

1-1-1992

# A Massively Parallel MIMD Implemented by SIMD Hardware?

H. G. Dietz

*Purdue University School of Electrical Engineering*

W E. Cohen

*Purdue University School of Electrical Engineering*

Follow this and additional works at: <http://docs.lib.purdue.edu/ecetr>

---

Dietz, H. G. and Cohen, W E., "A Massively Parallel MIMD Implemented by SIMD Hardware?" (1992). *ECE Technical Reports*. Paper 280.

<http://docs.lib.purdue.edu/ecetr/280>

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact [epubs@purdue.edu](mailto:epubs@purdue.edu) for additional information.



# **A Massively Parallel MIMD Implemented by SIMD Hardware?**

H. G. Dietz  
W. E. Cohen

TR-EE 92-4  
January 1992

---

<sup>†</sup> This work was supported by the Office of Naval Research (ONR) under grant number N00014-91-J-4013 and by the National Science Foundation (NSF) under award number 9015696-CDA.

## Table of Contents

|   |    |
|---|----|
| 1. Introduction .....                       | 2  |
| 1.1. Interpretation Overhead .....          | 2  |
| 1.2. Indirection .....                      | 3  |
| 1.3. Enable Masking .....                   | 3  |
| 1.4. Our Approach .....                     | 4  |
| 2. Instruction Set Design .....             | 5  |
| 2.1. Memory Reference Model .....           | 6  |
| 2.1.1. Local Memory Model .....             | 6  |
| 2.1.2. Global Memory Model .....            | 6  |
| 2.2. Assembly Language Model .....          | 7  |
| 2.3. Prototype Instruction Set .....        | 8  |
| 3. Emulator Design .....                    | 10 |
| 3.1. Shortening The Basic Cycle .....       | 10 |
| 3.2. Minimizing Operation Time .....        | 10 |
| 3.2.1. Maximizing Instruction Overlap ..... | 11 |
| 3.2.2. Reducing Operation Count .....       | 11 |
| 3.2.2.1. Subemulators .....                 | 12 |
| 3.2.2.2. Frequency Biasing .....            | 15 |
| 4. Performance Evaluation .....             | 15 |
| 4.1. High-Level Language Peak MFLOPS .....  | 16 |
| 4.2. Emulation Overhead .....               | 17 |
| 4.3. A Many-Thread Example .....            | 18 |
| 4.3.1. The Program .....                    | 18 |
| 4.3.2. Performance .....                    | 21 |
| 5. Room for Improvement .....               | 23 |
| 5.1. Compiler (mimdc) .....                 | 23 |
| 5.2. Assembler (mimda) .....                | 23 |
| 5.3. Emulator (mimd) .....                  | 23 |
| 6. Conclusions .....                        | 24 |

# A Massively Parallel MIMD Implemented By SIMD Hardware?

*H. G. Dietz and W. E. Cohen*

Parallel Processing Laboratory  
School of Electrical Engineering  
Purdue University  
West Lafayette, IN 47906  
`hankd@ecn.purdue.edu`

## Abstract

Both conventional wisdom and engineering practice hold that a massively parallel MIMD machine should be constructed using a large number of independent processors and an asynchronous interconnection network. In this paper, we suggest that it may be beneficial to implement a massively parallel MIMD using microcode on a massively parallel SIMD microengine; the synchronous nature of the system allows much higher performance to be obtained with simpler hardware. The primary disadvantage is simply that the SIMD microengine must serialize execution of different types of instructions — but again the static nature of the machine allows various optimizations that can minimize this detrimental effect.

In addition to presenting the theory behind construction of efficient MIMD machines using SIMD microengines, this paper discusses how the techniques were applied to create a **16,384**-processor shared memory barrier MIMD using a SIMD **MasPar MP-1**. Both the MIMD structure and benchmark results are presented. Even though the **MasPar** hardware is not ideal for implementing a MIMD and our microinterpreter was written in a high-level language (**MPL**), peak MIMD **performance** was **280 MFLOPS** as compared to 1.2 GFLOPS for the native SIMD instruction set. Of course, comparing peak speeds is of dubious value; hence, we have also included a number of more realistic benchmark results.

**Keywords:** MIMD, SIMD, Microcode, Compilers, Common Subexpression Induction.

<sup>†</sup> This work was supported in part by the Office of Naval Research (ONR) under grant number **N00014-91-J-4013** and by the National Science Foundation (NSF) under award number 9015696-CDA.

## 1. Introduction

Before discussing how a highly efficient MIMD machine can be built using a SIMD microengine, it is useful to review the basic issues in interpreting MIMD instructions using a SIMD machine. In the simplest **terms**, the way in which one interprets a MIMD instruction set using SIMD hardware is to write a SIMD program that interpretively executes a MIMD instruction set. There is nothing particularly difficult about doing this; in fact, one could take a completely arbitrary MIMD instruction set and execute it on a SIMD machine.

For example, [WiH91] reported on a simple MIMD interpreter running on a **MasPar** MP-1 [Bla90]. Wilsey, et. al, implemented an interpreter for the MINTABS instruction set and indicated that work was in progress on a similar interpreter for the MIPS R2000 instruction set. The MINTABS instruction set is very small (only 8 instructions) and is far from complete in that there is no provision for communication between processors, but it does provide basic MIMD execution. In fairness to [WiH91], their MIMD interpreter was built specifically for parallel execution of mutant versions of serial programs — no communication is needed for that application.

Such an interpreter has a data structure, replicated in each SIMD PE, that corresponds to the internal registers of each MIMD processor. Hence, the interpreter structure can be as simple as:

### Basic MIMD Interpreter Algorithm

1. Each PE fetches an "instruction" into its "instruction register" (IR) and updates its "program counter" (PC).
2. Each PE decodes the "instruction" from its IR.
3. Repeat steps 3a-3c for each "instruction" type:
  - a) Disable **all** PEs where the IR holds an "instruction" of a different type.
  - b) Simulate execution of the "instruction" on the enabled PEs.
  - c) Enable all PEs.
4. Go to step 1.

The only difficulty in implementing an interpreter with the above structure is that the simulated machine will be very inefficient. There are several reasons for this inefficiency.

### 1.1. Interpretation Overhead

The most obvious problem is simply that interpretation implies some overhead for the interpreter, even MIMD hardware simulating a MIMD with a different instruction set would suffer this overhead. In addition, SIMD hardware can only simulate execution of one instruction type at a time, hence, the time to execute a simulated instruction is proportional to the sum of the execution times for each instruction type.

## 1.2. Indirection

Still more insidious is the fact that even step 1 of the above algorithm cannot be executed in parallel across all **PEs** in many SIMD computers. The next instruction for each **PE** could be at any location in the **PE's** local memory, and many SIMD machines do not allow multiple **PEs** to access different memory locations simultaneously. Hence, on such a SIMD machine, any parallel memory access made will take time proportional to the number of different **PE** addresses being fetched from<sup>1</sup>. For example, this is the case on the TMC CM-1 [Hil87] and TMC CM-2 [Thi90]. Note that step 3b suffers the same difficulty if load or store operations must be performed.

Since many operations are limited by (local) memory access speed, inefficient handling of these memory operations can easily make **MIMD** interpretation on a SIMD machine infeasible.

This overhead can be averted only if the SIMD hardware can indirectly access memory using an address in a **PE** register. Examples of SIMD machines with such hardware include the PASM Prototype [SiN90] and the MasPar MP-1 [Bla90].

