Computer Science Department Technical Report
Artificial Intelligence Laboratory
University of California
Los Angeles, CA 90024-1596

MULTIPLE INSTRUCTION MULTIPLE DATA EMULATION ON THE CONNECTION MACHINE

Robert J. Collins

February 1991
CSD-910004

Multiple Instruction Multiple Data Emulation on the Connection Machine[1]

Robert J. Collins[2]
Artificial Life Laboratory
Department of Computer Science
University of California, Los Angeles
Los Angeles, CA 90024

February 19, 1991

[1]A thesis submitted in partial satisfaction of the requirements for the degree of Master of Science in Computer Science, University of California, Los Angeles, 1991.

[2]Electronic mail address: rjc@cs.ucla.edu

Abstract

A common criticism of SIMD computers such as the Connection Machine is that the data parallel programming style is only appropriate for a small class of problems. The parallelism in data parallel algorithms comes from the simultaneous application of the same operation to each datum in a large set of data, rather than from multiple threads of control. Many data parallel programs require slightly different sequences of operations on different data items, and thus contain **where** statements, describing one or more statement sequences, and where in the data set each statement is to be executed.

On the Connection Machine, **where** statements are usually implemented by issuing all of the embedded statements, turning on the appropriate processors before executing each sequence. Given this usual implementation, the execution time for a **where** statement is the sum of the execution times for each of the **where**'s statement sequences.

This thesis describes an alternative implementation for **where** statements, which interprets all of the statement sequences in parallel on an emulated MIMD computer. The MIMD emulation implementation executes a **where** statement in time proportional to its longest alternative sequence. We discuss MIMD emulation in detail, including paradoxical results that techniques used to speed processors for real MIMD machines do *not* work for SIMD–emulated MIMD machines, and that global throughput can be enhanced by selectively slowing down local computations. Emulated MIMD throughput of 50–350 MIPS is typical for small MIMD instruction sets (on 64K Connection Machine processors). We also describe a number of source code optimizations that can be automatically performed to **where** statements to improve run–times when MIMD emulation is used.

Chapter 1

Introduction

Two major classes of highly parallel architectures are Multiple Instruction Multiple Data (MIMD) and Single Instruction Multiple Data (SIMD). A MIMD machine is a collection of connected serial computers, each of which can be thought of as executing its *own* sequential program on its own data (control parallelism). MIMD architectures are generally designed around either shared memory or message passing communication. A SIMD machine is a collection of processing elements, each executing the *same* sequential program on its own data (data parallelism). Each SIMD processor independently either executes or ignores each instruction. Processors that are executing instructions on any given cycle are said to be *selected* for that cycle.

The Connection Machine system is a hybrid computer, consisting of a standard serial front end (e.g. Sun4/330) and the SIMD back end [4]. The front end operating system supports program development, networking, and I/O operations. The Connection Machine itself consists of up to 65,536 1-bit processing elements, an interconnection network called the router, and connections to the front end. Each processing element has 65,536 bits of local memory.

The execution of Connection Machine programs occurs both on the front end and on the Connection Machine. The front end executes the sequential portions of the program, and sends parallel instructions to the Connection Machine in a buffered stream. Sequential execution on the front end can occur in parallel with the execution of previously issued Connection Machine instructions.

Several Connection Machine programming languages are available. The high level languages are C* [8, 12] and *LISP [13], which are extensions of C [7] and Common LISP [9] respectively. A C++ [10, 3] interface is provided by CM++ [2]. The "assembly" language of the Connection Machine is called PARIS (PARallel Instruction Set) [11], which can be called from any of C, C++, Common LISP, C*, or *LISP.

The Connection Machine languages present a data parallel computing model [6, 5, 13], in which there are a large number of small processors that are all driven simultaneously by a shared instruction stream. A typical decomposition strategy is to map one element or datum of the problem to each processing element. For example, in a VLSI logic simulation, each transistor in the circuit might be assigned to a different Connection Machine processor. This allows all of the processors to apply the transformations that are specified by the single instruction stream to their as-

signed transistors in parallel. On the Connection Machine, when more processors are logically required than are physically present, the Connection Machine can transparently operate in "virtual processor mode." Each physical processor emulates several virtual Connection Machine processors, with the local memory shared by the virtual processors (not to be confused with the virtual MIMD processors that we study in this thesis).

Given an additional layer of software, both MIMD and SIMD machines can run programs written for the other, with only a constant overhead per emulated instruction [4]. Whether MIMD or SIMD hardware is better depends on the application:

> "For well–structured problems with regular patterns of control, SIMD machines have the edge, because more hardware is devoted to operations on the data. This is because the SIMD machine, with only one instruction stream, can share most of its control hardware among all processors. In applications in which the control flow required of each processing element is complex and data dependent, MIMD architecture has the advantage. The shared instruction stream can follow only one branch of code at a time, so each possible branch must be executed in sequence, whereas the uninterested processor is idle. The result is that processors in a SIMD machine may sit idle much of the time [4, pg. 25]."

A common criticism of SIMD computers is that they are appropriate for only a small class of problems. This criticism is becoming less true as we gain more experience in developing data parallel algorithms [6, 1]. However, many applications require non–trivial amounts of data–dependent control flow, resulting in reduced parallel execution (due to idle processors). We refer to the statements that specify data–dependent control flow as **where** statements (as does C*, version 6.0).

In this thesis, we will show that the current loss of parallel execution in **where** statements is a feature of the Connection Machine language implementation (which we refer to as the *selection* implementation). We introduce an alternative implementation (called MIMD emulation), based on emulating a von Neumann processor on each SIMD processor, that can maintain a greater degree of parallel execution in many applications. The two implementation styles are not mutually exclusive: the standard implementation is faster when there are few different paths of control flow, and the MIMD emulation implementation is faster when there are many. We also describe a number of automatic source code transformations that can be performed to improve run–time performance. In addition, we describe the technique of MIMD emulation in detail, illustrated with empirical performance data from Connection Machine implementations, and we report some paradoxical results: (1) many techniques used to speed processors for real MIMD machines do *not* work for SIMD–emulated MIMD machines, and (2) global emulated throughput can be enhanced by selectively slowing down certain local computations.

1.1 Where Statements: The Selection Implementation

In current Connection Machine language implementations, **where** statements are a structured way for the programmer to turn processors on and off for particular instruction sequences. Consider the **where** statement S:

$$\S \; S\text{:} \quad \texttt{where } (\textit{Expr})\ S_1 \\ \qquad\quad \texttt{else } S_2$$

where *Expr* evaluates to a parallel Boolean value and S_1 and S_2 are arbitrarily complex statements, possibly including scalar (front end) code and nested **where** statements. (Note that the `else` clause is optional.) Informally, the C* (version 6.0) semantics for S are (1) to evaluate *Expr*, (2) execute S_1 on processors where *Expr* is true (non-zero), (3) execute S_2 on all other processors (that were selected when S began execution), and (4) select the processors that were selected when S began execution. Note that S_1 *always* executes before S_2, and that side effects from S_1 can affect the execution of S_2.

The selection implementation of the **where** statement S is straightforward and is the method used by all of the Connection Machine languages:

1. Evaluate *Expr* on the original selected set of processors.
2. Select the subset of the original selected set of processors that evaluated *Expr* to true (non-zero).
3. Execute S_1.
4. Select the subset of the original selected set of processors that evaluated *Expr* to false (zero).
5. Execute S_2.
6. Select the original selected set of processors.

The selection implementation of **where** statements executes every case—and every combination of nested cases—sequentially, selecting (turning on) the appropriate combination of processors for each case, so the time to execute the **where** statement S is always

$$\mathrm{T}_{\mathrm{SELECTION}}(S) = \mathrm{T}_{\mathrm{SELECTION}}(\textit{Expr}) + \mathrm{T}_{\mathrm{SELECTION}}(S_1) + \mathrm{T}_{\mathrm{SELECTION}}(S_2). \quad (1.1)$$

This implementation leads to reduced parallel execution, due to idle processors. For statement S, *Expr* evaluates to true on some fraction p of the selected processors, so while S_1 and S_2 are executing, only a fraction of the processors (p and $(1-p)$, respectively) are selected. If both branches take equal time, the fraction lost is 50%.

1.2 Where Statements: The MIMD Emulation Implementation

The combinatorial execution of **where** statements with the selection implementation (Equation 1.1) can have a major impact on the performance of applications, even those that are otherwise well suited for SIMD execution. This section briefly introduces the MIMD emulation implementation method for **where** statements, which can sometimes take better advantage of the available parallelism in such code.

Many Connection Machine applications require data dependent control flow, which is expressed in **where** statements. The **where** statement provides the illusion of the control parallelism of MIMD–style programming, but the performance properties of the selection implementation (Equation 1.1) ruin the illusion. The MIMD emulation implementation of **where** statements produces the effect of control parallelism by providing the expected performance properties.

