Compiler Techniques for Loosely-Coupled Multi-Cluster Architectures

Bryan Chow
Scalable Concurrent Programming Laboratory
California Institute of Technology
Pasadena, California 91125

In Partial Fulfillment of the Requirements
for the Degree of Master of Science

June 17, 1996

Acknowledgments

Stephen Taylor, my advisor, whose guidance, comments and criticisms were pivotal in the completion of this work. Daniel Maskit and Rich Lethin for lengthy discussions on the subject, and answering my questions about the Multiflow compiler. Yevgeny Gurevich for providing the runtime system and tools essential for this project, and also for answering my questions and requests in a timely manner, day or night. Stephen Keckler, for help with the M-Machine simulator.

Contents

Chapter 1

Introduction

1.1 Motivation

Recent advances in semiconductor technology have made it possible to have multiple execution units reside on a single chip[17]. These multiple execution units can be used to execute individual machine operations in parallel (instruction-level parallelism). However, it is an open question as to how to best organize the units in order to exploit instruction-level parallelism. The traditional method for exploiting instruction-level parallelism has been the use of Very Large Instruction Word (VLIW) architectures. A VLIW processor has multiple functional units and a single long instruction encodes many operations, typically one for each functional unit, as illustrated in Figure 1.1 (a). These operations execute in parallel and in lock-step. It is the responsibility of the compiler to ensure that there are no dependencies between operations. There have been a number of commercial VLIW implementations[15] [4].

Unfortunately, the classic VLIW approach has its disadvantages. The increasing disparity between clock cycle and memory latency limits the performance of VLIW, since each time a single functional unit stalls, the whole machine stalls. Also, the amount of instruction-level parallelism in a program is inherently limited due to dependencies in the code, the limited number of registers, and other factors.

An alternative method is the use of loosely-coupled clusters. Each cluster, consisting of a subset of the functional units, has its own instruction pointer as shown in Figure 1.1 (b). Instead of having all the functional units in the machine synchronized at every cycle, clusters are allowed to “slide” in relation to each other. When one unit stalls, others are unaffected. This thesis concerns the investigation of this alternative architectural model and compiler techniques for this model.

An example of this organization is the experimental massively parallel computer, the M-Machine, being designed by the Concurrent VLSI Architecture Group at the Massachusetts Institute of Technology.

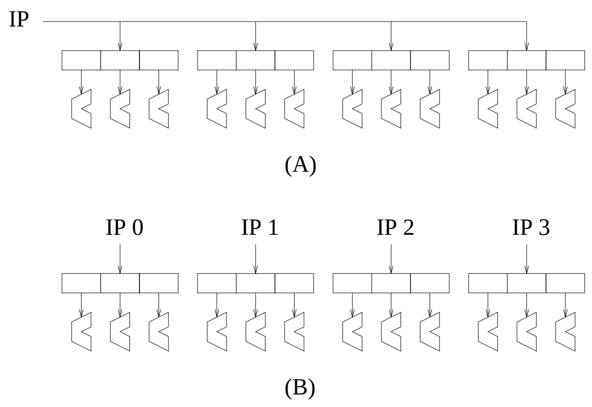

Figure 1.1: Cluster comparison. A) A classic VLIW with a single instruction pointer, and all functional units execute in lockstep. B) A loosely-coupled architecture with an instruction pointer for each cluster.

1.2 Compilation Issues for Loosely-Coupled Clusters

Keckler[17] has shown that loosely-coupled clusters provide better latency hiding and functional unit utilization than a classic VLIW architecture, especially when memory latencies can be variable. On an architecture such as the M-Machine in which the memory latency can vary from 3 to 32 cycles or worse if a memory operation misses in the translation lookaside buffer and requires software handling, loose coupling appears to be a good alternative.

Unfortunately, loose-coupling complicates the job of the compiler since it can no longer determine the state of the machine at each cycle. In particular, code scheduling becomes more difficult because the full state of the machine (including resource utilization) is not known at compile-time.

Clusters may exchange information by register transfers. Communication between clusters is complicated by the fact that one cluster is not able to predict the point of execution of another cluster. Hence synchronization is required. Additionally, any instruction on a cluster can take a variable number of cycles. In other words, if cluster 0 executes 2 instructions, it is not known how many instructions cluster 1 has executed in that same period of time. Again, this complicates communication, performance optimization, and also impacts correctness.

Finally, since clusters communicate via shared registers and block when they are empty, deadlock may occur. This will happen if clusters get out of synchronization with each other. Therefore the compiler has to ensure that deadlock does not occur by implementing communications in such a way as to make any possible execution of the schedule free of deadlock.

1.3 Goals

The goals of this research are:

1) To investigate compilation techniques for loosely-coupled architectures. The M-Machine is the first computer to use this concept.
2) To implement a compiler that generates multi-cluster code, and to integrate multi-cluster compilation with the assembler, linker, and other tools for the M-Machine system.
3) To evaluate the performance of code generated by the multi-cluster compiler by running compiled benchmarks using the M-Machine simulator.
4) To propose refinements, optimizations, and future directions in multi-cluster compilation.

1.4 Approach

In order to bootstrap on existing technology, the Multiflow Compiler was chosen as the substrate for the M-Machine compiler. The Multiflow Trace Scheduling Compiler is a compiler for VLIW architectures that provides an ideal basis for experimentation. This compiler has been retargetted to compile code for the M-Machine. This thesis concerns additions to the compiler to support loosely-coupled clusters.

1.5 Contributions

The contribution of this thesis is a preliminary evaluation and experiment in compiling for loosely-coupled multi-cluster architectures. This experiment serves to asses opportunities for multi-cluster scheduling and present a preliminary design for a multi-cluster scheduling algorithm. This exercise has isolated some of the central issues and allowed an analysis to a preliminary implementation of multi-cluster scheduling. It has also resulted in changes to the architectural design.

Chapter 2

Compiler Philosophy

2.1 M-Machine Architecture

The M-Machine[6] is a massively parallel computer being designed at the Massachusetts Institute of Technology. It consists of a collection of nodes connected by a 3-D mesh network. Each node on the M-Machine consists of memory and a multi-ALU processor (MAP) chip, each of which contains 4 clusters, a network router, network interface, internal cache and communication switches. This is shown in Figure 2.1.

A cluster is an execution unit that has its own program counter. It includes two integer ALUs (one of which also acts as the memory unit) and a floating-point ALU, register files, and instruction cache. Clusters communicate with other clusters through shared registers. Each cluster has two kinds of registers - public and private. Private registers may only be written by operations executed within the cluster. Public registers can be written to by all clusters but read only by one, and are the basis for inter-cluster communication and synchronization on the M-Machine.

Each register has an associated presence bit. There is an instruction that marks a register EMPTY. Reading a register that is marked EMPTY blocks the cluster until it is filled (marked FULL) by another cluster or an earlier instruction on the same cluster.

The M-Machine also supports conditional execution through the use of conditional registers. Each node has 8 global conditional registers which may be read by any of the 4 clusters, but each cluster can only write to 2 of the global conditional registers. Conditional execution is useful for generating code for simple if...then statements. Since conditional registers can also be FULL or EMPTY, they are convenient for implementing synchronization primitives such as barriers.

2.2 Loosely-Coupled Clusters vs VLIW

In a VLIW architecture, all the functional units work in lock-step, with a single instruction counter. The functional units never get out of synchronization with each other, which simplifies compile-time scheduling. This allows the compiler to know exactly what state the system is in at each cycle.

On the M-Machine the clusters are loosely-coupled. Each cluster has its own instruction

