COMPUTER SYSTEMS ARCHITECTURE AT YALE

The Enormous Longword Instruction (ELI) Machine
Progress and Research Plans

Joseph A. Fisher
July, 1982

Research Report #241

This research is sponsored in part by the National Science Foundation (NSF) under grants no. MCS-81-06181 and MCS-81-0746, in part by the Office of Naval Research (ONR) under grant number N00014-76-C-0277, and in part by the Army Research Office (ARO) under grant number DAAG29-81-K-0171. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the NSF, the ONR, the ARO, or the US Government.

Table of Contents

COMPUTER SYSTEMS ARCHITECTURE AT YALE

The Enormous Longword Instruction (ELI) Machine
Progress and Research Plans

1. Introduction

The overall goal of the Yale Computer Science Department's Attached Processor Project systems group is to improve dramatically the *practical* state of the art in CPU-bound scientific computing. Specifically, we are building a very long (probably over 500 bits) instruction word machine, the ELI-512. A machine with this much irregular parallelism can reasonably be coded only in high-level languages; this requires state-of-the-art techniques in compiling horizontal microcode. An effective approach to this problem, **trace scheduling**, has been developed at Yale over the past three years.

Longword instruction machines are now quite popular and may offer the best alternative for obtaining supercomputer power at a fraction of its current cost. Unfortunately, they are already being built too wide for people or today's compilers to generate significant quantities of good code for them. *Without this or similar work, there is little chance that more usable wide-word architectures will be commercially developed.* It is an aim of this project to cause such commercial development to occur.

1.1. Project Goals

The following are key steps in building the ELI-512:

- **Parallelism measurements.** We are carrying out experiments to measure the parallelism available for Longword Instruction Architectures to exploit. The measurements are being made on commonly used scientific programs, and will aid architectural decisions in the design of the ELI-512.
- **Compiling for existing machines.** The most popular Longword Instruction machines, though quite limited in their parallelism, are very difficult to code. Trace scheduling can significantly improve this situation at low cost. Writing compilers for current machines will have an immediate payoff as well as honing our code generation techniques.
- **Compiling for, designing, and building the highly parallel ELI-512.** The Enormous Longword Instruction architecture will not be usable unless and until we can compile ordinary high level languages into its long instructions. The extensions to trace scheduling necessary for this still need proving out and tuning. We will not build the ELI-512 until our ability to compile for it has been demonstrated.
- **Allowing the compiler to design highly parallel customized processors.** Specialized VLSI attached devices will be tailored to especially important compute-bound code. Techniques for compiler-specified design of such devices will be a natural outgrowth of the code compaction techniques.

We are cooperating with the computer science department's numerical analysis group. They are studying attached processor issues themselves, will provide a significant applications test-bed, and will be the initial user community for the ELI-512.

2. Longword Instruction Architectures and Compaction

2.1. What Are Longword Instruction Architectures?

The defining properties of Longword Instruction Architectures are:

1. There is one central control unit issuing a single wide instruction per cycle.
2. Each wide instruction consists of many independent operations.
3. Each operation requires a small, statically predictable number of cycles to execute. Operations may be pipelined.

Restrictions 1 and 3 distinguish these from typical multiprocessor organizations.[1] The most familiar examples of Longword Instruction architectures are horizontal microcode and some specialized scientific processors, such as the Floating Point Systems machines. These machines are, however, quite limited in their available parallelism.

Longword Instruction machines may have large numbers of identical functional units. When they do, we do not require that they be connected by some regular and concise scheme such as shuffles or cube connections. A tabular description of the somewhat ad hoc interconnections suffices for our purposes. This makes the use of Longword Instruction machines very different from machines with regular interconnection structures and/or complex hardware data structures.

2.2. Automatic Code Parallelization For Longword Instruction Architectures is Practical Today

Machines with highly regular hardware structures are elegant and appealing. Writing code for them can be a pleasant, if quite challenging, experience. When the regularity of an algorithm can be altered to fit that of the hardware, the results are most satisfactory. By contrast, Longword Instruction machines are ad hoc and unintuitive. Writing barely passable code is almost always a wrenching process, error prone and tedious. So much for hand coding. What about compiling? Both regular machines and Longword Architectures (like all parallel architectures) present special problems in compiling. There has not been a single popular compiler for either class of machine that could compete with hand code.