Consider again the **where** statement S in Section 1.1. We redefine the informal semantics of S to simply be that statement S_1 is executed on the processors where *Expr* evaluates to true and statement S_2 is executed on the processors where *Expr* evaluates to false. The order of execution of S_1 and S_2 is not defined, and in fact they can be executed in either order or arbitrarily interleaved. This definition is superior to that used by the Connection Machine languages because (1) it is typical of control–parallel languages, and (2) it treats S_1 and S_2 symmetrically.

When discussing the emulation of a MIMD multiprocessor on the Connection Machine, we use the terms "virtual processor," "virtual instruction," "virtual memory," etc. to refer to the components of the emulated MIMD machine, and the terms "Connection Machine processor," "Connection Machine instruction," "Connection Machine memory," etc. to refer to components of the Connection Machine.

The MIMD emulation of the **where** statement S (Section 1.1) implementation consists of 3 steps:

1. At compile time, generate semantically equivalent virtual MIMD machine code for S. This code is loaded as data into Connection Machine memory at run time.

2. When S is encountered at run time, evaluate *Expr* in normal SIMD mode and then invoke an interpreter for the virtual MIMD machine. The interpreter emulates one processor of the virtual machine on each Connection Machine processor.

3. When all of the virtual threads of control reach the end of S, the interpreter exits and normal SIMD execution continues at the statement following S.

The MIMD emulation implementation executes every path of control flow in parallel, so the time to execute the **where** statement S is

$$\mathrm{T}_{\mathrm{EMULATE}}(S) = \mathrm{T}_{\mathrm{SIMD}}(Expr) + \mathrm{MAX}(\mathrm{T}_{\mathrm{EMULATE}}(S_1), \mathrm{T}_{\mathrm{EMULATE}}(S_2)). \qquad (1.2)$$

The overhead associated with the emulation of each virtual instruction is constant ([4] and Chapter 2), so $T_{EMULATE}(S)$ differs from $T_{MIMD}(S)$ (the time to execute S on real MIMD hardware corresponding to the virtual MIMD machine) by at most a multiplicative constant.

Chapter 2

MIMD Emulation

In this chapter, we describe how to emulate a MIMD multiprocessor on the Connection Machine, with at most a constant slowdown of execution per MIMD instruction (as compared to execution on MIMD hardware). The focus of this discussion is not only how to minimize the slowdown for a given virtual MIMD machine, but also the design of virtual MIMD machines that allow efficient emulation on SIMD hardware.

2.1 The Interpreter

In this study, a virtual MIMD computer consists of thousands of virtual von Neumann processors, each running its own sequential program, where each virtual processor executes asynchronously (at the virtual instruction level) with respect to the others. The virtual MIMD computer is implemented by an interpreter (a data parallel program) that emulates one virtual processor on each Connection Machine processor, executing virtual instructions on every virtual processor in parallel.

Figure 2.1 demonstrates the basic form of an interpreter for a virtual MIMD computer. This virtual machine uses 2–operand virtual instructions, and no virtual general registers. The virtual internal registers (e.g. the program counter, etc.) are represented as Connection Machine parallel variables, and the virtual memory of each virtual processor as a per–processor Connection Machine array. During initialization (omitted for clarity), the virtual program for each virtual process is loaded into the virtual memories, the virtual program counters are set to point to the first virtual instruction, the virtual stack pointer is initialized, etc. The basic interpreter cycle is

1. Fetch the next virtual instruction on each virtual processor.
2. Update the virtual program counters.
3. Decode the virtual instructions and perform virtual to Connection Machine address translation (in this example, address translation is a no–op).
4. Fetch the virtual operands.
5. Execute the virtual instructions.

```
shape [64 * 1024] procs;
int:procs pc;           /* program counter */
int:procs cc;           /* condition codes */
int:procs instr;        /* current instruction */
int:procs opcode;       /* opcode */
int:procs src, dst;     /* operand addresses */
int:procs s, d;         /* source and destination operands */
int:procs sp;           /* stack pointer */
int:procs mem[2048];    /* virtual memory */

/* the opcodes */
enum opcodes { add, subtract, multiply, divide, move /* etc. */ };

main()
{
    with (procs) {
      while (1) {
            instr = mem[pc];   /* fetch the next instruction */
            pc++;              /* update program counter */
            opcode = (0xF00000000 & instr) >> 28;   /* decode */
            src = (0x0FFFC000 & instr) >> 14;
            dst = 0x00003FFF & instr;
            s = mem[src]; d = mem[dst];     /* fetch operands */
            /* execute the current instruction */
            where (opcode == add) d += s;
            where (opcode == subtract) d -= s;
            where (opcode == multiply) d *= s;
            where (opcode == divide) d /= s;
            where (opcode == move) d = s;
            /* etc. */
            mem[dst] = d;      /* store result */
        }
    }
}
```

Figure 2.1: A partial interpreter for a virtual MIMD machine, written in C* (version 6.0). This virtual machine uses 2–operand virtual instructions, and no virtual general registers. The virtual internal registers are implemented as normal C* parallel variables, and memory of each virtual processor as a C* parallel array. During initialization, the virtual code is loaded into the virtual memory and the virtual pc is set to point to the first virtual instruction. The initialization and cleanup code has been omitted in this example. This example is written in C* for clarity, but in practice such interpreters are written in Paris, so that the virtual condition codes can be set correctly.

6. Store the virtual results.

A virtual processor can execute any virtual instruction in the virtual instruction set on any interpreter cycle. The execution of the virtual instructions (step 5 above) is implemented by a number of **where** statements, so each Connection Machine processor executes only the Connection Machine instructions that are needed to emulate the particular virtual instruction that it must execute. This example is written in C* for clarity, but in practice such interpreters must be written in Paris so that virtual condition codes can be set correctly (condition codes have been ignored in this example).

When we design a virtual MIMD computer, we must make decisions about the underlying method of virtual interprocessor communication and synchronization (not addressed in Figure 2.1). The three basic options are no communication, communication via message passing, and communication via shared memory.

2.1.1 Independent Virtual Processors

Some applications simply require many thousands of independent virtual machines. When it is sufficient to emulate independent virtual machines, no virtual communication or synchronization primitives are needed. The result is a fast interpreter with high virtual throughput, because the interpreter does not have to emulate as many virtual instructions, and it does not need to perform any (Connection Machine) interprocessor communication operations (which are relatively slow).

For example, this type of virtual machine can be used in developing a simulator for a new piece of hardware, along with a diagnostic test suite. The simulator is implemented as the interpreter, and a different diagnostic test can be run on each virtual copy of the (simulated) hardware. This allows many thousands of tests to be run in parallel, rather than one after another on a sequential computer.

Another application that requires no interprocessor communication is a set of sequential discrete event simulations. Typically, a large number of simulations must be performed, differing only in the initial conditions and input. Emulation of many independent sequential machines allows all of the different simulations to execute concurrently.

2.1.2 Loosely Coupled Virtual Processors

All communication and synchronization among the various processors in loosely coupled MIMD architectures occurs through messages. One of the simplest virtual message passing protocols uses a *rendezvous* between processes. The rendezvous consists of the synchronization of the two communicating processes, followed by the passing of a message. We will describe the implementation of rendezvous with one–way naming (i.e. sender specifies the destination).

The virtual send primitive is `send sdst src`, where `sdst` is the virtual processor address of the destination and `src` is the virtual memory address of the message in the virtual processor executing the virtual send. The virtual receive primitive is `receive`

rdst, where rdst is a virtual memory address. When a virtual message is received, it is placed in the virtual memory of the receiving virtual processor, and the virtual address of the message is placed in rdst.

Rendezvous communication requires that both the sending and receiving virtual processors be ready to communicate. This means that when a virtual processor executes a **send**, it will remain blocked until the receiver executes a **receive**, and the receiver will block until some virtual processor executes a **send** to it. It is possible that many virtual processors will simultaneously **send** messages to the same destination. When this occurs, the interpreter selects one of the contending virtual messages and attempts delivery, blocking all others (as if the destination were not executing **receive**). The interpreter uses three temporary variables to perform the synchronization and arbitration:

- **receiver-ready** indicates whether the virtual processor is executing a virtual **receive** instruction
- **message-received** indicates whether a message has been received by this virtual processor
- **arbitrator** is used to determine which sending virtual process may attempt to deliver its message

In addition, each virtual processor requires a unique processor identifier (PID). Since we have assumed the emulation of one virtual processor on each Connection Machine processor, we let the virtual PID be the Connection Machine PID.