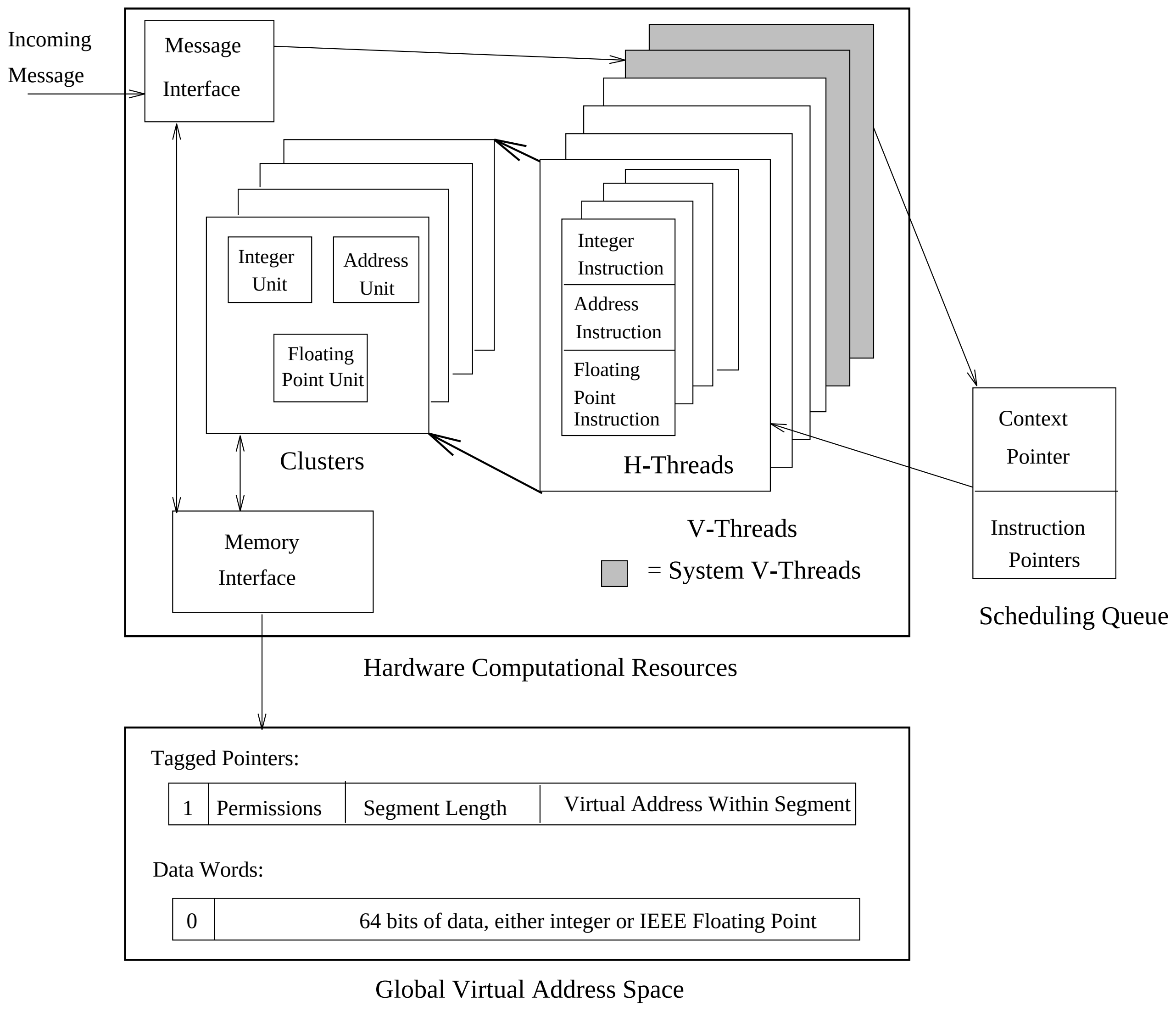

Figure 2.1: Abstract view of the MIT M-Machine.

pointer and can execute independently of other clusters. This is intended to provide better hiding of latency, hence better functional unit utilization. Instead of stalling the whole node on a cache-miss, for example, only a single cluster needs to stall while the others can continue execution. Also, the M-Machine has a non-blocking memory system with variable latencies which makes it impossible to predict exactly the number of cycles a memory operation can take.

As the speed of processors increase and their size decrease, loose coupling becomes more attractive. We can pack many loosely coupled functional units on a chip and hide latency better than other methods such as VLIW.

2.3 Multiflow Trace Scheduling Compiler

The compiler used for the M-Machine is the Multiflow Trace Scheduling Compiler[15], which was designed to exploit instruction-level parallelism for traditional VLIW architectures.

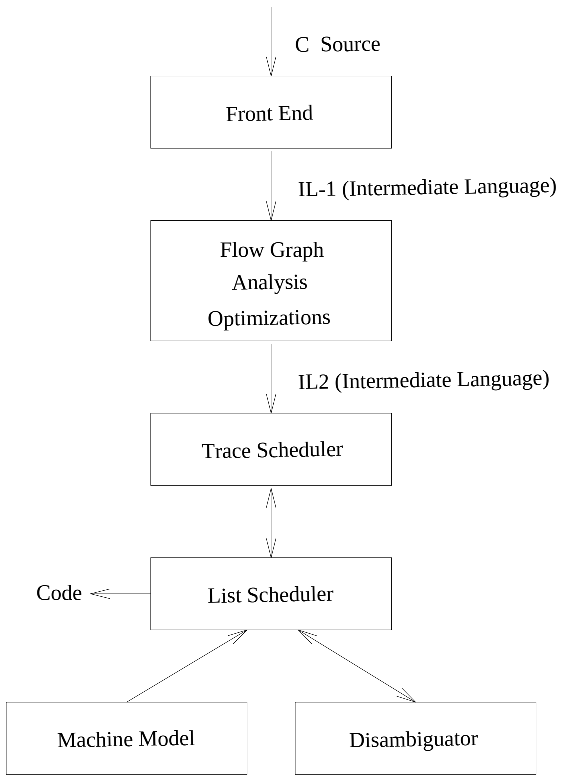

Figure 2.2: Structure of the Multiflow compiler

Figure 2.2 shows the structure of the Multiflow compiler. The front end accepts C and Fortran source code and converts it into a high-level intermediate representation called IL-1.

Traditional code optimizations such as loop unrolling, constant folding and common subexpression elimination are applied to this intermediate representation. It is then converted to a flow graph or DAG (directed acyclic graph) of lower level operations called IL-2 operations.

This flow graph is then fed into the trace sheduling module (described in the next section). The traces are then scheduled onto functional units by the list scheduler, which receives information on the target architecture from the machine model and generates code. The disambiguator performs memory-reference analysis.

2.4 Trace Scheduling

Trace scheduling[8] is an algorithm that allows instruction scheduling beyond basic blocks, so that more operations can be considered for scheduling and hence provide a greater amount of parallelism. It allows loops, conditional expressions, and straight stretches of code to be handled in a consistent and uniform manner. A trace is a linear sequence of basic blocks that can potentially be a path of execution.

The trace scheduler first annotates the flow graph with expected execution frequencies. These are generated using branch probabilities and loop trip counts, taken from heuristics or runtime measurements. With the execution frequencies in place, the trace scheduler then:

1) Selects the trace through the graph with the highest execution count that hasn't yet been scheduled.
2) Removes this trace from the flow graph, and schedules it. By optimizing the most likely execution path first, it ensures that the most frequently executed path is the most optimized.
3) Replace the original trace in the flow graph with the scheduled trace. If instructions have been moved such that the flow graph is no longer correct (for example, if an instruction is moved above a branch), add compensation code to correct it.
4) Repeat the above until all traces through the flow graph have been scheduled.

By looking at an entire trace instead of a basic block, a great deal of parallelism is exposed. The instruction scheduler can freely move instructions around to utilize as many functional units of the VLIW as possible. However, to preserve correctness, it is necessary to insert compensation code. This is called bookkeeping. For example, the instruction scheduler may decide that moving an instruction earlier in the schedule can reduce the execution time. In Figure 2.3, an instruction is moved above a branch target. To ensure that the moved instruction is always executed, the bookkeeper makes a copy of the moved instruction onto the incoming path.

In Figure 2.4, an instruction is moved below a branch target. In this case, we need to ensure that those instructions are *not* executed if we arrived from the side entrance. This is done by making a copy of the instructions that will be executed if the side entrance is taken, and moving the side entrance to join at a later point in the schedule as shown in the figure.

2.5 Differences between the M-Machine and TRACE VLIW Computer

The Multiflow TRACE computers[15], for which the compiler was originally designed, all share a common set of features. The differences between the M-Machine and TRACE that are signif-

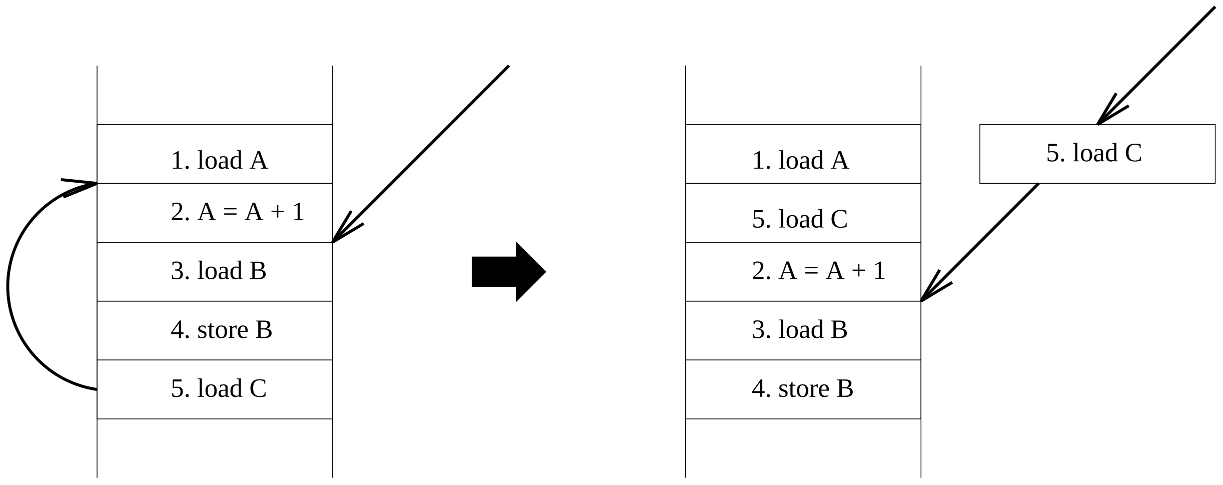

Figure 2.3: Moving an instruction above a side entrance.

