# ROPE:
# A new twist in computer architectures

Kevin Karplus
Alexandru Nicolau

87-885
November 1987

Department of Computer Science
Cornell University
Ithaca, New York 14853-7501

# ROPE: A new twist in computer architectures

Kevin Karplus and Alexandru Nicolau

Karplus:

Computer Engineering

University of California at Santa Cruz

Santa Cruz, CA 95064

Nicolau:

Computer Science Department

Cornell University

Ithaca, NY 14853

Abstract

Supercomputer architectures are not as fast as logic technology allows because memories are slower than the CPU, conditional jumps limit the usefulness of pipelining and prefetching mechanisms, and functional-unit parallelism is limited by the speed of hardware scheduling.

We propose a supercomputer architecture called Ring Of Prefetch Elements (ROPE) that attempts to solve the problems of memory latency and conditional jumps without hardware scheduling. ROPE consists of a pipelined CPU or very-large-instruction-word data path with a new instruction prefetching mechanism that supports general multi-way conditional jumps.

To get high-performance without scheduling hardware, ROPE relies on an optimizing compiler based on a *global* code transformation technique (Percolation Scheduling).

This paper describes both the promise and the limitations of ROPE.

## 1 Introduction

We propose a supercomputer architecture called Ring Of Prefetch Elements (ROPE) that attempts to solve the problems of memory latency and conditional jumps without hardware scheduling. To get high-performance without scheduling hardware, ROPE relies on an optimizing compiler based on a *global* code transformation technique, which we call percolation scheduling. ROPE has a powerful multi-way jump mechanism that meshes nicely with the code generated by percolation scheduling.

This paper starts with an explanation of the need for a new computer architecture, briefly mentions some existing architectures, and explains the usefulness of multi-way jumps. Then some details of the architecture are presented, followed by a very brief summary of percolation scheduling. We then show some preliminary results, including a detailed treatment of one example program (Livermore loop 24). We conclude with a discussion of the limitations of ROPE, and future work we plan to do.

## 2 Why a new computer architecture?

Traditional computer architectures use resources inefficiently, and their performance is disappointing when compared to the raw speed of the components. The main technique for obtaining high throughput, *pipelining*, has been limited by the difficulty of keeping the pipeline full. Pipeline limitations usually come from two sources: data dependencies and the slowness of memory.

In many supercomputer architectures, complex scheduling hardware keeps the almost independent processing units from violating the data dependencies implicit in the code. Even with far more complicated approaches than those in current machines, scheduling hardware achieves only threefold speedups over simple sequential machines [WS84]. The scheduling mechanism is expensive to build, and slows down the cycle time, since it must operate faster than the processing units [Smi84].

Large memories are slow compared with modern processing elements, limiting performance. Instructions must be fetched from memory, and data operations that need to read from or write to memory take a long time to complete, delaying other instructions. Interleaved memory banks and instruction caches are 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.

An instruction 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, instruction cycles are already 8–16 times faster than access times for large memory chips. Conventional machines try to avoid this speed mismatch either by decoding complex instructions into long sequences of operations which are fetched from a faster memory (microcode), or by caching instructions. Computer designers have discovered that it is hard to identify useful sequences of operations that are long enough to offer much speedup, but used frequently enough to be worth microcoding[PS82].

Instruction caches work well at moderate processor speeds, but are difficult to design for the short cycle times of supercomputers. Furthermore, many current code optimization techniques [Fis79,GHL*85,Nic85] increase code size by unrolling loops, which reduces the spatial and temporal locality required by instruction caches. Cache size puts strong limits on the amount of loop unrolling that can be done for current machines, since caches offer no speedup for loops larger than the cache size. Code duplication to avoid redundant computation is also limited by cache size. Even more important, many real-time applications cannot tolerate the possible degradation in performance caused by cache misses; they have to meet certain real-time deadlines and require guaranteed performance.

## 2.1 Why is ROPE a good architecture?

The ROPE architecture allows general, multi-way jumps with parallel prefetching, so that slow, inexpensive memory can be used without speed penalties. Our architecture is designed to take advantage of a new code technique, percolation scheduling, which schedules operations to avoid data dependency [Nic85]. Percolation scheduling developed from experience with trace scheduling in the ELI project at Yale, and attempts to overcome the problems that limit the effectiveness of trace scheduling.

Because percolation scheduling, like other sophisticated code transformation techniques, is expensive to run, we are examining applications where the large running time of programs justifies an expensive compilation. We are not trying to create a new microprocessor, but a supercomputer architecture that supports scientific computation, simulation, and complex signal processing. The architecture is designed to work well even with unpredictable jumps, so it should handle a significantly larger class of applications than traditional supercomputers. In particular, the design should be fast for AI and systems code as well as scientific code.

In a previous paper [KN85], we proposed using ROPE for the microcode engine in a conventional computer. Microcode has a high compute/compile ratio and small improvements in microcode speed are very valuable, so microcode is a promising target for both percolation scheduling and ROPE. Although the relatively small size of many microcode programs makes a ROPE prefetch system unnecessary for standard microengines, the trend toward direct compilation to (essentially) microcode makes ROPE more appealing. We believe that ROPE is appropriate for RISC machines, where low-level instructions are stored in main memory.

The central feature of our architecture is a novel mechanism for prefetching instructions that reduces memory bottlenecks and the need to flush pipes on conditional jumps. The mechanism is a Ring Of Prefetch Elements (ROPE) that allows general, multi-way jumps with parallel prefetching, so that slow, inexpensive memory can be used without speed penalties. The prefetching mechanism is distributed, so fast, wide memory buses are not needed.

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 prefetching. Our mechanism avoids both these problems.

The proposed architecture is technology-independent, assuming only that processing cycle times will be significantly shorter than main memory access times.

The main hardware innovation in ROPE is the *Ring Of Prefetch Elements* that gives the architecture its name. Each prefetch element controls part of the instruction memory, operating primarily at memory speeds. Nearest neighbor interconnection of the prefetch elements gives fast straight-line prefetching in a very direct way. Multiple jump targets and the code that follows them can be fetched in parallel. Multi-way branches are easily implemented on ROPE, but the basic ideas can be used to advantage even with conventional 2-way branching.

## 3 Previous work

### 3.1 Previous work—other super-computer architectures

In many supercomputer architectures, complex scheduling hardware keeps the almost independent processing units from violating the data dependencies implicit in the code[1]. Although scheduling hardware allows some overlapping of normally sequential operations, the final machine is only about twice as fast as a strictly sequential machine [NK85]. Even with far more sophisticated scheduling hardware than that in current machines, only another factor of one and a half is obtained [WS84]. Not only is the scheduling mechanism expensive to build, but it also slows down the basic cycle time, since it must operate faster than the processing units [Smi84].

Large memories are slow compared with modern processing elements, limiting the performance of a machine in two ways. First, instructions to be executed must be fetched from memory. Second, data operations that need to read from or write to memory take a long time to complete, delaying other instructions.

For straight-line code, conventional prefetch units and instruction caches remove most of the instruction-fetch delay, but substantial penalties are incurred for conditional jumps and cache misses. Small instruction caches have 5% to 10% miss rates, and large instruction caches have 0.3% to 2% miss rate [Smi82].

Data caches usually have much higher miss ratios than instruction caches. Even if the caches never missed, they would still be twice as slow as registers. One attempt to solve the data-fetch latency problem is the PIPE architecture [GHL*85,Smi84], which uses two separate processors, one for data-fetching and one for other operations. The two processors are joined by queues, so that all data operations retain the original serial order. Although the decoupling of data memory operations from other instructions provides significant execution overlap, the queues are hard to manage. In the data memory processor, data fetches have to be distributed to memory banks, and the returned data values put back into the queue in the correct serial order. Since fetched data values must be read in the order they were requested, code rearrangement in the instruction processor is limited by the need to flush

[1]A data dependency is a relationship between two instructions that use a common register or memory location. The second operation cannot begin until the first operation is finished using the register or memory.

unneeded data values from the queue.

RISC (Reduced Instruction Set Computer) designs try to gain performance without scheduling hardware by making all instructions take the same time. The more complex operations, such as floating-point arithmetic, have been broken into smaller operations that can be executed quickly. The approach works well for small machines [HJP*82,PS82], but is unsuitable for high-performance scientific computation, where floating-point operations are common. RISC architectures are designed to take advantage of mature compiler technology. Our proposed architecture is designed around more modern, experimental compiler technology.

## 3.2 Previous work—other optimizing compilers

Optimizing compilers usually optimize only within basic blocks, putting strong limits on the depth of pipelines that can be used effectively. Several experiments [FR72,TF70,NF84] show that basic blocks are small, about 3–5 operations on average. The smallness of basic blocks and corresponding frequency of jumps has usually limited the size of pipelines to two or three stages [Rus78]. 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 often have serial dependencies, code rearrangement has to range over several basic blocks if more than a limited rearrangement is required.

Until recently, code transformations were limited to source transformations or to code motions within basic blocks. Although source transformations are useful[Kuc76,KKP*81,AK82], they are not sufficient for dealing with problems arising at the machine level. Fine-grained, basic-block techniques can be successful when dealing with short pipelines and instructions of fixed length [HJP*82,PS82]. Such methods, however, do not take full advantage of the speed of a given technology, particularly when floating point units are available. Recent advances in optimization techniques—notably, trace scheduling [Fis81] and percolation scheduling [Nic85]—offer code rearrangement that crosses basic block boundaries.

