VLIW MACHINES: MULTIPROCESSORS WE CAN ACTUALLY PROGRAM

Joseph A. Fisher
John J. O'Donnell
Computer Science Department
Yale University New Haven, CT

Research Report YALEU/DCS/RR-298
January 1984

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

VLIW Machines: Multiprocessors We Can Actually Program

Joseph A. Fisher
John J. O'Donnell
Computer Science Department
Yale University
New Haven, CT

Abstract

VLIW machines are highly parallel architectures that offer an alternative to multiprocessors and vector machines. VLIWs resemble ordinary multiprocessors, but have a tightly coupled, single-flow control mechanism much like clean, very parallel horizontal microcode. Programs for VLIWs must specify fine-grained hardware control.

Unlike multiprocessors and vector machines, it is essentially impossible to hand code VLIWs. But also unlike multiprocessors and vector machines, a compiler now exists that can produce highly parallel code from a broad range of ordinary, sequential programs. This compiler, using a technique called *trace scheduling*, frees the programmer from the difficult task of locating the parallelism and matching it to the parallel structure of the hardware. This may make VLIWs the most practical way to obtain parallelism speedup for the typical compute intensive scientific application.

In this paper VLIWs are described and compared with other parallel architectures. The BULLDOG compiler, which generates highly parallel code from sequential high level source programs, is described. Included is a brief description of the target machine, the ELI (Enormously Long Instructions), a machine with an anticipated instruction word of over 1000 bits. Our compiler now generates code for an ELI with a parameterized description; we are currently picking values of the parameters to tune the architecture to a suite of scientific programs.

What Are VLIW Architectures?

VLIWs are architectures characterized by the large degree of timing and resource control in their programs. As a result, each instruction for a VLIW contains many bits, perhaps in the thousands. To picture a VLIW, imagine many (32, say) copies of your favorite RISC machine[1] all somehow connected to a memory system and all capable of communicating over some sort of interconnect. We'll refer to the RISC machines as the processors. There is no requirement or expectation that the processors resemble each other. Figure 1 contains a sketchy picture of a VLIW.

So far, this new definition fits the usual definition of a heterogeneous multiprocessor. What's the difference? The answer is in the instructions necessary to control the machine.

- Each long instruction contains operation fields to control each of the individual processors.
- The instructions are in a single flow of control[2]. Thus a single long instruction word is fetched, and all the processors do their individual operations. The operations differ for the various processors; there is no coherence as there would be in a vector machine. After an instruction is executed, the next instruction is chosen and fetched.

[1]Reduced Instruction Set Computer [Patterson 82], a load/store architecture with very simple operations and few addressing modes.

[2]As a result of this property, some have (disparagingly) referred to machines like this as still being Von Neumann Architectures. But unlike a Von Neumann machine, this is a highly parallel architecture. It, like a vector machine, does not have its parallelism in the control flow, but elsewhere.

- The instruction word completely controls all communications among the processors. That means that a piece of data is taken from a known location and placed into another known location, using known resources for a known amount of time. There is no sense of packets containing destinations, nor of hardware scheduling of a data transfer. Data transfers and their timings are completely choreographed in the code.

As a practical matter, there is no central program store. Instead, each processor fetches that portion of the instruction word relevant to it from its own program store. But on any given cycle, all processors fetch from the same address. There is a central control unit collecting test result bits and generating the next address information. This looks to the code generator as if there were one very long instruction word.

For those familiar with horizontal microcode, this control model is precisely the same. But there the similarity ends. VLIWs differ from horizontally microcoded machines in two key respects: they are far more parallel, and they are far cleaner. Although there have been lots of horizontally microcoded machines, no one has ever built a VLIW. Nor should they have; until recently there was no chance of producing reasonable code for them. A VLIW cannot be coded by hand — there are too many unrelated execution streams going on in parallel.

Figure 1 contains a table which points out some key differences between VLIWs, multiprocessors, and vector machines.

Why Get Excited About VLIWs?

