Compiler Techniques For Fine-Grain Execution On Workstation Clusters Using PAPERS†

H. G. Dietz, W. E. Cohen, T. Muhammad, and T. I. Mattox

Parallel Processing Laboratory
School of Electrical Engineering
Purdue University
West Lafayette, IN 47907-1285
hankd@ecn.purdue.edu
(317) 494 3357

Abstract

Just a few years ago, parallel computers were tightly-coupled SIMD, VLIW, or MIMD machines. Now, they are clusters of workstations connected by communication networks yielding ever-higher bandwidth (e.g., Ethernet, FDDI, HiPPI, ATM). For these clusters, compiler research is centered on techniques for hiding huge synchronization and communication latencies, etc. — in general, trying to make parallel programs based on fine-grain aggregate operations fit an existing network execution model that is optimized for point-to-point block transfers.

In contrast, we suggest that the network execution model can and should be altered to more directly support fine-grain aggregate operations. By augmenting workstation hardware with a simple barrier mechanism (PAPERS: Purdue's Adapter for Parallel Execution and Rapid Synchronization), and appropriate operating system hooks for its direct use from user processes, the user is given a variety of efficient aggregate operations and the compiler is provided with a more static (i.e., more predictable), lower-latency, target execution model. This paper centers on compiler techniques that use this new target model to achieve more efficient parallel execution: first, techniques that statically schedule aggregate operations across processors, second, techniques that implement SIMD and VLIW execution.

† 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

Combining the low cost and high performance of modern workstations with the ever increasing bandwidth of the networks that connect them, the differences between a workstation cluster and a traditional supercomputer blur. However, there are fundamental differences that severely limit the range of parallel applications for which a workstation cluster can achieve good performance. These differences are not inherent in the use of workstations as the processing elements of a parallel machine, rather, they relate to assumptions made about the ways in which the workstation network would be used.

If we assume that each workstation is owned and used almost exclusively by one user, and the network is used primarily for file sharing and email, then it is appropriate to organize the system so that:

- Each workstation runs an autonomous operating system that schedules user process activities and moderates interactions with other workstations.
- Networks are designed to transmit relatively large blocks of data formatted as self-contained messages.

The problem is that this loosely coupled, stateless, organization is not a good match for the aggregate operations within most parallel algorithms, and impedes many of the most important compiler optimizations.

Fortunately, it takes very little modification to the hardware and operating system of a workstation cluster to provide an execution model that is more appropriate for executing fine-grain parallel code. Such a set of modifications is PAPERS: Purdue's Adapter for Parallel Execution and Rapid Synchronization [DiM94].

1.1. PAPERS Hardware

Since our goal is to make a workstation cluster more effective as a parallel machine, the PAPERS hardware is implemented as a simple and inexpensive (less than $50 per processor) add-on unit — neither the workstation hardware nor the network hardware is altered. The PAPERS hardware is connected to the standard Centronics-compatible printer port of each workstation.

Although it is accessed using port I/O operations instead of `Load` instructions, PAPERS implements the full dynamic barrier synchronization mechanism described in [CoD94]. Because the set of processors participating in each barrier synchronization is specified by each processor giving a bit mask indicating which processors it should synchronize with, the hardware is also completely partitionable, and barriers with nonoverlapping masks have no effect on each other. In any case, the barrier logic is easily implemented to be faster than wire propogation delays, which are in turn negligible in comparison to the processor latency in accessing an external port.

As a side effect of each barrier synchronization performed using PAPERS, an unusual type of synchronous communication is effected. Data from each processor participating in the barrier is simultaneously made available to all other processors (a multi-broadcast communication). In the first PAPERS unit, just one bit of data is sent from each processor, and the gathered set of bits can be directly read by each processor such that each bit in the value read is the data bit that was sent by the processor corresponding to that bit position in the barrier mask. Thus, this hardware allows processors within a partition of the full machine to "vote" and have the result of the vote known by all processors that were involved. For example, this is how the machine can determine at runtime which processors should be included in a barrier mask. It also allows generalized testing of the aggregate machine state, for example, checking if any processor voted 1 (i.e., global or).