Trace scheduling was proposed as a global compaction technique for horizontal microcode and for high-level language compilation for Very-Large-Instruction-Word (VLIW) machines [Fis81]. Trace scheduling can extract substantial parallelism from ordinary code, as shown by the ELI project at Yale [FERN84]. For all its success, however, ELI has left several problems unsolved. For example, ELI ignores serious memory latency problems, and the conditional-jump mechanism used is not general. The wide instruction words of ELI, combined with its conditional-jump mechanism, makes the data path to memory enormously wide. Simultaneous data fetching requires wide data paths to the data memory, and significantly increases the chance of bank conflict.

Although ELI can achieve speedups more than tenfold on some scientific code [FERN84], these speedups are significantly lower than what is attainable for an idealized statically scheduled fine-grained architecture [NF84]. Much of the performance loss relative to the ideal model can be traced to the inadequacies of trace scheduling, memory latency, or the conditional-jump mechanism.

## 4 Multi-way Branching

Both trace scheduling and percolation scheduling tend to cluster conditional jumps. Since conditional jumps make up 15–33% of the initial program, combining the conditional jumps of a cluster into a single multi-way jump offers substantial improvements in speed. Unless several conditional jumps are evaluated at once, the ratio of jumps to straight-line code becomes unacceptably high.

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

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. One of the VAX microengines, 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 with the jump address field to specify the target address[PLT79]. Hard-wiring 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.

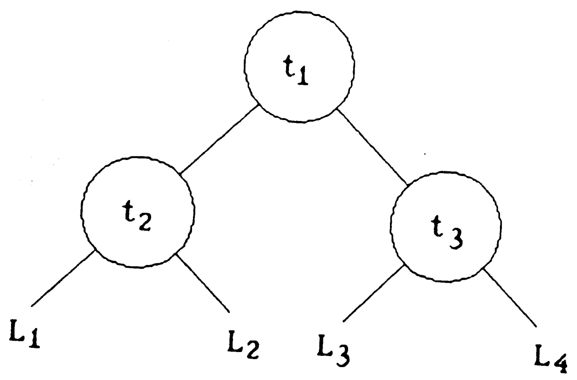


Figure 1: Conditional jumps not handled by Fisher's multi-way jump scheme.

This is typically used for microcoded case statements such as opcode decoding.

A more flexible approach was suggested by Fisher[Fis80]. His approach allows combining a sequence of $t$ arbitrary tests into a single *vine* in which the first true condition determines the branch to take. This scheme works well for trace scheduling[Fis81], which deals with only one path through the code at a time, but cannot handle some frequent combinations of jumps, such as those in Figure 1.

Significantly, none of the existing methods allow extensive prefetching; 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 memories. Ensuring that instruction fetching can be completed in 1 cycle requires either slowing down the instructions to the speed of the available memory, or investing in expensive fast memory.

The density of conditional jumps foils the effectiveness of traditional instruction prefetch mechanisms. Usually, only one path through the program graph is prefetched, 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.

Program size, a conventional measure of instruction set quality, is less important with our scheme than with previous ones. Because we use slow, cheap memory, we can easily afford lower code density than conventional systems. Global compilation techniques often increase the program size since they concentrate on the main goal, which is the speed of

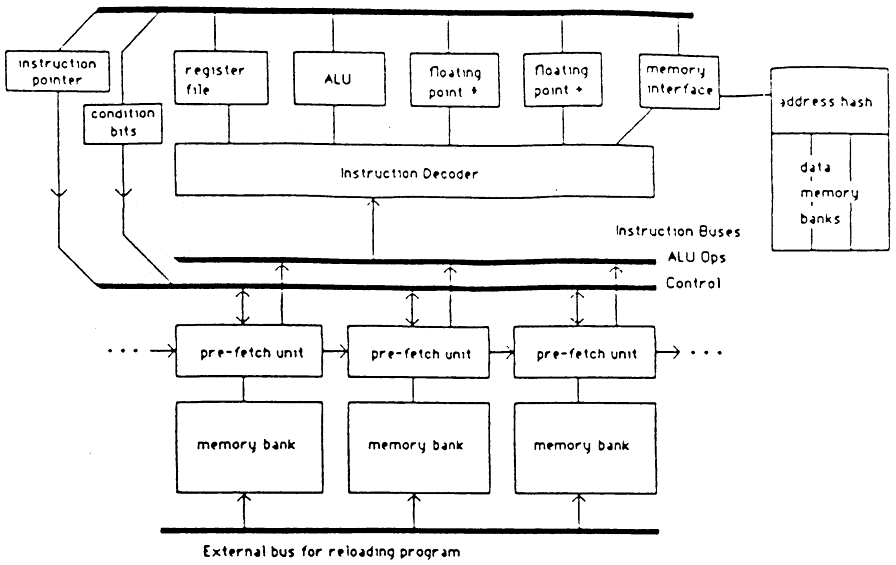


Figure 2: ROPE block diagram.

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 prefetch units easier, but no empty blocks of memory are needed, and the instruction word is only slightly larger than in conventional systems.

Although valuable, multi-way branching is not essential to the concept of ROPE. A simplified version that allows parallel prefetching, but uses only 2-way jumps, is also attractive. The prefetch units provide a cheap alternative to instruction caches or branch predictors for prefetching instructions. The compiler for such a system can be relatively simple, as it need only schedule prefetches, not rearrange data path and control operations to build multi-way jumps.

## 5 Architecture

The ROPE architecture is basically a Harvard architecture, with separate instruction and data memories. As in PIPE [GHL*85], control flow is handled primarily by the instruction memory system, while computation is done by the data path and data memory. On each clock cycle, one data path instruction and one control flow instruction are begun. Instructions need not finish in one cycle, but a functional unit that takes longer than a cycle must be pipelined or duplicated, so a new operation can begin on every cycle.

The data path and data memory are fairly conventional—most of the innovation of ROPE is in the control system and instruction memory. Figure 2 shows the block diagram for the ROPE machine.

## 5.1 Data Path

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 main 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.

We'll describe a fairly conventional data path design, one that combines RISC and array processor designs. It has the following features:

- All arithmetic and logic operations are register-to-register operations. The instructions for controlling the data path are essentially vertical microcode, as in RISC architectures (or horizontal microcode, if a wider data-path is supported). The register file must be large enough to reduce memory traffic significantly, and it may be broken into separate banks to allow more register operations per cycle. Since we are aiming at scientific computation, we use the registers primarily to reduce memory traffic during arithmetic operations, rather than to reduce procedure call overhead. When a procedure call occurs in an inner loop of a program, the compiler will do in-line expansion to eliminate the overhead of the call.

- Besides the conventional integer arithmetic unit, pipelined floating-point hardware is provided. The main requirements on the arithmetic units are that an operation may be started on every cycle, and that each operation take a fixed time to complete. Pipelined floating-point multipliers and adders are standard on machines intended for scientific computing, since the speed at which floating point operations can be done is the main performance limitation for scientific calculation.

- As in RISC architectures, addressing modes are not provided, and all memory operations are explicit loads and stores. The instruction and data memories are entirely separate, so no conflicts can occur between data and instruction fetching. Ideally, one load or store operation can be issued each cycle, although an operation may take several cycles to complete. The data and instruction memories are each banked and can accept one request per cycle, as long as no bank has to process two requests simultaneously.

When multiple requests are made to the same bank, the processor freezes until the first request is completed. For instruction fetches, the prefetch mechanism described in the next section can be used to guarantee that an instruction is always ready to execute. Various techniques are suggested in Section 5.2 below to make freezing adequately rare.

- The system is fully synchronous, in that all operations finish after some known multiple of the basic clock period. Although asynchronous systems can be built, such systems are difficult to schedule efficiently, and schedules are difficult to debug. Scheduling hardware is expensive, and would slow the basic operations of the machine. Our design requires no scheduling hardware. It is the responsibility of the compiler to create good fixed schedules for instruction execution, given the execution times for the various instructions as parameters.

- In the simplest implementation, 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 will use cheaper, slower 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. Because the percolation scheduling technique allows early instruction and data prefetching, we expect clock freezing to be rare enough not to degrade the performance of the machine (see Section 5.2). Section 5.3 explains a more sophisticated way to handle context switching.

- Different implementations of the architecture can have different numbers of each arithmetic unit, and the speeds of the units can vary. Of course, programs will need to be recompiled for different versions of the machine, since the available parallelism affects the mapping of the percolation schedule to machine resources. ROPE is an attractive control mechanism for VLIW data paths.

### 5.1.1 Context switching

In some systems, context switching and interrupts are needed, and simply freezing the processor is unacceptable since it may produce incorrect results. 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 prefetch 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. A simple solution to this problem is for the compiler to schedule operations so that no register or memory word is written within $n$ cycles after being read. Then the last $n$ instructions can be repeated after a trap or interrupt without changing the semantics of the program.

An alternative, and even simpler, approach permits interrupts only when inserting an arbitrary delay before an instruction will not affect the correctness of the program. The data operations already started can be completed while the prefetch units store their states. If different contexts use non-overlapping sets of registers, 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 that the maximum delay before accepting interrupts is sufficiently short, 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.

