound scientific code. It is difficult to extend VLIW parallelism beyond procedure calls; when we want to, we can expand such calls in line. Any VLIW architecture is likely to perform badly on dynamic code, including most systems and general-purpose code and some scientific code. We will be content to have ELI perform very well on most scientific code.

Acknowledgements

Invaluable contributions to the ELI design and the Bulldog compiler have been made by John Ruttenberg, Alexandru Nicolau, John Ellis, Mark Sidell, John O'Donnell, and Charles Marshall. Trish Johnson turned scribbling into nice pictures. Mary-Claire van Leunen edited the paper.

References

[Aho 77] A. V. Aho and J. D. Ullman.
Principles of Compiler Design.
Addison-Wesley, 1977.

[Dasgupta 79] S. Dasgupta.
The organization of microprogram stores.
ACM Computing Surveys 11(1):39-65, March 1979.

[Fisher 80] J. A. Fisher.
An effective packing method for use with 2^n-way jump instruction hardware.
In *13th Annual Microprogramming Workshop*, pages 64-75. ACM Special Interest Group on Microprogramming, November 1980.

[Fisher 81] J. A. Fisher.
Trace scheduling: A technique for global microcode compaction.
IEEE Transactions on Computers C-30(7):478-490, July 1981.

[Foster 72] 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.

[Gross 82] T. R. Gross and J. L. Hennessy.
Optimizing delayed branches.
In *15th Annual Workshop on Microprogramming*, pages 114-120. ACM Special Interest Group on Microprogramming, October 1982.

[Hennessy 82] J. Hennessy, N. Jouppi, S. Przbyski, C. Rowen, T. Gross, F. Baskett, and J. Gill.
MIPS: A microprocessor architecture.
In *15th Annual Workshop on Microprogramming*, pages 17-22. ACM Special Interest Group on Microprogramming, October 1982.

[Jacobs 82] D. Jacobs, J. Prins, P. Siegel, and K. Wilson.
Monte Carlo techniques in code optimization.
In *15th Annual Workshop on Microprogramming*, pages 143-148. ACM Special Interest Group on Microprogramming, October 1982.

[Nicolau 81] 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.

[Padua 80] 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.

[Patterson 79] D. A. Patterson, K. Lew, and R. Tuck.
Towards an efficient machine-independent language for microprogramming.
In *12th Annual Microprogramming Workshop*, pages 22-35. ACM Special Interest Group on Microprogramming, 1979.

[Patterson 82] D. A. Patterson and C. H. Sequin.
A VLSI RISC.
Computer 15(9):8-21, SEPT 1982.

[Riseman 72] E. M. Riseman and C. C. Foster.
The inhibition of potential parallelism by conditional jumps.
IEEE Transactions on Computers 21(12):1405-1411, December 1972.

[Tjaden 70] G. S. Tjaden and M. J. Flynn.
Detection and parallel execution of independent instructions.
IEEE Transactions on Computers 19(10):889-895, October 1970.

[Tokoro 78] M. Tokoro, T. Takizuka, E. Tamura, and I. Yamaura.
A technique of global optimization of microprograms.
In *11th Annual Microprogramming Workshop*, pages 41-50. ACM Special Interest Group on Microprogramming, 1978.