A COMPILER-DRIVEN SUPERCOMPUTER

KEVIN KARPLUS AND ALEXANDRU NICOLAU[1]
Computer Science Department
Cornell University Ithaca, NY 14853

ABSTRACT

The overall performance of supercomputers is slow compared to the speed of their underlying logic technology. This discrepancy is due to several bottlenecks: memories are slower than the CPU, conditional jumps limit the usefulness of pipelining and pre-fetching mechanisms, and functional-unit parallelism is limited by the speed of hardware scheduling. This paper describes a supercomputer architecture called Ring of Pre-fetch Elements (ROPE) that attempts to solve the problems of memory latency and conditional jumps, without hardware scheduling. ROPE consists of a very pipelined CPU data path with a new instruction pre-fetching mechanism that supports general multi-way conditional jumps. An optimizing compiler based on a *global* code transformation technique (Percolation Scheduling or PS) gives high performance without scheduling hardware.

INTRODUCTION

Traditional computer architectures use resources inefficiently, resulting in machines whose performance is disappointing when compared to the raw speed of their components. The main technique for running processors near the limits of a technology is *pipelining*. An operation in a pipelined machine may take several cycles to complete, but a new operation can be started on each cycle, so the throughput remains high. The benefits of pipelining, however, have been limited by the difficulty of keeping the pipeline full. The difficulty can be traced to two sources: data dependencies and the slowness of memory.

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. In many supercomputer architectures, complex scheduling hardware is used to keep the almost independent processing units from violating the data dependencies implicit in the code. Although scheduling hardware allows some overlapping of normally sequential operations, the final machine is only about twice as fast as a strictly sequential machine. Even with far more sophisticated scheduling hardware than that in current machines, only another factor of one and a half is obtained [18] . The scheduling mechanism is not only expensive to build, but it also slows down the basic cycle time, since it must operate faster than the processing units.

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 pre-fetch units and instruction caches remove most of the instruction-fetch delay, but substantial penalties are incurred for conditional jumps and cache misses. The smallness of basic blocks[17], [12] and corresponding frequency of jumps has usually limited the size of pipelines to two or three stages [16].

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 [8] [15], but is unsuitable for high-performance

[1] This work is supported in part by NSF grant DCR-8502884 and the Cornell NSF Supercomputing Center.

Applications of Supercomputers
Deborah F. Lockhart and Darrell L. Hicks, Editors

scientific computation, where floating-point operations are common. RISC architectures are designed to take advantage of mature compiler technology. ROPE is designed around more modern, experimental compiler technology.

The lack of sufficiently powerful compiler techniques has forced this choice of reduced performance, scheduling hardware, or sequences of simple instructions. Pipelined architectures allow instructions to overlap, but this overlapping helps only if the instructions can be ordered so that data dependencies do not slow execution unnecessarily. Such good execution schedules usually exist, but finding them is difficult.

Until recently, code transformations were limited to source transformations or to code motion within basic blocks. Although source transformations are often useful [10] [1], 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 [8], [15]. Such methods, however, do not take full advantage of the speed of a given technology, particularly when floating point units are available.

Our architecture is designed to take advantage of a new code transformation technique, Percolation Scheduling (PS)[13] that schedules operations globally to avoid data dependency. PS developed from our experience with Trace Scheduling [6] in the ELI project, and attempts to overcome the problems that limit the effectiveness of trace scheduling.

A novel mechanism for pre-fetching instructions reduces memory bottlenecks and the need to flush pipes on conditional jumps. This approach is used to obtain supercomputer performance at relatively moderate cost.

Since PS, like other sophisticated code transformation techniques, can be 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 micro-processor, but a supercomputer architecture that supports scientific computation, simulation and complex signal processing. We believe that ROPE architectures combined with percolation scheduling will provide a cost effective alternative to existing supercomputers and array processors.

THE ARCHITECTURE

Our proposed architecture has two parts: the data path and the instruction controller. 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.

Figure 1 shows the block diagram for the ROPE machine.

Data Path

The data path is a conventional design, in that it 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. The register file must be large enough to reduce memory traffic significantly. The register file 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.[2]

[2] Obviously recursive calls cannot be completely expanded. In practice only frequently executed calls (based on sample runs) would be expanded.