## 5.2 Data Memory

The data memory, like other units on the data path, must accept requests as often as one per cycle, but may take several cycles to respond to a request. The compiler schedules memory operations as if they take a constant time to execute. Memory operations may not take a fixed time, however, since data caching can speed memory operations, and page faults can slow them down. Faster memory operations pose no problems (although the execution schedule chosen by the compiler may no longer be optimal), but slower memory operations force the processor to freeze until the memory operations can be completed. The compiler can schedule load and store operations for nominal execution times (for example, assuming cache hits for sequential array access, and cache misses for random linked lists).

A multi-bank data memory will meet our needs. Each bank processes only one memory operation at a time, but different banks can execute memory operations concurrently. The memory as a whole can accept one operation per cycle, but may have to freeze the processor if a bank does not complete an operation in the scheduled time (due to access conflicts).

```
for i=m1 to n1 by k1 do
 for j=m2 to n2 by k2 do
     c[i]:=a[i] + b[j];
```

Figure 3: Indirect references that may conflict.

Caches are not needed for moderate performance (up to about 20 Mflops), but a cache may be needed for each bank at higher speeds.

We can alleviate the data-memory freezing by having the compiler do most of the work, carefully mapping the data onto the banks. Based on our experience with ELI, good mappings should be feasible most of the time, given good memory disambiguation techniques [Nic84b]. Unlike VLIWs, the ROPE machine issues instructions one at a time. The bandwidth of the data memory is therefore not as critical as in VLIWs and should not be a bottleneck.

The memory bank conflicts can be reduced by making the mapping from address to memory bank number be a hash function, rather than the usual low-order address bits. Hash functions can be computed quickly in hardware, but would add an extra cycle to the data memory access. Studies are needed to determine if memory bank conflicts are frequent enough to justify the extra cost of hashing.

Data memory conflicts and the resulting delays can sometimes be reduced by distributing multiple copies of read-only data into the banks[2]. Having many registers will enable the compiler to maintain fast access to heavily used data, further reducing the problem. Finally, if memory references in the code are ambiguous and cannot be guaranteed by the compiler to fall in different banks, the user could be queried at compile time for information about the likelihood of a conflict. If the chances of collision are high, the references can be scheduled with an interval that will avoid freezing, allowing percolation scheduling to use the intervening time for other operations. Since percolation scheduling supports global code motions, the chances that such operations could be found are good.

Figure 3 shows an example where user-supplied information can guide the compiler. We would like to issue the fetches from `a[i]` and `b[j]` one immediately after the other, but in

[2]Signal processing algorithms make particularly heavy use of read-only constants.

the absence of more information about i and j. we can't guarantee that a[i] and b[j] are in different banks. We can either create multiple copies of a[] and b[], or, if the user can supply information about i and j (for example, that i is always odd and j is always even), the compiler could try assigning addresses for a[] and b[] to minimize conflicts. Even if the user can guarantee only that conflicts are rare, without specifying exactly when they will occur, average performance can be improved. For example, if $i = 2x - 1$ and $j = x^2 - 1$, for $x \geq 0$, a conflict can still occur, but the code can still be scheduled as if there were no conflicts, since freezing will occur only once. Alternatively, the compiler could schedule the memory references apart far enough that no conflicts can occur, using the intervening cycles to perform housekeeping tasks like incrementing and testing the counter.

A critical bottleneck is the communication between the data memory and the functional units. The PIPE architecture proposed adding a queue to overlap the memory latency with other operations [GHL*85]. However, interleaving the results from multiple data banks into the queue requires fairly complex hardware. We propose queuing requests to memory, but providing each load request with a specific destination register, decided at compile time. The load registers need a tag bit each to indicate whether they contain data or are waiting for a fetch to finish. When a register that is waiting for a fetch is read, the processor must freeze until the fetch finishes. Since the waiting-for-fetch bit is only set when a new fetch is started, the values in the registers may be re-used without being moved to another register (unlike PIPE's queues, in which data is lost when it it popped off the queue). PIPE's queues make fetching data that turns out not to be needed doubly expensive, as the data must be removed from the queue before useful data can be read. Since our load registers can be accessed randomly, no such penalty exists in ROPE.

## 5.3 The $n$-way jump and prefetch mechanism

A standard conditional jump has 2 branches (true and false). Thus $n$ independent conditional jumps 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 one of $2^n$ paths.

Luckily, conditional jumps that are clustered when compacting sequential programs are usually not independent, instead they form a rooted DAG (directed acyclic graph), with tests

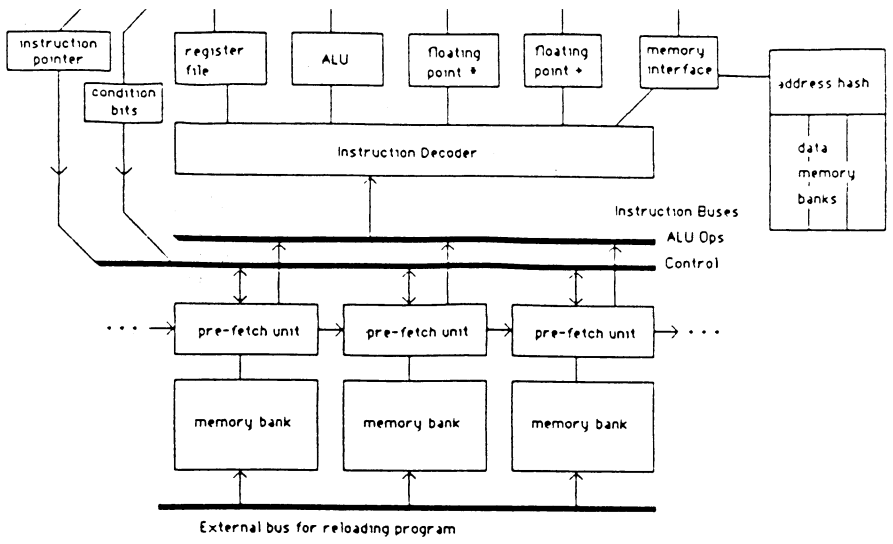


Figure 4: ROPE block diagram (duplicates Figure 2).

in the internal nodes and continuations at the sinks. The $n$ conditional jumps in such a DAG (of which Fisher's vine [Fis80] is a special case) choose one of at most $n+1$ continuations. In this section, we describe a mechanism tailored for such multi-way jumps.

### 5.3.1 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 supercomputers.

Our design has interleaved memory banks, but avoids the jump penalty by using intelligent prefetch units, connected in a ring. We have named the architecture *ROPE*, for *Ring Of Prefetch Elements*. Figure 4 shows the block diagram for the ROPE machine.

Each of the $2^n$ prefetch units has its own block of memory, independent of the other units. On each cycle, exactly one prefetch unit, called the *active* prefetch unit, provides an instruction to the decoder. The other prefetch 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 prefetch 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 prefetches must be started well before the instruction is executed. prefetches are started automatically for normal straight-line code.

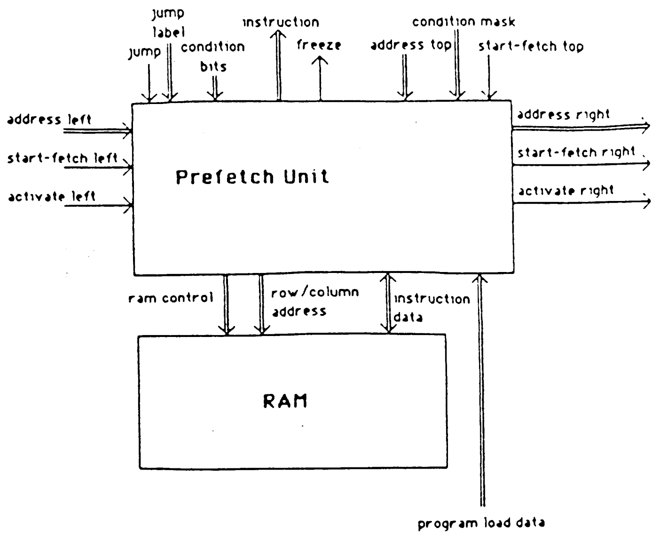


Figure 5: One prefetch unit.

Since straight-line code proceeds from left-to-right across the prefetch units, it suffices for each unit that starts a prefetch 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 prefetch instructions.

## 5.4 Structure of a single prefetch unit

Figure 5 shows a single prefetch 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 $2^n$ prefetch units, the bottom $n$ bits of address select the prefetch 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 prefetch unit has two state bits that determine its behavior: *busy* and *target*. A prefetch 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 prefetch unit is a target if the word fetched was requested by an explicit `prefetch` instruction, and a non-target if the word was requested from the unit to its left.

The prefetch 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 prefetch 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 prefetch 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 prefetch unit responds to a jump signal.

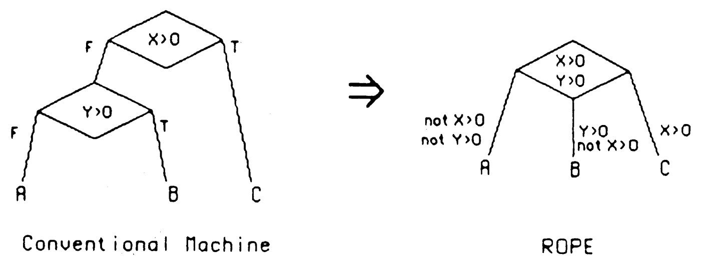


Figure 6: Combining two conditional jumps into a three-way jump.