Surprisingly, the multi-broadcast operation is actually sufficient for fully general aggregate communications, and can be used to implement random point-to-point "routing" such that all communication patterns are conflict-free. The basic concept is that each sending processor makes data available to every other processor, and the receiving processors simply select which data values they want to use. This is not the standard "shared memory" model, nor is it traditional "message passing," but it yields a more efficient implementation than either of these schemes:

- In both shared memory and message passing, the processor sending a datum must transmit both the datum and the destination address; in contrast, PAPERS' multi-broadcast allows the sender to transmit only the datum. Thus, given a fixed bandwidth (i.e., fixed number of wires), the multi-broadcast operation can send more data per unit time than is possible using either of the more conventional alternatives.
- There is no difference between a permutation and a multicast communication; the number of readers for each datum is irrelevant. This is because there is no dynamic routing. In fact, even if dynamic routing were used within the PAPERS unit, any reasonable delay within the box would be negligible in comparison to the propagation delay time for the cables that connect each workstation to the PAPERS box.
- Because each buffer within PAPERS has only one writer, write operations are always non-blocking. Although future versions of PAPERS might have blocking reads, delaying a read slows only one processor, while delaying a write slows all processors that would read that value.

Although the first PAPERS and TTL_PAPERS units send only one bit per processor, all later versions support four-bit wide communications as single operations.

1.2. PAPERS Workstation Interface

In discussing how each workstation is interfaced to the PAPERS unit, the first thing that should be understood is that adding PAPERS to an existing workstation cluster does not interfere with any of the other capabilities of that cluster. For example, PAPERS does not interfere with operation of Ethernet interfaces, etc. The second concept to understand is that common interactions with PAPERS should all be accomplished directly from a user process, without any operating system intervention.

It might surprise the reader, but most versions of UNIX allow user processes to directly access I/O ports. Typically, this is done by a privileged process calling `iopl()` to obtain permission to access all ports, and then simply accessing the ports by directly executing port I/O instructions. On 386/486/Pentium processors running the Linux operating system [Wie93], it is possible for a privileged process to obtain access to specific I/O ports by calling `ioperm()`; the hardware maintains an I/O port access map. Ideally, a new operating system call should be implemented so that a non-privileged user process can obtain access to the I/O ports connected to PAPERS if those ports are not in use by another process; in any case, port access permission need be obtained only once per user process. There is no operating system intervention required in accessing PAPERS aggregate functions.

Unfortunately, there are unavoidable overheads associated with direct use of standard Centronics printer ports:

- Most workstations have port hardware that is deliberately slowed to increase reliability of the printer interface. Depending on the port hardware, the port access time ranges from about 0.2μs to 4.0μs, much slower than the 0.1μs limit imposed by cable length and PAPERS circuit speed.
- The use of a unidirectional printer port for all PAPERS I/O severely limits the number of bits that can be transferred per port operation. Each printer port is actually three independently addressed ports: an 8-bit data output port, a 4-bit status output port, and a 5-bit status input port. PAPERS is designed so that normal operation requires use of only the 8-bit output and 5-bit input ports, treating one bit from each as a strobe.

Given these constraints, we can approximate execution time by simply counting the number of port operations required to perform each basic PAPERS function. Table 1 lists each of the basic aggregate functions implemented by PAPERS, the number of port operations needed to perform that operation on the 4-processor PAPERS1 prototype and on a *p*-processor PAPERS2, and a short description of each function. For example, if each port access takes about 1.0μs and we are using PAPERS2 to perform a `p_waitvec()` across a 16-processor workstation cluster that is already roughly synchronized, the approximate execution time would be less than $(2 + 16/2) * 1.0\mu$s, or 10.0μs.

