EFFICIENT HARDWARE FOR MULTI-WAY JUMPS AND PRE-FETCHES

Kevin Karplus and Alexandru Nicolau
Department of Computer Science
Cornell University
Ithaca, NY 14853

Introduction

Two recent trends in computer architecture have been increasing the size and complexity of microprograms: RISC machines, array processors, and VLIW machines are programmed directly in microcode, and CISC machines have large microcode programs that interpret higher-level machine instructions. The difficulty of developing and maintaining large microprograms suggests that they should be written in a high-level language and compiled by optimizing compilers. Conventional optimizing compilers have not been particularly effective for microcode compiling, because they optimize primarily within basic blocks (that is, segments of sequential code, uninterrupted by conditional jumps or jump targets), which are too small (3–5 instructions) to provide much code rearrangement. Hand coding, though slow and error-prone, has offered significant performance advantages over compiled microcode.

Recent advances in optimization techniques—notably, trace scheduling [Fisher81] and percolation scheduling [Nicolau84]—offer code rearrangement that crosses basic block boundaries. These code rearrangement techniques tend to cluster conditional jumps. Since conditional jumps make up 15–33% of the initial microcode, combining the conditional jumps of a cluster into a single multi-way jump offers substantial improvements in speed.

Various schemes have been proposed in the past for multi-way jumps, but they have generally been unsatisfactory. One common problem is insufficient generality to represent the clusters of conditional jumps found by the new optimization techniques. Another, potentially more serious, problem is that the multi-way jump mechanisms interfere with instruction pre-fetching.

A microcode memory system must operate at the speed of the instruction decoder and the data path. Although fast memories are available, they are small and expensive. Recent memory chip manufacturing trends are for cheap, large memories that are relatively slow. With current processor and memory speeds, microcode instruction cycles are already 8–16 times faster than access times for large memory chips.

In this paper we propose a mechanism that allows general, multi-way jumps with parallel pre-fetching, so that slow, inexpensive memory can be used without speed penalties. The architecture evolved as the target machine for a new optimization technique, percolation scheduling, which has been described elsewhere [Nicolau84].

Our pre-fetch mechanism relies on operating relatively slow pre-fetch units in parallel, each with its own memory. Pre-fetches can be started many cycles before the instructions are needed, *even when multi-way jumps intervene.* Pre-fetches for straight-line code are started automatically by the system, but targets for jumps have to be explicitly pre-fetched several cycles before the jump. With the pre-fetch units, instruction caches are not needed, and fast circuitry in the memory system is minimized.

Microcode size, a conventional measure of microcode quality, is less important in our scheme than in previous ones. Because we use slow, cheap memory, we can easily afford lower code density than conventional microcode systems. Global compilation techniques often increase the size of the microcode, since they concentrate on the main goal, which is the speed of the optimized program. Even so, our system does not waste memory profligately. Code will occasionally be duplicated for efficiency or to make the mapping into pre-fetch units easier, but no empty blocks of memory are needed, and the microcode word is only slightly larger than in conventional systems.

Although we stress the importance of compiling microcode, our pre-fetch system can be programmed by hand, with no more difficulty than in other pipelined microcode schemes. We have built a simulator for the architecture, and hand-compiled a few examples for it. In our hand-compilation tests, we have found that separating pre-fetches and jumps allows considerable flexibility, both in compacting code, and in choosing encodings for the microword.

A simplified version of the system could be used with more conventional architectures in place of caches or branch predictors for pre-fetching instructions. The compiler for such a system can be relatively simple, as it need only schedule pre-fetches, not rearrange data path operations and build multi-way jumps.

Previous Work

Several experiments [FostRise], [TadjFlynn], [NicFish] show that basic blocks are small, about 3–5 operations on average. On a pipelined or very-large-instruction-word machine, using

the hardware efficiently requires starting several instructions before the first one finishes. Since the instructions within a basic block usually have serial dependencies, code rearrangement has to range over several basic blocks. In horizontal microcode, several data path operations can often be started simultaneously. Unless several conditional jumps are evaluated at once, the ratio of jumps to straight-line code becomes unacceptably high.