## 1.3. Enable Masking

It is also important to note that the above algorithm assumes that it is possible for **PEs** to enable and disable themselves (set their own masks). Although most SIMD computers have some ability to disable **PEs**, in many machines it is either difficult to have the **PEs** disable themselves (as opposed to having the **control** unit disable **PEs**, as in the PASM prototype [SiN90]) or some arithmetic instructions cannot be disabled because they occur in a coprocessor, as in the TMC CM-2 [Thi90]. In such cases, masking can be circumvented by the use of **bitwise** logical operations, e.g. a C-like SIMD **wde** segment:

```
where (ir == CMP) {
    /* executed only by PEs in which
       ir has the value CMP; cc is
       not accessed by other PEs
    */
    cc = alu - mbr;
}
```

might be implemented by all **PEs** simultaneously executing the C code:

```
/* use C's bitwise logical operations so
   that cc = alu - mbr in those PEs where
   ir == CMP, and cc = cc in the others
*/
mask = ~(ir == CMP);
cc = ((cc & ~mask) | ((alu - mbr) & mask));
```

---

<sup>1</sup> Worse still, for some SIMD machines the technique used takes time proportional to the size of the address space which could be accessed.

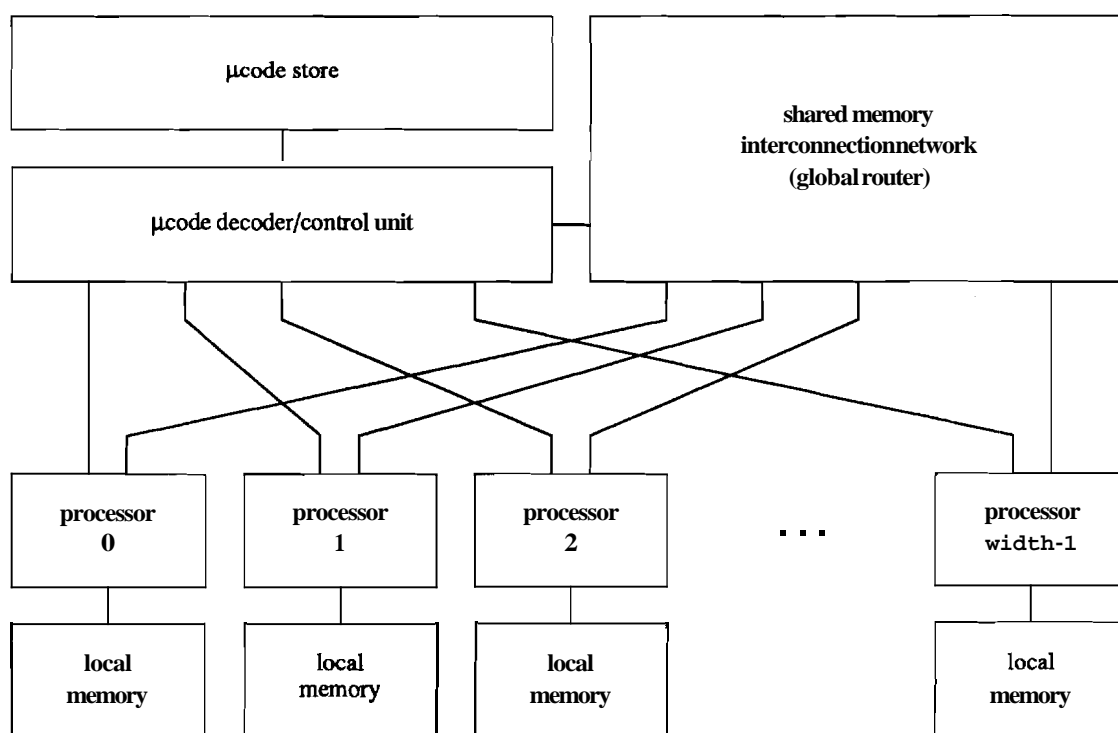
which is relatively expensive. Notice that in addition to the **bitwise** operations, the above implementation requires a memory access (*i.e.*, loading the value of `cc`) that would not be necessary for a machine that supports enable **masking** in hardware. Because **masking** is done for each simulated instruction, the **masking** cost effectively increases the basic interpretation overhead.

Examples of SIMD machines whose hardware can implement the appropriate **masking** include the TMC CM-1 and the MasPar MP-1.

#### 1.4. Our Approach

Now consider building a true MIMD machine using a specially designed SIMD **microengine** instead of simply implementing an interpreter on top of an existing SIMD machine.

Just as building an efficient interpreter would be infeasible unless the SIMD machine has hardware supporting both indirection and **masking**, the SIMD microengine must incorporate hardware for these functions. However, if we are designing a SIMD microengine, it is inexpensive to make it support both indirection and masking. How do we know this? Because **the MasPar MP-1's SIMD instruction set is actually implemented by microcode on a SIMD microengine that supports both indirection and masking**. We are not claiming that the MasPar MP-1 hardware is our ideal SIMD microengine, but it is close enough to allow us to implement a proof-of-concept MIMD emulation — as presented in this paper.



**Figure 1:** Block Diagram of MIMD using SIMD microengine

In our system, as shown in figure 1, the **MasPar's ACU** (Array Control Unit) becomes our microcode decoder and control unit, synchronously managing the parallel system. The ACU memory is thus the microcode store (with virtual memory paging support). Each SIMD PE becomes an essentially complete MIMD processor — except in that these processors do not have any local microcode control. The local memory for each PE functions identically in the MIMD organization, except in that the union of the local memories, with the help of the global router network, forms a global shared memory. Note that even though global memory references must pass through processors, this is done transparently under microcode control.

Given an appropriate SIMD microengine, the only remaining difficulty is the emulator (interpreter) overhead associated with decoding and performing operations within each SIMD PE. By careful construction of the MIMD instruction set and optimization of the emulation algorithm, the effective interpreter overhead and number of instruction types can be reduced greatly.

The result is a MIMD emulation that typically achieves a large fraction of the peak speed that a pure SIMD instruction set would obtain using the same SIMD microengine. As a true microcoded implementation, it is possible that the MIMD machine would have peak performance virtually identical to the equivalent SIMD machine.

Unfortunately, there are a number of compromises in the implementation of our proof-of-concept prototype MIMD emulator as presented in this paper. By far the most important compromise is that rather than directly using the SIMD microengine, our **current** version of the emulator is written in MPL [Mas91], a C language dialect that is compiled into the **MasPar's** SIMD macroinstructions. This results in between about 115th and 1140th the peak SIMD performance when executing pure MIMD code.

While these numbers rank our prototype 16,384-processor shared memory barrier MIMD as a "marginal" supercomputer peaking in the low 100's of MIPS, and the **MasPar** MP-I is cheap enough to even yield a reasonable **MIPS/dollar** rating using our **current** emulator, that is not our point. Our point is that, designing a SIMD microengine from scratch, the performance of this new type of MIMD implementation could be superior to that obtained by more conventional MIMD architectures.

The remainder of this paper explains the design, optimization, and prototype performance of a MIMD machine constructed using a SIMD microengine.

## 2. Instruction Set Design

Although there are many factors influencing the design of an instruction set, here we are concerned only with making the instruction set execute efficiently and be powerful enough to encode reasonable programs.



## 2.1. Memory Reference Model

In many computers, execution speed is more often limited by memory reference time than by the speed of arithmetic operations within a processor. The choice of memory reference model is even more important in the design of a massively parallel machine:

1. Although each processor generally has local memory nearby, the bandwidth is usually limited by VLSI pinout constraints and the need to minimize the number of memory chips per processor. For example, on the MasPar MP-1 each group of 16 PEs shares a single, time multiplexed, port to local memory.
2. Processors inevitably must communicate with each other, hence, some mechanism for accessing data from another PE is needed. Massively parallel machines can have massive amounts of memory distributed across all PEs; there is even a strong incentive to spread local data across the machine simply because it might not fit in local memory, which is typically small.

The following two sections address these issues.