Notice that, although we discuss PAPERS in terms of connecting workstations, it is perfectly reasonable to have multiple PAPERS connections (i.e., multiple printer ports) on each workstation. ISA bus printer port cards can be purchased for less than $10 each. Thus, for example, a multiprocessor workstation might well have one PAPERS connection for each processor it contains. This not only yields a more symmetric structure for the software, but also yields faster synchronization across processes within the machine than would normally be obtained using standard UNIX inter-process communication facilities. Using a 486DX/50 workstation running Linux, a simple 1-byte handshake between two processes using a pipe takes 225μs — at least 68 times as long as an arbitrary operation done using the current PAPERS1 hardware.

2. Static Scheduling of Interactions

Given the low-latency synchronization and communication abilities of the PAPERS hardware, a variety of new compiler optimizations for workstation cluster communication are feasible. The first of these uses the PAPERS hardware not for communication, but as a mechanism for semi-statically scheduling the use of other communication facilities. The second class of optimizations uses the PAPERS hardware instead of the traditional network for performing the actual communications. The ability of PAPERS to quickly examine aggregate state can also be used to impose deterministic simplifications that directly eliminate communications by detecting their irrelevance to the computation, and this forms the third class of compiler scheduling optimizations.

PAPERS Function (port ops 4-PE PAPERS1, *p*-PE PAPERS2)	Description

`extern inline void`
`p_enqueue(barrier mask);`
 	
(0 - 1, 0 - *p*/4)
	Enqueue the bit pattern given by `mask` as the group of processors that this processor will synchronize with. This grouping remains in effect until explicitly changed.

`extern inline void`
`p_wait(void);`
 	
(0 - 2, 0 - 2)
	Wait for a barrier across the set of processors specified by the last enqueued mask. Although the interface can be interrupt driven, the wait is currently implemented by polling the input port.

`extern inline barrier`
`p_waitvec(int flag);`
 	
(0 - 2, 0 - 2+*p*/2)
	Wait for a barrier, simultaneously broadcasting one bit (`flag`) to all processors that participate in the barrier. The resulting value is a bit vector in which bit *k* is the flag value from processor *k*; bits from processors not participating in the barrier are forced to be zero.

`extern inline int`
`p_all(int flag);`
 	
(0 - 2, 0 - 2)
	Like `p_waitvec()`, except this operation returns a value of 1 if all processors in the barrier had non-zero `flag` values, and returns 0 otherwise. Note that the all operation is performed in the PAPERS2 hardware, so that transmission of the result bit vector is not necessary. The `p_any()` operation is also supported.

`extern unsigned char`
`p_putget8u(`
`unsigned char datum,`
`int source);`
 	
(0 - 4, 0 - 4+$\log_{16}p$)
	This function provides generalized aggregate byte communication, sending the value `datum` and returning the value that was sent by the processor whose number is `source`.

Table 1: PAPERS Software Interface

2.1. Scheduling Conventional Network Use

In a recent paper by Brewer and Kuszmaul [BrK94], it was observed that performance of the CM-5 data network could be improved by a factor of 2-3x simply by inserting semantically unnecessary barrier synchronizations to help level network traffic. They also observed that an additional 25% gain could be obtained by slowing the rate of message injection to the network to match the rate of message removal. Given a workstation cluster augmented by the PAPERS hardware, we can achieve a similar gain by using PAPERS to schedule use of the conventional network.

The basic problem noted by [BrK94] is that, without barriers, some processors may race ahead of others, thus causing blocking that further worsens the variation in completion times. Although most conventional workstation cluster networks are relatively simple and do not have blocking properties like those of the CM-5's fat tree, similar problems arise.

For example, the next processor to be given access to a shared bus network (e.g., Ethernet) is essentially selected at random by granting the bus to the first processor that makes an uncontested attempt to access the bus. If multiple processors attempt to grab the bus at the same time, all detect the collision and back-off for a random period before trying again. This type of contention can be eliminated by using the PAPERS hardware to schedule bus accesses. The technique is very simple:

(1) Use a single `p_waitvec()` to determine which processors would like to access the traditional network. A processor sends a flag value of 1 if it wants traditional network access, 0 if not.

(2) In a statically-determined order, each processor that wants access is given its turn to perform a traditional network access. The accesses are separated by `p_wait()` operations as necessary to enforce the conflict-free scheduling of traditional network accesses.