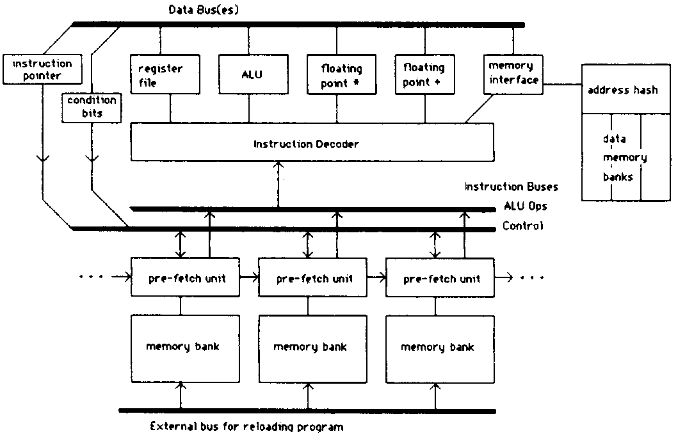


Figure 1: ROPE block diagram.

- 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 pre-fetch mechanism described in the next section can be used to guarantee that an instruction is always ready to execute. Various techniques are suggested in the Data Memory section 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.

- 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

Departments. We are deeply grateful to Bill Powers (Dean of the College of Science and Arts) and Tim Whitten (Vice President of Academic Affairs) for their continuing support and encouragement on this project in particular and our other endeavors at MTU in general. We extend our thanks to the people who prepared and/or presented lectures at the workshop; a complete list alphabetized by the author's last name follows:

Clifford Arnold (ETA Systems) "Simulating Very Strong Shocks in Three Dimensions"

R.C.Y. Chin, G.W. Hedstrom, J.R. McGraw and F.A. Howes (Lawrence Livermore National Laboratory and University of California at Davis) "Parallel Computations of Multiple Scales Problems"

Jack Dongarra and Danny Sorensen (Argonne National Laboratory) "Symmetric Eigenvalue Problems for Parallel Processors"

Michael Gilpin (MTU) "Applications of Parallel Processors to Some Hard Discrete Problems, Part 1"

G.W. Hedstrom (LLNL, see R.C.Y. Chin)

Darrell Hicks (MTU) "Introduction to Computational Continuum Dynamics"

Richard Hill and Gerald Ludden (Michigan State University) "Purcell's Method, Projective n-Space and Sparse Systems"

F.A. Howes (U. CA at Davis, see R.C.Y. Chin)

Glenn Ierley (MTU) "Solution of Nonlinear Boundary Value Problems by Spectral Methods: A Case Study"

Kevin Karplus and Alexandru Nicolau (Cornell University) "ROPE, A Statically-Scheduled Supercomputer Architecture"

Barry Kunz (MTU) "Electron Correlation in Large Systems"

Jon Larsen (KMS Fusion, Inc) "Computations in Fusion Physics"

Lorie Liebrock (MTU, see Joseph McGrath)

John Lowther (MTU) "AI Architectures and Parallelism"

Olaf Lubeck, James Moore and Raul Mendez (Los Alamos National Laboratory and the Naval Postgraduate School) "A Benchmark Comparison of the Japanese Supercomputer and the CRAY-XMP"

Gerald Ludden (MSU, see Richard Hill)

Henry Makowitz (CRAY Research, Inc.) "Numerical Experiments in Concurrent Multiprocessing with Thermal-Hydraulic Nuclear Reactor Systems Codes"

Joseph McGrath, Darrell Hicks and Lorie Liebrock (KMS Fusion, Inc. and MTU) "Algorithms for Computational Continuum Dynamics"

J.R. McGraw (LLNL, see R.C.Y. Chin)

Steve McGrogan (ELXSI, see Robert Olson) 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 jumps, control can pass to any pre-fetch unit that is ready.

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, some additional mechanism is needed for starting instruction fetches. Jump targets are started with explicit pre-fetch instructions.

Figure 3 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 as a result of 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 is ignored by target units, but starts a fetch of **address.left** on non-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. There are usually several start-fetch tokens being passed around the ring at once.