### 2.1.1. Local Memory Model

The standard solution to the first problem is to use either registers or cache. Fortunately, machines like the MasPar MP-1 have many registers... unfortunately, the same register must be accessed by all enabled PEs. Without the ability for each PE to access a register of its choosing, it is impossible for the PE to efficiently implement a register oriented model; each "register" would have to be stored in local memory. Although the modification of the MasPar MP-1 hardware to support indirect register references (similar to those on the AMD 29K [Adv89]) would be relatively simple, as a practical matter, such a modification is beyond the scope of academic research.

Hence, we are forced to reduce the number of memory references by using an instruction set in which every instruction accesses the same register for an operand. Either an accumulator-based or stack cache-based scheme is viable; we used a stack cache in which the top element on the stack is cached in a particular register. Larger stack cache sizes are impractical due to the overhead in manipulating registers to appear as a stack cache. The single element stack cache averts one operand fetch on all unary and binary operations.

### 2.1.2. Global Memory Model

Although many small MIMD computers allow all processors to share access to a common memory [ThG87][Cra91], it is very difficult to construct hardware that scales this feature up to thousands of processors. Hence, the primary question becomes whether one should try to make distributed memory hardware appear to software as slow shared memory or as distributed memory accessed by explicitly sending a message to the processor for whom that memory is local.

There are two reasons that we use a shared memory model:

1. The shared memory model implies that all packets sent through the network at a given time operate on the same size data — one memory word. This implies that SIMD network control can be used without loss of efficiency [BeS91] — and at a great savings in network switch hardware complexity.
2. If an explicit, asynchronous, message-passing scheme were used, it would be necessary to both buffer messages and to interrupt the receiving processor to process them. These overheads would both complicate (i.e., slow) the emulator and result in longer instruction sequences that could not be executed in parallel on the SIMD microengine.

For these reasons, supporting a shared memory model is actually likely to be more efficient than using explicit message passing. Of course, one still should program so that most references will be to objects in local memory, because global references will be slower. On the 16,384 PE **MasPar** MP-1 using the global router network, the ratio between global and local references is approximately 10:1<sup>2</sup>.

There is, however, one other difficulty that arises in the above treatment of shared memory: if every processor wants to access the same shared memory location, this may cause network contention that would serialize the operations. Effectively, this was the problem that inspired "Repetition Filter Memory" [Kla80] and "Fetch-and-Op" [Sto84] for shared memory MIMD computers.

Surprisingly, this problem is much easier to solve when the network control is SIMD [BeS91]. In effect, races can be resolved by the SIMD microengine's control unit — and the resulting value can simply be broadcast. For example, the current MIMD emulator allows a second type of shared memory which is implemented using a copy in each processor; loads are local memory references, stores have races resolved by the control unit and the result broadcast to all copies of the variable. The SIMD network control also makes "Fetch-and-Op" efficiently implementable without additional hardware.

## 2.2. Assembly Language Model

In implementing a **MIMD** machine using a SIMD microengine, it seems that the ultimate limit on performance must be the fact that the SIMD microengine must serialize execution of different types of MIMD instructions. Hence, one would expect that the slowdown for an emulated MIMD would be roughly proportional to the sum of the execution times of all instructions in the instruction set. Fortunately, this need not be the case, because:

1. Here we are talking about a SIMD microengine, and many of the microinstructions implementing different MIMD instructions are of the same type. Hence, it isn't a matter of not being able to overlap the MIMD **Mul** and **Div** instructions, but a

---

<sup>2</sup> This remarkably low ratio is due to the fact that the MasPar MP-1 router is fast and local memory accesses are slow due to 16-way sharing of local memory ports.

matter of overlapping all their constituent microinstructions except for the ALU operation. The problem is not lack of overlap, but rather the complexity of making the best choice among the many possible microinstruction overlaps. By designing the instruction set so that many microinstructions will form common subsequences, only a small amount of overhead is associated with having a larger instruction set.

2. Even if there are many instructions in the MIMD instruction set and there are few microinstructions in common, the emulation speed can be very good if only a few different instructions are to be executed in any given emulation cycle. For example, the MIMD instruction set supported by our prototype emulator contains 38 different instructions, many of which have little microinstruction overlap; however, even in the very asynchronous MIMD program given in section 4.3, there were only an average of 6.28 different types of MIMD instructions executed in each emulator cycle. In section 3.2.2.2, we also describe how the number of different MIMD instruction types whose execution is attempted in each emulator cycle can be artificially reduced.

In fact, the techniques used are so effective that the instruction set size and instruction execution time have only indirect impact on emulation speed.

The single most severe constraint on instruction set size for the current MIMD emulator is the desire to minimize the time taken for instruction fetch from local memory — ideally, the instruction set would have no more than 256 instructions so that all opcodes will fit in 8 bits. Because of memory address alignment constraints<sup>3</sup>, the use of 8-bit opcodes makes it difficult to fetch more than an 8-bit immediate operand. Hence, our instruction set incorporates a constant pool that holds up to 256 32-bit values.

### 23. Prototype Instruction Set

In the prototype MIMD emulator, we have implemented an instruction set that is as rich as we felt was useful. Even as this paper is being written, we are considering a number of changes including the addition of several new instructions.

A brief summary of the MIMD instruction set used in the current emulator appears in table 1. Mnemonics followed by *i* are operations using 8-bit immediate values, and those followed by *c* use 32-bit values taken from the constant pool. The processor number, *this*, and the number of processors, *width*, are actually special entries in the constant pool that are initialized at program load time; hence, they are accessed via *Const* instructions. The class membership column of this table is discussed in section 3.2.2.1.

---

<sup>3</sup> The MasPar MP-1 microengine has no alignment constraints, but the SIMD macroinstruction set unfortunately does.

| Mnemonic | Functional Description               | Class Membership                  |
|----------|--------------------------------------|-----------------------------------|
| Add      | int addition                         | Op_NOS                            |
| And      | bitwise and                          | Op_NOS Op_Rare                    |
| Const c  | push 32-bit constant                 | Op_Immed Op_CPool                 |
| Div      | int divide                           | Op_NOS Op_Slow Op_Rare            |
| Eq       | compare for ==                       | Op_NOS                            |
| FAdd     | float addition                       | Op_NOS                            |
| FDiv     | float divide                         | Op_NOS Op_Slow Op_Rare            |
| FMul     | float multiply                       | Op_NOS                            |
| FNeg     | float negate                         | Op_Rare                           |
| FPrint   | float print                          | Op_NOS Op_Slow Op_Rare            |
| FToI     | convert float to int                 | Op_Rare                           |
| GE       | compare for >=                       | Op_NOS Op_Rare                    |
| GT       | compare for >                        | Op_NOS Op_Rare                    |
| Halt     | stop this processor                  |                                   |
| IToF     | convert int to float                 | Op_Rare                           |
| Jump c   | unconditional jump                   | Op_Immed Op_CPool                 |
| JumpF c  | jump if false (zero)                 | Op_NOS Op_Immed Op_CPool          |
| Ld       | load                                 |                                   |
| LdD      | load from distributed memory         | Op_NOS Op_Slow Op_Rare            |
| LdL      | load local from stack                |                                   |
| LdS      | load from shared memory (same as Ld) |                                   |
| Mod      | int modulus                          | Op_NOS Op_Slow Op_Rare            |
| Mul      | int multiply                         | Op_NOS Op_Rare                    |
| NE       | compare for !=                       | Op_NOS Op_Rare                    |
| Neg      | int negate                           |                                   |
| Not      | bitwise not                          | Op_Rare                           |
| Or       | bitwise or                           | Op_NOS Op_Rare                    |
| Pop i    | remove items from stack              | Op_Immed Op_Rare                  |
| Print    | int print                            | Op_NOS Op_Slow Op_Rare            |
| Push i   | push 8-bit constant                  | Op_Immed                          |
| Ret i    | return and pop items                 | Op_Immed                          |
| SPrint c | string print                         | Op_Immed Op_CPool Op_Slow Op_Rare |
| ShL      | int shift left                       | Op_NOS                            |
| ShR      | int shift right                      | Op_NOS Op_Rare                    |
| St       | store                                | Op_NOS                            |
| StD      | store into distributed memory        | Op_NOS Op_Slow Op_Rare            |
| StL      | store local into stack               | Op_NOS                            |
| StS      | store into shared memory             | Op_NOS Op_Slow Op_Rare            |
| Wait     | wait for barrier synchronization     | Op_Slow Op_Rare                   |

Table 1: MIMD Instruction Set.