This scheduling can be further improved by selecting a static order for each communication so that the processors involved in consecutive communications are disjoint. This is the bus-based analog of matching injection and removal rates in the CM-5's network. For example, in the extreme case, if too much data is buffered in a workstation, that workstations network buffer pool can be exhausted, and additional data received will be discarded. This further damages performance because the lost data will have to be re-transmitted, yielding yet more network traffic. In contrast, the PAPERS hardware can even be used to broadcast information about which workstations currently have buffer space available.

It is also useful to note that a simple `p_all()` can be used to provide lightweight acknowledgements when data is received using the conventional network.

2.2. Low Latency Multi-Broadcast

As suggested earlier, the PAPERS hardware provides a very low latency multi-broadcast communication mechanism that can be used for general purpose communication, but there are several catches:

(1) Although PAPERS provides low latency, virtually zero communication startup and message formatting overhead, and apparently collision-free "routing," the data transmission bandwidth is severely limited by the use of a Centronics-compatible printer port. At most, just 4 data bits can be read per port operation — in contrast, DMA network interfaces can transfer up to 64 data bits per bus cycle (e.g., using PCI).

(2) The low latency is obtained by direct port access from the user program, implying polled I/O and very little potential for overlap of computation with communication. In fact, all communication using the current version of PAPERS is completely synchronous.

(3) The PAPERS mechanism requires that the sender broadcast data and that the receiver know who to read data from. Given that, however, data always arrives in order and transmission is 100% reliable without an acknowledgement. For the same reason, there is typically no need to construct, send,

receive, and decode message headers — a processor never needs to send information at runtime if that information was known to all processors at compile time.

Thus, although PAPERS works very well for most aggregate operations, it probably isn't the best way to do some types of communication. It is up to the compiler to optimize the choice of communication method to match the type of communication required.

A simple rule for compiler scheduling can be derived by examining communication block size. Points (1) and (2) above clearly make sending large blocks of data more efficient using a conventional network than using the PAPERS unit. Thus, for any system, the compiler's decision can be distilled into a single formula for predicting the expected communication time for a collection of communication operations. The cost for sending a block using PAPERS1 or PAPERS2 is four port operations per byte. Using 486DX/33 workstations under Linux, receiving each byte takes about 6.6μs... very low latency, but only 150K bytes/second bandwidth per processor, independent of processor count and routing pattern. In contrast, an Intel EtherExpress Ethernet card on the same system achieves about 580K bytes/second bandwidth, but, through the Linux TCP interface, has a latency of more than 5,000μs. Unless the communication is a simple broadcast, bandwidth available also decreases at least linearly as more processors share the same Ethernet. For this particular 4-processor workstation cluster, the point-to-point communication times always favor using PAPERS1; however, if the operation is a broadcast and the block size is at least 1K bytes, the Ethernet is faster. Incidentally, if the Ethernet were replaced by a 100M byte/second HiPPI, the crossover point for broadcast would have been at a block size of at least 750 bytes. Since PAPERS bandwidth per processor is constant (limited by port speed for reasonable numbers of workstations), adding more processors will make PAPERS faster than bus-based networks for even larger blocks.

A more sophisticated compiler analysis would take into account the issues described in (3) above. Most current compiler analysis is based on creating messages that are targeted by the sender, but the PAPERS hardware requires cooperation from both the sender and receiver. In most cases, this means using the same compiler analysis of communication structure to provide different information, but sometimes it invokes a radical transformation of the communication structure. For example, consider a CMF, MPF, Fortran-D, or HPF statement like:

```
A(1:N) = B( I(1:N) )
```

where `A`, `B`, and `I` all have a layout that places the j element on processor j. Using PAPERS, each processor would simply broadcast the value of `B(`j`)`. Each processor can then locally select the appropriate datum from the multi-broadcast. In contrast, a traditional message-passing implementation would first send a request message to processor `I(`j`)`, and then send messages containing `B(`j`)` to any processors that requested them. Thus, rather than one set of multi-broadcast operations, traditional message-passing uses 2`N` messages (not counting acknowledgements) — and there can be serializing conflicts on all 2N messages either due to network topology or destination/source conflicts (e.g., suppose every element of `I` has the same value).