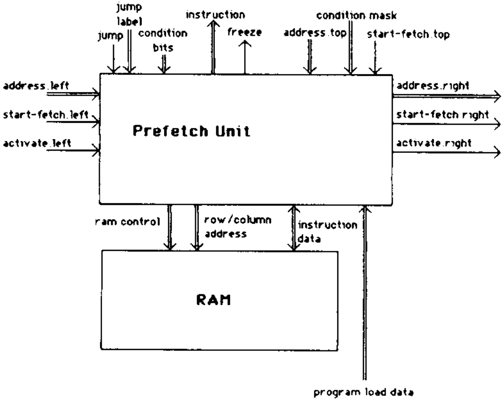


Figure 3: One pre-fetch unit

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 a token.

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 pre-fetch unit. The condition mask specifies under what conditions a pre-fetch unit activates. It is saved for future comparison with the condition bits[3]. The condition bits 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, `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 activate 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`. It is the compiler's responsibility to ensure that exactly one pre-fetch unit responds to a jump signal.

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 data operations are standard instructions for the data path discussed previously. 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 (to handle return from procedures and pointers to functions, for example). The jump label and condition mask are stored by the pre-fetch unit for later matching.

JUMP (*jump-label*): Activate one of the target pre-fetch units for the specified jump. Jumps are labeled so that pre-fetches can be moved past jumps while scheduling. Jump labels can be re-used when there is no danger of a conflict, so two or three bits probably would suffice. 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. This separation of the functions of a conditional branch is the main advantage of a ROPE architecture.

On a conventional machine, jumping to one of three addresses requires two conditional jumps, as in the left half of Figure 4. On a ROPE machine, a single three-way jump is used (right half of Figure 4). Figure 5 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. The tests in figure 5 are setting condition bits 1, 2. These tests correspond to those shown in figure 4. The masks in the prefetch operations (e.g., $\bar{1},\tilde{2}$) are expressed as functions of the condition bits set by (the separate) test operations.

Although ROPE's pre-fetch units are fairly cheap, it would be naïve to build a machine with thousands of them. How many do we need to support multi-way jumps? Let us assume that

[3] We have devised several different coding schemes for the condition mask, but not settled on one yet. We will have to examine many multi-way jumps to determine the relative importance of a short encoding and flexibility in expressing the conditions.

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, each target branch need only pre-fetch up to the next jump instruction, and fewer units 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 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.

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, since basic blocks are usually less than five instructions [12]. 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.

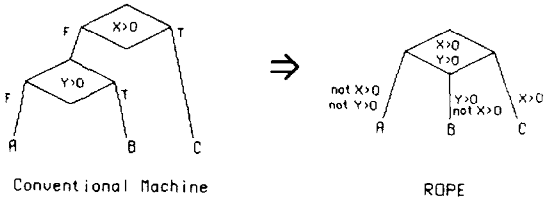


Figure 4: Combining two conditional jumps into a three-way jump.

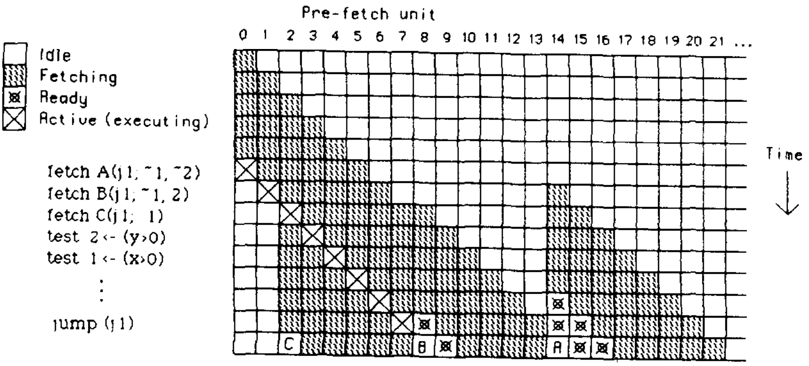


Figure 5: Possible execution sequence for a three-way jump.

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. 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 request is made to a bank that is busy, or if a bank does not complete an operation in the scheduled time. Caches are not needed for moderate performance (up to about 20 Mflops), but a cache may be needed for each bank at higher speeds.

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

Figure 6: Indirect references that may conflict.