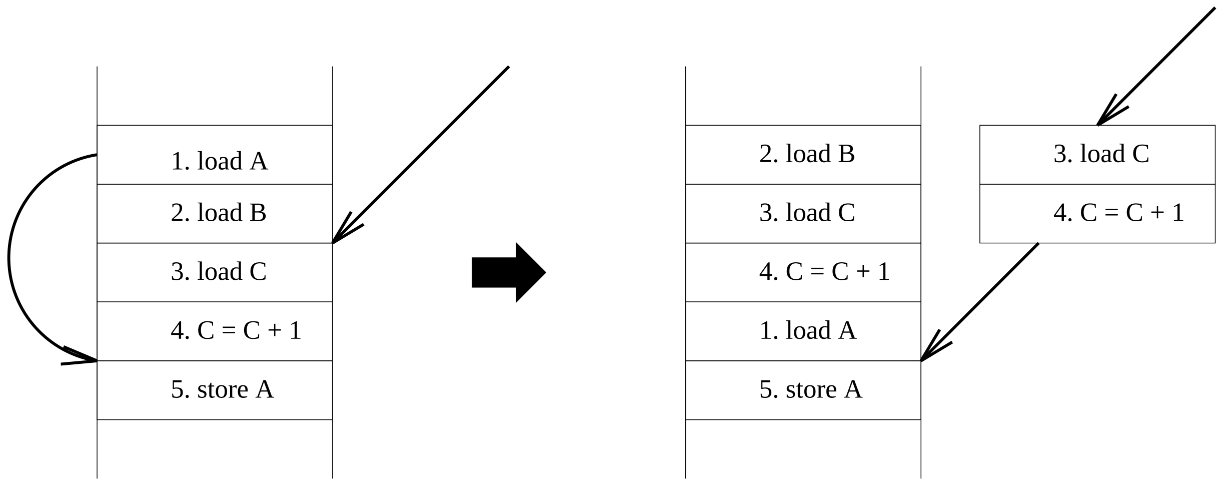

Figure 2.4: Moving an instruction below a side entrance.

icant in our retargetting effort are[1]:

1) TRACE computers were 32-bit machines, the M-Machine is a 64-bit machine. This necessitated many changes in the code generator, front-end, and optimizer.
2) Although TRACE computers were also multi-clustered (because the clock cycle was too short to propagate the instruction pointer to all the clusters), all the clusters are strongly coupled and always execute in lockstep.[2]
3) The TRACE computers did not have data caches, but instead relied on a two-level interleaved memory hierarchy. The absence of a cache made it easier to schedule memory operations at compile-time. The M-Machine has an integrated cache at each node.
4) The TRACE computers were completely deterministic in their execution. The number of cycles a memory fetch will take can be predicted precisely. On the M-Machine, however, due to the unpredictability of network latency, cache hits and misses, and the different threads executing, the execution cannot be fully predicted at compile-time. Running the same program at different times may result in slightly different execution times. This makes the next point extremely important.
5) The TRACE computers do not do any resource arbitration in hardware, so resources can be oversubscribed which will lead to incorrect executions. Therefore it is crucial that the compiler model the cycle-by-cycle state of the machine precisely. The M-Machine has hardware resource arbitration, thus the compiler only needs to model what is necessary for good performance.

[1] We are only concerned with one node of the M-Machine here.

[2] It is possible to decouple the clusters, in which case you end up partitioning the machine into independent computers.

Chapter 3

Implementation

3.1 Implementation using Multiflow Compiler

The first step in implementing multi-cluster support in the compiler was to expand the machine model. With a single cluster, there are integer (IALU), floating point (FALU) and memory (MEMU) functional units, all of which can write to each other's register banks (the IALU and MEMU share a single register bank), as shown in Figure 3.1

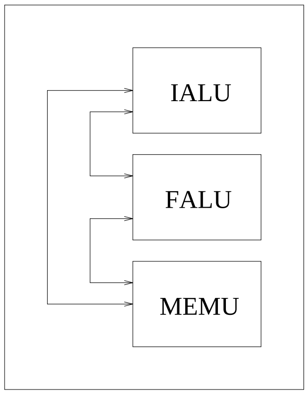

Figure 3.1: Connections within a single cluster.

Beginning with the single cluster machine model, 4 clusters were instantiated. Next, the correct relationships between the different sets of register files and functional units were modeled. Each cluster has private and public registers. Private registers are only visible to the cluster itself. All clusters can write to all public registers, but only the owning cluster can read from its public registers.

An additional resource that models the communication switch (c-switch) was created that restricts the number of communications that could proceed at once. Figure 3.2 gives a simplified diagram of the expanded machine model. Communication delays between each pair of register files and functional units were modeled to enable the instruction scheduler to allocate registers and schedule communications efficiently.

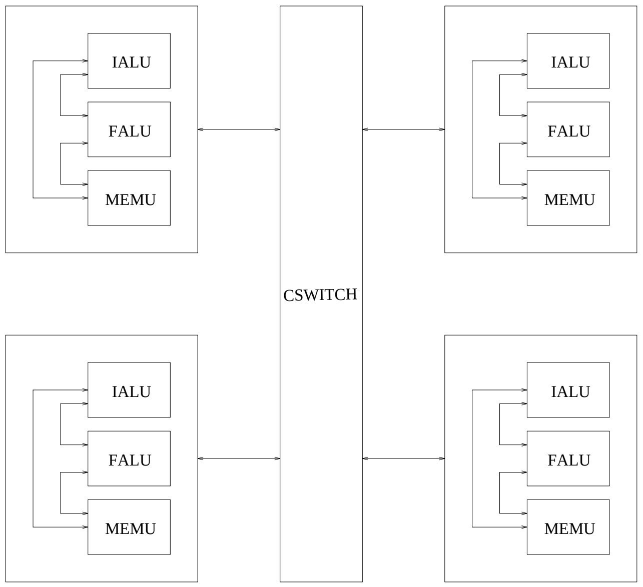

Figure 3.2: Connections within a 4-cluster node.

In the code generator, code sequences were written to enable register moves and copies among different clusters. For example, most ALU operations can write their destination to a different cluster. When this happens, additional synchronization operations are added.

3.2 Initiating a Multi-Cluster Program

The M-Machine runtime by default executes on a single cluster [1]. In order to execute a multi-cluster program, a stub is linked with the multi-cluster code. The source for the stub is given in Figure 3.3.

This stub replaces the entry point of the multi-cluster program (the function _main). It calls the system to spawn a thread on each clusters which starts executing the code for that cluster.

3.3 Communication and Synchronization

One of the main challenges of compiling for loosely-coupled clusters as opposed to VLIW is to manage communication and synchronization to ensure correctness and absence of deadlock. During a cross-cluster register copy the target cluster must be at the right point in its execution to receive the register. If not, the source cluster has to wait for the target cluster to be ready. This is achieved by setting up a barrier synchronization between the two clusters.

Synchronization is also required when one operation is constrained to execute only after another operation has executed. These are constraints on the order of the operations, but are not

[1] In the case of a single node.

```
#include <stdio.h>

int hspawn(int numargs, void *threadip, int dest_node, ...);

extern main_c0();
extern main_c1();
extern main_c2();
extern main_c3();

void main() {

    i = hspawn(0, main_c1, 1);
    if (!i) printf(``hspawn 1 failed\n'');
    i = hspawn(0, main_c2, 2);
    if (!i) printf(``hspawn 2 failed\n'');
    i = hspawn(0, main_c3, 3);
    if (!i) printf(``hspawn 3 failed\n'');
    main_c0();

}
```

Figure 3.3: Implementation of **stub.c**.

actual data dependencies. For example, operations are often constrained such that they are executed before a branch operation. The Multiflow compiler handles such constraints by ensuring that one operation happens in a later cycle than the other. For a VLIW machine, that is sufficient. However, on the M-Machine, if the constrained operations are placed on different clusters, a barrier has to be set up between the two clusters.

The main challenge is to communicate and synchronize as cheaply in terms of execution cycles as possible. Because register communications on the M-Machine are designed to be very fine-grained, the communication latency should not be larger than a few machine cycles.

Communication is organized by using two-phased synchronization operation. There is a setup phase and a communicate phase, separated by a barrier.

3.4 Managing Synchronization

Clusters communicate via shared registers. A cluster can write to any shared register of the other 3 clusters on the same node. Consider an instruction running on cluster 0 that copies register 8 to register 10 of cluster 2:

```
instr   cl0 ialu mov    i8, c2.i10;
```

The keyword “instr” informs the assembler that this is a new instruction. The segment “cl0 ialu” signifies the instruction is to be executed on the integer ALU on cluster 0. The instruction moves integer register 8 (i8) to integer register 10 on cluster 2 (cl2.i10).