During the phase of the interpreter cycle that actually emulates the virtual instructions, the following pseudo code is executed (where **procs[sdst][x]** means variable x on processor **sdst**):

```
receiver-ready = 0;
where (opcode == receive) {
    receiver-ready = 1;
    message-received = 0;
};
where (opcode == send) {
    where (procs[sdst][receiver-ready]) {
        /* sender and receiver ready */
        send-with-overwrite my-PID to procs[sdst][arbitrator];
        where (procs[sdst][arbitrator] == my-PID) {
            /* deliver message at procs[sdst][mem[ADDR]] */
            procs[sdst][mem[rdst]] = ADDR;
            procs[sdst][message-received] = 1;
        } else pc--;    /* lost in arbitration--retry */
    } else pc--;    /* receiver not ready--retry */
};
where (opcode == receive && !message-received)
    pc--;    /* no sender--retry */
```

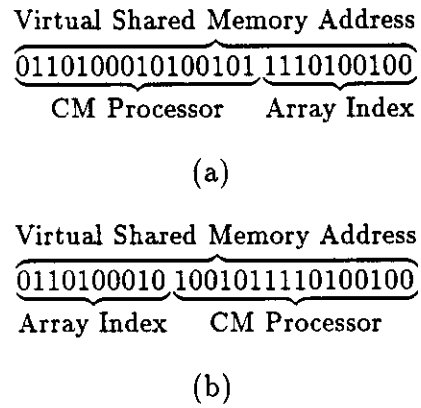

Figure 2.2: Translation of virtual MIMD shared memory addresses to Connection Machine addresses. Both (a) non–interleaved and (b) interleaved virtual memory layouts are shown. In either case, the address translation is trivial.

`Send-with-overwrite` is a Connection Machine communication primitive that ensures that exactly one of the (potentially) many contending values is delivered. In this interpreter, blocking is implemented using busy loops, so we do not have to directly emulate the interrupts of hardware MIMD machines.

On any interpreter cycle, if a virtual processor executes `receive` and one or more virtual processors execute a `send` to that virtual processor, a rendezvous will occur between one of the senders and the receiver on that cycle. Because the communication in the Connection Machine is deterministic, for a given set of virtual processors contending for a virtual destination, the same virtual process will always be given the opportunity to attempt message delivery. This means that under extreme conditions, starvation is possible (which is typical for rendezvous message passing with one–way naming).

2.1.3 Tightly Coupled Virtual Processors

In a tightly coupled architecture, all communication among the virtual processors occurs through virtual shared memory. To emulate a global shared address space the interpreter translates virtual shared memory addresses into Connection Machine memory addresses. The virtual shared memory is physically distributed across all Connection Machine processors.

The virtual shared memory in each Connection Machine processor is treated as an array (in terms of Connection Machine addressing) consisting of 2^{10} virtual machine (32–bit) words associated with each of the 2^{16} processors. The result is 2^{26} words of virtual shared memory and 26–bit virtual shared memory addresses. Figure 2.2 shows how the address translation is performed, for both non–interleaved and interleaved virtual memory layouts.

The simplest synchronization primitive for implementing mutual exclusion is a test–and–set–lock instruction in shared memory. When the lock tests clear, the pro-

cess may safely operate on the shared data that is protected by the lock. When the process completes the operation on the shared data, it clears the lock.

There is no built–in atomic test–and–set–lock instruction that operates on Connection Machine memory. Therefore, it is necessary to synthesize a test–and–set–lock instruction from the communication primitives of the Connection Machine. Again, the PARIS `send-with-overwrite` instruction is used, along with unique virtual processor numbers for arbitration among the processors contending for the lock.

A lock consists of two components: the 1–bit *state* of the lock and an *arbitrator*. The `test-and-set lock` virtual instruction, where `lock` refers to a virtual shared memory address, is emulated by the following pseudo code in the interpreter:

```
where (opcode == test-and-set) {
    send-with-overwrite my-PID to shared-mem[lock.arbitrator];
    where (shared-mem[lock.arbitrator] == my-PID) {
        /* test and set the lock */
        lock-condition-code = shared-mem[lock.state];
        shared-mem[lock.state] = 1;
    } else {
        /* lost arbitration--don't get to test lock
           (it is or will be set) */
        lock-condition-code = 1;
    }
}
```

If a lock is free, and at least one virtual processor attempts to set the lock, the lock will be set by exactly one of the contending virtual processors. Because communication in the Connection Machine is deterministic, for a given set of virtual processes contending for a lock, the same process will always be given the opportunity to perform the test–and–set. This means that under extreme conditions, starvation is possible (typical for shared memory multiprocessors).

2.2 Performance

In this section, we explore the design tradeoffs and implementation methods that affect the virtual throughput of the MIMD machine. To achieve high virtual throughput from an emulated MIMD machine, the most important aspects of the system are the regularity of virtual instruction encoding, the number of instruction types, and the details of the virtual instruction interpreter implementation.

2.2.1 Regularity in the Virtual Instruction Encoding

Regularity in the encoding of the virtual instructions minimizes interpreter overhead for virtual instruction decoding. If all of the virtual instructions are encoded in the same format, the interpreter can fetch and decode virtual instructions in all processors simultaneously. Otherwise, the different formats must be handled with

where statements. All of the virtual instructions should be the same length (so that a single fetch is sufficient) and the opcode and operands should be the same lengths and in the same positions in each of the different virtual instruction types.

2.2.2 Size of the Virtual Instruction Set

The execution time of the interpreter cycle grows linearly with the size of the virtual instruction set (for simplicity, we assume that each virtual instruction type takes approximately the same amount of time to emulate—all subsequent analyses make the more realistic assumption that instruction types may take *different* amounts of time to emulate). We measure the size of the virtual instruction set as the number of different virtual instruction types, where each virtual addressing mode and virtual instruction is a different instruction type. Figure 2.3 describes the throughput of virtual MIMD machines as a function of the size of the instruction set. Due to the linear slowdown of the interpreter with increasing virtual instruction set size, we achieve the best virtual throughput when we limit the virtual instruction set to a small number of simple (fast) instructions.

What criteria should be used for deciding which instruction types to include? We can model the effect of adding an additional instruction type to the virtual instruction set as follows. Let I be an instruction set, and let $I' = I \cup \{i\}$ be the same instruction set except for the addition of a new instruction type i. Let k be the number of virtual instructions needed to implement instruction i using only instructions in I. Let p be the run-time fraction of i in the program coded over I', let t be the interpreter cycle time over I, and let c be the the increase in interpreter cycle execution time due to the addition of i (cycle time over I' is $t + c$). Consider a virtual instruction stream S' of length n over I', and a semantically equivalent virtual instruction stream S over I. The mean length of S is

$$(1 - p)n + pkn.$$

The execution time for S' is

$$(t + c)n,$$

while the execution time for S is

$$((1 - p)n + pkn)t.$$

S' is executed faster than S whenever

$$(t + c)n < ((1 - p)n + pkn)t,$$

which simplifies to

$$\frac{c}{t} < p(k - 1). \tag{2.1}$$

Intuitively, the left hand side is the fraction the interpreter slows down due to virtual instruction type i and the right hand side is the fraction of the interpreter cycles that are no longer needed, due to using fewer virtual instructions to execute the same program ($|S'| < |S|$). When Inequality 2.1 holds, the addition of instruction i to the virtual instruction set will decrease the execution time of the program.