### 5.5 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 prefetch 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 prefetch ring.

`PREFETCH` *address jump-label condition mask*: instruct the appropriate prefetch 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 prefetch unit for later matching.

`JUMP` *jump label*: activate the appropriate target prefetch unit for the specified jump. Jumps are labeled so that prefetches 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.

`CLEAR-JUMP` *jump label*: tell all target units for specified jump that the jump will not be executed. Each target unit then behaves as if the jump was requested and the condition mask failed to match. The next unit in line is activated, as in the `NEXT` instruction. `CLEAR-JUMP` instructions are only needed when the compiler moves the pre-fetches for a branch forward past another branch. If the path that gets taken does not include the `JUMP` instruction, the target units for the jump would block normal sequential flow, and must be released.

A conditional jump on a ROPE machine thus has three parts: `PREFETCH` 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.

### 5.6 Example of multi-way jump on ROPE

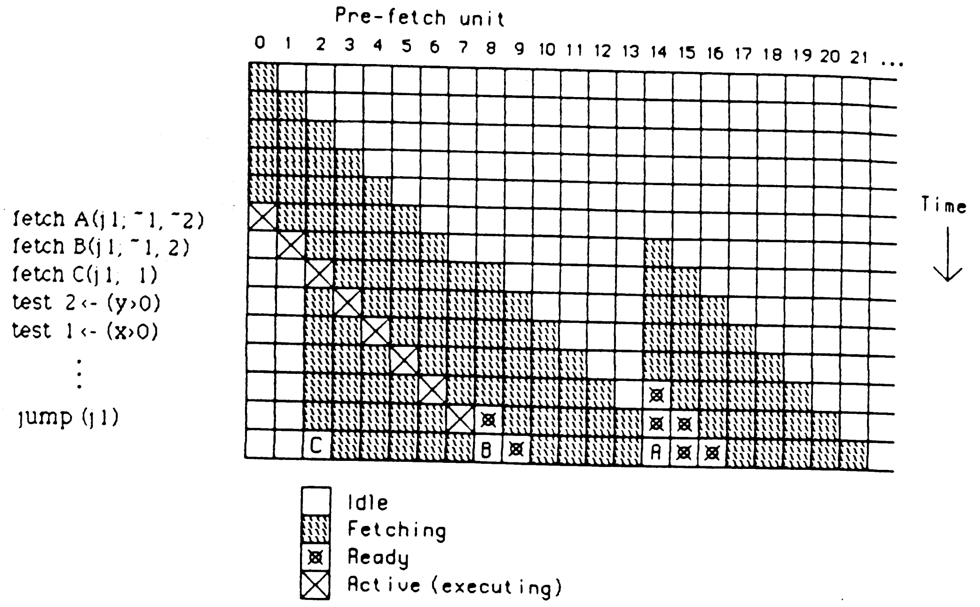


Figure 7: Possible execution sequence for a three-way jump.

On a conventional machine, jumping to one of three addresses requires two conditional jumps, as in the left half of Figure 6. On a ROPE machine, a single three-way jump is used (right half of Figure 6). Figure 7 shows how this example can be scheduled onto the multiple prefetch units of the ROPE machine. Each column shows the activity of one unit, each row representing a single cycle.

Although ROPE's prefetch units are fairly cheap, it would be naïveto 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 prefetch 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 prefetching. So a $k$-way jump requires at least $kF$ prefetch units. For example, a four-way jump with a six cycle fetch time requires 24 prefetch units.

If jumps are close together, as they are in the example shown in Section 8, each target branch needs to prefetch only up to the next jump instruction, and fewer units are needed. Numerous experiments [TF70,NF84] show that basic blocks are small. Even after compaction and combining several jumps into a multiway jump, the resulting basic blocks are still quite small. Thus, in fewer banks than $kF$ would be required. In our experience, $4k$ to $5k$ are enough for freeze-free execution.

Because we rarely construct branches with more than four targets, a ROPE machine with 16 or 32 prefetch units should achieve almost all the possible speedup of this architecture.

If an implementation has too few prefetch units, programs will be slowed down, but only by the number of prefetches that do not fit (not by the time required for each fetch), since prefetches have no data dependencies, just resource availability constraints.

The compiler (or hand-coder) must schedule the prefetches and assign code addresses to minimize the waiting for instruction fetches. For tree-structured control flow with infrequent branching, scheduling prefetches and assigning code addresses is easy. Assigning addresses gets 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. If branching is frequent and not enough prefetch units are available, delays are unavoidable with any schedule.

We believe that multi-way jumps will prove to be a valuable part of ROPE. Combining basic blocks and using multi-way jumps should allow us longer sections of straight-line code than compilers for conventional machines, which examine only basic blocks. The main cost of a jump instruction on conventional machines is the fetch time for a non-sequential instruction. With our architecture, the prefetches, tests, and jumps can be scheduled independently, and therefore do not slow the machine. Separating prefetches, tests, and jumps may improve performance significantly, even without multi-way jumps.

## 5.7 Format of the instruction word

We are still considering different formats for the instruction word. A possible format for the word on the instruction bus is presented here. Note that the instructions in memory can be more tightly encoded, since the prefetch 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 cycles 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.

Two bits are needed to encode the four control ops: `NEXT`, `JUMP`, `CLEAR-JUMP`, and `PREFETCH`. `NEXT` needs no further information, the rest need a 4-bit jump label, and `PREFETCH` needs to know under what circumstances the target will activate. Because prefetches will not move far from the jumps that use them, jump labels can be re-used, and 16 different jump labels probably suffice for all realizable ROPE architectures.

A PREFETCH instruction requires 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 hard-wired 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 8). 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 probably 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 8 cannot be encoded.

A simple extension to the subcube scheme allows multiple instructions to provide additional subcubes to a target. The jump in Figure 8 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 encod-

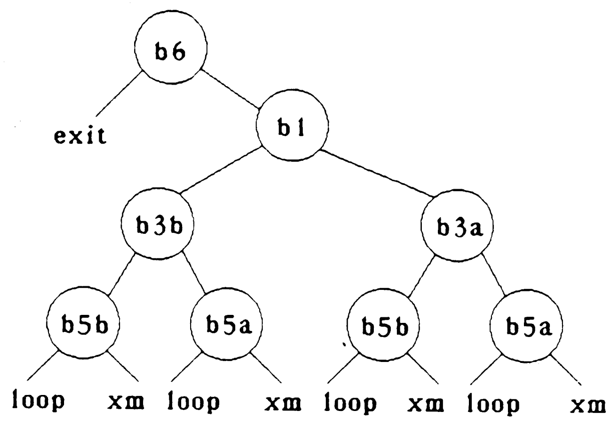


Figure 8: Boolean jump tree for 3-way branch using 6 condition bits.

ings of the useful jumps are possible. One scheme uses a full binary tree, with one condition bit in each internal node. A `prefetch` instruction needs only to specify the leaf nodes that will activate the unit. The complex 6-condition test of Figure 8 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. An alternative scheme allows a prespecified set of conditions to be loaded at compile-time in a table (a set of registers). The fetch instructions then specify the condition on which their prefetch unit will activate by indexing into this table. Note that the set of conditions need not be general-purpose, as they can be generated (from the actual multiway jumps in a given program) at compile-time. Thus the entries in the table are tailored to the program being run.

## 6 Percolation Scheduling

Percolation scheduling globally rearranges code across basic-block boundaries. Its core is a small set of primitive program transformations acting on adjacent nodes in a program graph. These transformations are easy to understand and implement, and are independent of any heuristics. They are the lowest level in a hierarchy of transformations and guidance rules. Higher levels of this hierarchy direct the core transformations and rearrange the program graph to allow more code motion by the core transformations. Aided by the other levels, the core transformations operate uniformly and can be applied to partially parallelized code, allowing percolation scheduling to improve code produced by other compilers.

The following is an overview of the hierarchy and the work we have completed. A more detailed discussion can be found in [Nic85].

### 6.1 Core Transformations

The four primitive transformations of this level are the core of percolation scheduling. They operate on adjacent nodes in a program graph. Repeatedly applying the transformations allows components to "percolate" (move towards the top of the program-graph) from the various parts of the program graph towards the start node, hence the name percolation scheduling.

The details of the transformations deal with maintaining the integrity of all affected paths. A brief description of each transformation is given below. Rigorous definitions can be found in [Nic85].

Delete Transformation A node in the program graph can be removed by the *Delete* transformation when the node contains no executable operations. Nodes without any components may occur as a result of the other transformations or as part of the original program graph. Since they do not affect the execution semantics of the program in any way, such nodes may be deleted, provided that the outgoing edges of their predecessors are reset to point to the successor of the deleted node. An illustration is given in Figure 9.