### 3. Emulator Design

Although the detailed design of the emulator is intertwined with the design of the instruction set and the SIMD **microengine**, for this paper we will make the simplifying assumption that the SIMD microengine is the machine on which we have implemented our prototype: the **MasPar MP-1**. Further, we will restrict the examples to the instruction set as given in table 1 and used in the prototype emulator.

The most important optimizations of the emulator can be grouped into two categories: reduction of the emulation overhead by shortening the basic emulator cycle or by maximizing overlap (parallelism) in emulated execution of different types of instructions.

#### 3.1. Shortening The Basic Cycle

There are many ways to reduce the basic emulator cycle time:

1. Keep processor state in microengine registers. In the case of our interpreter, the program counter (**pc**), instruction register (**op**), program relocation base address (**addr**), constant pool base address (**cp**), top-of-stack cache (**tos**), and various other internal variables are all kept in registers.
2. Don't use a linear sequence of enable-masking conditional tests to isolate an operation type. For our emulator, a helper program was written in C to automatically generate an **optimal binary search tree** for isolating operation types.
3. Either don't use a high-level language or use it, but take steps to ensure good code will be generated. Our emulator is written in MPL and **MasPar's** MPL compiler usually generates fairly efficient code, but not always. In particular, the **MPL** compiler is obsessed with performing needless conversion of quantities from 8 to 32 bits — a painful **error** when the processors are based on 4-bit slices. We repair this code generation blunder by using an AWK script to recognize and remove the needless conversions from the assembly code for the main emulation loop.

In addition to the above, there are a number of minor coding tricks employed.

#### 3.2. Minimizing Operation Time

With a small instruction set consisting of relatively cheap operations, the basic emulator cycle time is more important than the serialization of execution of different operations. However, a truly useful machine needs more operation types and must support at least a few expensive operations (**e.g.**, shared memory references). Hence, it is very **important** that there be some techniques used to reduce the operation time.

There are two basic way in which the operation time can be reduced:

1. Increase the overlap, at the microcode level, between the various instructions that are to be executed.

2. Reduce the number of different operations that must be executed in an emulator cycle — ideally to just one operation, i.e., to SIMD code.

The following sections detail the methods used in the current emulator.

### 3.2.1. Maximizing Instruction Overlap

The concept of maximizing the microcode overlap for a series of operations is not new. In fact, it is probably the single most common hand optimization used in writing microcode or code for a SIMD machine. Unfortunately, the process had not been formalized and automated until very recently.

The new compiler optimization, called "Common Subexpression Induction" (CSI) [Die91], accepts multiple independent threads of code and outputs a reorganized version of the code that shares instructions across threads so that the minimum execution time is obtained. Although the algorithm is far too complex to describe in this paper, the general **flavor** is that operations from various threads are classified based on how they could be merged into single instructions executed by multiple threads and then a heavily pruned search is executed to find the minimum execution time code schedule using these merges. In fact, the development of the CSI algorithm was the enabling technology that inspired our first MIMD interpreter.

Without the CSI algorithm, it is possible to find and factor-out common microinstruction subsequences by hand only for very small, simple, instruction sets. For the MIMD emulator presented here, hand tuning was inconvenient, but coding the emulator in MPL made it impossible to directly use our CSI tool (since the CSI tool generates unstructured control flow and masking). Hence, we used the CSI tool to locate the most advantageous subsequences and then hand coded them in MPL.

These sequences included the basic instruction fetch and program counter increment, fetching the value for the next-on-stack (NOS), fetching the value for an 8-bit immediate (Immed), and looking-up a 32-bit value in the constant pool (CPool). Without this factoring, the emulator would be several times slower.

### 3.2.2. Reducing Operation Count

The second "trick" in speeding up execution of multiple different operations involves the observation that the emulator need only be able to decode the instructions which might be executed in this cycle, rather than the entire instruction set. But how can we know which instructions might be executed in this cycle without actually decoding them first?

There are two answers, both of which are used heavily in the current emulator: subemulators and frequency biasing.

### 3.2.2.1. Subemulators

Because the **microengine** is completely synchronous, it is relatively easy to construct **hardware** that will allow the control unit to check to see if there exists a processor in which a particular value meets some condition. In the **MasPar**, this is implemented by an operation called **globalor**, which in just 10 clock ticks (less than  $1\mu\text{s}$ ) ors together values from all the **PEs**. By carefully **encoding** the instruction set, we can **use a globalor of the opcode values to index a control unit jump table to select the emulator that understands only those instructions that could appear within the or mask value.**

Within the current emulator, there are 32 such "subemulators." Obviously, the 32 **subemulators** could not reasonably be generated by hand, so a C program was written to **perform** this task.

In addition, the choice of how instructions are grouped together into subemulators should not be made at random. Instructions that share CSIs should be grouped together because factoring-out the CSIs will make those interpreters execute faster. Hence, the **NOS**, **Immed**, and **CPool** CSIs (described above) correspond to bit positions in both the opcode and the **globalor** mask. It is also useful to make similar divisions based on the expected cost (Slow) and frequency of execution (Rare), and these sets also correspond to opcode bits. The class membership of each instruction in our **MIMD** instruction set is given in table 1.

To illustrate the subinterpreter structure, we present a complete subemulator set. However, to keep the size reasonable, we have restricted the subemulator set to cover only the instructions used in the example in listing 1.

```

/* fact.mc
Recursive int factorial
*/
int
fact(int n)
{
    if (n) {
        return(n * fact(n-1));
    }
    return(1);
}

_fact:
    Const    0          ;if (n)
    LdL
    JumpF    L0
    Const    0          ;n
    LdL
    Const    retlab0 ;fact(n-1)
    Const    8
    LdL
    Const    -1
    Add
    Jump     _fact
retlab0:
    Mul      ;*
    Ret      2      ;return
L0:
    Const    1          ;return(1)
    Ret      2