The communication via shared registers appears to be computationally inexpensive. Unfortunately, because each instruction on a cluster can take a variable number of cycles in relation to another cluster, barrier synchronizations are required in order to properly communicate via shared registers. The following code segment illustrates how two clusters cooperate when one copies a register to another. It assumes that all global conditional registers are empty on entry:

```
0. instr cl1 ialu empty  i10
1. instr cl1 ialu ieq i0,#0, c0.cc1
2. instr
3. instr
4. instr cl0 ialu ct c1.cc1 ccempty c1.cc1
5. instr cl0 ialu mov i8, c1.i10
```

At cycle 0 the destination register on the target cluster is emptied. On cycle 1 the target cluster sets the conditional register to signal that it is ready to accept the register. There is a two cycle latency before this write reaches other clusters, therefore the earliest cycle for scheduling the instruction at the source cluster that waits on the conditional register is cycle four. In the fifth cycle the actual move takes place. It takes an additional two cycles before the register is filled in the destination register. If an attempt is made to use the register before it arrives at the target cluster, it will stall.

Note that “cl0 ialu ct c1.cc1 ccempty c1.cc1” waits for c1.cc0 to be filled, and then empties it again. This ensures that the next synchronization will again start with the global conditional register (GCC) empty. In other words, it acts as a test-and-set operation.

If the destination register is not used before a control flow change (a branch) in the schedule, then an extra instruction “mov i10, i0” is required. This is because it is possible that the control flow takes a path which does not use the received i10 register. If that happens, the register can arrive later and overwrite i10 which may already contain another value. However, in most schedules the use of the received register follows the cross-cluster register copy closely, so the additional instruction is not required.

Each communication takes 2 instructions at the source cluster and 3 at the destination cluster. There are also stall cycles due to the inter-cluster copying of the global conditional register and the data register. Additionally, one cluster will stall at the barrier waiting for the other cluster to reach the same point. This represents a tremendous amount of overhead for a single register copy which would have taken 1 cycle on a VLIW machine.

3.5 Deadlock Avoidance

Each pair of clusters has its own dedicated “channel” (a GCC register) for synchronizing communications. To prevent deadlock, a cluster cannot be communicating with two clusters at the same time. In other words, a cluster must finish communicating with one cluster before it can initiate another cross-cluster communication, and each pair of clusters communicate using a dedicated channel. This prevents any cyclic dependencies in the communication structure which might result in a deadlock situation.

Conditional branches which require all four clusters to synchronize use a separate set of channels (also GCC registers). A pair of communicating clusters have to complete their communications before initiating a conditional branch. This is implemented by ensuring that cross-cluster communications instructions do not overlap branch instructions.

3.6 Handling Branches and Function Calls

Each cluster has its own instruction pointer and can in fact execute completely independently. However, when executing as loosely-coupled clusters it is intended that the clusters follow the same branch sequences and simply "slide" in relation to one another.

This requires that both conditional and unconditional branches be taken (or not taken) by all clusters together. They may not actually take the branch on the same cycle since it is desirable to allow a faster cluster to branch earlier if possible, but eventually they will all execute the same sequence of branches. For unconditional branches, each cluster can branch as soon as it is ready. For conditional branches, all the clusters have to share the branch conditional register to decide whether to take the branch. Therefore all clusters have to barrier synchronize at the branch instruction.

The following code sequence illustrates conditional branching using the conditional register on cluster 0:

```
instr   cl0 ialu cf c0.cc0 br LABEL_c0          // All 4 clusters
        cl1 ialu cf c0.cc0 br LABEL_c1          // inspect c0.cc0
        cl2 ialu cf c0.cc0 br LABEL_c2
        cl3 ialu cf c0.cc0 br LABEL_c3

instr   cl0 falu fmula f13, f14, f15, f0        // Cluster 0 waits
                                                // for other clusters
                                                // to finish

        cl1 ialu ccempty c0.cc0                 // Empty c0.cc0 for
        cl2 ialu ccempty c0.cc0                 // next branch
        cl3 ialu ccempty c0.cc0

instr   cl1 mov f0, c0.f13                      // Clusters 1,2,3
        cl2 mov f0, c0.f14                      // tell cluster 0
        cl3 mov f0, c0.f15                      // they are ready

        cl0 falu empty c0.f13, c0.f14, c0.f15   // Empty these on
                                                // cluster 0 for
                                                // next branch
```

Floating point registers f13, f14 and f15 are reserved solely for synchronizing conditional branches to avoid deadlock. The second and third instructions above fit in the delay slots of the branch instruction, and thus are executed regardless of which way the branch takes. They empty the conditional register on each cluster to prepare for the next conditional branch. The cluster where the branch originates waits until the other clusters have emptied their conditional registers

```
--mark_begin;
--mark_entry caller_saves, regs;
_main::
                    --  Begin schedule(1) aregion(1)
--mark_trace 1;
instr   cl2 ialu ash    i0,i0, i6  -- line 22
        cl0 ialu mov    #0, i10;  -- 0(I64)

instr   cl0 ialu imm    ##(?2.1?2ras_p), i3
        cl1 ialu mov    #0, i9  -- 0(I64)
        cl2 ialu mov    #0, i9  -- 0(I64)
        cl3 ialu mov    #0, i9;  -- 0(I64)

instr   cl0 ialu lea    i2,i0, i11  -- line 11
        cl1 ialu mov    #10, i4;  -- 10(SI64)
        cl2 memu lea    i2,i3, i3
        cl3 memu lea    i2,i3, i3;

instr   cl0 memu st     i4, i3  -- sp/re t18
        cl0 ialu mov    #10, i4;  -- 10(SI64)

instr   cl0 ialu jmp    i4;

--mark_return;
                    --           End schedule(1) aregion(1)
--mark_end;
```

Figure 3.4: Sample multi-cluster compiler output. The compiler generates code annotated with the cluster number for each instruction.

before proceeding. Therefore, the minimum cost of a multi-cluster conditional branch is the two delay slots, plus the cycles required for a cross-cluster register write (for the “mov f0, c0.freg” to reach the originating cluster).

Most system calls are implemented as single-cluster code, so when a multi-cluster program makes such a call, only one cluster actually branches. The other clusters will continue to execute their own code without taking the branch, until they attempt to communicate with the cluster that took the branch, in which case they will stall.

Currently, all function calls are assumed to be single-cluster.

3.7 Integrating with the Linker and Assembler

The M-Machine Assembler and Linker were written for single cluster code.

A compiler post-pass converts the multi-cluster code generated by the compiler into four sets of single cluster code, each of which executes on a cluster.

As an example, Figure 3.4 gives a sample multi-cluster compiler output (the code is only meant to illustrate the post-processor). Figure 3.5 is the output after the compiler post-processor.

```
--Start of cluster 0 code
--mark_begin;
--mark_entry caller_saves, regs;
_main_c0::
    --  Begin schedule(1) aregion(1)
--mark_trace 1;
instr ialu mov    #0, i10;  -- 0(I64)
instr ialu imm    ##(?2.1?2ras_p), i3;  -- 0(I64)
instr ialu lea    i2,i0, i11  -- line 11;  -- 10(SI64)
instr memu st     i4, i3  -- sp/re t18
ialu mov    #10, i4;  -- 10(SI64)
instr ialu jmp    i4;
--mark_return;
    --           End schedule(1) aregion(1)
--mark_end;


--Start of cluster 1 code
--mark_begin;
--mark_entry caller_saves, regs;
_main_c1::
--mark_trace 1;
instr ialu mov    #0, i9;  -- 0(I64)
instr ialu mov    #10, i4;  -- 10(SI64)
--mark_return;
--mark_end;


--Start of cluster 2 code
--mark_begin;
--mark_entry caller_saves, regs;
_main_c2::
--mark_trace 1;
instr ialu ash    i0,i0, i6;  -- line 22
instr ialu mov    #0, i9  -- 0(I64)
memu lea    i2,i3, i3;
--mark_return;
--mark_end;


--Start of cluster 3 code
--mark_begin;
--mark_entry caller_saves, regs;
_main_c3::
--mark_trace 1;
instr ialu mov    #0, i9;  -- 0(I64)
memu lea    i2,i3, i3;
--mark_return;
--mark_end;
```

Figure 3.5: The same program after being post-processed. Each routine is separated into 4 parts, each of which executes on a single cluster.

This output is then assembled just like a single-cluster program, linked with the stub described earlier, and can then be loaded by the runtime system like any other single-cluster program. When executed, it spawns multiple hthreads and starts executing in parallel across multiple clusters.

Chapter 4

Case Studies

This chapter investigates the performance of the prototype Multiflow compiler in compiling a few representative programs on a single M-Machine cluster and on four clusters. The intent here is to establish a baseline of performance using the standard Multiflow scheduler with minimal changes. These examples demonstrate the performance of multi-cluster programs in the current implementation of the compiler and the factors affecting their performance. They also signify problems with the current instruction scheduler in compiling for loosely-coupled clusters and give insights for developing alternative algorithms.

Four programs are considered:

1) Matrix Multiplication (matmul)
2) Dirichlet Algorithm (dirichlet)
3) Sieve of Erastothenes (sieve)
4) LU Factorization (lu)

The programs are sequential and run on only 1 node. The timings in each program do not include the output portion of the execution.

