DRAFT SIAM

ROPE: a statically-scheduled supercomputer architecture

Kevin Karplus and Alexandru Nicolau

1 Introduction

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. The smallness of basic blocks ([TadjFlynn], [Nicolau84b]) and corresponding frequency of jumps has limited the size of pipelines to two or three stages [Russell78]. 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 only achieves threefold speedups over simple sequential machines [WeiSmith84]. The scheduling mechanism is expensive to build, and slows down the cycle time, since it must operate faster than the processing units.

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. Instruction caches work well at moderate processor speeds, but are difficult to design for the short cycle time envisioned for this architecture.

ROPE's novel mechanism for pre-fetching instructions reduces memory bottlenecks and the need to flush pipes on conditional jumps, obtaining supercomputer performance at relatively moderate cost.

ROPE is designed to take advantage of a new code transformation technique, Percolation Scheduling (PS) [Nicolau84c]. The core of PS is a small set of primitive program transformations acting on adjacent nodes in a program graph. Higher level heuristics and transformations direct the core transformations and rearrange the program graph to allow more code motion by the core transformations. This talk describes only the architecture, not the compiler techniques.

2 The Architecture

Our proposed architecture has two parts: the data path and the instruction controller. ROPE can work with data paths of varying degrees of parallelism, from simple pipelined units to very-large-instruction-word units. The important contribution of ROPE is to parallelize the instruction fetching and conditional jump evaluation.

On each cycle, one or more data-path instructions and one control-flow instruction are begun. A functional unit that takes longer than a cycle is pipelined or duplicated, so a new operation can begin on every cycle.

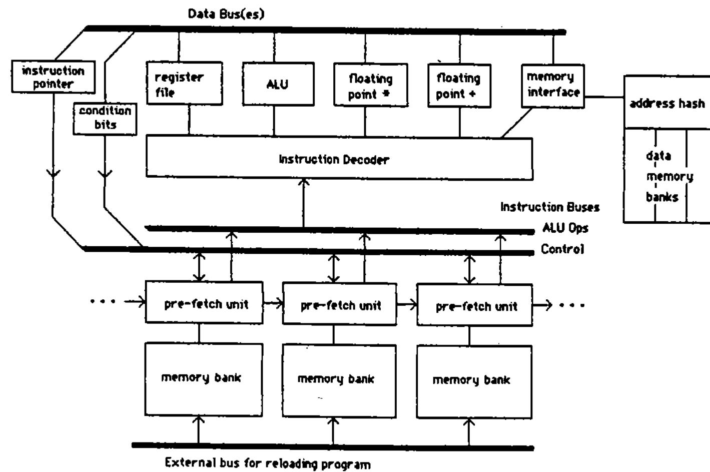

Figure 1: ROPE block diagram.

```
     if X>0 then C
else if Y>0 then B
else             A
```

Figure 2: Related conditional jumps.

Figure 1 shows the block diagram for the ROPE machine.

2.1 Data Path

Any of several different data paths can be used with ROPE. The essential characteristics are that the data path be able to start any operation on each cycle, that floating-point hardware is provided, and that the time taken for each operation to finish is known to the compiler.

Being geared to scientific computation, ROPE has no interrupt handling or fast context-switching. Rather than slow down the main processor for these rare tasks, ROPE uses cheaper, slower processors to handle instruction traps, page faults, and I/O.

2.2 Program Memory

The program memory must provide an instruction on each clock cycle even though the read time of the memory may be 10 clock cycles or more. Our design has interleaved memory banks, but avoids the jump penalty by using intelligent pre-fetch units.

Multi-way jumps are useful for increasing the ratio of sequential operations to jumps. Furthermore, the core transformations of PS will cluster conditional jumps, making multi-way jumps particularly attractive. Merging n completely independent jumps might be too expensive to consider, as n jumps can have $2n$ independent targets, for n of which execution would have to proceed in parallel. Luckily, most conditional jumps occurring in ordinary code are not independent, and for n conditionals only one out of $n+1$ targets is chosen. PS will cluster only such conditional jumps. An example of dependent conditional jumps is shown in Figure 2. For a more detailed discussion of the semantics of multi-way jumps, see [Nicolau84c].

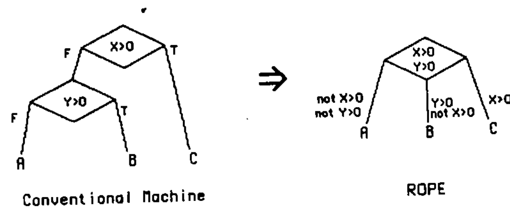

Figure 3: Combining two conditional jumps into a three-way jump.