```

Listing 1: MIMD Factorial — C and Assembly Code

The complete subemulator set is given in listing 2. The `Op_` references are opcode values or bit masks; the `m_` references are macros that actually perform the corresponding operation. If we assume that **all** processors in the MIMD machine call `_fact` at the same time, all processors would simultaneously execute the `Const` instruction using the subinterpreter for classes `Immed` and `CPool` (case `0x14`). Suppose that some processors wish to execute `Mul` (classes `NOS` and `Rare`) at the same time that others execute `JumpF` (`Immed`, `NOS`, and `CPool`); then the subemulator for `Immed`, `NOS`, `CPool`, and `Rare` would be executed (case `0x1d`). Notice that the C program that builds the subemulators factors-out identical subemulators (e.g., case `0x19` and case `0x1b`) to save **ACU** memory space.



```

switch ((globalor op) & opmask) {
case 0x0: /* Opcodes in every class */
case 0x1: /* Op-Rare */
case 0x2: /* Op_Slow */
case 0x3: /* Op_Slow Op_Rare */
case 0x4: /* Op_CPool */
case 0x5: /* Op_CPool Op_Rare */
case 0x6: /* Op_CPool Op_Slow */
case 0x7: /* Op_CPool Op_Slow Op-Rare */
    M_LdL
    break;

case 0x8: /* Op_NOS */
case 0xa: /* Op_NOS Op_Slow */
case 0xc: /* Op_NOS Op_CPool */
case 0xe: /* Op_NOS Op_CPool Op_Slow */
    if (op & Op_NOS) {
        M_NOS M_Add
    } else {
        M_LdL
    }
    break;

case 0x9: /* Op_NOS Op-Rare */
case 0xb: /* Op_NOS Op_Slow Op_Rare */
case 0xd: /* Op_NOS Op_CPool Op_Rare */
case 0xf: /* Op_NOS Op_CPool Op_Slow Op-Rare */
    if (op & Op_NOS) {
        M_NOS
        if (op <= Op-Add) {
            M_Add
        } else {
            M_Mul
        }
    } else {
        M_LdL
    }
    break;

case 0x10: /* Op_Immed */
case 0x11: /* Op_Immed Op-Rare */
case 0x12: /* Op_Immed Op_Slow */
case 0x13: /* Op_Immed Op_Slow Op-Rare */
    if (op & Op_Immed) {
        M_Immed M_Ret
    } else {
        M_LdL
    }
    break;

case 0x14: /* Op_Immed Op_CPool */
case 0x15: /* Op_Immed Op_CPool Op_Rare */
case 0x16: /* Op_Immed Op_CPool Op_Slow */
case 0x17: /* Op_Immed Op_CPool Op_Slow Op-Rare */
    if (op & Op_Immed) {
        M_Immed
        if (op & Op_CPool) {
            M_CPool
        }
    }
    if (op <= Op_Ret) {
        if (op <= Op_Const) {
            M_Const
        } else {
            M_Ret
        }
    } else {
        if (op <= Op_Jump) {
            M_Jump
        } else {
            M_LdL
        }
    }
    break;

case 0x18: /* Op_Immed Op_NOS */
case 0x1a: /* Op_Immed Op_NOS Op-Slow */
    if (op & Op_NOS) {
        M_NOS M_Add
    } else {
        if (op & Op_Immed) {
            M_Immed
        }
        if (op <= Op_Ret) {
            M_Ret
        } else {
            M_LdL
        }
    }
    break;

case 0x19: /* Op_Immed Op_NOS Op_Rare */
case 0x1b: /* Op_Immed Op_NOS Op_Slow Op-Rare */
    if (op & Op_NOS) {
        M_NOS
        if (op <= Op-Add) {
            M_Add
        } else {
            M_Mul
        }
    } else {
        if (op & Op_Immed) {
            M_Immed M_Ret
        } else {
            M_LdL
        }
    }
    break;

case 0x1c: /* Op_Immed Op_NOS Op_CPool */
case 0x1e: /* Op_Immed Op_NOS Op_CPool Op-Slow */
    if (op & Op_NOS) {
        M_NOS
        if (op & Op_Immed) {
            M_Immed M_CPool M_JumpF
        } else {
            M_Add
        }
    } else {
        if (op & Op_Immed) {
            M_Immed
            if (op & Op_CPool) {
                M_CPool
            }
        }
        if (op <= Op_Ret) {
            if (op <= Op_Const) {
                M_Const
            } else {
                M_Ret
            }
        } else {
            if (op <= Op_Jump) {
                M_Jump
            } else {
                M_LdL
            }
        }
    }
    break;

case 0x1d: /* Op_Immed Op_NOS Op_CPool Op-Rare */
case 0x1f: /* Op_Immed Op_NOS Op_CPool Op-Slow Op-Rare */
    if (op & Op_NOS) {
        M_NOS
        if (op & Op_Immed) {
            M_Immed M_CPool
        }
        if (op <= Op_JumpF) {
            if (op <= Op-Add) {
                M_Add
            } else {
                M_JumpF
            }
        } else {
            M_Mul
        }
    } else {
        if (op & Op_Immed) {
            M_Immed
            if (op & Op_CPool) {
                M_CPool
            }
        }
        if (op <= Op_Ret) {
            if (op <= Op_Const) {
                M_Const
            } else {
                M_Ret
            }
        } else {
            if (op <= Op_Jump) {
                M_Jump
            } else {
                M_LdL
            }
        }
    }
    break;
}

```

Listing 2: Example Subemulator Set

A vaguely similar type of improvement is suggested in [NiT90]. Nilsson and Tanaka envision a set of subinterpreters such that each subinterpreter emulates only a single type of instruction and all subinterpreters are executed once per interpreter cycle. Using statistics, they change

the order of the subinterpreters to maximize the expected number of instructions executed per processor per interpreter cycle. **E.g.**, if the subinterpreters **are** in the order A, B, C then the instruction sequence B, A takes 2 cycles — but B, A would take only one cycle if the **subinter-**preters were ordered B, C, A. The problem is that this improvement is small and is essentially incompatible with factoring-out portions of the emulated operations (*i.e.*, instruction fetch).

### 3.2.2.2. Frequency Biasing

The second way to reduce operation count is what we call frequency biasing. Suppose that a particular operation takes  $t_1$  ticks to execute and another operation takes  $t_2=5*t_1$ . If these two instructions were allowed to execute in each emulator cycle, the apparent execution time of both would be about  $6*t_1$ . Suppose that instead, we would allow up to five emulator cycles of the first instruction before attempting to execute the second instruction; the fast instruction will average one execution every 2 emulator cycles, and the slow instruction will average one execution every 10 cycles. This is essentially an instruction-level variation on the concept of shortest job first (SJF) scheduling, and yields the same benefits.

However, the benefits would be small were it not for an interesting property of most expensive operations: if several expensive operations would have been executed just one or two emulator cycles off from each other, **delaying the operations will cause them to group together in the same emulator cycle**. For most operations, having more processors execute the operation simultaneously does not significantly change the speed with which that operation is executed. Hence, this "alignment" effect dramatically improves performance.

Notice that if we consider not two instructions, but two groups of instructions, the same property holds.

In the current version of the emulator, only a small amount of frequency biasing is used. Instructions that are in either the Slow or **CPool** classes **are** only allowed to execute every other cycle. Of course, we need not be able to decode these instructions in the subemulator set that excludes these operations. Hence, there are actually two different subemulator sets, or a total of 64 subemulators, within the current emulator. Despite this, the complete emulator uses less than **80K** bytes of ACU memory.

## 4. Performance Evaluation

Our first proof of concept MIMD system was implemented on the **Purdue** University Parallel Processing Laboratory's 16,384-PE **MasPar** MP-1 in July 1991, shortly after developing the CSI algorithm and prototype implementation. The current version (January 1992) of the MIMD system includes:

`mimdc`

A compiler, written in C using **PCCTS** [PaD92]. The language is a parallel dialect of

C called MIMDC. It supports most of the basic C constructs. Data values can be either `int` or `float`, and variables can be declared as `mono` (shared) or `poly` (private) [Phi89].

There are actually two kinds of shared memory reference supported. The `mono` variables are replicated in each processor's local memory so that loads execute quickly, but stores involve a broadcast to update all copies. It is also possible to directly access `poly` values from other processors using "parallel subscripting":

$$x[i] = y[j] + z;$$

would use the values of `i`, `j`, and `z` on this processor to fetch the value of `y` from processor `j`, add `z`, and store the result into the `x` on processor `i`. In addition to using shared memory for synchronization, MIMDC supports barrier synchronization [Dis89] using a `wait` statement.

#### **mimda**

An assembler, written in C. The stack-based MIMD assembly `code` output by `mimdc` is assembled to generate both a listing file and an Intel-format hex load module.