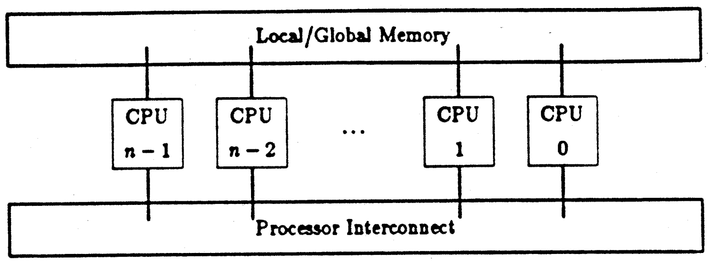
VLIWs aren't just another parallel architecture. Using a technique called trace scheduling [Fisher 83] [Fisher 81] [FERN 84], we can actually develop large amounts of highly parallel code for them without a great deal of programmer effort. We believe this sets VLIWs apart from multiprocessors and vector machines[3].

Trace scheduling was originally developed to generate less parallel code (horizontal microcode), and we had no idea of how it would scale up. Indeed, we were skeptical at first about the possibility of generating highly parallel code. But in fact, given code with appropriate properties (and we believe that most scientific code fits the model) trace scheduling does quite well.

Trace scheduling in all its glory is a complex procedure to explain, and a bear to implement. Nonetheless, the basic idea is quite simple. When the control flow of a segment of code is known at compile time, e.g. in code without conditional jumps, it's clear what to do — just schedule the operations. But there is very little parallelism in short segments of straight-line code. To handle conditional jumps, a trace scheduling compiler uses information about the dynamic behavior of the program to do greedy scheduling of operations. When the compiler can make good guesses — when many of the jumps are weighted heavily towards one leg — it's productive to be greedy. Otherwise, VLIWs are probably the wrong architecture to use.

Figure 2 goes into the mechanics of trace scheduling in slightly more detail. The only very detailed explanations of trace scheduling are found in [Fisher 79] and [Fisher 81], both of which are about code generation for horizontal microcode. More details of trace scheduling for VLIWs will be available soon.

[3]And especially from dataflow machines. Dataflow machines resemble VLIWs in their fine-grained parallelism, but they require untried hardware and architectural properties that may not eliminate all of the bottlenecks they're purported to. Furthermore, there is no evidence that human beings are capable of writing large amounts of side-effect free code. Worse still, code must be vectorized to handle large arrays, and it's not at all clear that other aggregates can be handled in any reasonable way without excessive copying.

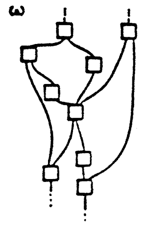


Multiprocessors, vector machines, and VLIWs all share the above structure. Despite their overall similarity, the architectures differ radically in their control flow, in the connecting of processors, in the accessing of memory, and (most importantly) in the programming requirements of each architecture. The below table compares these points.

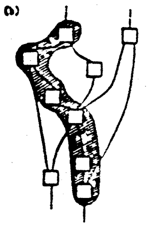
	Multiprocessors	**Vector Machines**	**VLIWs**
Flow of Control	Each processor has its own control flow. Processors must wait to synchronize on data availability.	One instruction at a time Von Neumann style, all processors doing the same operation. Mask bits may disable a subset of processors.	All processors fetch instructions from the same next address, but instructions differ from processor to processor. Von Neumann-style control flow with extremely long instructions.
Interprocessor Communications Bandwidth	Hardware scheduling of datapath resources, data packets with destination addresses, and buffering of unusable data.	Subvectors must be moved in patterns fitting the regularity of the interconnect and algorithm. Otherwise it is like a uniprocessor.	Individual movements of data completely specified at compile time. No need for addresses, runtime resource scheduling, or implicit buffering.
Memory Bandwidth	When possible, references are localized to a given processor. Otherwise expensive and slow global memory switch to allow processor/memory xbar.	Vectorized references allow full access to all banks. Unvectorized references like a uniprocessor.	Banks must be predicted at compile time to get maximal bandwidth for scalars and aggregates, which may be mixed in a single instruction. When the aggregate bank is unpredictable, it is like a uniprocessor.
Programming Requirements	Code must be broken into relatively independent tasks which minimize communications and synchronization.	Code must be expressed as regular operations on aggregates. Matchup must be made between inherent regularity of code and hardware.	Not likely to be hand-coded. Compiler must do greedy scheduling (e.g. *trace scheduling*) and elaborate run-time analysis to choreograph individual data movements and other resource selections.