Furthermore, the density of conditional jumps foils the effectiveness of the traditional pre-fetch mechanism in which the next instruction is pre-fetched sequentially. Traditionally, only one path through the program graph is pre-fetched, and programs are slowed at conditional branches, unless branch probabilities are predictably near zero or one. Fetching all branches simultaneously, on the other hand, requires prohibitively many data-paths from the test-unit to the memory banks and from the memory banks to the instruction register.

Many existing microengines allow several tests to be executed in parallel in each instruction. These tests, however, have limited flexibility, making them hard to use whether compacting microcode manually or automatically. The VAX, for example, has prespecified sets of tests that can be executed in any given instruction, using a mask to select any subset of these tests as active. The results of the tests are then and-ed into the jump address field to specify the target address[PLT]. Hardwiring the tests for the microinstruction loses the flexibility required to handle the variety of combinations of tests that arise when compacting general-purpose code. Other existing multi-way jumps allow a set of bits from the data path to modify the address calculation, generating 2^n way jumps. This is typically used for microcoded case statements such as opcode decoding.

A more flexible approach was suggested by Fisher[Fisher80]. His approach allows combining a sequence of n arbitrary tests into a single "vine" in which the first true condition determines the branch to take (see Figure 1). This scheme works well for the compaction method known as trace scheduling[Fisher81], which deals with only one path through the code at a time. Unfortunately, Fisher's approach cannot handle some frequent combinations of jumps, such as those in Figure 2.

Significantly, none of the existing methods allow extensive pre-fetching; they all assume that fetches started at the beginning of one microinstruction cycle will finish by the end of the same cycle. Although single-cycle fetches are feasible for small (1–4 k) microinstruction memories, they are impractical for large microstores and fast microengines. Ensuring that microinstruction fetching can be completed in 1 cycle requires either slowing down the microinstruction cycle to the speed of the available memory, or investing in expensive fast memory. We will describe a pre-fetch mechanism that starts fetching instructions several cycles ahead and can handle multi-way jumps.

Architecture for n-way jump and pre-fetch

A standard conditional jump has 2 branches (true and false). Thus n independent conditional jumps in a microinstruction can have $2n$ independent possible continuations. After evaluating n independent conditional jumps in parallel, execution would either have to continue in parallel on the resulting n mutually independent paths, or would require selecting 1 of 2^n paths.

```
        if t1 then L1
    elseif t2 then L2
              .
              .
              .
    elseif tn then Ln
    else Ln+1
```

Figure 1: Vine of tests, as handled by Fisher's multi-way jump.

```
if t1 then if t2 then L1
                 else L2
else       if t3 then L3
                 else L4
```

Figure 2: Decision tree that cannot be handled with vines.

Luckily, conditional jumps that are clustered when compacting microcode are usually not independent, instead they form a rooted DAG (directed acyclic graph), with tests in the internal nodes and continuations at the sinks. The n conditional jumps in such a DAG (of which Fisher's vine is a special case) choose one of at most $n+1$ continuations. In this paper, we describe a mechanism tailored for such multi-way jumps.

Our proposed architecture provides one data path instruction on each clock cycle. The data path can be simple, as in a RISC machine, or complex, as in a VLIW architecture. The only requirements are that any instruction can be started on any cycle, and that the time that each instruction takes is known to the compiler. The instruction and data memories are entirely separate, so that no conflicts can occur between data and instruction fetching.

No interrupt handling or fast context-switching is provided in our processor. Rather than slow down our main processor to handle these rare tasks, we could use auxiliary processors to handle instruction traps, page faults, and I/O. Sometimes, albeit rarely, we may need to "freeze" the clock on our main processor when data or instructions are not ready. State-of-the-art compiler techniques can schedule instruction and data pre-fetching early, so that clock freezing is rare enough not to degrade the performance of the machine.

In many contexts real context switching is needed, and simply freezing the processor is unacceptable. Saving the state of the entire data path and control system would require substantial extra hardware, which may impose a speed penalty even when no context-switching is done.