The ROPE pre-fetch mechanism easily supports multi-way jumps. Each of the 2^n pre-fetch units has its own block of memory, independent of the other units (see Figure 1). On any given cycle, one pre-fetch unit, called the *active* pre-fetch unit, provides an instruction to the decoder. The other pre-fetch units may be either busy fetching words from their memories, or ready to become active. 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 was specified in in a preceding pre-fetch instruction.

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 multiple target addresses must be ready at jump instructions, explicit pre-fetch instructions are needed to start fetches for jump targets. Each pre-fetch instruction includes a target address, a *jump label* indicating which jump it corresponds to, and a condition mask indicating what combination of condition bits will select this target. The target address is usually a constant, but may come from the data path (for subroutine returns and pointers to functions).

A jump instruction has only one operand, the jump label. Pre-fetch units that are targets for that jump check their condition masks against bits from the datapath, and the selected unit provides the next instruction.

On a conventional machine, jumping to one of three addresses requires two conditional jumps, as in the left half of Figure 3. On a ROPE machine, a single three-way jump is used (right half of Figure 3). Figure 4 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.

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. A ROPE machine with 32 or 64 pre-fetch units should achieve almost all the possible speedup of this architecture.

The compiler schedules the pre-fetches and assigns code addresses to minimize waiting for instruction fetches. For tree-structured control flow with infrequent branching, scheduling pre-fetches and assigning code addresses is easy. If branching is frequent and pre-fetch units are scarce, delays are unavoidable with any schedule. A code block that has multiple predecessors, such as loop or the statement after an if-then-else, may need to be duplicated to avoid conflicting placement requirements.

2.3 Data Memory

The data memory, like other units on the data path, can start new operations every cycle, but may take several cycles to finish. Since large data memory is essential for many problems, slow, cheap

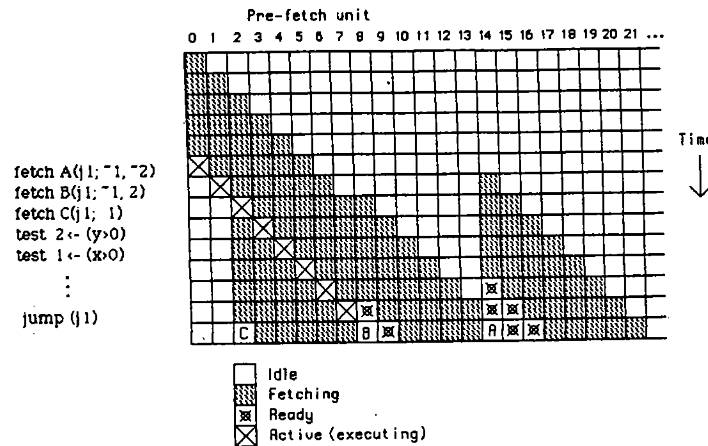

Figure 4: Possible execution sequence for a three-way jump.

RAM is used. The RAM is split into banks, and each bank processes only one memory operation at a time, but different banks can execute memory operations concurrently. If a requested bank is busy, the processor freezes until it becomes available.

The compiler schedules memory operations as if they take a constant time to execute. Cache misses and page faults force the processor to freeze until the memory operations can be completed.

We can reduce data-memory freezing by having the compiler carefully map 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 [Nicolau84a]. Memory bank conflicts could also be reduced by making the mapping from address to memory bank be a hash function. Studies are needed to determine if memory bank conflicts are frequent enough to justify the extra cost of hashing.

References

[Nicolau84a] A. Nicolau. *Parallelism, Memory Anti-Aliasing, and Correctness for Trace Scheduling Compilers.* Yale University Ph. D. Thesis, New Haven, Connecticut, 1984.

[Nicolau84b] A. Nicolau and J. A. Fisher. Measuring the Parallelism Available for Very Long Instruction Word Architectures. *IEEE Transactions on Computers,* C-33(11), Nov. 1984, 968–976.

[Nicolau84c] A. Nicolau. *The design of a Global Parallel Compilation Technique—Percolation Scheduling.* Cornell University, Computer Science Technical Report, 1984.

[Russell78] R. M. Russell. The CRAY-1 Computer System. *Comunications of the ACM* 21(1), Jan. 1978, 63–72.

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

[WeiSmith84] S. Weiss and J. E. Smith. Instruction Issue Logic in Pipelined Supercomputers. *IEEE Transactions on Computers* C-33(11), Nov. 1984, 1013–1022.