2.3. Scheduling To Remove Needless Communications

Because the PAPERS unit can provide rapid access to a small amount of global state information, it is possible for the compiler to statically detect communication patterns that are likely to involve needless transmission of data, and to then insert PAPERS operations to detect and eliminate these transmissions at runtime. By far the most common reason that a transmission may be redundant is because it is involved in a race. Fundamentally, if there are *n* data values racing to be stored in the same location, *n*-1 of these data transmissions will have no effect on the computed result, and can be eliminated.

Such races are commonly used in MPL [MCC91] code to select representative cases for sets of values. Consider:

```
plural int i; plural double a, b;
router[i].a = b;
```

in which each processor attempts to store its value of `b` into the variable `a` residing on processor `i`. A race exists if the value of `i` is the same for multiple processors.

Reguardless of whether the data communication (i.e., transmission of `b` values) is done using PAPERS or the conventional network, it is possible to use PAPERS to rapidly detect these races and select the winners, so that only they need to be transmitted. This is done by simply using `p_waitvec()` for each processor to transmit the destination it wishes to send to. Each processor can then apply a statically determined algorithm to determine if it was the loser of a race, since it knows which other processors had the same target.

The algorithm can be summarized as follows. For each bit in the destination address, use `p_waitvec()` to transmit the bit. The returned bit vector is inverted if the bit we broadcast was a 0. By `and`ing all the resulting bit vectors together, we create a race bit mask with a 1 in each position corresponding to a processor that is racing with this processor. Each processor can then determine if it was the loser of a race (and hence, should not send data) by examining the race bit mask. For example, if the race bit mask `and` the bit mask representing all higher-numbered processors yields 0, then this processor is the highest-numbered processor involved in this race, and hence the winner.

Of course, other methods can also be used to determine the winner from the race bit mask. For example, it is particularly effective to ensure that any processor wishing to send to itself is always the winner of any race involving that communication.

3. Execution Models

An essential part of successfully programming a parallel machine is making effective use of available programming models [BeS91]. A cluster of workstations that provides low-latency aggregate operations can support a wide range of programming models. Different programming models, or even individual programs, may obtain better performance with a particular execution model. Even within the same program, different parts of the program may benefit from using different execution models, for example, mixing MIMD and SIMD execution [BeS91] [Wat93].

Obviously, multiple workstations can run multiple programs, and it is natural to use a cluster as a large-grain MIMD (Multiple Instruction streams, Multiple Data streams) system. However, a PAPERS cluster is a good target for the fine- to medium-grain MIMD barrier scheduling work discussed in [Pol88] [Gup89] [DiO92]. It should be noted, however, that executing fully static schedules for MIMD code may require temporarily disabling interrupts — although this is permitted under most versions of UNIX, disabling interrupts for too long may have strange consequences in the UNIX environment.

The two most common fine-grain models are SIMD (Single Instruction stream, Multiple Data streams) and VLIW (Very Long Instruction Word). Both these models assume a single program counter for the entire parallel machine, and both distribute parallel data across the processing elements. The difference is that SIMD implies that all processors execute the same opcode from each instruction, whereas VLIW allows each processor to use a different opcode field within each instruction. Although it is not practical for workstations to share a single core image and program counter, the same effect can be obtained by placing the appropriate opcode sequence in each processor's memory and using aggregate operations to enforce SIMD or VLIW semantics.

However, these execution models seem to imply instruction granularity. In fact, there is no reason to require that an operation or instruction is an atomic parallel function for the hardware; it could instead be an entire subroutine or sequence of machine instructions. In Table 2, we briefly summarize the effective *minimum* grain sizes for a variety of parallel machines in terms of the minimum communication latency (end-to-end time) measured both as an absolute time and as the maximum number of floating point operations that could be executed by the machine in that time period. Although the Thinking Machines CM-5 and Intel Paragon XP/S both claim to support SIMD execution, they clearly do not support instruction granularity — neither does PAPERS. However, PAPERS is fine-grain enough to enable a large class of SIMD and VLIW codes to execute efficiently.