Saving the state of the control system is straight-forward, as each pre-fetch unit can save its state in the portion of the instruction memory that it handles. Saving the state of the data path is messier, as each functional unit may have several internal pipeline registers. An alternative approach permits interrupts only when inserting an arbitrary delay before an instruction will not affect the correctness of the program. The

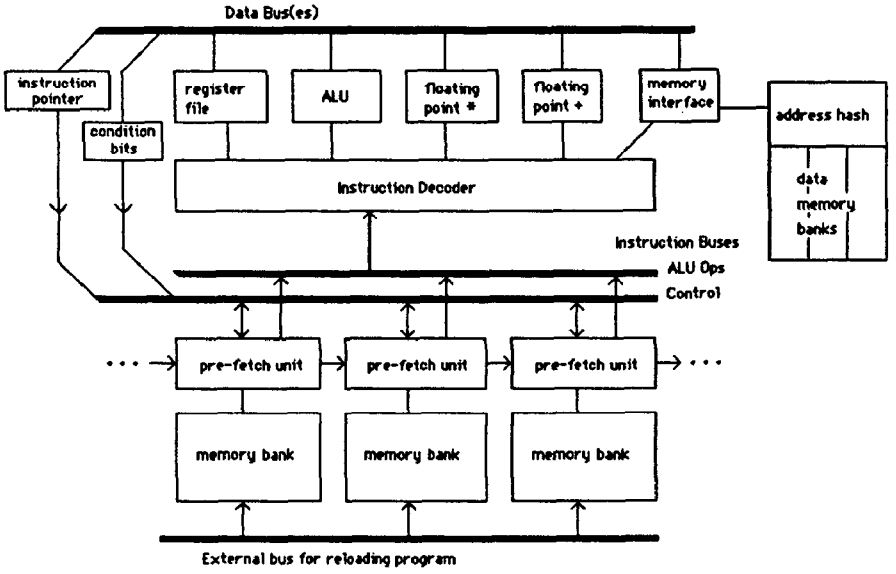

Figure 3: ROPE block diagram

operations in the data path can be completed while the pre-fetch units store their states. Context switches need take only a little more time than the latency of the slowest data path operation.

Since the compiler does complete flow analysis, it can easily mark the instructions where delays can be inserted. To guarantee short delays on servicing interrupts, delayable instructions must occur fairly frequently. If part of the code is so tightly scheduled that no instructions can be delayed, the compiler may have to schedule some no-ops to make interrupts feasible.

Program memory

The program memory must provide a data path instruction and a control instruction on each clock cycle even though the read time of the memory may be 10 clock cycles or more. Interleaved memory banks and instruction caches are the two standard approaches for obtaining high-performance program memory. Interleaved memory banks are fast for straight-line code, but impose large delays on each jump. Instruction caches work well at moderate processor speeds, but are difficult to design for the short cycle time envisioned for fast micro-engines.

Our design has interleaved memory banks, but avoids the jump penalty by using intelligent pre-fetch units, connected in a ring. We have named the architecture *ROPE*, for *Ring Of Pre-fetch Elements*. Figure 3 shows the block diagram for the ROPE machine.

Each of the 2^n pre-fetch units has its own block of memory, independent of the other units. On each cycle, exactly one pre-fetch unit, called the *active* pre-fetch unit, provides an instruction to the decoder. The other pre-fetch units may be either busy fetching words from their memories, or ready to become active. After an instruction is passed to the decoder, control passes from the active unit to a different pre-fetch unit. For normal straight-line code, control passes to the right from unit i to unit $(i+1) \bmod 2^n$.

For the machine to work without delays, instruction pre-fetches must be started well before the instruction is executed. Pre-fetches are started automatically for normal straight-line code. Since straight-line code proceeds from left-to-right across the pre-fetch units, it suffices for each unit that starts a pre-fetch to tell its right hand neighbor to start fetching the next address on the next cycle. Since multiple target addresses must be ready at jump instructions, jump targets are started with explicit pre-fetch instructions.