When dealing with machines having regular hardware structures, the match between

---

[1]Longword Instruction Architectures do not fit neatly into any of the taxonomies of parallel processor organization.

algorithm regularity and hardware regularity is typically poor. It takes the talents of a very sophisticated programmer to rearrange the algorithm to fit. Commonly even this cannot be done, and it is often impossible to realize *any* parallelism at all. Without the ability to find such a match, it is unlikely that we can use these architectures for general purpose scientific code. Despite interesting research into automating this process, real progress seems a long way off.

By contrast, using extensions of trace scheduling, we feel we have solved every major theoretical problem in the way of good code generation for Longword Instruction machines. Our scheduling techniques find an adequate match between Fortran (for example) programs and hardware. The techniques, described below, require neither the discovery of elegant algorithm transformations nor sophisticated parallel programming skills.

2.3. Compaction

Compaction is the scheduling of several independent small instructions into common cycles to take advantage of wide instruction word parallelism. For basic blocks[2] of code, this resembles job-shop scheduling, but conditional control flow greatly complicates compaction. Since most parallelism is found beyond block boundaries, special techniques are needed to deal with conditional flow of control. *Trace Scheduling* [1] is a technique for replacing the block-by-block compaction of code with the compaction of long streams of code, possibly thousands of instructions long. The major characteristics of trace scheduling are:

1. Dynamic information is used at compile time to select streams with the highest probability of execution.[3]
2. Pre- and post- processing allows the entire stream to be compacted as if it were one basic block. *This is the key feature of trace scheduling. By using good scheduling heuristics on long streams, large quantities of far reaching parallelism may be efficiently identified.* Thus we avoid a case by case search, which would be far too expensive on such large streams and would be unlikely to yield much parallelism.
3. Preprocessing is done to prevent the scheduler from making absolutely illegal code motions. The constraints on code motion are encoded into the data precedence graph, making those constraints look just like the usual data precedence constraints that a basic block scheduler would have to cope with. The scheduler is then permitted to behave just as if it were scheduling a single basic block.
4. After scheduling has been completed on a stream, the scheduler has made many code motions which, while potentially legal, will not correctly preserve jumps from (or rejoins to) the stream to the outside world. To make these

---

[2]No jumps in except at the beginning, no jumps out except at the end.

[3]This cuts down on the potential code explosion in the state recovery code.

code motions actually legal, 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.

5. The most frequently executed code beyond the stream, including the new state recovery code, is then compacted as well, possibly producing more state recovery code.

Eventually, this process works its way out to code with little probability of execution, and more mundane compaction methods are used which do not cause the possible code explosion of the state recovery operations.

Trace scheduling won't do well on code which is dominated by highly unpredictable jumps (e.g. parsing loops or tree searches). Fortunately, one can make a good guess at the direction of most jumps in typical scientific code. These guesses may be derived using a tool to run debugged code on samples of data, or may be programmer supplied.

Inner loops of code present a special opportunity for code compaction. *Software pipelining* is a technique used by hand coders to increase parallelism. It involves rewriting a loop so that pieces of several consecutive original iterations are done simultaneously in each new iteration. Although interesting work has been done towards automating software pipelining (e.g. by Leslie Lamport), the techniques do not seem to lend themselves well to general code. Trace scheduling, on the other hand, may be trivially extended to do software pipelining on any loop. It is sufficient to simply unroll the loop for many iterations. This will yield a stream, which may be compacted as above. All the intermediate loop tests will now be conditional jumps in the stream; they require no special handling beyond that always given conditional jumps. While that 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. The extra space is not of major importance here.

3. Architecture Type Evaluation

We are experimentally measuring just how much parallelism is available for Longword Instruction Architectures to exploit. The series of three experiments is now halfway completed.