Figure 1: A Comparison of Multiprocessors, Vector Machines, and VLIWs

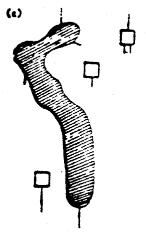
LOOP-FREE CODE



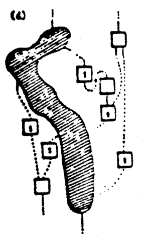
We start with loop-free code that has no back edges. Given a reducable flow graph, we can find loop-free innermost code. Figure (a) to the left shows a small flow graph without back edges.



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 figure (b), a trace has been selected from the flow graph.



Preprocessing prevents the scheduler from making illegal code motions between blocks. 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 scheduler is then permitted to behave just as if it were scheduling a single basic block, paying no attention whatsoever to block boundaries. In figure (c), the trace has been scheduled but not rejoined to the rest of the code.



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 figure (d), the new, uncompacted code appears at the code splits and rejoins. Then we look for our second trace, again looking 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 follow this proceedure until all the code is compacted.

Trace scheduling can be trivially extended to do software pipelining on any innermost loop. We simply unroll the loop for k iterations; the loop body can contain arbitrary flow of control. The unrolled loop gets compacted just as loop-free code to the left. At right is a flow graph of a fully scheduled loop that has been rejoined to the rest of the code.

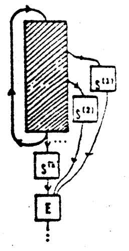


Figure 2: A Brief Description of Trace Scheduling

The ELI Project

The ELI (Enormously Long Instructions) Project has been underway at Yale since 1980. It centers about the construction of the BULLDOG compiler[4] (now written in compiled LISP on a DEC 2060), and, eventually, the construction of the ELI, a VLIW machine. Notice that we have the horse before the cart. The compiler has been running for over a year. We still haven't fully designed the machine it generates code for, instead we generate code using a tabular description of the ELI. By designing a small, pre-prototype machine and building code generators at the same time, both efforts polish each other. This allows us, within the bounds of our general model, to change aspects like:

- The repertoires, number, and placement of functional units,
- The size, placement, timings, and number of ports in register banks,
- The sizes and speeds of memory banks,
- The topology of the processor interconnect,

and so on. Our machine model prohibits features which might not scale across implementation technologies or to large, highly parallel machines. Thus we include no N-port memories, massive alignment networks for connecting memories to processors, or "hotspots" which need faster logic than the rest of the machine.

We're tuning ELI characteristics by running the compiler on a wide range of scientific code. Therefore it's impossible to give a fixed description of the machine. On any given day, we have our current best guess at the setting of the parameters. That notwithstanding, here's a current view intended only to give a sense of the scale of the machine:

Processors: Each of the 8 processors contains several functional units, all capable of initiating an operation each cycle. Our current best guess is two integer ALUs, one pipelined floating ALU, one memory port, several register banks, and a limited crossbar for all of these to talk to each other. Thus if we build this to a 200ns. cycle time, we will have 4 (operations per processor) x 8 (processors) x 5 (million cycles per second), or a potential of 160 RISC MIPS using technology that is far from the fastest available.

Memory System: Our memory system is a general alignment network, allowing the delivery of any memory word to any processor, which is very narrow: one word per cycle. This alignment network is locally bypassed at each processor, allowing 8 references per cycle when banks can be statically determined. When the compiler can decide what bank a particular reference is to, we get high memory bandwidth; when it can't, we default to the global network. Bank disambiguation almost always succeeds in the programs we look at, as long as we use a few tricks to help it out. This allows us to have have enormous bandwidth without vectorizing our code, and without building an expensive or unrealistic alignment network. This is more fully described in [Fisher 83].

Processor Interconnect: This is the most unsettled design parameter. As VLSI changes the cost balance between interconnect and function, one is led to evaluate system cost/performance in terms of the effective use of the provided interconnect. From simulations we have found that various topologies do not seem to differ greatly in their performance; what matters is the overall functional unit to interconnect ratio. Currently we have 4 half-duplex bidirectional lines to/from each processor. With the processors arranged in a circle, 2 lines connect to the nearest neighbors, and the other lines each go about 1/3 of the way around, one clockwise, one counter-clockwise.