#### **mimd**

The MIMD interpreter, written in MPL (**MasPar's** SIMD C dialect) with the aid of several specialized interpreter construction programs written in C and AWK. The structure of `mimd` was described in detail in section 3.

Benchmark programs were written in MIMDC and their performance was evaluated. A high level language was used for the benchmarks because we feel that it both "keeps us honest" and provides a friendlier, more realistic, interface for program development.

### **4.1. High-Level Language Peak MFLOPS**

Although we have measured the peak floating point performance of hand-coded MIMD programs at from 280 MFLOPS to over 350 MFLOPS, we felt that the fairest comparison would be to take essentially SIMD codes, written in MIMDC and MPL, and compare the MFLOPS obtained. Listing 3 shows these two equivalent programs.

```

/*  gflop.mc

Do 1 GFLOP worth of float adds.
*/

int
main()

int count = 10000000/16384;
float sum = 0.0;

while (count) {
    /* do 100 float adds per loop... */
    sum = sum + sum + sum + sum + sum +
    sum + sum + sum + sum + sum +
    sum + sum + sum + sum + sum +
    sum + sum + sum + sum + sum +
    sum + sum + sum + sum + sum +
    sum + sum + sum + sum + sum +
    sum + sum + sum + sum + sum +
    sum + sum + sum + sum + sum +
    sum + sum + sum + sum + sum +
    sum + sum + sum + sum + sum +
    sum + sum + sum + sum + sum +
    sum + sum + sum + sum + sum +
    sum + sum + sum + sum + sum +
    sum + sum + sum + sum + sum +
    sum + sum + sum + sum + sum +
    sum + sum + sum + sum + sum +
    sum + sum + sum + sum + sum +
    sum + sum + sum + sum + sum +
    count = count - 1;
}

/* mimd emulator automatically prints time */
}

/*  gflop.m

Do 1 GFLOP worth of float adds...
*/

#include <mpl.h>

extern double dpuTimerElapsed();

int
main0
{
    int count = 10000000/16384;
    plural float sum = 0.0;

    dpuTimerStart();

    while (count) {
        /* do 100 float adds per loop... */
        sum = sum + sum + sum + sum + sum +
        sum + sum + sum + sum + sum +
        sum + sum + sum + sum + sum +
        sum + sum + sum + sum + sum +
        sum + sum + sum + sum + sum +
        sum + sum + sum + sum + sum +
        sum + sum + sum + sum + sum +
        sum + sum + sum + sum + sum +
        sum + sum + sum + sum + sum +
        sum + sum + sum + sum + sum +
        sum + sum + sum + sum + sum +
        sum + sum + sum + sum + sum +
        sum + sum + sum + sum + sum +
        sum + sum + sum + sum + sum +
        sum + sum + sum + sum + sum +
        sum + sum + sum + sum + sum +
        sum + sum + sum + sum + sum +
        sum + sum + sum + sum + sum +
        sum + sum + sum + sum + sum +
        count = count - 1;
    }

    printf("Done: %gs DPU usage\n", dpuTimerElapsed());
}

```

**Listing 3:** Peak FLOPS benchmark in MIMDC and MPL.

In the MIMD program, all processors execute the same code sequence, only one instruction is executed in the emulator for each MIMD cycle and processors are rarely idle. The other code, written in MPL, executes with all processors enabled and is completely SIMD. Neither program does any useful calculations, but the performance provides a good estimate of peak floating point speed<sup>4</sup>. The emulator achieved 97.2 MFLOPS, or about 10% of the MPL program's 986 MFLOPS. Note that the **MasPar's** theoretical peak speed is 1,200 MFLOPS.

## 4.2. Emulation Overhead

The above numbers also allow us to compute something much more meaningful: the emulation overhead. Since our emulator records the number of emulation cycles executed, and we know that the actual operations must have taken the time that the MPL program ran for, we were able to determine that each emulator cycle had an overhead of about **48 $\mu$ s**. This number was also confirmed by other benchmarks.

Aside from the fact that **48μs** is surprisingly fast, it is important to note that most of this overhead could be eliminated by **recoding** the emulation in a different language. An obvious way

<sup>4</sup> Note that the MasPar floating point operation time is not dependent on operand value, hence adding 0 values yields a valid time without the potential for overflow.

to reduce the overhead is to write the emulator in the **MasPar's SIMD** assembly language instead of in **MPL**; however, unless the emulator algorithm also is changed, the improvement would be quite small. This is partly because **MPL** is low-level enough (e.g., **register** declarations) to usually generate good code, and partly because we already use an AWK script to patch the few obvious blunders made by **MPL**.

The insight that could remove most of the **48 $\mu$ s** overhead is that **the MasPar's 32-bit RISC SIMD instruction set is implemented by microcode executed on 4-bit PEs**. By implementing the MIMD emulator as a single new microcoded instruction, **emulateMIMD**, the emulation overhead per emulator cycle would almost certainly drop to less than **10 $\mu$ s**.

Small additional improvements, at either the assembly or microcode level, could result by slightly altering the emulation algorithm. Essentially, **MPL** only allows **structured** mixing of control flow and enable **masking**; there are a few portions of the emulation that could profit from directly manipulating enable masks.

### 43. A Many-Thread Example

While the above numbers are impressive, they should be impressive because for each emulator cycle, all MIMD threads were executing the same instruction taken from the same relative location in PE memory. Such code sequences are actually common in massively parallel MIMD code, but it is much more important that most cases typically encountered perform reasonably. In fact, the emulator **structure** is **not** designed to maximize best-case execution speed.

Recall that different instruction types execute serially in the SIMD microengine. Hence, a MIMD program that tends to have a wide range of different instructions being encountered within each emulator cycle should provide much poorer performance. These are also the cases that most of the emulator's optimizations attempt to improve. A MIMD program with this property makes a much tougher test case.

#### 43.1. The Program

Unfortunately, most parallel benchmarks are more SIMD in nature, and would yield better performance. Lacking a good "standard" example MIMD program, we selected a recursive algorithm in which processors take radically different paths through the code based on their processor numbers. The problem selected, and implemented in MIMDC, was a graph fault-tolerance problem in which each of the 16,384 processors analyzed a unique graph derived from a master graph. The master graph is shown in figure 2; each line represents two arcs, one in each direction.

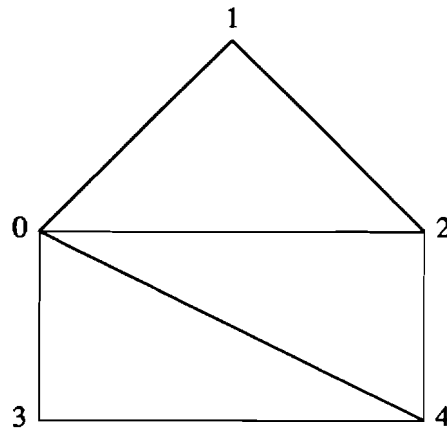


Figure 2: Initial Graph for **graph.mc**.

The program in listing 4 uses an exhaustive recursive search to determine, for all possible combinations of faulty arcs in the master graph, the total number of faulty states for which it is still possible to travel from node 0 to node 4. We do not claim that this is a good algorithm for this problem, but it is a good example of "true MIMD" code.