4.1 Study 1: Matrix Multiplication

The matrix multiplication program (matmul) takes two 8 by 8 floating point matrices and multiplies them, then outputs the result. The main body of the code is given below:

```
for (column = 0; column < MATSIZE; column++) {
  for (j = 0; j < MATSIZE; j++) {
    total = 0;
    for (i = 0; i < MATSIZE; i++)
      total += matrix1[column][i]*matrix2[i][j];
    final[column][j] = total;
  }
}
```

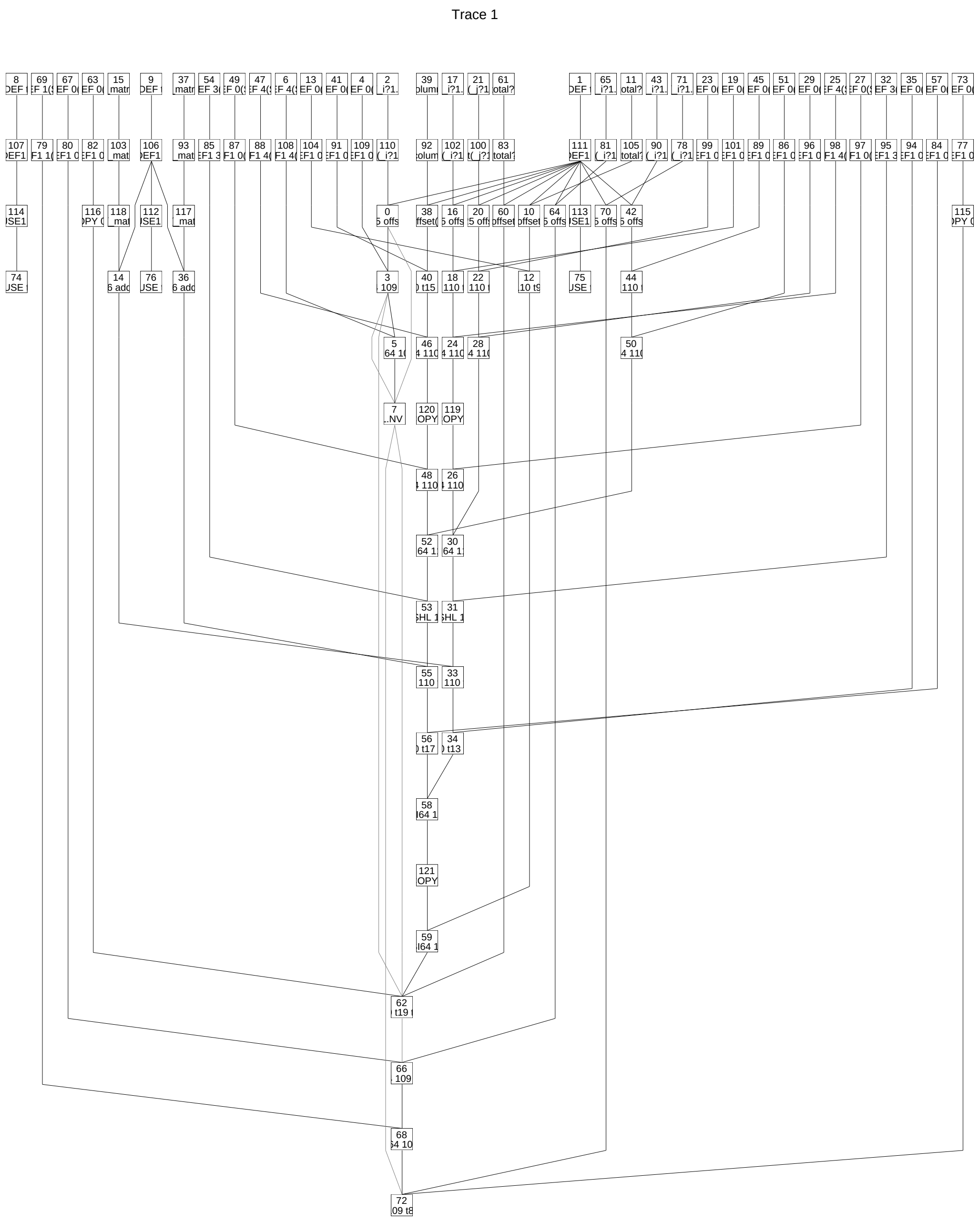

Figure 4.1: DAG for matmul.c. The solid edges represent data dependencies, and the dotted edges represent constraints in the order of operations. Each node is an IL2 operation.

The DAG for the trace with the highest frequency count is given in Figure 4.1. The DAG is labelled with IL2 operations, which is the compiler's intermediate language. Each IL2 operation expands to one or more M-Machine operations.

There are two cross-cluster register copies in the innerloop of the algorithm. A barrier is required for each iteration of the second nested loop. Except for those synchronization points, only the conditional branches require synchronization.

The number of cycles required is given in the following table:

Number of Clusters	Cycles
1	35230
4	24812

Running on a single cluster, *matmul* took 35230 cycles to calculate the product of two 8 by 8 matrices. With 4 clusters, the cycle count dropped to 24812. This is an improvement of 30% over the single cluster code. The innerloop of *matmul* is simple and does not require many registers, which reduces the number of cross-cluster register copying.

Potentially conflicting memory writes executing on different clusters require barriers and hence reduce the performance. There is only a single memory write in the body of the innerloop (not including the loop index variable), hence no additional barriers are required. In other words, there are few operations that are constrained across multiple clusters in each iteration of *matmul*. These factors lead to the speedup seen in the execution of *matmul*.

Notice that there is only a single line in the innerloop of *matmul*, relatively few operations are executed for each iteration. This reduces the potential speedup because the overhead for loops and branches for multiple clusters is significantly higher than for a single cluster. In the single cluster execution, branches took 2444 cycles to execute, or 7% of the total execution time. For 4 clusters, branches consumed approximately 4133 cycles or 17% of the total execution time.

4.2 Study 2: Dirichlet

The Dirichlet program implements a gauss-seidel algorithm to compute $\nabla^2 U = 0$ and returns $norm = abs(U_t - U_{t-1}))/N$. It calculates 10 iterations of the Dirichlet algorithm.

The algorithm for a single iteration or timestep[1] is shown in Figure 4.2.

The DAG for the trace with the highest frequency count is given in Figure 4.3.

The number of cycles required is given in the following table:

Number of Clusters	Cycles
1	58134
4	59453

[1] The routine below is actually inlined in the test program.

```
/*
 * timestep()
 *   Computes one timestep of gauss-seidel algorithms to compute
 *   Del^2 U = 0 and returns norm = abs(Ut-Ut-1))/N
 */
static void timestep(bp)
BlockP bp;
{ double u1,u2,anorm,temp;
  int i,j,iend,jend;

  iend=imax(bp)-1;                          /* Find upper bndries     */
  jend=jmax(bp)-1;
  anorm=0.0;                                /* Initialize norm@time   */
  for(j=1; j<jend; j++)                     /* Sweep vertically       */
    for(i=1; i<iend; i++) {                 /* Sweep horizontally     */
      u1=block(bp,i,j);                     /* Save old u(i,j)        */
      u2=                                   /* Compute new u(i,j)     */
        (block(bp,i+1,j)+block(bp,i-1,j)+   /* Add i neighbors to     */
         block(bp,i,j+1)+block(bp,i,j-1))/4.0;/*  j neighbors & average */
      block(bp,i,j)=u2;                     /* Store new u(i,j)       */
      temp = u2-u1;
      anorm += (temp>0?temp:-temp);         /* Accumulate norm        */
    }
  norm(bp)=anorm/(float)(imax(bp)*jmax(bp));  /* Compute new norm       */
  tb(bp) = tb(bp)+1;
}
```