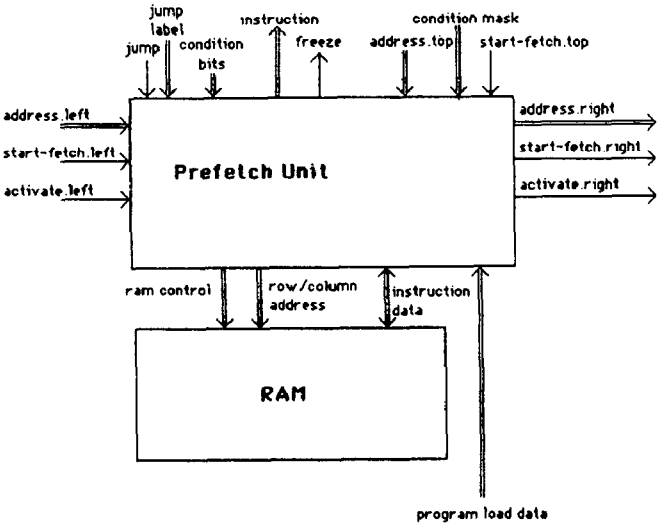

Figure 4: One pre-fetch unit.

Figure 4 shows a single pre-fetch unit. The signals on the left and right sides are passed around the ring, and the signals on the top communicate with the instruction buses. Note that in a ROPE with 2^n pre-fetch units, the bottom n bits of address select the pre-fetch unit, and do not have to be passed around the ring. The high-order part of the address that is passed

around the ring does not need to be changed between units, except when passed from unit $2^n - 1$ to unit 0, where it must be incremented.

A pre-fetch unit has two state bits that determine its behavior: *busy* and *target*. A pre-fetch unit is busy if it is in the middle of a fetch, and ready when it has data to be put on the instruction bus. A pre-fetch unit is a target if the word fetched was requested by an explicit **PRE-FETCH** instruction, and a non-target if the word was requested from the unit to its left.

The pre-fetch units can best be understood by examining what they do with the signals passed to them from the top or left.

- A **start-fetch.left** signal starts fetching **address.left** on non-target units, but is ignored by target units. Whenever a fetch is started, the unit sends a **start-fetch.right** signal on the next cycle, passing the address it is fetching to **address.right**. The unit becomes busy until the fetch is completed. Note that a **start-fetch.left** signal could be received by a non-target unit that is busy with a previous fetch, in which case the previous fetch is aborted. Several start-fetch tokens are usually being passed around the ring at once.
- The **activate.left** signal can be thought of as passing a unique *activate* token around the ring to execute straight-line code. A ready unit puts the available instruction onto the instruction bus, and signals **activate.right** on the next cycle. A busy unit freezes the data path until the fetch is completed, then puts the available information onto the instruction bus and signals **activate.right**. A jump signal inhibits the usual left-to-right passing of the activate token. Target units are not allowed to receive the activate token from the left.
- The **start-fetch.top** signal makes any unit a target unit. Fetching is started for the address **address.top**. The jump label is saved for comparison with future jumps. Only jumps with the same label can activate the target unit. The condition mask specifies under what conditions a pre-fetch unit becomes active. It is saved for future comparison with the condition bits, which are registers on the data path, set by explicit test instructions. The usual setting of condition-bits as side-effects of other instructions is too difficult to control for multi-way branching.
- On the next cycle after starting any fetch, **start-fetch.right** is signalled, with **address.right** set to the new address. If a unit receives a **start-fetch.top** and **start-fetch.left** signal on the same cycle, the **start-fetch.top** signal has priority, and the **start-fetch.left** signal is ignored.
- The **jump** signal is sent with a jump label. Each target with a pre-fetch for the labeled jump compares the current condition bits with the condition mask saved from the beginning of the fetch. If the mask matches, the unit will become active as soon as it is ready, freezing the processor if the fetch is not yet completed. If the mask does not match, the unit reverts to being a non-target unit, and starts fetching the instruction at **address.left**. The compiler is responsible for ensuring that exactly one pre-fetch unit responds to a jump signal.

Control instructions

The instructions for the machine consist of two parts: the data op, which controls the data path, and the control op, which controls the pre-fetch units. The control ops are:

NEXT: activate the next unit in line. This is the normal behavior for straight-line code, and requires no action from a controller outside the pre-fetch ring.