Move-op Transformation This transformation moves a simple operation (that is, one that does not affect the control-flow) from node $n$ up through edge $(m, n)$ to node $m$, provided

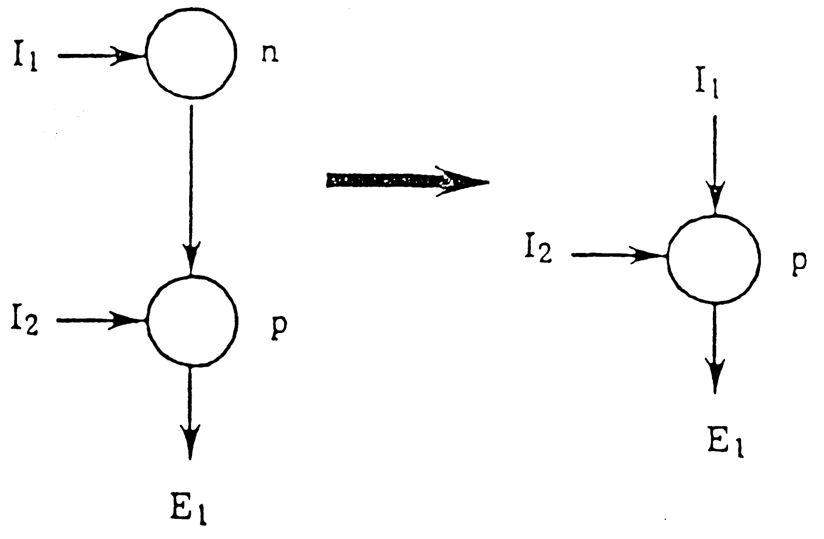


Figure 9: Delete Transformation

no data-dependency exists between operations in $m$ and the operation being moved. In performing the movement, care must be taken not to affect the semantics of paths passing only through $n$ but not through $m$. To ensure this, these paths are split and provided with a copy of the original $n$. An illustration is given in Figure 10.

Move-cj Transformation A conditional jump can be moved up from node $n$ to node $m$ through edge $(m, n)$, provided that no dependency exists between $m$ and the component being moved. In performing the movement, care must be taken not to affect paths passing only through $n$ but not through $m$. To ensure this, the paths are split and a copy of $n$ (called $n'$ in Figure 11) is provided. In addition, since we allow an arbitrary tree of conditional jumps in a node, and the conditional jump being moved may come from an arbitrary spot in that tree, $n$ will be split into $n$ and $n''$, to correspond to the true and false branches of the moving conditional jump (see Figure 11). The details of the splitting and a proof that the transformation indeed preserves the semantic correctness of the original program is beyond the scope of this paper and can be found together with proofs of correctness and termination in [Nic85]. An illustration is given in Figure 11.

Unification Transformation This transformation merges identical operations from a set of nodes $\{n_0, n_1, n_2, \ldots\}$ to a predecessor node $m$. This is done only when no dependency exists between $m$ and the component being moved and when paths $(m, n_i)$ exist for all nodes in the

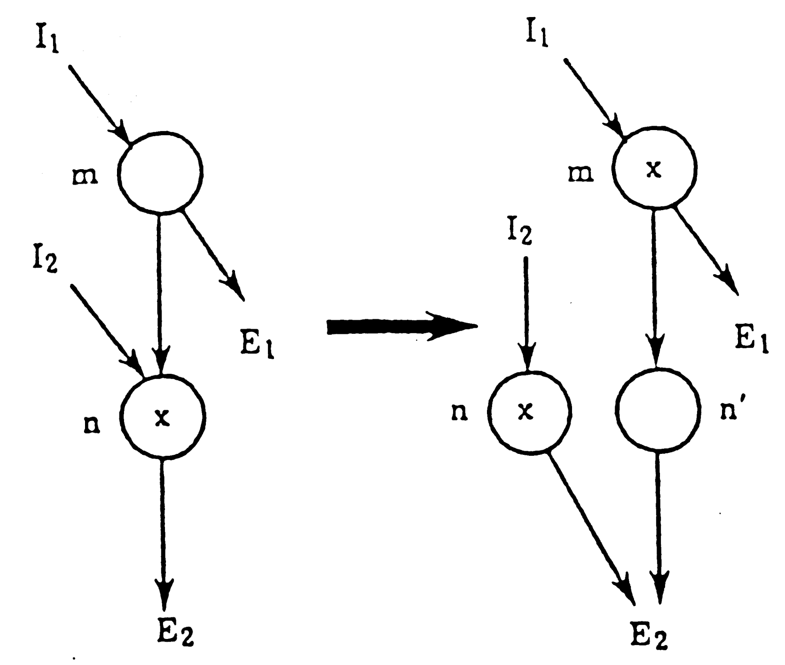


Figure 10: Move-op Transformation

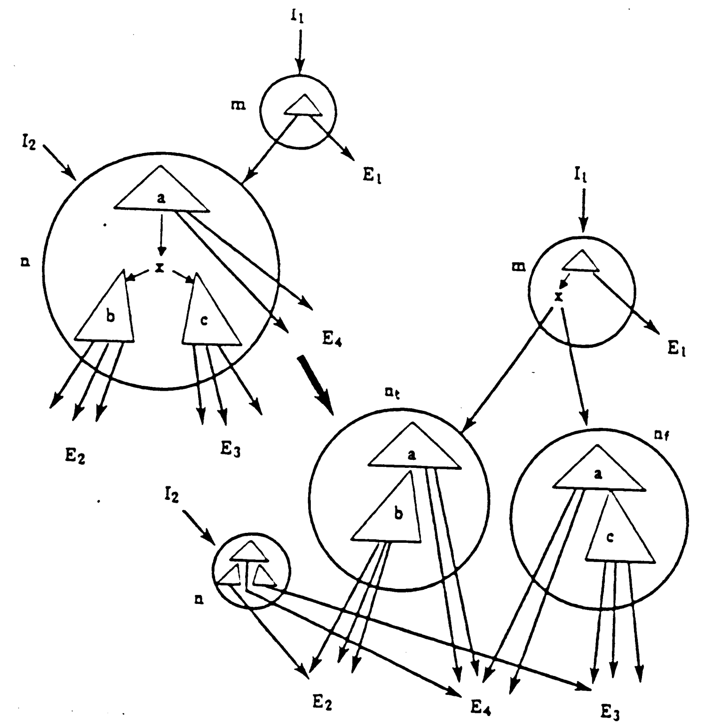


Figure 11: Move-cj Transformation

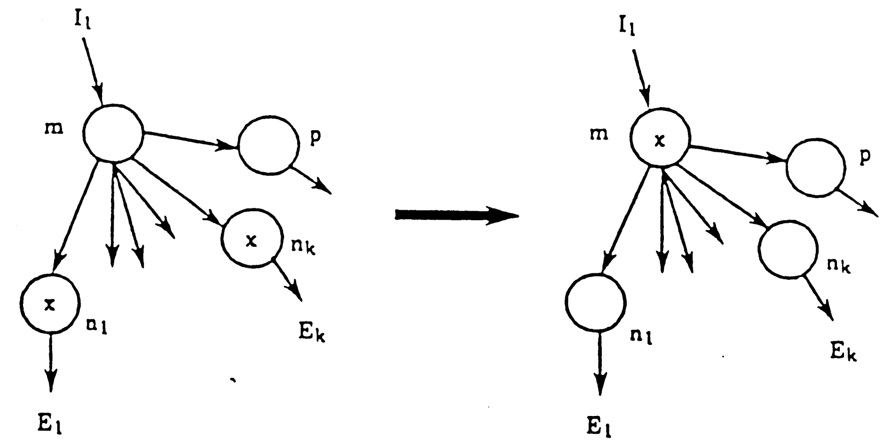


Figure 12: Unification Transformation

set. In performing the code motion, care must be taken not to affect paths going only through the $n_i$'s but not through $m$ and, as usual, splitting and copying is used. An illustration is given in Figure 12.

## 6.2 Beyond the Core Transformations

The core transformations do most of the code motion, and higher levels direct the core transformations and rearrange the program graph to allow more code motion by the core transformations.

*General Support*: At this level, the data dependencies are found and recorded. Memory disambiguation and enhanced flow analysis methods [Nic84b] increase the accuracy of data dependencies and permit more code motions. Traditional optimizations, such as dead code removal, are also used at this level.

*Guidance Rules*: This level consists of a set of rules that decide when and where to apply the core transformations. Operation and branch probability, reverse depth-first ordering, dependency chain length and many other heuristics [Nic85] can be used to direct the primitive transformations[3].

[3]In trace scheduling this level consists of only one rule (trace picking) and is inseparable from the code transformations. This rigidity reduces the potential benefits of trace scheduling.

*Enabling Transformations*: This level provides transformations of the program graph that cannot be accomplished by the simple code motions of the core transformations. The program graph rearrangements are done primarily to allow the core transformations greater freedom to move code. Examples of such rearrangements are: folding, tree-height reduction [Kuc76], and loop quantization (a multiple loop unrolling technique) [Nic84a].

The computational model and the core transformations are formally defined in [Nic85]. The semantic correctness and termination of the transformation process have been proven. We are also studying the completeness of the transformations. Guidance rules and transformations for the higher levels of our hierarchy have been defined. These topics are all discussed in [Nic85].