3.1. Experiment 1:

Hypothesizing infinite hardware and a compiler that can reconcile all dynamic ambiguities, we have found about a factor of 50-90 speed up available on common number-crunching Fortran code [3]. Many programs offered unlimited parallelism, constrained only by the size of the data. This is a considerable improvement over earlier experiments which stopped locating parallelism at basic block boundaries, and thus found little parallelism. This optimistic result leads us naturally to experiment 2, in which we are trying to more narrowly assess how much of this available parallelism we can cull from programs.

3.2. Experiment 2:

Hypothesizing much more realistic hardware and using a completely realistic compiler, we are generating actual code and seeing how much parallelism can be found. The compiler uses trace scheduling and memory disambiguation techniques. It has no more information than any compiler we would be likely to build. The architectures we generate code for are in several respects unrealistic, in that they will not have the arbitrary details typically found in hardware structures making code generation more painful and less effective. They will also have memories that allow n accesses per cycle, with no register structure needed. While this unrealistically avoids the memory communication bottlenecks of most current machines, we will design much of that bottleneck away later anyway (see below). Otherwise, these are realistic designs. John Ellis is completing experiment 2, using Josh Fisher's trace scheduling routines, and Alex Nicolau's memory disambiguation code,

Experiment 2 will tell us how much parallelism we have a chance to recover. The potential impediments are: lack of dynamic information (which we hope Alex Nicolau's code and trace scheduling will sufficiently deal with) and code explosion from trace scheduling (for which there are untried "space saving" techniques).

3.3. Experiment 3:

After completion of experiment 2, John Ruttenberg will take intermediate code traces it produced and will generate code for an early draft version of the ELI. He will use his *delayed binding* techniques (described below). This will still be far from a full-blown compiler in two respects. First, he will not do full trace scheduling — he will stop at the compaction of each trace rather than produce the complex state recovery code. Second, he will be generating code for a machine that will probably never be designed completely enough to simulate. However, we do not expect that generating code for a realistic machine will generate proportionately more recovery code than for the architecture used for experiment 2. We also do not expect the architecture used here to execute code in significantly fewer instructions than the ELI. Thus by comparing his compacted traces to those produced in experiment 2, we feel we can legitimately extrapolate how successful we will be in compiling for machines like the ELI.

4. Compiling for existing machines

Whether or not there is a large amount of parallelism available for trace scheduling to exploit, trivial (in their parallelism) Longword Instruction machines are being built and are the most popularly sold attached processors. These have a history of being terribly difficult to program. This is due to their small amount of parallelism, which is still enough to vex most programmers and all compiler writers.

We have a Floating Point Systems FPS-164 (delivered 3/15/82) for which we will be producing a trace scheduling compiler. This compiler will only begin to use the code generation techniques being developed at Yale. Nevertheless, we expect it to produce code

competitive with hand code. It will be a drastic improvement in the usability of the system. The FPS-164 is similar enough to the FPS AP-120b that our compiler could be adapted to generate AP-120b code. There have been 1500 AP-120b's sold to date, so there will be a considerable audience for such a compiler.

Neta Amit wrote portions of a first pass throw-away compiler generating FPS-164 code from a subset of FORTRAN. This has given us a handle on some of the issues, and as soon as we see where the FPS software stands, Charles Marshall will start the compiler in earnest. We will use existing code fragments (e.g. the FPS FORTRAN compiler) wherever possible and we do not expect our result to be a polished, commercial product. We expect that such a product could be produced from our results in a straightforward fashion.

5. The Enormous Longword Instruction (ELI) Machine

Current Longword Instruction machines are not only badly limited in their degree of available parallelism, but also hinder the ability of a very smart compiler to generate highly parallel code for them. It is not likely that commercial products will change dramatically in these respects unless we build a dramatically successful prototype. This phase of the project will have three main tasks:

1. The design and simulation of a very wide attached processor
2. Code generation for the processor
3. Building the processor

This is a continuation of the work done in the earlier stages of the project, but also implies work in design automation.

5.1. Processor Design

We will define, simulate, and build a very wide attached processor. The ELI-512 will be unique not only because of its very wide instruction word, but also by being completely tailored to a trace scheduling compiler. The design will be guided by the results of experiments 2 and 3. The CPU will completely rely on a very smart compacting compiler — no other means of code production (e.g. by hand) will be reasonable. Since the machine will be attached to some popular host, we will avoid many difficult details.