PRE-FETCH *address jump-label condition mask*: instruct the appropriate pre-fetch unit to start fetching the specified address. Normally the address will be a constant contained in the instruction, but sometimes will have to come from the data path—for example, to handle return from procedures and pointers to functions. The jump label and condition mask are stored by the pre-fetch unit for later matching.

JUMP *jump label*: Activate the appropriate target pre-fetch unit for the specified jump. Jumps are labeled so that pre-fetches can be moved past jumps during scheduling. Jump labels can be re-used when jumps do not conflict, so only a few different jump labels are needed. A conditional jump on a ROPE machine thus has three parts: **PRE-FETCH** instructions for the first instruction of each branch; test instructions to set the condition bits; and a **JUMP** instruction to start executing the desired branch. Separating the functions of a conditional branch allows considerable code rearrangement for efficient pipeline use.

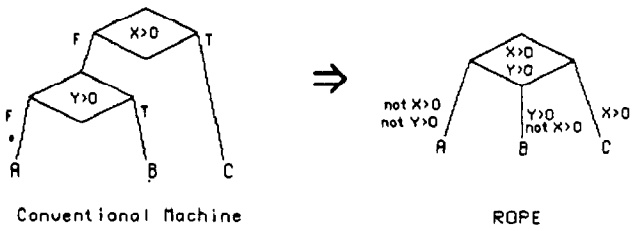

Figure 5: Combining two conditional jumps into a three-way jump.

Example of multi-way jump on ROPE

On a conventional machine, jumping to one of three addresses requires two conditional jumps, as in the left half of Figure 5. On a ROPE machine, a single three-way jump is used (right half of Figure 5). Figure 6 shows how this example can be scheduled onto the multiple pre-fetch units of the ROPE machine. Each column shows the activity of one unit, each row representing a single cycle.

Although ROPE's pre-fetch units are fairly cheap, it would be naïve to assume that we could build a machine with hundreds or thousands of them. How many do we need to support multi-way jumps? Let us assume that a pre-fetch unit becomes ready F cycles after a fetch is requested for it. On the cycle after a jump each of the targets will need to be ready, and the $F - 1$ units to the right of each target must have started pre-fetching. Thus a k-way jump requires least kF pre-fetch units. A four-way jump with a six cycle fetch time requires 24 pre-fetch units. If jumps are close together, as they are in the example shown in the next section, each target branch needs to pre-fetch only up to the next jump instruction, and fewer units

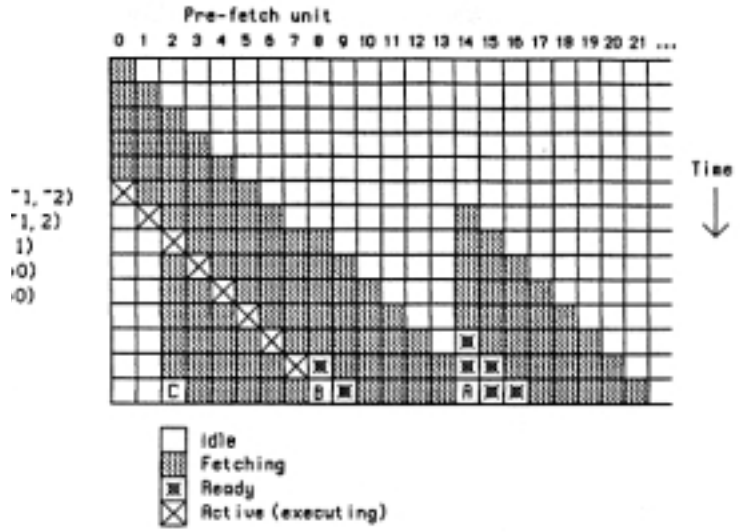

Figure 6: Possible execution sequence for a three-way jump.

are needed. A ROPE machine with 32 or 64 pre-fetch units should achieve almost all the possible speedup of this architecture. If not enough pre-fetch units are available, programs will be slowed down, but only by the number of pre-fetches that do not fit (not by the time required for each fetch), since pre-fetches have no data dependencies, just resource availability constraints.