**All 16,384** processors begin by executing `main()`. Each processor reads the master graph and modifies it to produce a unique faulty graph by removing arcs corresponding to 0 bits in the processor number (called `this` in MIMDC). Since **16,384** is  $2^{14}$ , we use a graph with **14** arcs so that each processor will have a unique task. The function `foundpath()` determines whether a path exists by a depth first search. It returns as soon as it has found a node that it has already visited, has reached the destination, or has explored all arcs leaving the node. If it finds an arc that goes to a node it has not visited, it recursively calls itself with the unexplored node. When **all** faulty graphs have been explored, the `reduceAdd()` function uses **barrier** synchronization and distributed memory accesses to total the number of successful path traversals.

```

/* graph.mc

A simple little program to compute some basic
fault tolerance properties....

*/

#define LINKS 14

mono int master_from[LINKS] = {
    0, 1, 0, 2, 0, 3, 0, 4, 1, 2, 2, 4, 3, 4
};
mono int master_to[LINKS] = {
    1, 0, 2, 0, 3, 0, 4, 0, 2, 1, 4, 2, 4, 3
};
mono int master-arcs = 14;

poly int from[LINKS];
poly int to[LINKS];
poly int been_there[LINKS];
poly int arcs = 0;

int
main()
{
    /* Initialize poly copies of the master graph
    with all arcs removed corresponding to the
    0 bits in the processor number

    */
    int i = 0;
    int mybits = thin;
    int gotpath = 0;

    while (i < master-arcs) {
        if (mybits & 1) {
            from[arcs] = master_from[i];
            to[arcs] = master_to[i];
            arcs = arcs + 1;

            t
            mybits = mybits >> 1;
            i = i + 1;
        }
        I

        /* Try to find the paths from node 0 to 1 */
        if (this < (1<<master_arcs)) {
            gotpath = foundpath(0,4);
        } else {
            gotpath = 0;
        }
        I

        gotpath = reduceAdd(gotpath);

        /* Let processor 0 speak for all... */
        if (this == 0) {
            print "There were ", (1<<master_arcs),
                " networks checked.\n";
            print "Of these, ", gotpath,
                " could reach 4 from 0.\n";
        }
        I
    }

    int
    foundpath(int here, int there)
    {
        int i = 0;
        int k = 0;

        /* Are we there yet? */
        if (here == there) return(1);

        I* We are here. */
        been_there[here] = 1;

        /* Try each arc outta here... */
        while (i < arcs) {
            I* Found an arc out... */
            if (from[i] == here) {

                /* Have we been where it goes? */
                if (been_there[ to[i] ] == 0) {

                    /* Nope. Go there now... */
                    if (foundpath(to[i], there)) {
                        return(1);
                    }
                }

                /* No luck yet, try any other arcs... */
                i = i + 1;
            }
            I

            /* Can't get there from here... */
            return(0);
        }

        poly int reduce-tmp;
        poly int rk;
        poly int rj;
        poly int ri;

        int
        reduceAdd(int val)
        {
            /* Recursive doubling summation */
            ri = 1;

            reduce-tmp = val;
            while (ri < width) {
                wait;
                rk = (this + ri) * (width);
                rj = reduce_tmp[ri rk];
                wait;
                reduce-tmp = reduce-tmp + rj;
                ri = ri << 1;
            }
            t
            return(reduce-tmp);
        }
    }
}

```

Listing 4: Code for graph.mc.

Under the MMD emulator, `mimd`, each processor executes its own path through the code. Hence, the execution of this program differs greatly from a path search program for a SIMD machine.

On average, there were **16.3** unique program counter (PC) values active for each emulation cycle, which is roughly equivalent to **16** completely different programs executing. This average was obtained in a program that only has a total of **234** instructions. The number of PCs is also limited by the wide range in processor work loads, which causes many processors to wait at the first barrier in `reduceAdd()`. Averages of over **50** different PC values have been seen on a version of `graph.mc` that builds many different permutations of the graph for each processor before summing the number of networks with paths between **0** and **4** — but that program was

more complicated and the statistics were very similar. By any standard, the example code is very dynamic.

### 43.2. Performance

Complete emulation statistics for the code of listing 4 are given in tables 2 and 3. For table 2, the interpret count gives the number of emulator cycles in which that instruction was executed; the execute count is the total number of times that instruction was executed. Table 3 shows the number of times each particular subemulator mask occurred.

The execution speed was determined in two steps. First, the code was timed using the hardware timer available on the **MasPar** (80ns per tick). Then, the code was run using an instrumented version of the emulator to get the total number of emulator cycles needed to complete the program, number of cycles that had a particular instruction, number of each instruction executed, and the total number of instructions run. The instrumented MLMD emulator takes longer to run the program, but **preserves** the relative time between processors, number of instructions executed, and the number of emulator cycles in the MIMD program — this repeatability in itself is an important advantage of our **MIMD** implementation technique. The total number of instructions found by the instrumented emulator, divided by the run time of the **uninstrumented** emulator, yields the average number of instructions executed per second for the uninstrumented emulator.

For **graph.mc**, the average speed was 54.6 MIPS (excluding output from the print instructions at the end of the program). On average each processor was executing 3,300 MIMD instructions per second. This seems to be a very low number, but the processors on the MP-1 are implemented by 4-bit slices and each cluster of 16 shares a single interface to its local memory. Assuming that all of the processors are totally asynchronous, the maximum rate at which they would be able to fetch an 8-bit instruction, execute a simple 32-bit operation, and update the program counter is 123,000 instructions per second. Thus, the MLMD emulator had a slowdown of less than 37 times the maximum performance that would be obtained if each processor had its own instruction decoder and control — which would imply many times more hardware to implement each processor. We suspect, but cannot yet prove, that the additional hardware would actually increase processor hardware complexity by more than a factor of 37 (primarily due to the complexity of floating point and network control). Also keep in mind that we are still talking about the MPL-coded emulator speed versus pure MLMD hardware....

It should also be noted that, although **graph.mc** does not use floating point, this makes little difference in performance. Actually, the 32-bit floating point instructions for multiplication and division take significantly less time than the 32-bit integer versions; this is due to the lower precision — just 24 bits of mantissa. Much of the run time of the emulator is due to decoding the instruction and fetching operands (as was shown in the **gflop** benchmark; see section 4.2).



| Operation            | Interpret Count | Execute Count |
|----------------------|-----------------|---------------|
| Add                  | 1805            | 1899520       |
| And                  | 105             | 229376        |
| Const                | 1333            | 5257088       |
| Eq                   | 992             | 344320        |
| GT                   | 816             | 761472        |
| Jump                 | 889             | 773632        |
| JumpF                | 1028            | 1373440       |
| Ld                   | 1945            | 2882560       |
| LdD                  | 14              | 229376        |
| LdL                  | 1980            | 2411904       |
| Mod                  | 14              | 229376        |
| Pop                  | 1               | 16384         |
| Push                 | 2244            | 5637376       |
| Ret                  | 705             | 87424         |
| ShL                  | 1141            | 1131392       |
| ShR                  | 210             | 229376        |
| st                   | 1156            | 1336832       |
| StL                  | 1044            | 702592        |
| Wait                 | 1018            | 10197504      |
| <b><i>totals</i></b> | 18440           | 35730944      |

**Table 2:** Instruction Statistics for graph.mc.

| Subemulator Mask                         | Interpret Count |
|--|-----------------|
| 0  | 99              |
| Op_Slow Op-Rare                          | 30              |
| Op_NOS                                   | 118             |
| Op_NOS Op_Rare                           | 20              |
| Op_NOS Op_Slow Op-Rare                   | 35              |
| Op_Immed                                 | 104             |
| Op_Immed Op-Rare                         | 1               |
| Op_Immed Op_Slow Op-Rare                 | 5               |
| Op_Immed Op_CPool                        | 225             |
| Op_Immed Op_CPool Op_Slow Op_Rare        | 26              |
| Op_Immed Op_NOS                          | 980             |
| Op_Immed Op_NOS Op_Rare                  | 121             |
| Op_Immed Op_NOS Op_Slow Op-Rare          | 3               |
| Op_Immed Op_NOS Op_CPool                 | 56              |
| Op_Immed Op_NOS Op_CPool Op_Rare         | 165             |
| Op_Immed Op_NOS Op_CPool Op_Slow Op_Rare | 947             |
| <b><i>total</i></b>                      | 2935            |

**Table 3:** Subemulator Statistics for graph.mc.

## 5. Room for Improvement

Although the current MIMD emulation system works quite well, there are quite a few improvements that should be made. The following is a summary of a few of the more significant possible enhancements.