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 [11]. Unlike VLIW's, the ROPE machine issues instructions one at a time. The bandwidth of the data memory is therefore not as critical as in VLIW's 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.

Alternatively, data memory conflicts and the resulting delays can sometimes be reduced by distributing multiple copies of read-only data into the banks[4]. 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 whether ambiguous (indirect) memory references are aliases of each other. An other possibility is the use of statistics gathered in test runs to help in the allocation of data to memry banks. If the chances of collision are high, the references can be scheduled with an interval that will avoid freezing, allowing PS to use the intervening time for other operations. Since PS supports global code motions, the chances that such operations could be found are good.

Figure 6 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 the absence of more information about i and j, we can't guarantee that a[i] and b[] 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

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

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.

PERCOLATION SCHEDULING

Percolation Scheduling (PS) 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 PS to improve code produced by other compilers.

An overview of the PS hierarchy and the work we have completed is presented in a companion paper in this issue. A more detailed discussion can be found in [13].

Mapping Percolation Schedules to ROPE

The transformations in PS 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. 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 schedule is NP-hard, unless every operation takes only one cycle. The greedy algorithm outlined below, however, appears to be adequate. It is a refinement of List Scheduling, which is reportedly very effective [4].

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 must be assigned address 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 pre-fetching 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.

CURRENT WORK

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, pre-fetching, 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 [2],[11] are used. Although most existing compilers do not provide such support, we include the assumption so our comparison is not biased towards the PS 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 [8].

The third simulated machine is our ROPE architecture. It has the structure described in section 2, and uses PS and the mapping techniques described in section 3. 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 the instruction after they have been read, and that they are read in a fixed cycle of the instruction. This assumption removes most read-write dependencies and is realistic for a pipelined architecture.

The fourth model is an *idealized* VLIW machine [3]. 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 pre-fetching 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 pre-fetch and uses a percolation scheduling compiler. The combination of ROPE and PS make the VLIW architecture much less sensitive to branch probabilities.

	Machine model				
Instruction Types	Sequential	Pipelined	VLIW	ROPE	VLIW/ROPE
Register to register transfers	1	1	1	1	1
Integer add, compare	2	2	2	2	2
Floating-point add, compare	4	4	4	4	4
Indexed data-fetch	6	6	6	6	6
Sequential instruction fetch	0	0	0	0‡	0
Non-sequential instruction fetch	6	6	6	6	6
Conditional-jump	compare + 6	compare + 0	compare + 0/6	1†	1†
Goto	6	0	0	1	1

† ROPE compares (tests) are separate instructions.
‡Sequential instruction fetches are handled automatically by the ring.

Figure 7: Operation times in cycles

For the following example we use the timings shown in Figure 7. These timings are consistent with current off-the-shelf components. We have also considered other timings (for example, those of the Cray-1). Choosing other times will not affect the ROPE architecture or the percolation scheduling compiler, but may of course change the speed of programs.

Despite the greater hardware complexity of the conventional pipelined architecture and of the VLIW model, our preliminary results show significant speedups for PS and the ROPE architecture, even on small problems (binary search, bubble sort, Livermore Loop 24, and matrix multiplication) over all other models.[5] 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.

An Example

The code in Figure 8 (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 for an average of 71.5 cycles.