Figure 4.2: Dirichlet Algorithm

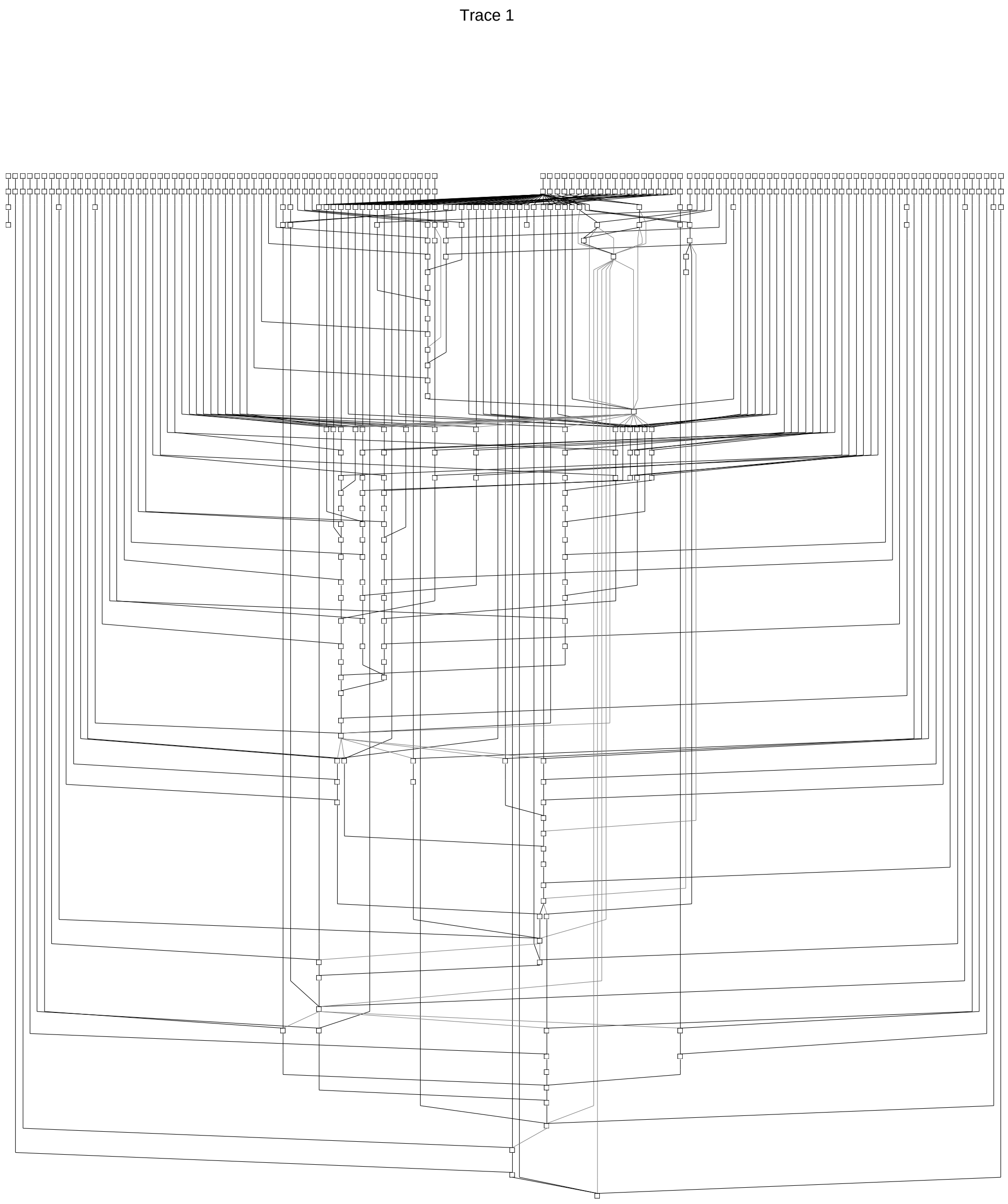

Figure 4.3: DAG for dirichlet.c

In this case executing on all 4 clusters is 2% *slower* than executing on a single cluster.

Compared with *matmul*, the innerloop of *dirichlet* requires many more variables and hence there is a large number of cross-cluster register copying. The scheduler tries to move the variables to each cluster as needed, resulting in excessive communications and synchronizations. There are 23 cross-cluster register copies in the innerloop of *dirichlet* compared to 2 for *matmul*. Even taking into account the fact that the innerloop for diricihlet contains about five times more operations than *matmul*, this is a large number of communication operations.

Additionally, there are many dependencies in the code. For example, the writes to u1 and u2 must complete before block[i,j] is updated. The same goes for the write to temp, and anorm must in turn wait for temp to be written. These writes cannot all be scheduled on the same cluster (or the schedule will be too poorly balanced across clusters) so barriers are required. Since cross-cluster register copying includes a pairwise barrier, only one additional barrier was required in each iteration of the innerloop.

However, even with the above factors limiting performance, a good scheduler should still be able to extract some parallelism and generate a code schedule that is faster than in single-cluster mode.

4.3 Study 3: Sieve of Erastothenes

The sieve program implements the sieve of Erastothenes to find all the prime numbers less than 1000 (using an array size of 500).

The number of cycles required is given in the following table:

Number of Clusters	Cycles
1	43591
4	39411

In this case running on four clusters is ten percent faster than a single cluster.

```
if(!array[i])
  {
    count++;
    prime = i + i + 3;
    for(k = i + prime ; k<SIZE ; k+=prime)
      {
        ci++;
        array[k] = 1;
      }
  }
```

Examining the innerloop of *sieve,* we see that it consists of just a single increment operation and an array set operation. Including the loop operations, the innerloop consists of just 20 instructions. Because of the small number of operations in the innerloop, it is difficult for the compiler to schedule operations evenly across the clusters.

4.4 Study 4: LU Factorization

lu uses Crout's algorithm without pivoting to factorize a 16 by 16 matrix.

The number of cycles required is given in the following table:

Number of Clusters	Cycles
1	91128
4	87574

In this case running on four clusters is only four percent faster than a single cluster.

This is because *lu* requires a large number of loop iterations, while each iteration is only a single line of code. The high cost of conditional branches in multi-cluster execution negates most of the performance gained from the additional functional units.

4.5 Conclusions

The four examples show that the performance gain in multiple clusters varies a great deal, from significant speedup to no speedup.

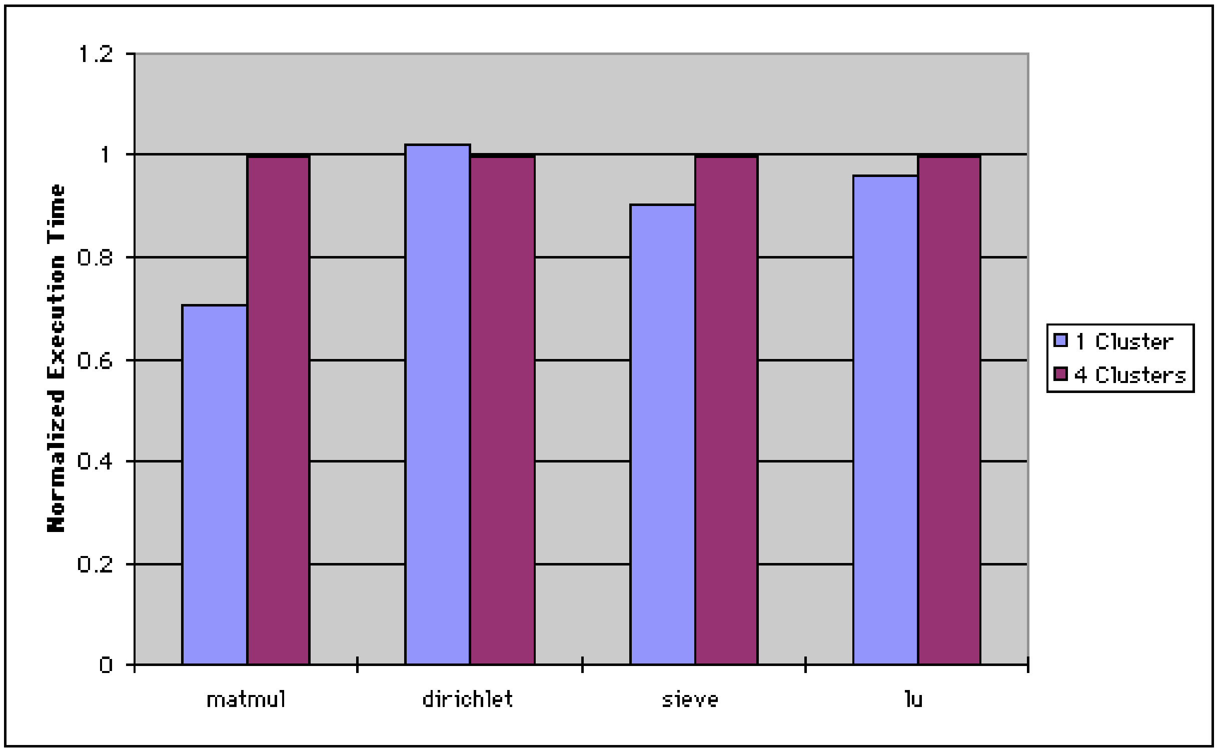

Figure 4.4: Speedup of the example programs.

One of the reasons for the poor performance is that conditional branches are inefficient in multi-cluster mode. By profiling *matmul* we see that the amount of time spent executing branches as a percentage of the total execution time increased two and a half times in multi-cluster mode compared to single cluster mode. There are some possible optimizations that can potentially reduce the cost of multi-cluster conditional branches, such as:

1) Duplicating the computation of the conditional on each cluster. This reduces the conditional branch on each cluster to a single cluster conditional branch that does not require synchronization.
2) Allowing a single cluster to execute a branch if the number of operations that is executed in that branch path is short.
3) Using conditional execution instead of branches if the number of operations is small.

Another reason is that the list scheduler, which is responsible for scheduling operations onto clusters, was not designed for large discrepancies in the latencies between different functional units. List scheduling works well when the communication cost is zero, as in most VLIW machines. Graham [9] showed that in such cases any list scheduler will be within 50% of optimum, and Adam, Chandy and Dickson[1] showed experimentally that the critical path list scheduling heuristic is within 5% of the optimum 90% of the time. However, when the communication cost is nonzero, the performance of list scheduling degrades significantly. List scheduling is a one-stage method. A one-stage method cannot accurately estimate the critical path because when communication cost is nonzero, the edge weights in the DAG are no longer deterministic before the cluster assignments have been determined. If an edge falls within the same cluster it has zero cost, but across clusters it has nonzero cost.

The list scheduler uses a bottom-up greedy (BUG) algorithm to assign operations to functional units, and in the cases where the operands of the operation have not yet been assigned, it disregards the latencies in moving the data from their sources to the functional unit (a reasonable thing to do, since it does not know where they will come from). This works reasonably well if assigning operations to different functional units do not significantly affect performance. In other words, all functional units of the same type should be roughly equivalent for the algorithm to work well. This is certainly not true in the case of loosely-coupled clusters, where assigning an operation to a functional unit different from the source registers would incur a significant penalty.