### 5.1. Compiler (`mimdc`)

Aside from being a rather stupid compiler (*i.e.*, the only optimization performed is constant folding), the compiler makes no attempt to perform code scheduling of any kind.

Within an individual processor, code scheduling can be used to improve performance by matching generated code sequences to the order in which different operations are encountered in the emulator. First, the compiler should attempt to be consistent in generating the same instruction pattern wherever possible. Second, because not **all** instructions are executed in every cycle, some permutations of an instruction sequence will have lower expected execution **times** than **oth-**ers.

Even greater performance improvements can be made by code scheduling across all processors. This involves complex timing analysis and static scheduling as a bamer MIMD architecture [DiO90][BrN90][DiO92], but a MIMD implemented with a SIMD microengine provides exactly the features needed for these optimizations to be applied.

The compiler is also guilty of a few obvious coding blunders. For example, `while` loops should be coded to only have one `JumpF` rather than a `JumpF` and a `Jump` per iteration.

### 5.2. Assembler (`mimda`)

Although the assembler was constructed using a parameterized assembler (a local invention called ASA) that is capable of minimizing length of span-dependent instructions, we do not currently use this feature. Hence, the compiler often conservatively generates `Const` instructions for which the assembler blindly generates `Const` instructions. Instead, the assembler should recognize `Const` as the long form of the `Push` instruction, and should substitute `Push` wherever possible.

### 5.3. Emulator (`mimd`)

Throughout this paper we have noted a number of things about the emulator that mark it clearly as a proof-of-concept prototype rather than a "real" machine. Obvious improvements include rewriting the emulator in the **MasPar** microcode, or at least in **MasPar** assembly language, instead of MPL. There **are** also some optimizations that result in unstructured manipulation of control flow and masking, and these could not be done in MPL, so the emulation algorithm will change in future versions.

Various changes and additions to the instruction set **are** also likely. In particular, the bamer mechanism will be made more general (currently it is an SBM, but will be upgraded to a DBM [DiO90]) and some provision for explicitly switching to pure SIMD execution will be added.

This would allow the machine to more efficiently execute parts of algorithms that are inherently SIMD, such as communication or reduction operations. The **MIMD/SIMD** switching will vaguely resemble the facility provided in the PASM prototype [BeS91].

In the immediate future, the emulator will be modified to provide a rudimentary operating system so that multiple users will be able to submit MIMD jobs asynchronously. In the current version, the complete MIMD environment is set up when the emulator begins executing.

## 6. Conclusions

In this paper, we have presented the theory behind construction of efficient MIMD machines using SIMD **microengines**. Further, we have detailed how we created a 16,384-processor shared memory **barrier** MIMD using a SIMD **MasPar MP-1**, and we have given measured performance figures that validate the approach.

The MIMD emulation software discussed in this paper, `mimdc`, `mimda`, and `mimd`, are being set up as a public domain Beta test release, and will be available via an **email** server. The **email** address will appear in the final version of this paper.

## References

- [Adv89] Advanced Micro Devices, *29K Family 1990* Data Book, Sunnyvale, California, 1989.
- [Bla90] T. Blank, "The MasPar MP-1 Architecture," 35th IEEE Computer Society **International** Conference (COMPCON), February 1990, pp. 20-24.
- [BrN90] C.J. **Brownhill** and A. Nicolau, Percolation Scheduling for *Non-VLIW* Machines, Technical Report 90-02, University of California at **Irvine, Irvine**, California, January 1990.
- [BeS91] T.B. Berg and H.J. Siegel, "Instruction Execution Trade-offs for SIMD vs. MIMD vs. Mixed Mode Parallelism," 5th International Parallel Processing Symposium, April 1991, pp. 301-308.
- [Cra91] Cray Research Incorporated, The *CRAY Y-MP C90* Supercomputer System, Eagan, Minnesota, 1991.
- [Die91] H.G. Dietz, "Common Subexpression Induction," Midwest Society for Programming Languages and Systems (MWSPLS) Spring Meeting, Digital Computer Laboratory (DCL), University of Illinois at Urbana-Champaign, **Urbana**, Illinois, April 20, 1991.
- [DiO90] H.G. Dietz, M.T. O'Keefe, and A. Zaafrani, "An Introduction to Static Scheduling for MIMD Architectures," *Advances in Languages and Compilers for Parallel Processing*, edited by A. Nicolau, D. Gelertner, T. Gross, and D. Padua, The MIT Press, Cambridge, Massachusetts, 1991, pp. 425-444.
- [DiO92] H.G. Dietz, M.T. O'Keefe, and A. Zaafrani, "Static Scheduling for Barrier MIMD Architectures," *The Journal of Supercomputing*, accepted to appear.
- [DiS89] H.G. Dietz, T. Schwederski, M. T. O'Keefe, and A. Zaafrani, "Static Synchronization Beyond **VLIW**," *Supercomputing 1989*, November 1989, pp. 416-425.
- [Hil87] W.D. **Hillis**, "The Connection Machine," *Scientific American*, June 1987, pp. 108-115.
- [Kla80] A.D. Klappholz, "An Improved Design for a Stochastically Conflict-Free **Memory/Interconnection** System," 14th Asilomar Conference on Circuits, Systems, and Computers, November 1980.
- [Mas91] MasPar Computer Corporation, *MasPar* Programming Language (*ANSI C compatible MPL*) Reference Manual, Software Version 2.2, Document Number 9302-0001, Sunnyvale, California, November 1991.
- [NiT90] M. Nilsson and H. Tanaka, "MIMD Execution by SIMD Computers," *Journal of Information Processing*, Information Processing Society of Japan, vol. 13, no. 1, 1990, pp. 58-61.
- [PaD92] T.J. **Parr**, H.G. Dietz, and W.E. Cohen, "PCCTS Reference Manual (version 1.00)," *ACM SIGPLAN Notices*, accepted to appear, February 1992.

- [Phi89] M.J. Phillip, "Unification of Synchronous and Asynchronous Models for Parallel Programming Languages" Master's Thesis, School of Electrical Engineering, **Purdue University**, West Lafayette, Indiana, June 1989.
- [SiN90] H.J. Siegel, W.G. Nation, and M.D. **Allemang**, "The Organization of the PASM **Reconfigurable** Parallel Processing System," Ohio State Parallel Computing Workshop, Computer and Information Science Department, Ohio State **University**, Ohio, March 1990, pp. 1-12.
- [Sto84] H.S. Stone, "Database Applications of the Fetch-And-Add Instruction," IEEE Transactions on Computers, July 1984, pp. 604-612.
- [ThG87] S. **Thakkar**, P. Gifford, and G. Fielland, "Balance: A Shared Memory Multiprocessor System," International Conference on **Supercomputing**, May 1987, pp. 93-101.
- [Thi90] Thinking Machines Corporation, Connection Machine Model CM-2 Technical **Summary**, " version 6.0, Cambridge, Massachusetts, November 1990.
- [WiH91] P.A. **Wilsey**, D.A. Hensgen, C.E. Slusher, N.B. Abu-Ghazaleh, and D.Y. Hollinden, "Exploiting SIMD Computers for Mutant Program Execution," Technical Report No. TR 133-11-91, Department of Electrical and Computer Engineering, University of Cincinnati, Cincinnati, Ohio, November 1991.