A very rough first draft of the ELI-512 has the following characteristics:

1. 16 *clusters*, half of them centered around memory banks, and half around floating point ALUs. Each memory cluster (MCL) will have a memory bank (or banks), an address/integer ALU, and two register banks. Each floating point ALU cluster (FCL) will have an ALU (probably differing in repertoire from FCL to FCL), and 2 register banks. Within each cluster will be a nearly full crossbar among all of the access ports of the functional units. The clusters themselves will probably be arranged in a ring, with nearest neighbor connections and somewhat random connections going among nonneighbors.

The intercluster connections will participate in the limited crossbars of the clusters involved. As VLSI gets faster, we may want to build single chip clusters. For the time being, we're planning that the first prototype will be Shottky or ECL MSI.

2. An (n+1)-way conditional jump control mechanism, allowing each instruction to do a conditional jump based on n independent test conditions.[4] An earlier, similar mechanism for doing this efficiently has been described in [2]. The compiler used for experiment 2 has implemented (n+1)-way jumps.

3. An interleaved main memory with heavy reliance on bulk memory. A key aspect of the ELI-512 is that we expect the compiler to be able to deduce the bank, if not the exact address, of almost all the memory references.[5] Thus the compiler will be able to schedule memory references without the overhead of dynamic address distribution. It should be possible, when things work out right, to have each of the banks busy almost every cycle, providing a huge memory bandwidth. This would be impossible using a typical memory system. For those cases when the bank cannot be predicted, there will be a "back-door" into the memory system, with hardware arbitration of the address. This will never steal cycles from individual banks (getting the processor out of synch). Instead, only when the result is needed, but not available, will the rest of the CPU freeze until the value pops out. Chasing down pointers will be quite expensive in this system, since the back door will be as slow as ordinary memory systems, and will severely constrain parallelism.

4. Possible memory system numbers are:

Main Data Store	16 8kw. banks (that's only 1 Mbyte of main store at 64 bits/word). This will enable us to build very fast distributed memory, and keep it on chip when the clusters are single chips. We'll rely on bulk memory and on paging designed around the trace scheduler's dynamic control flow estimates to make running large scientific programs on a 1 Mbyte store practical.
Bulk Memory	8 32 Mbyte bulk memories. (That's 256 Mbytes worth. At \$1K/Mbyte, a reasonable number for 2-3 years from now, cost is \$256,000. Rather acceptable as a disk replacement, considering what one gets!) We can think of each MCL as having its own bulk back-up, though the interconnect may be more complex than that. Each bulk memory will have a 64 Mbyte/sec. bandwidth after a small startup time.
Control Store	This will be distributed within the clusters and will be

[4]Probably 3-4 independent tests will suffice.

[5]Were this not the case, trace scheduling would probably fail to keep such a wide CPU busy enough for other reasons as well.

cached, though we haven't decided how or from where yet. Again, the trace scheduler's dynamic estimates will be used to improve caching performance.

The above has come out of many conversations, principally involving John O'donnell, John Ellis, John Ruttenberg, and Josh Fisher, along with Charles Marshall, Bill Gropp and Olin Shivers. We hope to get a first draft high level simulation working by October 1982.

5.2. Compiling for the very wide attached processor

Compiling for the ELI will necessitate the much more general code generation techniques being developed by John Ruttenberg, especially those concerned with the memory hierarchy and functional unit selection. Generally, these techniques fall under the heading of "delayed bindings", in which compiler bindings are not done until compaction time. This makes the ordinary search procedures associated with scheduling, already time-consuming, too broad to be tractable. Thus, to do this effectively, intermediate level constructs are bound in their entirety and scheduled whenever they are bound or considered at all. This is called *operation scheduling* and leads to a different style of scheduling from that encountered in *list scheduling*, as previously used in compaction.

John Ruttenberg has versions of his code working on toy architectures and will be extending them to the larger machine. This will also require that Alex Nicolau develop a more robust set of memory disambiguation tools, and it will require a front end far better than the one used for the experiments.

5.3. Building the very wide attached processor

We will not wire a single component to a board until we have an effective (if not polished) compiler for the device being built, nor until we have completely simulated and debugged the design.