Also, the list scheduler is being excessively conservative in assigning operations to clusters. Because assigning operations on other clusters impose a heavy latency penalty, a purely greedy algorithm ends up favoring the current cluster most of the time. This results in the uneven utilization we see in the table. Since the argument registers are initialized on cluster 0, it has the highest number of scheduled instructions.

To perfectly partition a DAG so as to minimize the execution time is an NP-complete problem, but given the strong similarities between DAGs generated by programs, we can take advantage of this similarity and come up with a reasonably good partitioning scheme that runs in polynomial time.

4.6 Proposed Solution: DAG Partitioning

We have seen that greedy list scheduling does not work well in the case of loosely-coupled clusters with large communication overhead. Cross-cluster communications are not minimized, and the balancing of computation across clusters is poor.

A solution to this problem is to partition the DAG to minimize cross-cluster communications and increase utilization. In the next chapter I will propose an algorithm for scheduling loosely-coupled clusters using DAG Partitioning.

Chapter 5

Proposed Multi-cluster Scheduling Algorithm

5.1 Approach to Multi-Cluster Scheduling

The first part of the trace scheduling algorithm, trace picking, selects the most likely paths to be executed within the program and optimize them most heavily. This works at a high level and benefits both VLIW and loosely-coupled clusters. The trace picker, which is guided solely by estimated execution frequencies and not the architecture, is left unchanged.

The second part of trace scheduling, the list scheduling algorithm, schedules the operations of the program and is being redesigned to work with loosely-coupled clusters with longer communication latencies. A new compiler phase is being added that partitions the DAG into clusters. After the partitioning is performed, it is followed by a modified list scheduler that also implements a task ordering algorithm. The new structure of the compiler is illustrated in Figure 5.1.

5.2 DAG Partitioning

Consider a directed acyclic graph (DAG) composed of a collection of nodes connected by directed edges. The nodes represent operations and the edges represent dependencies between the operations. Nodes of the DAG are marked with the cost (in cycles) of the operation, and a list of functional units it can be assigned to. Each edge in the DAG is potentially a communication or synchronization.

In the sample DAG given in Figure 5.2, the LEA (load effective address) operation receives its operand from the DEF B (define) operation, and the MLD (memory load) operation receives its two operands from the result of the LEA operation and DEF A. If all the operations are scheduled on the same cluster, no communcation delays occur. If, however, the DEF A and MLD operations are on cluster 0 while the DEF B and LEA operations are on cluster 1, synchronization is needed and the result of the LEA operation must be sent across clusters.

The goal of DAG partitioning is to minimize the parallel execution time. Assume that the DAGS have the following properties:

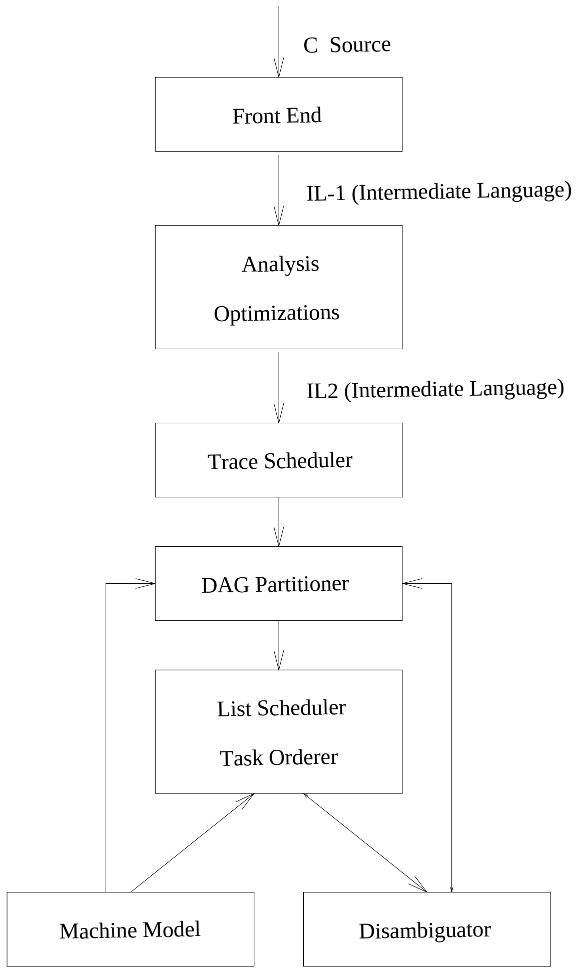

Figure 5.1: Structure of the Multiflow compiler with DAG Partitioning

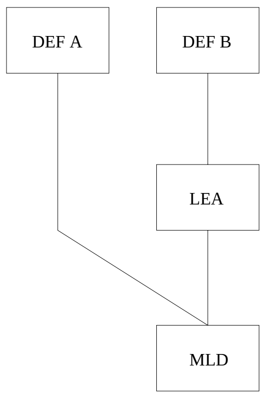

Figure 5.2: A directed acyclic graph (DAG) of IL2 operations.

1) Edges within a cluster have zero cost. This is true except when moving from the integer unit to the floating point unit and vice versa, in which case an explicit move operation MAY be required. Since this does not happen very often and the cost is low when it does, it appears to be a valid simplifying assumption.
2) Edges that cross clusters have fixed cost, which is the latency of the synchronization operations (described later).
3) The cost (latency) of each node is a 3-tuple, one for the integer unit, one for the floating point unit, and one for the memory unit. [1] These are estimated number of cycles the operation requires on each of the functional units.

Since a 3-tuple is used for the cost of each node, the latency will depend on the largest of the three costs. Therefore *max(i_cost, f_cost, m_cost)* is used when balancing a DAG. In almost all cases this will be *i_cost* because of the much larger number of integer operations compared to floating point or memory operations. For example, a memory load instruction requires a few integer operation to set the address, followed by a memory operation.

A good partition is one which results in a low execution time. There are many DAG partitioning algorithms in the literature, such as Sarkar's [18] algorithm and Kim and Browne's linear clustering algorithm [13]. The algorithm is based on Yang and Gerasoulis' Dominant Sequence Clustering (DSC) algorithm[24] for unbounded number of processors. The DSC algorithm combines the best features of several algorithms, and typically gets better partitions than Sarkar's or Kim and Browne's algorithms, and also run in less time. It has a time complexity of *(e + v) log v* where *e* is the number of edges and *v* is the number of vertices or nodes.

The longest path of the scheduled DAG is called the *dominant sequence (DS)* of the DAG. In the DSC algorithm, the DS is identified at each step and the nodes along the DS are placed in the same cluster. An outline of the DSC algorithm is given below. A more detailed description of the algorithm can be found in [24].

A DAG is defined by *G = (V, E, C, T)* where *V* is the set of nodes, *E* is the set of edges, *C* is the set of communication costs (in cycles), and *T* is the set of node costs. Initially, each node is assumed to be in its own cluster and hence *C* contains the cross-cluster communication delay for each edge. During the execution of the algorithm, the graph consists of the examined part *EG* and the unexamined part *UEG*. Initially, all nodes are marked unexamined.

The value *tlevel(n)* is defined as the length of the longest path from an entry node to *n*. *blevel(n)* is the length of the longest path from *n* to an exit node. Thus the parallel time of a scheduled graph is the maximum *tlevel(n) + blevel(n)* where n is any node from the set of nodes. This path is the critical path, or dominant sequence. We also define the priority of a node as *tlevel(n) + blevel(n)*.

The algorithm is given in Figure5.3.

Figure 5.4 shows how DSC works.

The above algorithm assumes an unbounded number of processors. Yang also described a simple but effective cluster merging algorithm which was implemented in the PYRROS[23] par-

[1] The Multiflow compiler is more general and keeps track of operations using resource request lists. However, since the M-Machine has just an IALU, FALU and MEMU on each cluster, it is more general than necessary and we can do a good job with just cycle counts on each functional unit.

```
1. EG = 0, UEG = V.
2. Compute blevel for each node and set tlevel = 0 for each node.
3. Every task is marked unexamined and assumed to constitute one unit cluster.
4. While there is an unexamined node Do
5.         Find a free node n with highest priority from UEG.
6.         Merge n with the cluster of one of its predecessors such that tlevel(n) decreases in a
maximum degree. If all zeroings increase tlevel(n), n remains in a unit cluster.
7.         Update the priority values of n's successors.
8.         UEG = UEG - {n}; EG = EG + {n}.
9. EndWhile
```

Figure 5.3: The Dominant Sequence Clustering algorithm.