Besides the actual machine, we also generate code for an idealized version of the ELI. It considers all of the above parameters at their most optimistic, and gives us a measure of the parallelism we're actually finding, as well as a notion of how well our choices are doing.

[4]The BULLDOG compiler is partially described in [FERN 84] and [Fisher 83], and will be more fully described elsewhere soon.

How Much Parallelism Do We Find?

We informally divide our suite of source programs into three broad classes:

Type P1 Those which (we believe) stand a chance of obtaining significant parallel speedups using an ordinary multiprocessor or a vector machine without significantly changing the algorithm. Doing so still might require significant programmer effort. Examples (among many) of these include an FFT, convolution, and matrix multiply.

Type P2 Those which appear on the surface to have potential parallelism but are too irregular or interwoven to fit into type P1. These include, among others, a prime number sieve and LU-decomposition[5].

Type S Those which seem too sequential to allow any significant speedup due to parallelism. Examples of these include various series summations, dot product[6], iterative solvers, etc.

We believe that types P2 and S are much more common in real life, while type P1 is found more commonly in Museums of Parallel Processing. Type P1 is also found in the inner loops of programs which are otherwise types P2 and S. It might seem sufficient to build parallel hardware to handle type P1, and not deal with types P2 and S, and sometimes it is. But we believe that even among most applications with type P1 code in the inner loops, not enough of the running time is in the inner loops to justify special hardware.[7] A few applications do have that property, signal processing probably being the most notorious. We believe that because of this property, these applications have had an influence on parallel processing almost to the exclusion of all the other things we want to do fast.

[5]A small algorithmic transformation makes LU-decomposition suitable for a vector processor. We use the straightforward statement of the algorithm. It evidently took years before people in the scientific programming community noticed this transformation, even though this code was the subject of much attention.

[6]This is a place where both vector machines and the BULLDOG compiler may be helped by the work of David Kuck's group at The University of Illinois [Padua 80]. Their source transformer can probably solve the recurrence and improve performance on this program.

[7]For example, if 80% of the code fits very well, and parallelism reduces that nearly to zero, the maximum possible speed up is under a factor of 5.

Using the code generator for our idealized ELI, we get the following general results on the test bed of programs we currently run:

Type P1 We find all the parallelism the hardware will allow. Although these programs might also run in parallel on other architectures, our parallelism is found without the programmer having to alter the natural expression of the algorithms[8].

Type P2 We again find all the parallelism the hardware will allow. We find it even if the inner loops contain conditional flow of control and data-precedence between loop iterations. We find it even with widely scattered references to array elements. Our individual control over memory accesses, and our ability to differentiate the banks being referenced, allow us to maintain a high memory bandwidth in situations that would reduce other processors to uniprocessor speed. This is perhaps the most encouraging fact about VLIWs and the BULLDOG Compiler.

Type S We generally get a factor of five or more speed up. These are programs that we believe would be impossible to speed up at all with other processors.

For more details about the workings of the BULLDOG Compiler, see [FERN 84].

Summary

The BULLDOG compiler, which extracts a large amount of parallelism from sequential code, has demonstrated that VLIWs may be the most attractive alternative for scientific, highly parallel computation. This has encouraged us to actually build a VLIW, and we are currently designing one. The BULLDOG compiler generates code for the machine we are

[8]Occasionally the compiler asks the programmer simple questions which have always (so far) had obvious answers. For details of the BULLDOG interactive assertion facility, see fern8.

designing, and we are now altering the machine characteristics to perform well in our simulations on a wide selection of code.

Acknowledgements

Main participants in the ELI project have included John Ellis, John Ruttenberg, Alexandru Nicolau, Mark Sidell, Doug Baldwin, Abhiram Ranade, Richard Kelsey and Charles Marshall. Dennis Philbin edited this paper. Judy Sexton helped in the paper's preparation.

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

References

[FERN 84] Joseph A. Fisher, John R. Ellis, John C. Ruttenberg, and Alexandru Nicolau. Parallel Processing: A Smart Compiler and a Dumb Machine. In *Submitted to SIGPLAN '84: Symposium on Compiler Construction* . ACM, June , 1984 .

[Fisher 79] 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.

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

[Fisher 83] Joseph A. Fisher. *Very Long Instruction Word Architectures.* 253, Yale University, Apr, 1983.

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