## 6.3 Mapping Percolation Schedules to Hardware

The transformations given above expose the parallelism available in the code and provide a partial ordering on the issuing of the operations. The transformed program graph can be viewed as the code for an idealized machine, in which no resource conflicts ever occur. Obviously this ideal is unrealizable, and can only serve as a bound on the effectiveness of the transformations. To execute the resulting code on realistic architectures, we need a mechanism to change the ideal schedule into a schedule that respects resource limitations. Even for our simple architectural model, finding an optimal resource-constrained schedule is NP-hard, unless every operation takes only one or two cycles. The greedy algorithm outlined below, however, appears to be adequate. It is a refinement of list scheduling, which is reportedly very effective [Fis79].

The main resource limitation in the ROPE architecture is that only one instruction can be executed per cycle. To satisfy this restriction, a total ordering must be derived from the partial ordering of the instructions. The total ordering is built one instruction at a time. The algorithm keeps two lists of instructions: the *ready* list and the *busy* list. The ready list contains all instructions that can be scheduled to begin on the current cycle, and the busy list contains all instructions being executed during the cycle. Initially, the ready list contains all instructions that have no predecessors, and the busy list is empty.

For each cycle, the algorithm chooses the instruction from the ready list that is expected to lengthen the total schedule length least. Our current heuristic is to choose the instruction with the longest dependency chain—that is, the least slack. To break ties, the instruction with the most dependents is chosen. If the ready list is empty, a no-op is scheduled.

After scheduling an instruction for the cycle, the algorithm checks the busy list for instructions that were completed during the cycle. Completed instructions are removed from the busy list and their immediate dependents are added to the ready list. Both the ready list and the busy list can be effectively implemented with priority queues.

The technique above works well for straight-line code. At each jump the algorithm must decide which branch to continue scheduling. Other branches are pruned from the graph and scheduled separately. The result of the greedy rescheduling algorithm is a collection of code fragments joined by jump instructions.

Once the rescheduling is completed, a starting address must be assigned for each code fragment. Within a code fragment, instructions are assigned sequential memory locations. The first code fragment can be assigned an arbitrary address, but the other fragments cannot be arbitrarily placed. All the targets of a conditional jump need to be sufficiently separated so the prefetching mechanism can have all the targets ready simultaneously. Each code fragment is placed so that the jumps into or from already placed fragments are all satisfied. If no such placement can be found, part of the code fragment may need to be duplicated.

## 7 Preliminary Results

We have constructed five simulated architectural models: two correspond to conventional machine architectures, one is the ROPE architecture, one is a VLIW architecture, and the last combines ROPE and VLIW ideas. We are implementing a percolation-scheduling compiler for all five architectures, but for the comparisons in this section percolation scheduling is used only for the ROPE machines.

The simplest architectural model executes each instruction in the object program sequentially. The machine contains no cache, so all jumps require memory accesses. Standard optimizations of the code (such as dead code removal) are assumed, but code rearrangement, prefetching, and hardware scheduling are all irrelevant for this machine.

The second model is representative of several existing supercomputers (for example, the Cyber 205, Cray-1, and CDC7600). It has a fully pipelined data path, identical to the

ROPE data path. This machine also has a hardware scheduling mechanism that guarantees executing dependent operations in the same order as they were issued, but allows independent instructions to be executed in any order. The machine contains a program cache, and for fair comparison with our architecture, we assume 100% hit ratios. That is, the machine can fetch and decode any instruction in one cycle, but instructions cannot be issued beyond a conditional jump until the condition has been resolved.

The data memory is banked, and we assume that the data layout permits memory accesses to start every cycle. This is unrealistic unless sophisticated tools for memory disambiguation and layout [Ban79,Nic84b] are used. Although most existing compilers do not provide such support, we include the assumption so our comparison is not biased towards the percolation scheduling and ROPE approach. We also allow this architecture an optimizer that can perform code reorganization within basic blocks. For example, jumps can be moved upwards inside basic blocks when not hindered by dependencies—as in MIPS [HJP*82].

The third simulated machine is our ROPE architecture. It has the structure described in Section 5, and uses percolation scheduling and the mapping techniques described in section Section 6. No cache or runtime scheduling hardware is provided in this machine; the compiler is completely responsible for the correct execution and the efficiency of the machine, including proper data/program bank accesses.

Both of the architectures with pipelined data paths assume that the registers read by an instruction are not needed again by that instruction after they have been read, and that they are read in a fixed cycle of the instruction. This assumption alleviates read-write dependencies and is realistic for a pipelined architecture.

The fourth model is an *idealized* VLIW machine [FERN84]. While the instruction timings are realistic, we allow as many resources (such as functional units, buses, memory ports, register ports) as required for peak performance. The functional units are pipelined to accept one operation per cycle, and trace scheduling is used to schedule the input program. We assume sufficient instruction prefetching on the on-trace path and on unconditional jumps so that instructions in that path can be issued every cycle. Off-trace conditional jumps require a memory access and are therefore slower.

The last model combines VLIW functional-unit parallelism with ROPE instruction prefetch and uses a percolation scheduling compiler. The combination of ROPE and per-

| | Cray-1 measured | Cray-1 counted | PIPE counted | ROPE counted |
|---|---|---|---|---|
| Livermore 5: | 37.8 | 62 | 48 | 12 |
| Livermore 6: | 57.3 | 64 | 47 | 5 |
| Livermore 11: | 29.0 | 27 | 21 | 6 |
| Livermore 24: | 36.0 | | | 5 |

Table 1: Cycles per iteration for sample Livermore loops.

colation scheduling make the VLIW architecture much less sensitive to branch probabilities.

Despite the greater hardware complexity of the conventional pipelined architecture and of the VLIW model, our preliminary results show significant speedups for the ROPE architecture (and for percolation scheduling), even on small problems (binary search, bubble sort, Livermore Loops 5,6,11, and 24 [McM83], and matrix multiplication inner-product) over all other models. The speedups range from factors of about two to three over the Cray-1, despite the slower technology assumed for ROPE.

Table 1 show the number of cycles per iteration for 4 of the Livermore loops (each of which is a "hazard-bound" loop [GHL*85]). The first column is derived from measured performance of the Cray-1 using the CIVIC30i compiler, the second and third are reported in [Smi84], and the fourth is from hand-compilation for ROPE. The columns are not directly comparable, as the ROPE simulation assumed faster memory (6 cycle latency instead of 11) and slower processing (9 cycle multiply instead of 7). Even if deflated by a factor of two, ROPE is clearly superior to either the CRAY or PIPE on these loops.

Assuming the same technology as that of the Cray-1, our architecture performed up to 5 times faster than the Cray-1 on our sample programs; using more functional-unit parallelism and the same cycle time as the Cray, the above speedup is nearly doubled. A further speedup is expected in a hardware implementation of ROPE, since the simple, uniform architecture should allow a shorter cycle time than a machine with hardware scheduling [NK85].

Although the density of conditionals in our small samples is not necessarily typical of scientific code, unpredictable conditionals occur often in scientific programs (such as Monte-

Carlo simulations and finite element computations), as well as in other applications, such as systems code or artificial intelligence code. Furthermore, an examination of the code produced from these samples indicate that ROPE can do even better on larger programs where there is greater freedom for code rearrangements. Considering the shortness of the code we dealt with so far, the speedups achieved are surprisingly good.

The ROPE machines have far fewer wasted cycles than the other approaches. Since the instruction prefetches, tests, and jumps are scheduled independently, we don't need to lengthen our schedule to wait for program memory, despite the absence of a cache. Our preliminary results indicate that the multi-way jumps and prefetch mechanisms of our architecture may perform very well on such programs.

## 8 Sample Program Fragment—Livermore loop 24

The code in Figure 13 (Livermore loop 24) will be used to illustrate our approach. Loop 24 finds the location of the minimum of an array of floating-point numbers. For all the architectures discussed, the loop has been unwound three times to increase potential pipelining, and traditional optimizations have been done.

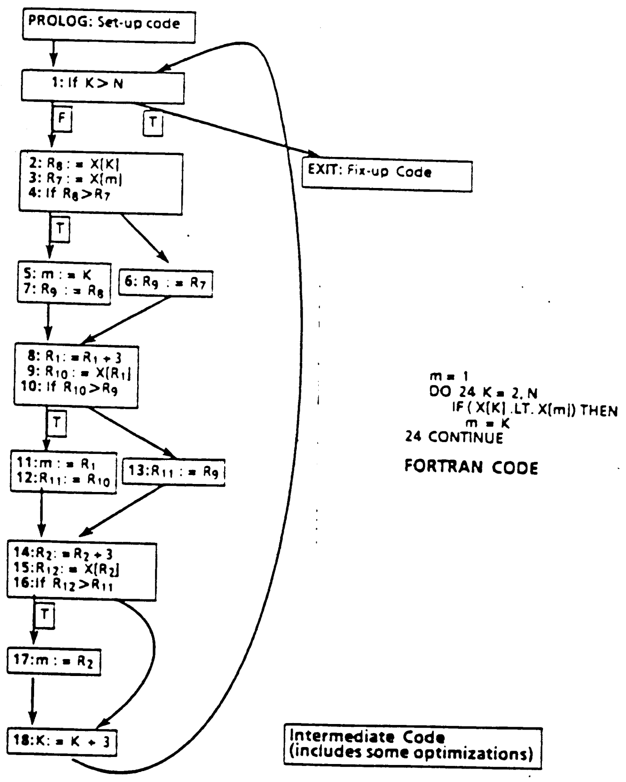
Executing the loop sequentially requires between 70 or 73 cycles (depending on which branches of the conditionals are taken) averaging 71.5 cycles.