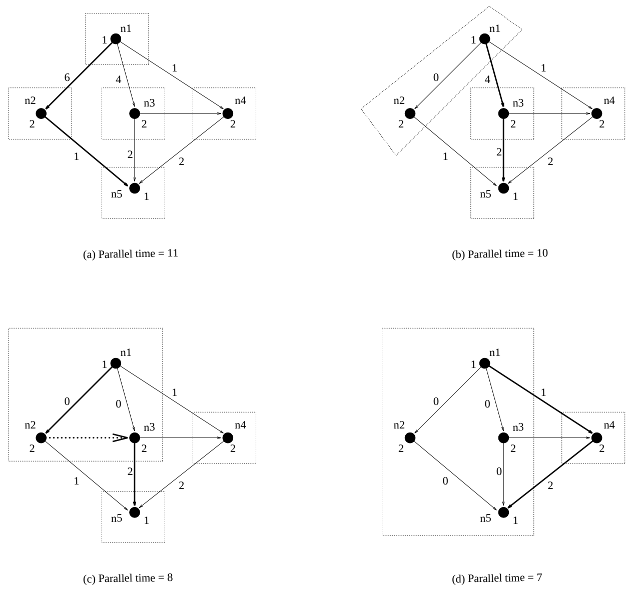

Figure 5.4: DAG Partitioning using DSC. Initially each node is in its own cluster, which results in a parallel time of 11 units. Clusters are merged if it reduces the parallel time, and this leads to the partitioning of the DAG in (d).

allel compiler system. It computes the load for each cluster, sorts them in increasing order, then uses a simple load balancing algorithm to map the u clusters to p processors.

Using this algorithm it should be possible to assign operations to clusters more effectively. Scheduling the tasks for each cluster using the list scheduler can then be achieved as before. This algorithm is currently not implemented in the compiler and details are currently under investigation.

Chapter 6

Conclusion

6.1 Summary

Loosely-coupled clusters have been shown to have the potential to exploit instruction-level parallelism while maintining high function unit utilization[17].

A compiler that generates code for loosely-coupled clusters has been implemented on top of the Multiflow Trace Scheduling Compiler. The performance of this compiler has been evaluated by compiling and executing benchmarking programs on the M-Machine simulator.

Using the execution profiles, bottlenecks have been found and optimizations proposed to improve the performance.

Greedy list scheduling as implemented in the Multiflow Compiler does a poor job of scheduling for loosely-coupled clusters. This thesis investigated the performance of using the greedy instruction scheduler in the Multiflow compiler in compiling for loosely-coupled clusters with large communication latencies, proposed a DAG partitioning algorithm to address some of its problems and described its implementation within the Multiflow compiler. The intent of this work is to establish a baseline for an in-depth investigation.

6.2 Limitations of the Current Implementation

There are a number of limitations in the current implementation of the multi-cluster compiler.

1) There are no multi-cluster function calls. All function calls are assumed to be single-cluster, as necessary for calling runtime routines. The functions in the case studies have all been completely inlined. To correctly implement multi-cluster function calls would require modifying the compiler and linker, and the object-file format to denote whether a function is single- or multi-cluster. The code for multi-cluster function calls will be similar to the code of multi-cluster branches except that the branch target will be a register instead of a constant offset.
2) Constraints in the compiled code are added by hand. If an operation is constrained to occur after another operation, the Multiflow compiler simply schedules them such that one

occurs before the other in their assigned cycles. However, this does not work with multiple clusters since they do not execute in lockstep, hence explicit synchronizations are required.

The only operations that might be constrained across clusters are: control flow changes, cross-cluster communications, and memory reads and writes. The first two are implemented such that they are self-synchronizing, but for memory operations barriers might be required. For example, a write to an address followed by a read of the same address on a different cluster would require a pairwise barrier between the two operations. This is straightforward, but has not yet been added to the compiler.

6.3 Related Work

Loose coupling is a new architecture, and there have not been any available implementations of it: the M-Machine will be the first. However, many research topics are relevant to compiling for loosely-coupled clusters.

Yang[22, 23, 24], SarkarSARKAR89, and Kim and Brown[13] have conducted extensive research on the partitioning of computational graphs.

The work on automatic parallelization, optimizations, interprocedural analysis, and partitioning techniques for parallelizing compilers by Lam[14], Hall[11], Gupta[10] and others are relevant when compiling for loosely-coupled clusters. Loosely-coupled clusters behave similar to processors running in parallel, and many of the techniques in compiling for parallel architectures can be applied to loosely-coupled clusters also.

6.4 Status

The algorithm described for compiling for loosely-coupled clusters is currently being implemented in the Multiflow compiler, together with the rest of the M-Machine software development system and runtime.

The compiler currently can compile simple programs for multiple clusters. The compiled assembly code is then fed through a post-processor that creates separate instruction streams for each cluster, adds the required synchronization to handle branches, and the output is then linked with the multi-cluster stub and executed on the simulator.

The M-Machine is currently being designed and built at MIT. The first prototype is expected to be built during winter 1997.

Bibliography

[1] T. Adam, K. M. Chandy and J. R. Dickson, "A comparison of list schedules for parallel processing systems", *CACM*, 17:12 (1974), 685-690.

[2] Aho, Alfred V., Jeffery D. Ullman, "Principles of Compiler Design" Addison-Wesley Publishing Company, Reading, Mass., 1977.

[3] S.T. Barnard and H.D. Simon, "A Fast Multilevel Implementation of Recursive Spectral Bisection for Partitioning Unstructured Problems", *Proc. Sixth SIAM Conf. on Parallel Processing for Scientific Computing,* SIAM, 1993.

[4] Gary R. Beck, David W. L. Yen and Thomas L. Anderson, "The Cydra 5 Minisupercomputer: Architecture and Implementation" *The Journal of Supercomputing,* 1993.

[5] Robert P. Colwell, Robert P. Nix, John J. O'Donnell, David B. Papworth and Paul K. Rodman. "A VLIW Architecture for a Trace Scheduling Compiler" *IEEE Transactions on Computers,* 37-8, August 1988.

[6] William Dally J., Stephen W. Keckler, Nick Carter, Andrew Chang, Marco Fillo, and Whay S. Lee, "M-Machine Architecture v1.0" Massachusetts Institute of Technology, *Artificial Intelligence Laboratory Concurrent VLSI Architecture Memo,* Number 58, January 1994.

[7] John R. Ellis, "Bulldog: A Compiler for VLIW Architectures" MIT Press, 1986.

[8] Joseph A. Fisher, "Trace scheduling: A technique for global microcode compaction" *IEEE Transactions on Computers,* C-30(8):478-490, July 1981.

[9] R. L. Graham, "Bounds for certain multiprocessing anomalies", *Bell System Technical Journal,* 45 (1966), 1563-1581.

[10] M. Gumpta and P. Banerjee, "Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers." *IEEE Transactions on Parallel and Distributed Systems*, 3(2):179-193, March 1992.

[11] M. W. Hall, B. R. Murphy, and S. P. Amarasinghe, "Interprocedural parallelization analysis: A case study." *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing,* San Francisco, Feb 1995.

[12] Hudak, David E. “Compiling parallel loops for high performance computers: partitioning, data assignment, and remapping” *The Kluwer international series in engineering and computer science*, 1993.

[13] Kim, S. J. and Browne, J. C. “A general approaach to mapping of parallel computation upon multiprocessor architectures”, *International Conference on Parallel Processing*, vol 3, 1988, pp. 1-8.

[14] J.Anderson and M. Lam. “Global optimizations for parallelism and locality on scalable parallel machines.”, *Proceedings of the SIGPLAN ’93 Conference on Programming Language Design and Implementation*, Albuquerque, NM, June 1993.

[15] P. G. Lowney, S. G. Freudenberger, T. J. Karzes, W. D. Lichtenstein, R. P. Nix, J. S. O’Donnell and J. C. Ruttenberg. “The Multiflow Trace Scheduling Compiler” *The Journal of Supercomputing,* 1993.

[16] D. Maskit and S. Taylor. “A Message-Driven Programming System for Fine-Grain Multicomputers.” *Software: Practice and Experience,* 24(10):953-980, 1994.

[17] S. Keckler, “A Coupled Multi-ALU Processing Node for a Highly Parallel Computer” Massachusetts Institute of Technology, *Artificial Intelligence Technical Report 1355,* September 1992.

[18] Sarkar, Vivek. “Partitioning and Scheduling Parallel Programs for Multiprocessors”, MIT Press, 1989.

[19] R. Van Driessche and D. Roose. “An Improved Spectral Bisection Algorithm and Its Application to Dynamic Load Balancing.” *Parallel Computing,* 21:29–48, 1995.

[20] R. Williams. “Performance of Dynamic Load balancing Algorithms for Unstructured Mesh Calculations.” *Concurrency: Practice and Experience,* 3:457–481, 1991.

[21] Wolfe, Michael Joseph. “High Performance Compilers for Parallel Computing”, Addison-Wesley, 1995.

[22] T. Yang and A. Gerasoulis, “List scheduling with and without communication delays”, Rutgers University, Department of Computer Science, 1992.

[23] T. Yang and A. Gerasoulis, “PYRROS: Static scheduling and code generation for message passing multiprocessors”, *Proc. of 6th ACM International Conference on Supercomputing,* Washington D.C., July 1992, pp. 428-437

[24] Tao Yang and Apostolos Gerasoulis, “DSC: Scheduling parallel tasks on an unbounded number of processors”, *IEEE Transactions on Parallel and Distributed Systems,* Vol. 5, No. 9, 951-967, 1994.