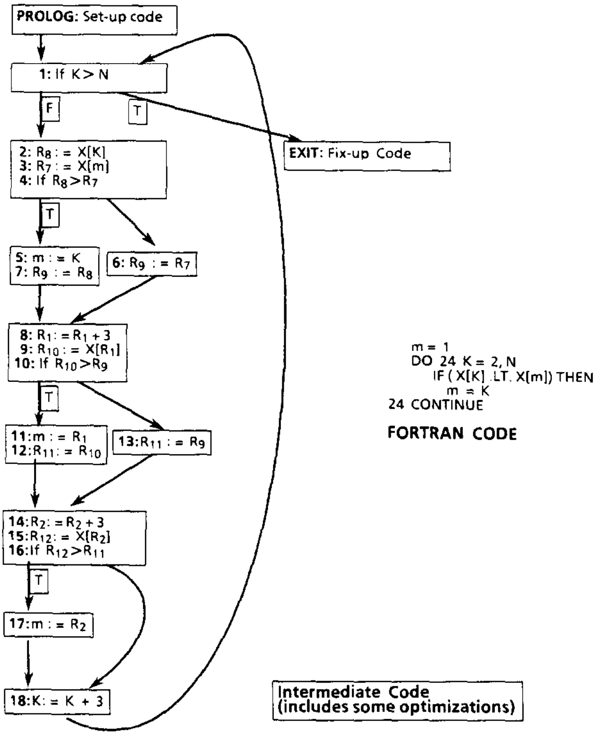
The loop body requires between 38 and 41 cycles to execute on the machine with hardware scheduling, for an average of 39.5. cycles[6]. This architecture is about 1.8 times as fast as the sequential machine on this example, which is consistent with the speedups reported in [18]. The actual performance of the Cray-1 (cycle time 12.5 nanoseconds) for this loop is of about 2.3 Mflops according to the Livermore benchmarks.[7]

For the VLIW machine, the intermediate (NADDR) code and the trace scheduled code is shown in Figure 9. On-trace jumps and unconditional jumps are assumed to be pre-fetched. 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 and

[5] Since our compiler is not fully implemented, the experimenting requires significant human help, limiting our ability to deal with large programs. However, the conditional structure of these programs is not amenable to standard pipelining/vectorization and is representative of a large class of problems (for example, Monte-Carlo simulation). Inspection of the resulting schedules makes it obvious that even better performance is to be expected on larger programs.

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

[7] 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 8: Sample Program Fragment (Livermore Loop 24)

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 8 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 branches r and 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 10 shows the machine instructions for the loop unrolled three times, and Figure 11 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.

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. While the density of conditionals in this small piece of code is not necessarily typical of scientific code, unpredictable conditionals often occur in scientific programs, as well as in other applications, such as systems code or AI. Our preliminary results indicate that PS and ROPE can do even better on larger programs. 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 pre-fetches, 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.