The compiler (or hand-coder) must schedule the pre-fetches and assign code addresses to minimize the waiting for instruction fetches. For tree-structured control flow with infrequent branching, scheduling pre-fetches and assigning code addresses is easy. If the branching is frequent and not enough pre-fetch units are available, delays are unavoidable with any schedule. Assigning addresses is difficult for a code block that has multiple predecessors, such as the entrance to a loop or the statement after an if-then-else. Such a code block may need to be duplicated to avoid conflicting requirements on its placement.

The main cost of a jump instruction on conventional machines is the fetch time for a non-sequential instruction. With our architecture, the pre-fetches, tests, and jumps can be scheduled independently, and therefore do not slow the machine. Separating pre-fetches, tests, and jumps may improve performance significantly, even without multi-way jumps.

Example

To illustrate the ROPE machine, we've chosen a simple loop that finds the subscript of the minimum in a one-dimensional array of floating-point numbers. This loop is number 24 in the Lawrence Livermore floating-point benchmarks [McMahon], so comparisons to existing machines are easily made. Figure 7 shows the loop in its initial form. The code was hand-compiled and optimized, using transformations that are feasible for a state-of-the-art compiler.

The data path assumed for the example is a simple pipelined unit that accepts one instruction per cycle. The architecture is a simple load-store architecture, with 3-register arithmetic operations. Register-register operations take 1 cycle, integer arithmetic takes 2, and loads, stores, and floating-point operations take 6. Instruction pre-fetch is assumed to take 6 cycles,

```
m=1;
for (k=2; k<=n; k++)
    {if (x[k] < x[m])
        {m=k;
}
    }
```

Figure 7: Livermore loop 24, find the minimum of an array.

except when a pre-fetch is requested for the address already in the pre-fetch unit, when no extra delay occurs.

With no code optimization, the loop takes 25 cycles per iteration. Standard optimization techniques do not shorten the dependency chains in this example, yielding a loop that still takes over 20 cycles. Using sophisticated transformations that allow code motion past branches reduces the cycle time to 7 cycles. Unrolling the loop to double its length allows us to reduce the time to 11/2 cycles per iteration, but only at the expense of very careful instruction ordering. Unrolling the loop to triple its length reduces the time to 15/3 cycles per iteration. Figure 8 shows the machine instructions for the loop unrolled three times, and Figure 9 shows a trace of one iteration. Note that some jumps appear to start before their pre-fetches are finished, since we are taking advantage of the pre-fetch units not starting a new fetch when the address is the same as for the previous fetch.

Format of the microinstruction word

We are still considering different formats for the microword. The format of the instruction word on the instruction bus is presented here. Note that the instructions in memory can be more tightly encoded, since the pre-fetch units can easily act as instruction expanders.

The encoding of data path operations is, in principle, orthogonal to the encoding for the control instructions. Only the control encoding is presented in this paper. Tight encoding may only be possible, however, if data path operations requiring large instruction words do not occur in the same cy-

```
xm=x[1];    xk1=x[2];    xk2=x[3];
n=1;        ak=&x+4;     nop^2;
loop:
     b1 = xk1<xm;
     b2 = ak>n+&x+2;
     xk3=*ak;            ak++;
     b3a = xk2<xm;       b3b= xk2<xk1;
     b4 = ak>n+&x+2;
          if (b2) goto exit
          if (b1) {xm=xk1; m=ak-(&x+3)}
     b5a = xk3<xm;       b5b= xk3<xk2;
     xk1=*ak;            ak++;
          if (b4) goto exit
          if (!b1&&b3a || b1&&b3b) {xm=xk2; m=ak-(&x+3)}
     b6 = ak>n+&x+2;
     xk2=*ak;            ak++;
     if (b6) goto exit
     if (    b1 && !b3b && b5a
         ||  b1 &&  b3b && b5b
         || !b1 && !b3a && b5a
         || !b1 &&  b3a && b5b)
                {xm=xk3; m=ak-(&x+3)}
     goto loop;
exit:
```

Figure 8: ROPE machine instructions for 3× unrolling.