Machine	Minimum Communication Latency (μs)	Float Operations per Com.	Source of Performance Data
486DX/33 Ethernet (TCP)	5,000	20,000	*measured*
Intel Paragon XP/S	240	18,000	[BrG94]
Thinking Machines CM-5	12-65	1,536-8,320	[LiC94][BrG94]
nCUBE/2	32-110	96-330	[BrG94]
486DX/33 PAPERS1	3.3	13.2	*measured*

Table 2: Comparison Of Minimum Hardware Grain Sizes

3.1. Implementation Of SIMD Execution

SIMD is implemented by each workstation having its own copy of the program, augmented by aggregate operations as needed. Conceptually, SIMD execution would require a barrier synchronization after each instruction. Fortunately, omitting this synchronization between most instructions yields no ill effects [HaL91]. The rule is simply that synchronization is needed for communication or selection of a control flow path based on aggregate state (i.e., does any processor want to take this path?). Because, following SIMD semantics, all processors must conceptually follow the same flow path, processors that did not want to take that control flow path can be "disabled" with enable masking.

3.1.1. Enable Mask Simulation

The SIMD execution model is generally expected to provide support for enable masking: the ability to "turn off" selected processing elements for a sequence of operations.

There are a variety of simple and effective ways in which hardware could disable particular processing elements, but the most practical implementation is probably to have all processing elements perform every operation, nulling the effects on the processing elements that should not have been active. For example, some processors have conditional store instructions that would allow "disabled" processing elements to perform computations, but not update variables with the results. With somewhat greater cost, this nulling can even be done using conventional arithmetic; for example, the parallel assignment:

```
c = a + b;
```

Can be made to execute only when `enabled` is a bit mask containing all `1`s (for true, as opposed to `0` for false) by all processing elements simultaneously executing:

```
c = (((a + b) & enabled) | (c & ~enabled));
```

Another possible implementation is to `JUMP` over code for which the processor is disabled. However, this must be done very carefully to avoid accidentally skipping operations that should be executed (enable mask operations, scalar code, function calls, etc.). Further, using `JUMP`s can cause pipeline bubbles that seriously degrade performance.

For the purposes of this paper, we will hide the details of how disabling is implemented by enclosing code that should be executed only by enabled processors within a block marked as `EVAL_ENABLED`. Of course, having the compiler select the cheapest method for implementing each such block is a fundamental optimization.

In addition to nullifying the effects of disabled operations, there is also the issue of tracking enable status as nested constructs change the set of enabled processors. Traditionally, enable status has been tracked using a special hardware stack of one-bit enable flags, however, this is not practical for most workstations. Alternatively, one word could be used to represent each enable flag on the stack, but this wastes memory space and requires a memory operation for each scope entry or exit. The best method for a workstation cluster using PAPERS is a variation on the "activity counter" scheme proposed by Keryell and Paris [KeP93].

This scheme takes advantage of a simple property of the enable stack deriving from use of structured SIMD enable constructs: the enable stack for each processor always contains a sequence of zero or more 1 bits (enable states) followed by zero or more 0 bits (disable states). Thus, if the program uses only structured enable masking, the same information can be recorded by simply counting the number of disable states that would have been pushed in each processor's enable stack. A counter called `disabled` is used in each processor to track the number of properly nested constructs for which each processor has been disabled. If this counter is zero, the processor is enabled; otherwise, the processor is disabled. Thus, entry or exit of an enable scope is accomplished by a simple update of the `disabled` counter. For best performance, `disabled` would normally be allocated to a register.

However, some SIMD languages provide features that allow "unstructured" changes to processor enable status. For example, MPL's `all` construct [MCC91] enables all processing elements for execution of the following statement — enabling processors that were disabled at entry to the enclosing region of code. In fact, a statement affected by an `all` can be a compound statement that contains additional structured and unstructured masking. Another example is MPL's `proc` construct; `proc[i].j` refers to the value of `j` on processor `i`, reguardless of whether processor `i` was enabled in the enclosing region of code. Fortunately, this situation can be correctly handled by mixing the activity counter and traditional stacking methods. The `disabled` counter value is pushed onto a stack at entry to an unstructured change of enable status and is restored at exit from that region.