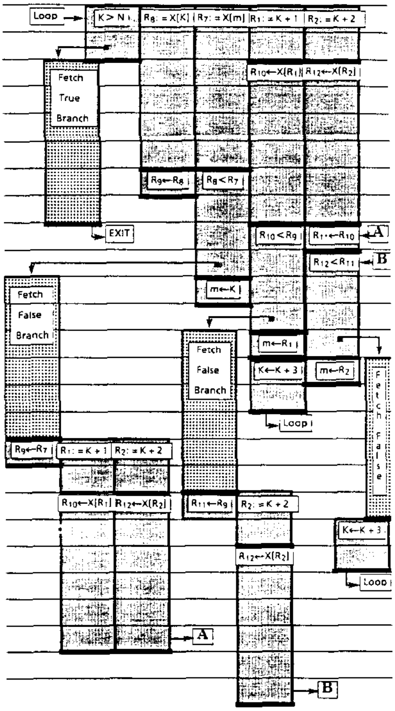


Figure 9: VLIW Machine Code and Execution Schedule

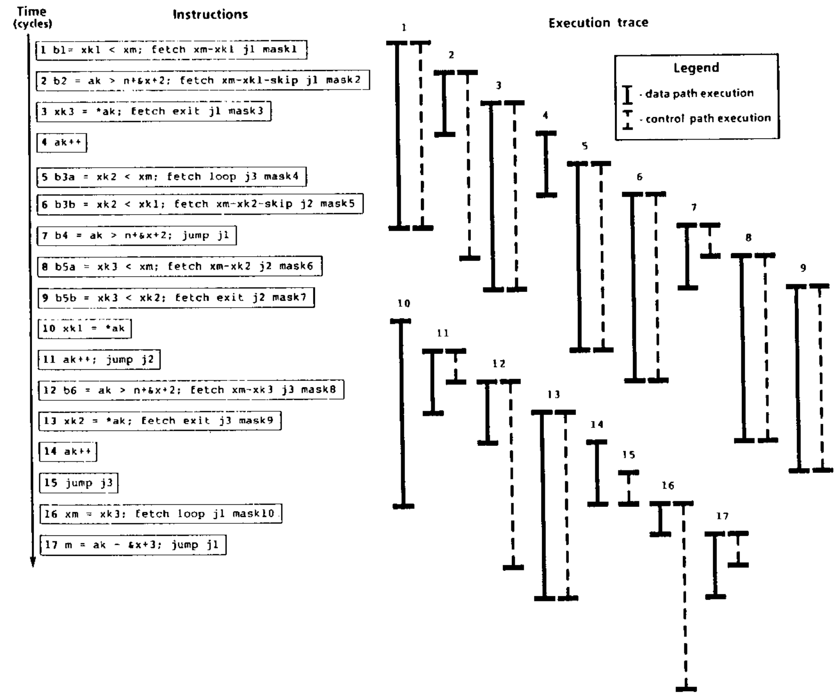
```
xm=x[1];   xk1=x[2];   xk2=x[3];
m=1;       ak=&x+4;    nop^2;
loop:
     b1 = xk1<xm;
     b2 = ak>n+&x+2;
     xk3=*ak;          ak++;
     b3a = xk2<xm;     b3b= xk2<xk1;
     b4 = ak>n+&x+2;
          if (b2) goto exit
          if (b1) {xm=xk1; m=ak-(&x+3)}
     b5a = xk3<xm;     b5b= xk3<xk2;
     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 10: ROPE machine instructions for 3× unrolling.

FUTURE WORK

Our next goal is to refine our model into a hardware implementation. To do so, further study is required:

- Extensive experimentation with our simulated machine is needed to evaluate the effectiveness of our approach and to choose the appropriate number of registers, pre-fetch units, and data memory banks. Experimentation can help us resolve questions such as:
  - How valuable are multi-way jumps?
  - How wide a multi-way jump can be used effectively?
  - How useful is binary branching with multiple pre-fetching?
- The complexity of the mapping algorithm has to be investigated, and efficient implementations are required.
- Alternative approaches for data and program memory layout (graph coloring algorithms, for example) should be investigated.



```
1 b1= xk1 < xm; fetch xm-xk1 j1 mask1
2 b2 = ak > n+&x+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+&x+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+&x+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 - &x+3; jump j1
```

Figure 11: Trace of one execution for 3× unrolled loop.

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 VLIW's are attractive.

References

[1] J. R. Allen and K. Kennedy. *PFC: a program to convert Fortran to Parallel form.* Rice University Technical Report MASC TR 82-6, (1982).

[2] U. Banerjee. *Speedup of Ordinary Programs.* University of Illinois Computer Science Technical Report UIUCDS-R-79-989, (1979).

[3] J. A. Fisher, J. R. Ellis, J. C. Ruttenberg, and A. Nicolau. Parallel Processing: A Smart Compiler and a Dumb Machine. *Proceedings of the ACM Symposium on Compiler Construction.*, (1984).

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

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

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

[7] J. A. Fisher. *Very long instruction word architectures and the ELI-512.* Yale University Department of Computer Science, Technical Report 253, (1982).

[8] J. Hennessy, N. Jouppi, S. Przbyski, C. Rowen, T. Gross, F. Baskett, and J. Gill. MIPS: A Microprocessor Architecture. 15th *Annual Microprogramming Workshop,* Palo Alto, CA, Oct 5–7, *SIGMICRO Newsletter,* 13(4), 17–22, (1982).

[9] D. J. Kuck. Parallel Processing of Ordinary Programs. *Advances in Computers,* vol. 15, 119–179, (1976).

[10] D. Kuck, R. Khun, D. Padua, B. Leasure, and M. Wolfe. Dependence Graphs and Compiler Optimizations. 8th *Annual ACM Symposium on the Principles of Programming Languages,* Williamsburg, VA, Jan. 26, 207–218, (1981).

[11] A. Nicolau. *Parallelism, Memory Anti-Aliasing, and Correctness for Trace Scheduling Compilers.* Yale University Ph. D. Thesis, New Haven, Connecticut, (1984).

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

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

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

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

[16] R. M. Russell. The CRAY-1 Computer System. *Comunications of the ACM* 21(1), 63–72, (1978).

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

[18] S. Weiss and J. E. Smith. Instruction Issue Logic in Pipelined Supercomputers. *IEEE Transactions on Computers* C-33(11), 1013–1022, (1984).