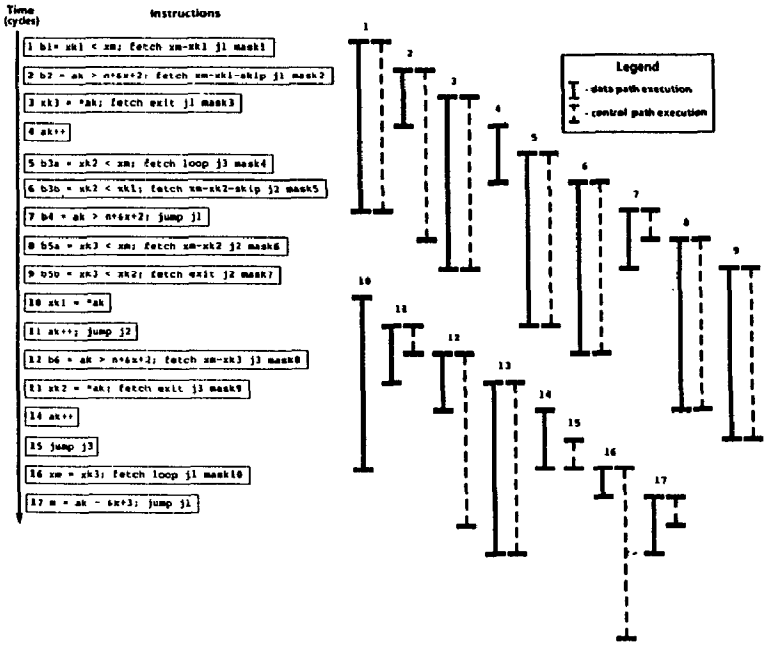

Figure 9: Trace of one execution for 3× unrolled loop.

cles as control operations requiring many bits. For horizontal microcode, particularly VLIW architectures, the large number of bits needed to specify the data operations means that relatively few extra bits are required for the multi-way jump specifications.

Every control operation has a 4-bit field for the jump label. If the jump label is zero, the instruction is a no-op (NEXT). If the jump label is not zero, one more bit is needed to distinguish JUMP and PRE-FETCH instructions.

Since jump labels are used only for distinguishing pre-fetches, and pre-fetches will not move far from the jumps that use them, jump labels can be re-used, and 15 different jump labels probably suffices for all realizable ROPE architectures. Thus four bits suffice for NEXT instructions, and five bits for JUMP instructions.

A PRE-FETCH instruction requires more care in the encoding, as it must specify the four bit jump label, the address of the target, and the conditions that cause it to become active. Most jumps are short, so the address could contain just the unit number and a few bits of offset for the higher-order address bits. Long branches can be encoded either by requiring that distant target addresses be taken from the data path, or by allowing only unconditional long branches, and stealing bits from the condition mask. In any event, the coding of the target address is straightforward, and requires only about 8 bits.

The encoding of the condition mask, however, is complicated. Most two-way branch instructions test either a single condition bit, or a hardwired combination of selected condition bits. An n-way branch, however, requires testing at least $\lg n$ condition bits. In one hand-coded example, a three-way branch depends on 6 different condition bits (see Figure 10). In the most general case, a target can be selected as an arbitrary boolean function of k bits selected from a larger group of m bits, and at least $2^k + \lg \binom{m}{k}$ bits are needed to specify each target. For functions of 6 bits chosen out of 16 bits, each target would need 77 bits to specify when it becomes active! This is too large even for VLIW machines. To reduce the condition mask to a reasonable size, we have to restrict how many condition bits are examined in a single target, or restrict which functions can be used to combine condition bits, or both.

One scheme that is particularly simple to build in hardware allows selecting any sub-cube of the m-dimensional hypercube. For each of the m condition bits, the mask may specify that the bit be 0, 1, or X (don't care). This scheme requires only $2m$ bits in the mask (32 for our hypothetical machine with 16 condition bits), and can encode any tree of two-way jumps. It cannot encode branches where two paths merge to a single target, such as $(b_1 \wedge b_2) \vee (\neg b_1 \wedge \neg b_2)$. In particular, the example of Figure 10 cannot be encoded.