The loop body requires between 38 and 41 cycles to execute on the machine with hardware scheduling, for an average of 39.5. cycles[4]. This architecture is about 1.8 times as fast as the sequential machine on this example, which is consistent with the speedups reported in [WS84]. The actual performance of the Cray-1 (cycle time 12.5 nanoseconds) for this loop is of about 2.3 Mflops according to the Livermore benchmarks.[5]

For the VLIW machine, the intermediate (NADDR) code and the trace scheduled code is shown in Figure 14. On-trace jumps and unconditional jumps are assumed to be prefetched. Off-trace jumps are shown explicitly by arrows and take longer than on-trace jumps. A few optimizations have been performed to ensure a fair comparison with percolation scheduling

[4]In this discussion we ignore initialization time and the time required to finish pending operations after the loop exits. For realistically large arrays, this time is negligible compared to the execution time of the loop.

[5]Loop 24 is the slowest of the Livermore loops on the Cray.



```
   m = 1
   DO 24 K = 2, N
      IF ( X(K) .LT. X(m)) THEN
         m = K
24 CONTINUE
```

Figure 13: Sample Program Fragment (Livermore Loop 24)

and ROPE. In particular, intermediate exit-tests (resulting from the unwinding of the loop) have been removed. With the idealized model, the time required by one loop iteration ranges from 15 to 54 cycles, for an average of 34.5. Even assuming several difficult post-scheduling optimizations (such as removing redundant memory fetches from the alternate paths), the time to execute the loop ranges from 15 to 40 cycles for an average of 27.5.

For the ROPE machine, the original loop in Figure 13 is first transformed by the enabling transformations (in particular, the loop is broken at point $(a)$ to minimize dependency chains). Next, the core transformations are applied. In the process, several simple algebraic manipulations are performed to support the core transformations. For example, operations of the form $m := K$ are changed to $m := K - 3$ as a side-effect of allowing $K := K + 3$ to percolate upward. Similarly, standard flow-analysis and peephole optimizations allow the removal of redundant fetches. Dead-code elimination removes the redundant assignments to $m$. 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

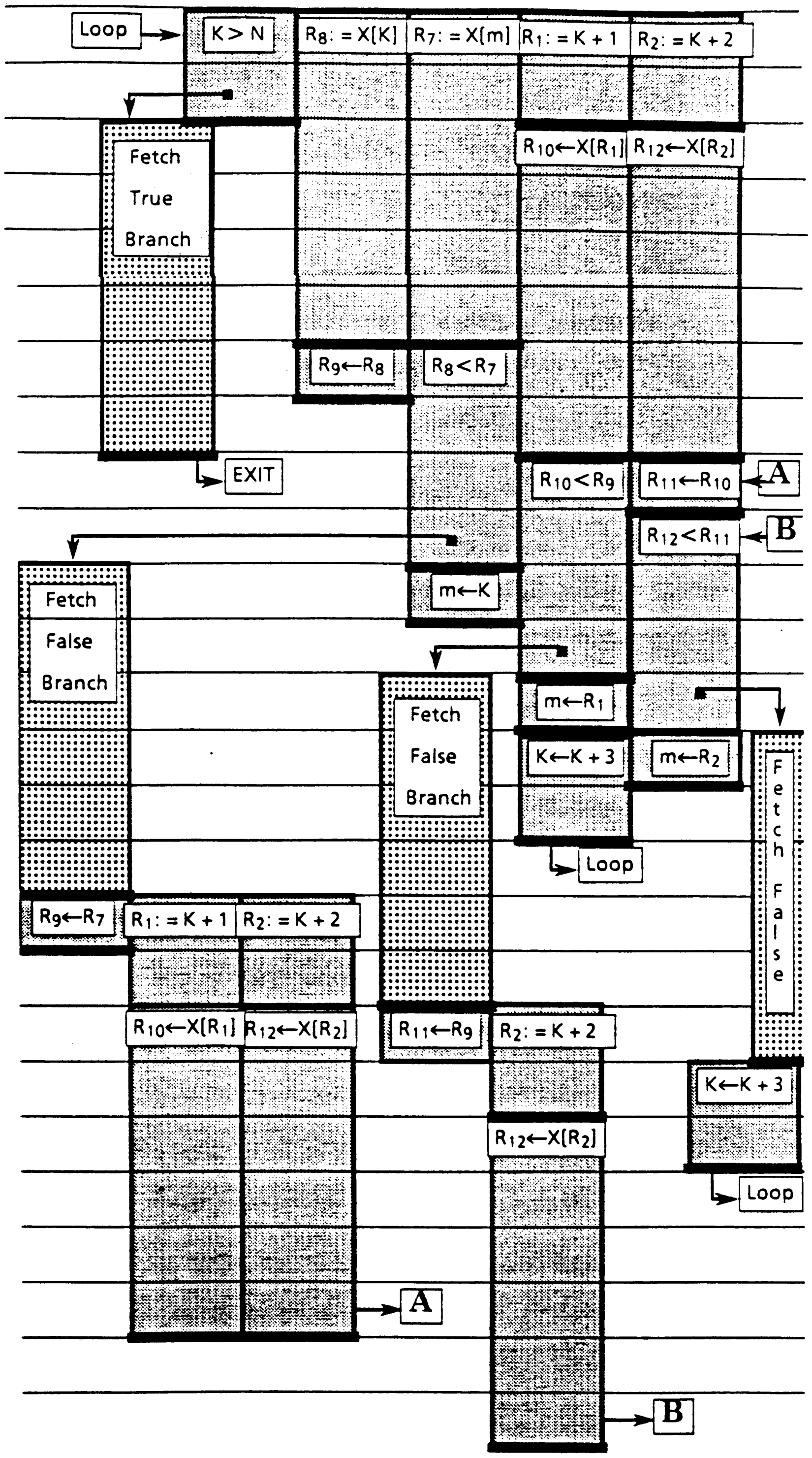


Figure 14: VLIW Machine Code and Execution Schedule

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 15 shows the machine instructions for the loop unrolled three times, and Figure 16 shows a trace of one iteration. Note that some jumps appear to start before their prefetches are finished, since we are taking advantage of the prefetch units not starting a new fetch when the address is the same as for the previous fetch.

For this loop, ROPE is 1.8 to 2.3 times faster than the VLIW machine, 2.6 times faster than the machine with hardware scheduling, and 4.8 times as fast as the sequential machine. Assuming a conservative 30 nanosecond cycle time, ROPE does loop 24 at 6.7 Mflops, as compared to 2.2 Mflops for the Cray-1, a 3-fold speedup, despite the slower clock rate of ROPE. ROPE is achieving 20% of its peak rate, while the Cray-1 gets about 1% on this loop. With the same clock speed as a Cray, we get 30 Mflops (13 times the Cray's performance).

The combined VLIW/ROPE architecture (machine model 5), of course, performs better than either the VLIW or the ROPE models. Its schedule only takes 8 cycles, for a speedup of 1.9 over pure ROPE and between 3.4 and 4.3 times over the pure VLIW.

This tiny example was chosen for ease of explanation. It also serves to illustrate the multi-way jump mechanism and the relative insensitivity to unpredictable conditionals of ROPE. Although this small loop is not necessarily typical of scientific code, similar unpredictable conditionals often occur in scientific programs. Our preliminary results indicate that percolation scheduling does better as program size increases. Considering the shortness of the code (only 17 instructions), the speedup achieved here is surprisingly good.

The ROPE machines have far fewer wasted cycles than the other approaches. Since the instruction prefetches, tests, and jumps are scheduled independently, we don't need to lengthen our schedule to wait for program memory, despite the absence of a cache.

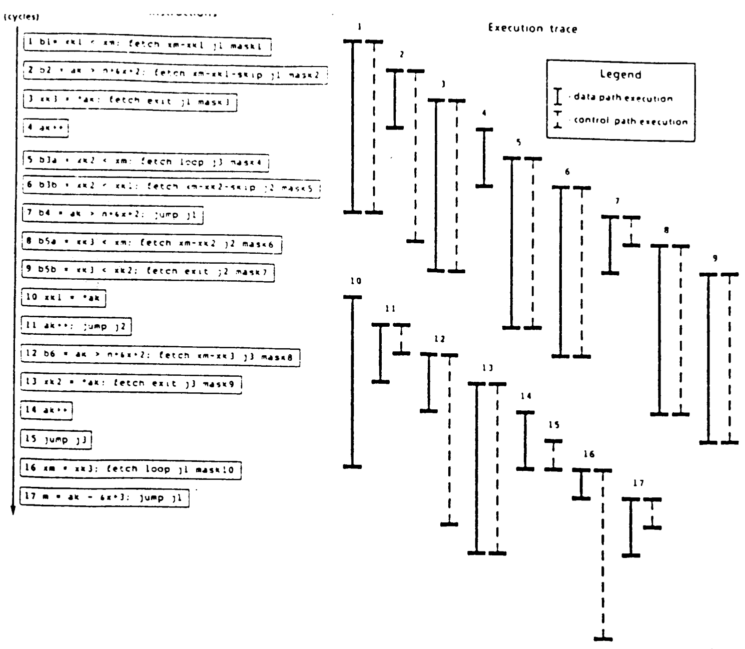
## 9 Limitations of ROPE

ROPE is an exciting idea to work on, but it has several limitations.