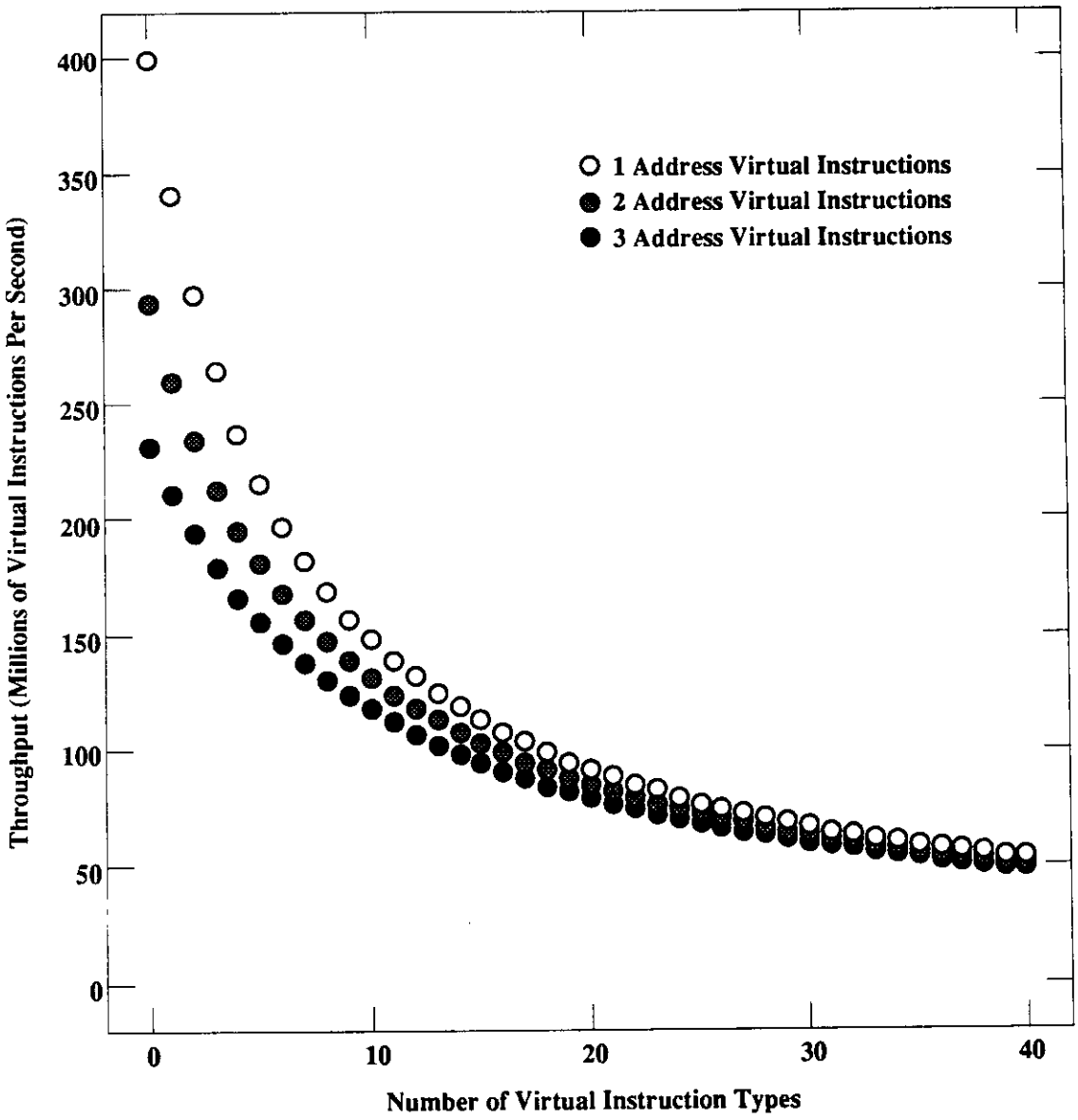

Figure 2.3: Virtual throughput as a function of the size of the virtual instruction set and number of non-immediate operands per virtual instruction. This is empirical data gathered on the Connection Machine, where every virtual instruction type takes as long to emulate as a 32-bit signed addition. The throughput values are scaled for a 64K processor Connection Machine.

2.2.3 Delayed Execution of Virtual Instructions

Virtual instruction types that are slow to emulate can sometimes be included in the virtual instruction set without causing performance degradation (as measured by Inequality 2.1). We can avoid performance degradation by not executing the slow virtual instruction type every time through the interpreter loop (Figure 2.4), reducing the impact of the slow virtual instruction type on the average interpreter cycle time (c in Inequality 2.1). In addition, on the cycles when the interpreter emulates the delayed virtual instruction type, greater utilization of this portion of the interpreter is achieved.

Which virtual instruction types should be delayed, and for how long? Assuming that the slow virtual instruction type appears in the virtual instruction stream at random with constant probability, and independently on all virtual processors, then the relevant factors are the probability of the appearance of the virtual instruction type in the virtual instruction stream, and the ratio of the time required to emulate the slow virtual instruction type to that of all the other virtual instruction types. The more often the virtual instruction type is used, the more often is should be executed by the interpreter, while the longer it takes to execute, the less often it should be executed.

We can model the effect of delaying the execution of a virtual instruction type as follows. Let I be the set of virtual instruction types that are not delayed, and let $I' = I \cup i$ be the whole virtual instruction set, where i is the delayed virtual instruction type. Let t be the execution time of the interpreter cycle over I, and c be the execution time of i (so the interpreter cycle time over I' is $t+c$). For this analysis, we model the execution of the interpreter as a sequence of virtual instruction *slots*. There are two types of slots: a *short* slot can execute any of I, and a *long* slot can execute only i. The sequence of slots is a repeating pattern of n short slots followed by exactly one long slot. The cost of a short slot is t and the cost of a long slot is c, so the period of the slot sequence is $w = nt + c$. Let q be the probability that the next virtual instruction type in the virtual instruction stream is i, and let $p = 1 - q$ be the probability that it is one of I. We will assume that $q > 0$, in order to simplify the calculations.

The expected number of the n short slots that will be filled per w time units is

$$\begin{aligned} E(\text{short slots used}) &= \sum_{x=0}^{n-1} x p^x q + n p^n \\ &= \frac{p - p^{n+1}}{q}, \end{aligned} \tag{2.2}$$

and the probability that a long instruction slot is utilized is

$$P(\text{long instruction slot utilized}) = 1 - p^{n+1}. \tag{2.3}$$

Combining Equations 2.2 and 2.3, the utilization (U) of the interpreter slots is

$$U = \frac{t\frac{p - p^{n+1}}{q} + c\left(1 - p^{n+1}\right)}{w}$$

```
shape [64 * 1024] procs;
int:procs pc;           /* program counter */
int:procs cc;           /* condition codes */
int:procs instr;        /* current instruction */
int:procs opcode;       /* opcode */
int:procs src, dst;     /* operand addresses */
int:procs s, d;         /* source and destination operands */
int:procs sp;           /* stack pointer */
int:procs mem[2048];    /* virtual memory */

/* the opcodes */
enum opcodes { add, subtract, multiply, divide, move,
               power /* etc. */ };

main()
{
    with (procs) {
      while (1) {
            instr = mem[pc];   /* fetch the next instruction */
            pc++;              /* update program counter */
            /* decode */
            /* fetch operands */
            /* execute the current instruction */
            where (opcode == add) d += s;
            where (opcode == subtract) d -= s;
            where (opcode == multiply) d *= s;
            where (opcode == divide) d /= s;
            where (opcode == move) d = s;
            where (opcode == power) {
                if (execute_now(power)) d = pow(d, s);
                else pc--;     /* ''block'' */
            }
            /* etc. */
            /* store result */
        }
    }
}
```

Figure 2.4: The interpreter of Figure 2.1 plus `power`, a virtual instruction type that is not emulated every interpreter cycle. The function `execute_now()` determines when the delayed virtual instruction type should be executed. During cycles when `power` is not to be executed, all virtual processors waiting to execute `power` become blocked (busy wait for `power` to execute). When `power` finally is executed, all waiting virtual processors execute in parallel.

$$= \frac{p - (1 + q\frac{c}{t})p^{n+1} + q\frac{c}{t}}{q\left(n + \frac{c}{t}\right)}. \quad (2.4)$$

We find the appropriate delay for a given virtual instruction type by maximizing the utilization U in Equation 2.4, based on the particular $\frac{c}{t}$ and q associated with the virtual instruction type. The delay value can be determined either statically or dynamically. A static delay makes assumptions about q, while a dynamic delay requires run–time estimation of q. Figures 2.5 and 2.6 plot the effects of the delayed execution of a long virtual instruction on U for various combinations of p and $\frac{c}{t}$. The delay of a slow and rarely used virtual instruction type increases throughput of the virtual MIMD computer *up to a point*, but further delay decreases throughput: throughput increases with delay until too many virtual processors become blocked between the long slots. The global increase in the system throughput as a result of delayed execution of virtual instructions is surprising, because this global *increase* in virtual throughput is the result of local *decreases* in virtual throughput.

2.2.4 Virtual Communication and Synchronization

The performance of an emulated MIMD machine is greatly influenced by the method of virtual interprocessor communication. On the Connection Machine, non–local memory accesses are significantly slower than local references. The difference is usually between a factor of 12 to 1000, depending on the reference pattern. The independent virtual processor interpreter never makes non–local memory references. The message passing virtual machine interpreter only accesses non–local memory when sending messages, which probably should be a delayed virtual operation (see Section 2.2.3). A shared memory virtual machine interpreter might access non–local memory several times each interpreter cycle.

For a shared memory virtual machine to obtain performance similar to that of a message passing virtual machine, certain restrictions must be observed. First, the virtual code, stack, and local data must be stored in local virtual memory. In addition, the number of virtual instructions types that access shared virtual memory must be small, since these are much slower to emulate than those that access local virtual memory. For instance, the only instruction types that access shared memory might be ***test–and–set–lock*** and only one or two addressing modes. All other operations should require only register and/or local memory accesses.