A simple extension to the subcube scheme allows multiple instructions to provide additional subcubes to a target. The jump in Figure 10 requires 9 sub-cubes. The machine could easily be limited by the time taken to issue all the subcubes.

Since multi-way jumps are generated by compacting trees of binary jumps, smaller encodings of the useful jumps are possible. One scheme uses a full binary tree, with one condition bit in each internal node. A PRE-FETCH instruction needs only to specify the leaf nodes that will activate the unit. The complex 6-condition test of Figure 10 requires only a 4-deep tree, which can be specified with 16 bits per target.

If each internal tree node has a different condition bit, test instructions may have to set multiple condition bits, as the same test is commonly needed in different nodes. A fixed mapping of condition bits to tree nodes creates resource-limitation dependencies between different test instructions, making code reorganization difficult.

The most promising scheme allows changing which condition bit is tested in each internal node, using explicit instructions. If a reconfiguration instruction could put arbitrary conditions into each internal node of a 4-deep tree, it would take $15 \lg m$ (60 for $m = 16$) bits to specify a reconfiguration. Since the other control operations are all less than 24-bits long, a better encoding of the useful configurations is desirable. Characterizing the useful tree configurations is difficult, so we are delaying the final decision on this scheme until after extensive experimentation.

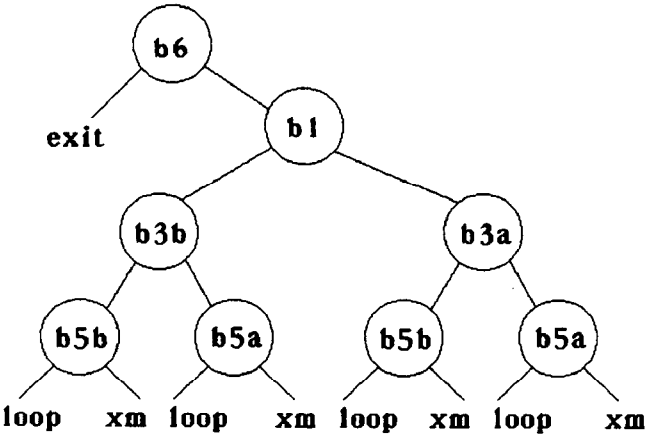

Figure 10: Boolean jump tree for 3-way branch using 6 condition bits.

Conclusions

The ROPE architecture provides a simple mechanism for implementing multi-way jumps with full instruction pre-fetching on machines where the instruction fetch time is much longer than the cycle time. The mechanism provides a convenient target architecture for new optimization techniques, such as percolation scheduling.

The ring of pre-fetch elements introduces parallelism in the memory system, obtaining high performance from relatively slow memory. The pre-fetch units operate primarily at memory speeds, rather than processor speeds, making them simpler to design and construct than conventional caches.

References

[Fisher80] J. A. Fisher. An Effective Packing Method for Use with 2^n-way Jump Instruction Hardware. 13^{th} *Annual Microprogramming Workshop*, Colorado Springs, Nov. 1980, *SIGMICRO Newsletter*, 11(3&4), 64–75.

[Fisher81] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers* C-30(7), July 1981, 478–490.

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

[McMahon] F. H. McMahon. Lawrence Livermore National Laboratory FORTRAN Kernels: MFLOPS. Livermore, CA. 1983.

[NicFish] A. Nicolau and J. A. Fisher. Measuring the Parallelism Available for Very Long Instruction Word Architectures. *IEEE Transactions on Computers*, C-33(11), Nov. 1984, 968–976.

[Nicolau84] A. Nicolau. *The design of a Global Parallel Compilation Technique—Percolation Scheduling.* Cornell University, Computer Science Technical Report, 1984.

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

[TadjFlynn] G. S. Tadjen and M. J. Flynn "Detection and parallel execution of independent instructions" *IEEE Transaction on Computers* **19**:10 (October 1970), 889–895.