First, we have made the assumption that freezing a processor is less complicated than adding scheduling hardware. Although the concept of freezing is certainly simple, imple-

```
xm=x[1];     xk1=x[2];     xk2=x[3];
m=1;         ak=&x+4;      no-op; no-op;
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 15: ROPE machine instructions for 3× unrolling.



```
1 b1= xk1 < xm: fetch xm-xk1 j1 mask1
2 b2 = ak > n+6x+2: fetch xm-xk1-skip j1 mask2
3 xk3 = *ak: fetch exit j1 mask3
4 ak++
5 b3a = xk2 < xm: fetch loop j3 mask4
6 b3b = xk2 < xk1: fetch xm-xk2-skip j2 mask5
7 b4 = ak > n+6x+2: jump j1
8 b5a = xk3 < xm: fetch xm-xk2 j2 mask6
9 b5b = xk3 < xk2: fetch exit j2 mask7
10 xk1 = *ak
11 ak++: jump j2
12 b6 = ak > n+6x+2: fetch xm-xk3 j3 mask8
13 xk2 = *ak: fetch exit j3 mask9
14 ak++
15 jump j3
16 xm = xk3: fetch loop j1 mask10
17 m = ak - 6x+3: jump j1
```

Figure 16: Trace of one execution for 3× unrolled loop.

menting it in a high-speed processor may not be. Wiring delays across the processor may be close to the cycle time of the processor, so freeze requests would arrive at some parts of the processor too late to be accepted. Our best approach so far is to allow the processor to start execution eagerly, and abort the register writes when the freeze signal arrives. In order to ensure the correctness of this approach, we rely on compiler techniques similar to those for handling interrupts, redoing all the instructions that were in the pipe. Great care will be needed to be sure that no combination of memory references can cause deadlock.

Second, instructions are being fed to the data path at high speed, and more instructions are being fetched than are being used. Because ROPE attempts to prefetch all possible branch targets and the instructions that follow the targets, ROPE inherently requires a greater instruction memory bandwidth than more conventional prefetch designs. Fortunately, this bandwidth is distributed across the prefetch units not concentrated in a single bottleneck.

Some packaging problems may arise when partitioning the memory system into chips or boards. The high-speed bus to the data path and the ring itself must be kept physically small to avoid time-of-flight problems, but many connections are needed from the ring to the memory banks controlled by the ring. Pin limitations (at both the chip and board levels) will probably be the limiting factor in speed and performance of the ROPE system. The best solution would be to put the prefetch unit on the same chip as the memory it controls. Unfortunately, mixing RAM and logic on the same chip increases the cost and decreases the RAM density significantly.

Many of the prefetches requested by ROPE are repetitive, so the wide bandwidth may be reduced by adding caches in the prefetch units. Note that, unlike conventional caches, the caches in the prefetch units do not have to respond at data path speeds. The compiler can schedule prefetches to match the cache hit time.

Finally, much of the promise of ROPE rests on the quality of the optimizing compiler. Since an effective compiler will have to be quite slow, ROPE will be applicable mainly to real-time or compute-intensive applications with relatively long life spans.

## 10 Future Work

Before we can plan to build a hardware prototype of ROPE, we still have several interesting research questions to address. Most of the research will focus on the percolation scheduling compiler, both to determine how effective percolation scheduling is, and to determine what architectural features would be most valuable in a target architecture for percolation scheduling.

An interactive, graphical interface has been built for a preliminary version of the compiler. This interface allows us to play with program graphs without getting bogged down in bookkeeping details, so we can discover the heuristics needed to guide the core transformations. We are also beginning to explore heuristics for mapping the idealized schedules into realistic resource-limited schedules.

Although percolation scheduling can occasionally use very wide jumps, we do not believe that there will be significant speedups for jumps wider than 4 or 5 targets. Once the compiler is working, we hope to determine how valuable multi-way jumps are: to be able to plot execution speed versus the width of the widest legal jump for many different programs. Not only will this data allow us to evaluate different ROPE implementations, it will give us an idea how important it is for computer architects to look for other multi-way jump mechanisms.

Similarly, we hope to determine how deep a prefetch can be effectively scheduled. We are convinced that percolation scheduling can handle fairly large instruction fetch latencies without significantly increasing the number of cycles needed for a program. We need to find out how far percolation scheduling can be pushed. To give hardware designers some hard data for making pipelining trade-offs, we'd like to plot the number of cycles needed for various programs as a function of the latency of instruction fetches assumed by the compiler.

## 11 Conclusions

Our preliminary results are encouraging and we believe that our approach has significant advantages for the development of a cheap, high performance machine.

ROPE can be used by itself, or combined with VLIW architectures. The ability to handle complex and unpredictable flow of control could significantly enlarge the class of applications for which VLIWs are attractive.

## References

[AK82] J. R. Allen and K. Kennedy. *PFC: a Program to Convert Fortran to Parallel Form.* Technical Report MASC TR 82-6, Rice University, 1982.

[Ban79] U. Banerjee. *Speedup of Ordinary Programs.* Technical Report UIUCDS-R-79-989, University of Illinois Computer Science Department, October 1979.

[FERN84] J. A. Fisher, J. R. Ellis, J. C. Ruttenberg, and A. Nicolau. Parallel processing: a smart compiler and a dumb machine. In *Proceedings of the ACM Symposium on Compiler Construction*, 1984.

[Fis79] J. A. Fisher. *The Optimization of Horizontal Microcode within and beyond Basic Blocks: an Application of Processor Scheduling with Resources.* PhD thesis, New York University, New York, 1979.

[Fis80] J. A. Fisher. An effective packing method for use with $2^n$-way jump instruction hardware. In 13th *Annual Microprogramming Workshop*, Colorado Springs, November 1980. also in *SIGMICRO Newsletter*, 11(3&4), 64–7.

[Fis81] J. A. Fisher. Trace scheduling: a technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–49, July 1981.

[FR72] C. C. Foster and E. M. Riseman. Percolation of code to enhance parallel dispatching and execution. *IEEE Transactions on Computers*, 21(12):1411–141, December 1972.

[GHL*85] J. R. Goodman, J. Hsieh, K. Liou, A. R. Pleszkun, P. B. Schechter, and H. C. Young. PIPE: a VLSI decoupled architecture. In *The* 12th *Annual International Symopsium on Computer Architecture*, pages 20–27, Boston, MA, June 17–19 1985.

[HJP*82] J. Hennessy, N. Jouppi, S. Przbyski, C. Rowen, T. Gross, F. Baskett, and J. Gill. Mips: a microprocessor architecture. In 15th *Annual Microprogramming Workshop*, pages 5–7, Palo Alto, CA, October 1982. also in *SIGMICRO Newsletter*, 13(4), December 1982, 17–2.

[KKP*81] D. Kuck, R. Khun, D. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In 8th *Annual ACM Symposium on the Principles of Programming Languages*, pages 207–218, Williamsburg, VA, January 26 1981.

[KN85] Kevin Karplus and Alexandru Nicolau. Efficient hardware for multi-way jumps and pre-fetches. In *Micro18: the 18th Annual Microprograming Workshop*, pages 11–18, Monterey, CA, December 1985.

[Kuc76] D. J. Kuck. Parallel processing of ordinary programs. *Advances in Computers*, 15:119–179, 1976.

[McM83] F. H. McMahon. Lawrence livermore national laboratory FORTRAN kernels: MFLOPS. 1983.

[NF84] A. Nicolau and J. A. Fisher. Measuring the parallelism available for very long instruction word architectures. *IEEE Transactions on Computers*, C-33(11):968–97, November 1984.

[Nic84a] A. Nicolau. *Loop Quantization, or Unwinding Done Right.* Technical Report, Cornell University Computer Science Department, 1984.

[Nic84b] A. Nicolau. *Parallelism, Memory Anti-Aliasing, and Correctness for Trace Scheduling Compilers.* PhD thesis, Yale University, New Haven, Connecticut, 1984.

[Nic85] A. Nicolau. *Percolation Scheduling: A Parallel Compilation Technique.* Technical Report TR 85-678, Cornell University Computer Science Department, May 1985.

[NK85] Alexandru Nicolau and Kevin Karplus. ROPE: a statically scheduled supercomputer architecture. In *First International Conference on Supercomputing Systems*, St. Petersburg, FL, December 1985.

[PLT79] 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.

[PS82] D. A. Patterson and C. H. Sequin. A VLSI RISC. *Computer*, 15(9):8–21, September 1982.

[Rus78] R. M. Russell. The CRAY-1 computer system. *Comunications of the ACM*, 21(1):63–7, January 1978.

[Smi82] A. J. Smith. Cache memories. *Computing Surveys*, 14(3):473–53, September 1982.

[Smi84] J. E. Smith. Decoupled access/execute computer architectures. *ACM Transactions on Computer Systems*, 2(4):289–30, November 1984.

[TF70] G. S. Tadjen and M. J. Flynn. Detection and parallel execution of independent instructions. *IEEE Transaction on Computers*, 19(10):889–89, October 1970.

[WS84] S. Weiss and J. E. Smith. Instruction issue logic in pipelined supercomputers. *IEEE Transactions on Computers*, C-33(11):1013–102, November 1984.