2.2.5 Other Implementation Tricks

The interpreter contains a large number of **where** statements (see Section 2.1), and thus contains many Connection Machine processor selection operations (see Section 1.1). A faster interpreter, and thus an increase in virtual throughput, could result if we were able to remove these selection operations. With no Connection Machine processor selection statements, the virtual results and updated virtual condition codes from all virtual instructions types are generated by each Connection Machine processor on every interpreter cycle, but only the desired virtual result/condition code

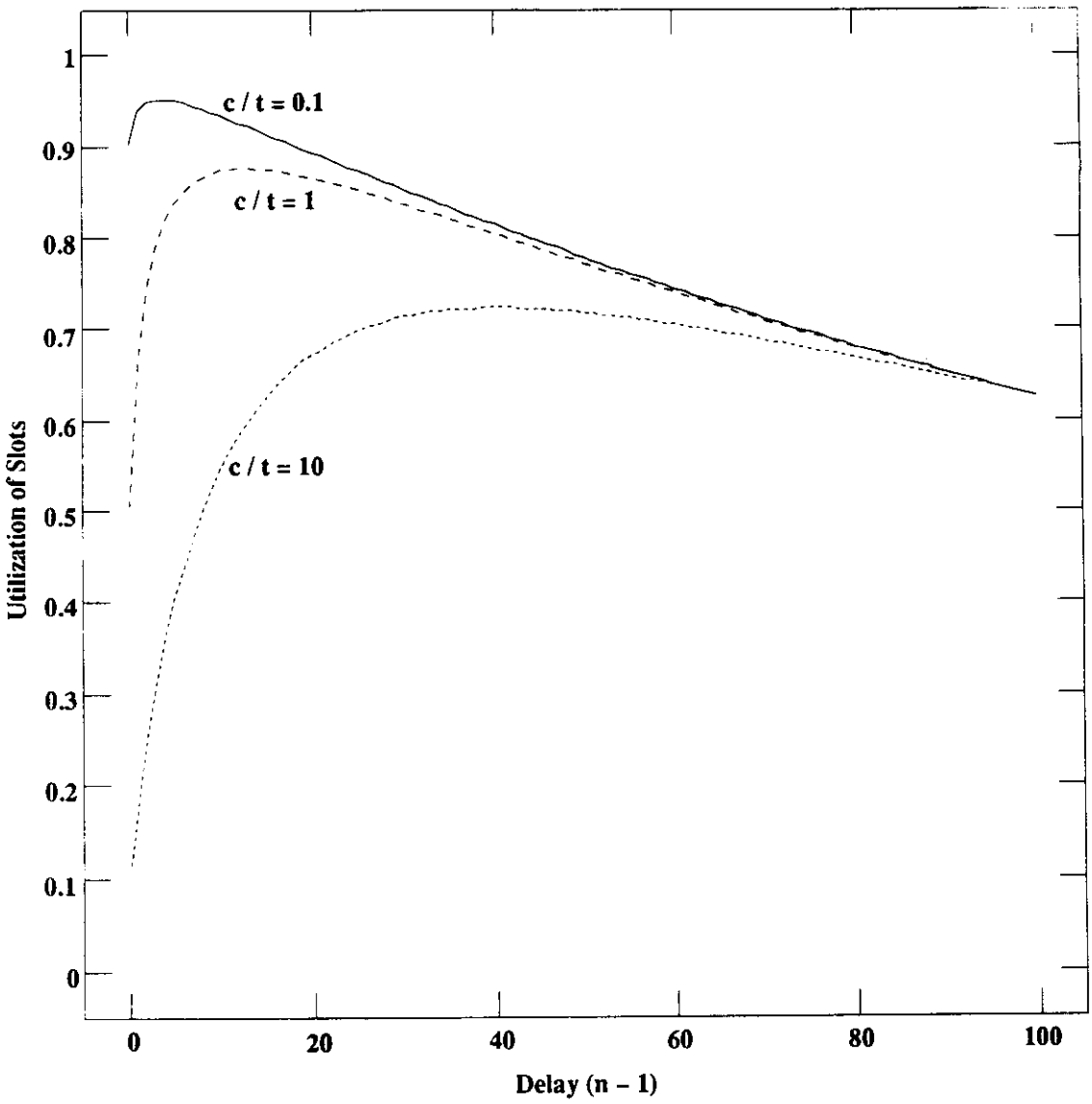

Figure 2.5: The utilization of the system as a function the delay for a long, rarely used instruction. This is a plot of Equation 2.4, with $q = 0.01$ ($p = 0.99$), for three values of $\frac{c}{t}$. In Figure 2.6, the same equation is plotted, with $\frac{c}{t}$ held constant and various values of p. Note that the center curve in both figures is the same.

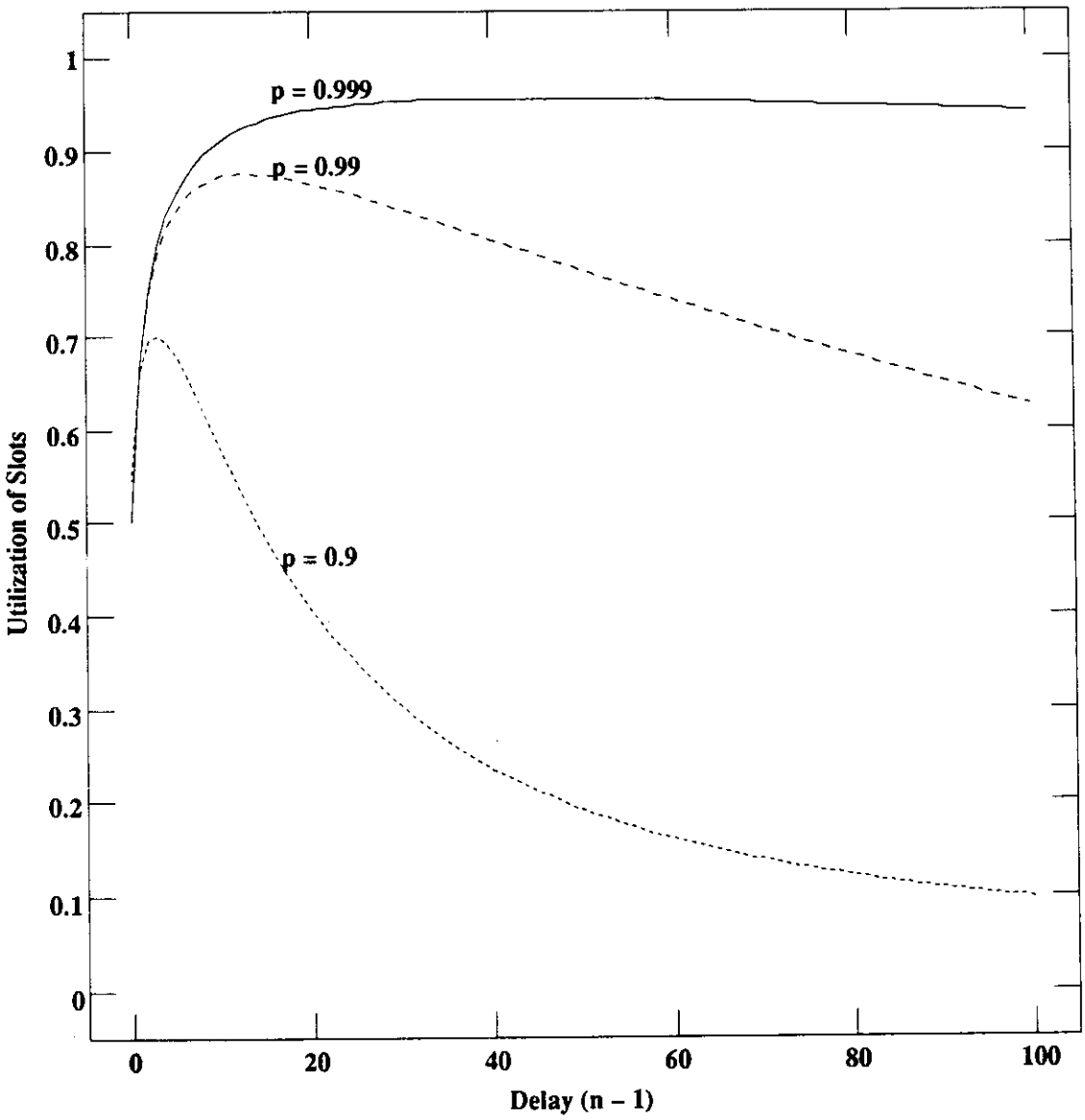

Figure 2.6: The utilization of the system as a function the delay for a long, rarely used instruction. This is a plot of Equation 2.4, with $\frac{c}{t} = 1$, for three values of p. In Figure 2.5, the same equation is plotted, with p held constant and various values of $\frac{c}{t}$. Note that the center curve in both figures is the same.

pair is retained. A virtual instruction set of size N requires N copies of the virtual processor state. That this could result in a speedup is quite surprising, because every Connection Machine processor will then emulate every virtual instruction in the virtual instruction set on every cycle—a dramatic increase in Connection Machine computation may result in increased global virtual throughput.