Our assumption is that it will be 5 or more years before VLSI speeds are competitive with today's bipolar MSI devices. Thus we will initially concentrate on MSI design, but will make sure that we are ready with VLSI expertise and tools so that we can make the move when appropriate. The state of the art in physical design tools appears to be fairly good, and we will be importing whichever seem to fit our needs. Logical design tools seem to be nearly absent. Thus we are concentrating on building a logical design system.

At the center of the system is a textual simulator, called the *Simyalelator* [4]. It uses a message passing scheme to allow clean communication among several levels of use. The Simyalelator has as its atomic device a *box*. Boxes are externally connected to other boxes via a powerful vector connection method. The behavior of a box may be described using two modes — through a Lisp description of its behavior or through boxes contained within. Typically, both forms of description are given, and the box is tested by comparing the behavior of the two. When a box is used in larger devices, it's Lisp description may be used for efficiency. A flag controls the selection of simulation mode for a given box. Since

this is embedded in Lisp, we have hooks into its innards allowing other systems to interface easily. The Simyalelator was designed by John Ruttenberg and John Ellis and implemented by John Ruttenberg.

The Simyalelator is the lowest level design automation tool we are building. We will interface it to lower level physical tools, which we will port, and to higher level logical tools, which we will build. Among the logical level tools will be a powerful graphics editor. Using Apollo graphics and backend computing (if necessary), we will build upon the Yale Computer Science Department's expertise in screen editors [5] to produce the "best ever" box editor. This will include facilities for editing on screen circuits, and for viewing circuits with logical zooming and locus of interest sub-circuits, as well as more mundane facilities. It will probably be *constraint oriented.* The graphics system is being implemented by Doug Baldwin and Richard Kelsey.

As of July, 1982, we have implemented (besides the Simyalelator itself) a locus of interest viewing system. This lays fully labeled boxes out on an Apollo screen, according to how close they are to a user specified set of "interesting" boxes. The layout is done to facilitate wire routing, which is done automatically. The simulator may then operate in "movie mode", in which one watches the values of wires changing as time progresses.

6. Compiler Directed Design of Customized Processors

VLSI will make it economical to design processors to carry out specialized tasks. The numerical analysis group will be identifying important algorithms, from which we will be generating code for the ELI-512. Eventually, however, we will want to build devices *tailored to running the compute bound kernels of these important algorithms.* We anticipate that this will significantly speed up the execution of these algorithms.

Trace scheduling lends itself well to allowing the compiler to pick its own design. At each operation scheduling point, the compiler is asking "do we have enough resources and the right interconnects to execute this instruction now?" There is no reason not to change that question to: "Can we specify enough resources and the right interconnects to execute this instruction now? At what cost?" We will need to develop at least the following:

- A parameterized version of the very wide attached processor.
- A way of allowing the code generator to specify parameter settings.
- Cost/speed tradeoffs to permit some estimation short of "full speed ahead" when selecting resources.
- Some minimum criteria for assuring that there is enough there to allow *any* code to be generated on the device, so that the rest of the program (beyond the kernel) may be generated, and so the program may be slightly changed after fabrication without catastrophe.
- A tie-in to the design automation system, so that the processors may be built without human intervention. This should also generate programming manuals, etc., as well as compilers for the new device.

Important parts of this portion of the project are certainly a few years off.

References

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

2. Fisher, J. A. An Effective Packing Method for Use with 2^n-Way Jump Instruction Hardware. 13th Annual Microprogramming Workshop, ACM SIGMICRO, November, 1980, pp. 64-75.

3. Nicolau, Alexandru, and Joseph A. Fisher. Using an Oracle to Measure Parallelism in Single Instruction Stream Programs. 14th Annual Microprogramming Workshop, ACM SIGMICRO, October, 1981, pp. 171-182.

4. Ruttenberg, J. C., J. R. Ellis, and J. A. Fisher. The Simyalelator: User's Manual and Report. Yale University Department of Computer Science, In Preparation.

5. Wood, Steven R. Z: The 95% Program Editor. Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation, ACM/SIGPLAN, Portland, Oregon, June, 1981, pp. 1-7.