We performed experiments to test this observation, but we did not observe improved performance from this optimization, because the implementation requires additional overhead to allow fast indirect addressing into the multiple virtual states, negating the improvements due to the elimination of the Connection Machine processor selection operations. The removal of the processor selection steps will result in a speedup if and only if

$$\mathrm{T}(I) < \mathrm{T}(D) + \mathrm{T}(S), \tag{2.5}$$

where I is an indirect addressing operation, D is a direct addressing operation, and S is a processor selection operation.

2.3 Emulation of Additional Hardware

In this section, we consider the emulation of additional hardware techniques that are used to speed up processors for real MIMD machines, such as pipelines, caches, etc. It turns out that all of these techniques fail to increase the virtual throughput, unless the interpreter uses modified algorithms to emulate the hardware. Paradoxically, we again must slow down certain virtual processors, in order to get global speedup.

2.3.1 Pipelined Virtual Processors

In the simplest case, a pipelined processor consists of two stages. The first stage fetches and decodes the next instruction while the second stage executes the current instruction (in parallel). This type of pipeline need only be restarted when a conditional branch is taken (in typical code, every five to ten instructions). In the best case, the effective instruction time of the processor approaches $Max(F, E)$, where F is the time to fetch an instruction and E is the time to execute an instruction, compared to $F + E$ when pipelining is not used.

Surprisingly, although pipelining is a standard technique on real processors, in a virtual MIMD system emulated on the Connection Machine, standard pipelining cannot enhance throughput. The improvements that are achieved in the real processor are a result of two or more *different* operations occurring at the same time. In the interpreter, this type of parallelism cannot improve performance because the Connection Machine has only a single instruction stream. The cycle time of the emulated pipelined processor is at best $F + E$. Second, the pipeline typically will have to be restarted on ten to twenty percent of the virtual machine cycles. In the emulation, this means that the pipeline would be restarted on ten to twenty percent of the virtual processors on *each* cycle. The worst-case costs of a broken pipe (in this case, an additional instruction fetch/decode) would be paid on nearly *every* interpreter cycle.

2.3.2 Emulated Cache Memory

Another method that is often used to improve the throughput of a von Neumann processor is cache memory. Typically, about ninety–five percent of the memory references will hit the cache. Nevertheless, emulation of a standard cache cannot improve the performance of a virtual MIMD machine. On virtually every interpreter cycle, some virtual processors will experience cache misses, so all processors will have the effective memory access time of a cache miss.

This observation, along with the one above about the ineffectiveness of pipelining lead to a surprising conclusion. Hardware that employs probabilistic methods (e.g. a pipeline or cache) to improve performance on a von Neumann processor *almost never* results in speedup when emulated on the Connection Machine, unless the standard algorithms are modified. A speedup can only be realized if the resolution of the rare, but slow, worst case situation (e.g. a cache miss) is not performed immediately. For instance, if we emulate a virtual cache, virtual processors that experience a cache hit proceed, but the virtual processors that suffer a cache miss block for some number of interpreter cycles, until a time when all pending cache misses are handled. In this way, the cost of a cache miss is incurred on only a fraction of the interpreter cycles. This technique is the same as the delayed execution of slow, rarely used virtual instructions (Section 2.2.3).

Caches in a shared memory computer give rise to the additional problem of maintaining coherence. Although many solutions to the cache coherency problem have been published, all involve significant overhead and interprocessor communication. This overhead will almost certainly negate the payoff of an emulated cache for a virtual shared memory MIMD machine.

If the virtual instructions are read–only, then emulated instruction caches do not suffer from coherency problems. Consider an emulated system with the following parameters: cache memory access requires three time units, shared memory access requires an average of fifty time units, and the execution phase of the interpreter cycle requires eighty time units. The analysis is shown in Figure 2.7. When we satisfy virtual cache misses immediately, the interpreter cycle is slower than with no cache at all (since on every cycle, some processor out of the many thousands will experience a miss). The interpreter with an emulated cache that satisfies misses every sixteenth cycle is faster than the interpreter with no virtual cache. In order to determine the optimal frequency at which cache misses are serviced, use Equation 2.4, where the satisfaction of a cache miss is treated as a slow virtual instruction.

Virtual caches and other virtual hardware that rely on probabilistic methods can be used to improve the performance of a the virtual MIMD machine, but only if the interpreter uses modified algorithms. The virtual instruction caches result in a speedup only because we deliberately slow down virtual processors with cache misses. Again, global speedup is produced by local slowdown.

OPERATION	COST
fetch instruction	50
fetch operand_1	50
fetch operand_2	50
execute virtual instructions	80
store result	50
Total	280

(a)

OPERATION	COST
fetch instruction from cache	3
satisfy cache misses	50
fetch operand_1	50
fetch operand_2	50
execute virtual instructions	80
store result	50
Total	283

(b)

OPERATION	COST
fetch instruction from cache	3
satisfy cache misses	$50 \div 16$
fetch operand_1	50
fetch operand_2	50
execute virtual instructions	80
store result	50
Total	236.125

(c)

Figure 2.7: Emulation of virtual instruction caches. In this analysis, we assume that cache memory access requires three time units, shared memory access requires an average of fifty time units, and the execution phase of the interpreter cycle requires eighty time units. (a) No instruction cache. (b) Typical instruction cache, satisfying cache misses immediately. (c) Instruction cache that satisfies cache misses every sixteenth interpreter cycle.

Chapter 3

MIMD Emulation of Where Statements

The combinatorial execution of **where** statements with the selection implementation that is used by the Connection Machine languages (Equation 1.1) can have a major impact on the performance of SIMD applications, even those that are otherwise well suited for SIMD execution. We briefly introduced an alternative **where** implementation method based on MIMD emulation in Section 1.2, including improved **where** statement semantics that treat the two cases symmetrically, allowing the implementation to interleave them arbitrarily.

In this chapter, we describe how to incorporate the MIMD emulation of **where** statements into a compiler for a high level language. MIMD emulation is used to complement, rather than replace, the selection implementation of **where** statements. The circumstances where MIMD emulation becomes beneficial depend on the number of cases which must be handled, the code for each case, and the characteristics of the virtual MIMD machine that is emulated.

3.1 Applying MIMD Emulation to Where Statements

The technique we want to explore is the replacement of a **where** statement by a call to an interpreter for a special purpose virtual MIMD machine. The code for the **where** statement is compiled for the virtual MIMD machine (as a sequential conditional statement), and placed in the memory of the Connection Machine. Then, during execution of the program, all of the different cases of the **where** statement are interpreted in parallel. The selective use of MIMD emulation for **where** statements can be automated, and implemented in a compiler for a SIMD language.

Consider an N-way **where** statement (Figure 3.1) with fifty branches in which each branch calculates a different product and assigns it to a different destination. Using standard SIMD execution, this statement requires time for fifty multiplications and assignments, plus processor selection overhead. If this statement is compiled into virtual MIMD code and interpreted, a single multiplication and assignment, plus

```
S:  where  (Expr) {
               case Const_0: S_0
               case Const_1: S_1
               case Const_2: S_2
               :
               case Const_{N-1}: S_{N-1}
           }
```

Figure 3.1: An N–way **where** statement S. Statement S_i is executed where $Expr = Const_i$.

the overhead of a virtual instruction fetch, virtual instruction decode, and virtual operand fetch and store suffices. When using the selection implementation, the critical path for a **where** statement includes all of the code contained in the statement, but MIMD emulation reduces the critical path to the longest branch used by some virtual processor (see Equations 1.1 and 1.2). The MIMD emulation clearly results in speedup in this case, but note that the selection implementation might win if the **where** statement had only one or two branches.

3.2 The SIMD Compiler and Virtual MIMD Targets

The construction of a compiler for a SIMD language L that uses MIMD emulation to optimize the execution time of **where** statements is complex but straightforward. The compiler is a combination of (1) a compiler for L for the SIMD machine, (2) a policy for deciding when to use MIMD emulation, (3) an algorithm for designing an appropriate virtual MIMD machine M, (4) a MIMD interpreter for M at run time, and (5) a compiler for L targeted to virtual von Neumann machine M.

3.2.1 A Single Virtual MIMD Target

The simplest compiler that uses MIMD emulation for **where** statements would use a predefined virtual MIMD instruction set. Since the virtual instruction set is statically defined, it will not be well suited for all programs. For instance, if a **where** statement never needs to do virtual multiplications, a multiply virtual instruction is extraneous. On the other hand, if no virtual multiply instruction is provided, multiplication-intensive statements will take much longer to execute.

3.2.2 Compiler–Designed Virtual MIMD Targets

It is clear that no particular virtual MIMD machine will be well suited for all **where** statements. A better method is for the compiler to design a special purpose virtual instruction set and interpreter for each **where** statement that is to be translated to virtual MIMD code; only the virtual instructions that are actually present in the

```
(cycle 0)  add c, b
(cycle 1)  delay
(cycle 2)  delay
(cycle 3)  mul a, c
(cycle 4)  add d, b
(cycle 5)  add d, c
(cycle 6)  add a, d
```

Figure 3.2: A sample of intermediate code with delayed execution of `mul`. The instructions are in the form `op destination, source`.

```
(cycle 0)  add c, b
(cycle 1)  add d, b
(cycle 2)  add d, c
(cycle 3)  mul a, c
(cycle 4)  add a, d
```

Figure 3.3: The reordered intermediate code, optimizing for delayed execution. The instructions are in the form `op destination, source`.

virtual code will be emulated by the interpreter. It is feasible to have hundreds, or even thousands of virtual instructions from which the compiler may pick and choose a small subset. The compiler will decide which virtual instructions to use based on the relative costs and benefits of adding additional instructions to the virtual instruction set (Section 2.2.2), and will generate an interpreter to implement exactly the virtual instructions that it has generated. The compiler-designed virtual machine will generally perform better than a statically-designed virtual machine for any particular **where** statement.

3.2.3 Additional Optimizations

We can achieve faster MIMD emulation by adding more complexity to the compiler. The compiler may be able to reduce the average interpreter cycle time by creating an interpreter that delays execution for certain virtual instruction types (Section 2.2.3). The compiler will deduce appropriate delays, based on the heuristically estimated dynamic execution frequency and expected emulation time of the various virtual instruction types.

When the compiler knows the delay for each virtual instruction type, it is possible for instructions to be reordered to fill "delay" cycles. For example, consider the virtual intermediate code in Figure 3.2. If the calculated delays are such that the *add* virtual instruction is not delayed, and the *mul* virtual instruction is delayed up to two cycles (i.e. will be executed only on cycles zero, three, six, etc.), then a virtual processor executing this section of code will be blocked from cycle one to cycle three, waiting for *mul* to be executed. Since there are no data dependencies between the cycle 3 virtual instruction and the cycle 4 and cycle 5 virtual instructions, the sequence can

```
int a, b, c;
float x, y, z;
where (Expr) {
     case 0: a *= b; /* int */
     case 1: c *= a; /* int */
     :
     case K: b *= c; /* int */
     case K + 1: x *= y; /* float */
     case K + 2: z *= x; /* float */
     :
     case N - 1: y *= z; /* float */
}
```

Figure 3.4: A **where** statement with two different classes of operations. Cases 0 through K perform integer multiplication, while cases $K+1$ through $N-1$ perform floating point multiplication.

be reordered to yield the sequence in Figure 3.3, saving two interpreter cycles.

The compiler is likely to encounter **where** statements for which reduction to two or more simpler statements results in better performance. For instance, half of the branches of the **where** statement might require only floating point arithmetic, while the other half require only integer arithmetic (Figure 3.4). This statement can be emulated faster if it is broken into a statement that performs only floating point arithmetic and a statement that performs only integer arithmetic (Figure 3.5). The two **where** statements are emulated sequentially, each with a different, faster interpreter.

Another example is a **where** statement in which many of the branches perform one class of operations, followed by a very different class of operations. For instance, each branch might update some floating point variables, then update some integer variables (Figure 3.6). The compiler would break this **where** statement into two statements. Each of the two statements would have the same number of branches as the original statement, but only half of the code (Figure 3.7). Again, the two **where** statements are emulated sequentially, each by a faster interpreter.

Another example is a **where** statement that contains one or two branches that are very different from all the other branches (Figure 3.10). The differences might be in terms of the required operations or in the number of virtual instructions that would need to be executed. The compiler can generate a selection **where** for the "odd" branch, and use MIMD emulation for the rest of the **where** statement, resulting in better overall performance (Figure 3.11).

```
int a, b, c;
float x, y, z;
tmp = Expr;
where (tmp) { /* integer where statement */
     case 0: a *= b; /* int */
     case 1: c *= a; /* int */
     :
     case K: b *= c; /* int */
}
where (tmp) { /* floating point where statement */
     case K + 1: x *= y; /* float */
     case K + 2: z *= x; /* float */
     :
     case N - 1: y *= z; /* float */
}
```

Figure 3.5: A **where** statement with two different classes of operations (Figure 3.4), optimized into two statements. One statement contains only integer multiplications, while the other contains only floating point multiplications.

```
int a, b, c;
float x, y, z;
where (Expr) {
     case 0:
          x *= y; /* float */
          a *= b; /* int */
     case 1:
          z *= y; /* float */
          c *= a; /* int */
     :
     case N - 1:
          y *= z; /* float */
          b *= c; /* int */
}
```

Figure 3.6: A **where** statement in which each case has two different classes of operations, a floating point multiplication followed by an integer multiplication.

```
int a, b, c;
float x, y, z;
tmp = Expr;
where (tmp) { /* floating point */
     case 0: x *= y; /* float */
     case 1: z *= y; /* float */
     :
     case N - 1: y *= z; /* float */
}
where (tmp) { /* integer */
     case 0: a *= b; /* int */
     case 1: c *= a; /* int */
     :
     case N - 1: b *= c; /* int */
}
```

Figure 3.7: A **where** statement in which each case has two different classes of operations (Figure 3.6), optimized into two **where** statements. The first handles the floating point multiplications, while the second handles the integer multiplications.

3.3 Speedup Using MIMD Emulation of Where Statements

We have gathered empirical performance data for a variety of **where** statements, comparing MIMD emulation to C* The intent is to provide some understanding of when MIMD emulation can result peedup of a SIMD **where** statement.

I performed all of the tests on -sun.think.com, a Sun 4/280 front end, running SunOS 4.0 and a Connection Ma 2 (8192 processing elements), running version 5.0.1 of the Connection Machine software (courtesy of the CMNS Pilot Facility, supported under terms of DARPA contract DACA76-88-C-0012). In each case, a C* version is compared to a MIMD emulation version. The C* versions were compiled with the C* translator version 5.0.21 and the Sun C compiler. The MIMD emulation versions consist of a hand-optimized virtual instruction sets and hand-optimized interpreters written in C/Paris and compiled with the Sun C compiler. All compilations were performed with the '-O' switch, to turn on optimizations. All tests were run with one virtual MIMD processor per physical Connection Machine processor. The test results are described below and summarized in Table 3.1.

In the first example, each branch of the **where** statement computes one product (Figure 3.8). The execution time of the MIMD emulation is constant, no matter how many branches in the statement, while the C* execution time increases linearly. The MIMD emulation runs faster than the corresponding C* version when there is more than one branch of control in the **where** statement. This example is a very good candidate for MIMD emulation, because all of the arms of the **where** are the same

```
where (Expr) {
        case 0: a *= b;
        case 1: c *= a;
        case 2: b *= c;
        :
        case N - 1: d *= k;
}
```

Figure 3.8: A **where** statement with one multiplication per case.

Example	Number of Branches Before MIMD Emulation is Faster Than C*
Example 1	2
Example 2	6
Example 3 (naive)	37
Example 3 (optimized)	6

Table 3.1: An empirical comparison of selection (C*) and MIMD emulation implementations of **where** statements. For each example (described in the text), the crossover point where MIMD emulation results in faster execution is shown. Despite the empirical nature of these experiments, the curves were very linear, so only the intersections are reported.

length and perform only one type of operation (multiplication).

The second example contains a greater variety of instructions. Each branch of the **where** contains five statements using five different instructions (Figure 3.9). This is still a good case for MIMD emulation. MIMD emulation results in faster execution when the **where** is more than six branches wide.

The third example demonstrates a **where** statement for which naive use of MIMD emulation does not yield good performance. The **where** statement is exactly the same as in the previous example, except for the first branch, which consists of twenty–five statements rather than five (Figure 3.10). MIMD emulation only begins to result in faster execution when this statement is more than thirty-seven branches wide.

The reason for the dramatic difference between the second and third examples is the fact the branches in the third example contain very different amounts of code. This is a case where the compiler should split the single **where** statement into two (Figure 3.11). The first statement consists of all but five of the operations (the length of the other arms of the **where**) from the long arm of the original statement. The second **where** statement consists of the remaining five instructions from the long branch, along with the rest of the cases. The first **where** is executed in the usual SIMD manner, while the second uses MIMD emulation. When these optimizations

```
where (Expr) {
     case 0: a *= b; b += a; c -= d; a &= 4; b /= 3;
     case 1: c *= a; c += 3; c /= d; c -= z; a &= 32;
     case 2: b *= c; c /= 54; b &= c; b += b; k -= 1;
     :
     case N - 1: d /= k; z &= 8; k *= 7; d += k; a -= 2;
}
```

Figure 3.9: A **where** statement with five statements per case. The five operations include addition, subtraction, multiplication, division, and logical and.

```
where Expr
     case 0: a *= b; b += a; c -= d; a &= 4; b /= 3;
          a *= b; b += a; c -= d; a &= 4; b /= 3;
          a *= b; b += a; c -= d; a &= 4; b /= 3;
          a *= b; b += a; c -= d; a &= 4; b /= 3;
          a *= b; b += a; c -= d; a &= 4; b /= 3;
     case 1: c *= a; c += 3; c /= d; c -= z; a &= 32;
     case 2: b *= c; c /= 54; b &= c; b += b; k -= 1;
     :
     case N - 1: d /= k; z &= 8; k *= 7; d += k; a -= 2;
}
```

Figure 3.10: A **where** statement. The first branch performs twenty-five operations and each other branch performs only five operations chosen from addition, subtraction, multiplication, division, and logical and.

are made, MIMD emulation is faster when the **where** is more than six branches wide (the same as the second example).

An additional optimization that might be used in such a case is reordering of the virtual instructions in the long case in order to put the most expensive instructions inside the **where** statement that will be executed with MIMD emulation. The expensive operations will be performed by the interpreter anyway, and this will reduce the execution time of the first statement.

Bibliography

[1] Guy Blelloch. Scans as primitive parallel operations. Technical report, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 1986.

[2] Robert J. Collins. CM++: A C++ interface to the Connection Machine. In *Proceedings of the Symposium on Object Oriented Programming Emphasizing Practical Applications*, September 1990.

[3] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley Publishing Company, 1990.

[4] W. Daniel Hillis. *The Connection Machine.* The MIT Press, Cambridge, Massachusetts, 1985.

[5] W. Daniel Hillis and Joshua Barnes. Programming a highly parallel computer. *Nature*, 326(6108):27–30, 1987.

[6] W. Daniel Hillis and Guy L. Steele Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.

[7] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice-hall, Englewood Cliffs, New Jersey, 1978.

[8] John R. Rose and Guy L. Steele Jr. C*: An extended C language for data parallel programming. Technical Report PL-87.5, Thinking Machines Corporation, Cambridge, Massachusetts, March 1987.

[9] Guy L. Steele Jr. *Common LISP: The Language.* Digital Press, 1984.

[10] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley Publishing Company, 1986.

[11] Thinking Machines Corporation, Cambridge, Massachusetts. *Connection Machine Parallel Instruction Set (Paris): The C Interface, Version 4.0*, 1987.

[12] Thinking Machines Corporation, Cambridge, Massachusetts. *C* Programming Guide, Version 6.0 (preliminary, pre-beta release)*, April 1990.

[13] Introduction to data level parallelism. Technical Report 86.14, Thinking Machines Corporation, Cambridge, Massachusetts, April 1986.

[14] Connection Machine model CM-2 technical summary. Technical Report HA87-4, Thinking Machines Corporation, Cambridge, Massachusetts, April 1987.

Acknowledgments

I would like to thank my committee members, Stott Parker, Rajive Bagrodia, and especially my committee chair David Jefferson. David's limitless patience, advice, and enthusiasm have been very important, as have the many hours of stimulating discussions. I also would like to thank all of my friends, in particular Valerie Aylett, for providing endless emotional support.

JP Massar was very helpful when it came to Connection Machine performance, timing issues, and problems. In addition, he suggested the the application of diagnostics for simulated hardware on independent virtual MIMD processors.

I was supported financially in part by Jet Propulsion Laboratory contract number 957523 and W. M. Keck Foundation award number W880615. This funding was greatly appreciated. The bulk of this research occurred on the UCLA Connection Machine, which is supported in part by the National Science Foundation Biological Facilities grant number BBS 87 14206. The performance data was gathered on the Connection Machine Network Server (CMNS) Pilot Facility, supported under terms of DARPA contract DACA76-88-C-0012, and a CM-2 computer at the Advanced Computing Laboratory of Los Alamos National Laboratory under the auspices of the U.S. Department of Energy, contract W-7405-ENG-36.

the effective size of the virtual instruction set must be kept as small as possible. The effective size of the virtual instruction set can be smaller than the actual size, if one or more rarely used virtual instruction types are not emulated during every interpreter cycle. Paradoxically, this optimization results in global speedup by occasionally causing those virtual processors that need to execute a delayed instruction to block for one or more interpreter cycles, and thus reducing the number of Connection Machine processors that are executing.

Another surprising result is that the emulation of hardware that employs probabilistic methods (e.g. a pipeline or cache memory) that improve performance of real MIMD processing elements *almost never* result in speedup of the virtual MIMD machine, unless the standard algorithms are modified. A speedup can only be realized if the resolution of the rare, but slow, worst case situation (e.g. a cache miss) is not performed immediately. Again, global speedup is achieve by stopping the computation of some of the Connection Machine processors for one or more interpreter cycles.

The use of MIMD emulation to implement **where** statements can be automated and incorporated into compilers for SIMD languages. The compiler can design and optimize both the virtual MIMD machine, and the interpreter for each particular **where** statement. In addition, a number of optimizations can be performed to the virtual MIMD code (the translated **where** statement) to dramatically improve the run time performance of the **where** statement. These optimized interpreters will often attain emulated throughput in the range of 200 to 350 million virtual instructions per second (on 64K Connection Machine processors), delivering about 20 percent of the available Connection Machine cycles to the virtual MIMD machine (typical Connection Machine applications achieve on the order of 1000 MIPS [14]). This performance could almost certainly be significantly increased by providing microcode or perhaps even hardware support for MIMD emulation.

Chapter 4

Conclusions

A common criticism of SIMD computers such as the Connection Machine is that the data parallel programming style is only appropriate for a small class of problems. The parallelism in data parallel algorithms comes from the simultaneous application of the same operation to each datum in a large set of data, rather than from multiple threads of control (control parallelism). Many data parallel programs require slightly different sequences of operations on different data items, and thus contain **where** statements, describing one or more statement sequences and where in the data set each is to be executed (control parallelism).

On the Connection Machine, **where** statements are usually implemented by issuing all of the embedded statements, turning on (selecting) the appropriate processors before executing each sequence. Given this usual implementation, the execution time for a **where** statement is the sum of the execution times for each of the **where**'s embedded statement sequences. Such selection implementations result in a significant decrease in parallel execution.

This lack of parallel execution is a feature of the selection implementation of the **where** statement in the Connection Machine languages, not an inherent feature of the **where** statement. In this thesis, we have described an implementation based on MIMD emulation that executes a **where** statement in time proportional to its longest alternative statement sequence (control parallel critical path). In this implementation, we use an interpreter to emulate a massively parallel MIMD computer on the Connection Machine, allowing the execution of control–parallel code with only a constant slowdown per emulated instruction. The interpreter maintains the state of an emulated von Neumann processor in each Connection Machine processing element, and treats the other parallel data on the Connection Machine as virtual MIMD machine instructions and data. MIMD emulation empirically results in speedups even for **where** statements with relatively few (less than 10) branches of control.

The heart of the interpreter is a series of **where** statements that emulates each of the virtual instruction types on the appropriate Connection Machine processors. The combinatorial nature of the selection **where** implementation has a major influence on the design of the virtual MIMD instruction set and the implementation of the interpreter. In order to minimize the virtual instruction fetch/decode overhead, all virtual instruction types should be encoded in exactly the same format. In addition,

```
tmp = Expr;
where (tmp == 0) { /* normal SIMD execution */
     a *= b; b += a; c -= d; a &= 4; b /= 3;
     a *= b; b += a; c -= d; a &= 4; b /= 3;
     a *= b; b += a; c -= d; a &= 4; b /= 3;
     a *= b; b += a; c -= d; a &= 4; b /= 3;
}
where (tmp) { /* emulated MIMD execution */
     case 0: a *= b; b += a; c -= d; a &= 4; b /= 3;
     case 1: c *= a; c += 3; c /= d; c -= z; a &= 32;
     case 2: b *= c; c /= 54; b &= c; b += b; k -= 1;
     :
     case N - 1: d /= k; z &= 8; k *= 7; d += k; a -= 2;
}
```

Figure 3.11: An optimized version of Figure 3.10. Twenty of the twenty-five operations of the first case have been moved out of the **where** statement. Only the second **where** statement will be transformed for MIMD emulation.