3.1.2. Example SIMD Code

To illustrate how code with SIMD semantics would be implemented on a group of workstations augmented by PAPERS, code for two parallel constructs will be generated: the parallel `if-else` and the parallel `while`. The code for both of the examples makes use of the aggregate test function `p_any()` and supports operations that may enable all the processing elements in the machine, e.g. the `all` statement in MPL and the `everywhere` statement in C*.

3.1.2.1. SIMD Parallel `if-else`

SIMD semantics do not make a parallel `if-else` construct change the control flow of the program. Rather, the construct changes enable/disable status of processors as all the processors pass through the code for both the "then" and `else` clauses.

Although both the new C* [TMC90] and MPL [MCC91] use SIMD semantics, the C* language more literally adheres to the principle that control flow cannot be altered by a parallel construct. Even if no processors are enabled, C* semantics suggest that the complete code sequence should be "executed." In contrast, MPL semantics suggest that code for which no processor is enabled should not be "executed." Thus, before each section of conditionally-executed code, MPL inserts instructions that jump over the code if no processors would be enabled for its execution. This difference in semantics is reflected by the fact that the code of Listing 2 implements the MPL semantics as is, and implements the new C* semantics if the **`bold`** code is removed.

```
if (parallel_expr){
    stat_a;
}else{
    stat_b;
}
```

Listing 1: Parallel `if-else`

```
    if (disabled) ++disabled;
    EVAL_ENABLED {
        if (!parallel_expr) disabled = 1;
    }
    if (p_all(disabled)) goto Else;
    EVAL_ENABLED {
   stat_a;
    }
Else:
    if (disabled < 2) disabled ^= 1;
    if (p_all(disabled)) goto Exit;
    EVAL_ENABLED {
   stat_b;
    }
Exit:
    if (disabled) --disabled;
```

Listing 2: SIMD parallel `if-else`

3.1.2.2. SIMD Parallel `while`

Because the new C* does not allow control flow to be altered by a parallel construct, there is no parallel `while` in C*. However, there is a parallel `while` construct in MPL. This construct is implemented by the code given in Listing 4.

```
while(parallel_expr){
    stat;
}
```

Listing 3: Parallel `while`

```
    if (disabled) ++disabled;
Loop:
    EVAL_ENABLED {
        if (!parallel_expr) disabled = 1;
    }
    if (p_all(disabled)) goto Exit;
    EVAL_ENABLED {
        fIstat;
    }
    goto Loop;
Exit:
    --disabled;
```

Listing 4: SIMD parallel `while`

3.2. Implementation Of VLIW Execution

VLIW, or Very Long Instruction Word, computation is based on use of a single instruction sequence with an instruction format that allows a potentially different opcode for each function unit. Treating each processing element as a function unit, genuine VLIW [Fis84] [CoN88] execution can be obtained without any hardware changes. Each VLIW instruction is striped across the various local (instruction) memories. I.e., the VLIW instruction at address α is encoded by having address α in instruction memory β be the opcode field for function unit β.

According to Brownhill and Nicolau [BrN90], a workstation cluster would be an effective vehicle for VLIW execution only if it provides:

- Fast communication
- Fast (barrier) synchronization
- An effective implementation for multi-way jumps

Of these, we have already discussed how PAPERS provides the first two; however, PAPERS also is essentially the ideal mechanism with which to implement multi-way jumps.

In order to facilitate more aggressive code motions, ideal VLIW machines allow multiple branch-tests to be executed in parallel. Executing multiple branch-tests within single VLIW operation results in a multi-way jump. In the idealized VLIW model, a central control unit collects these test result bits and selects the appropriate jump target. However, as suggested in [BrN90], the same effect can be achieved by having each processor evaluate its branch-test, set the bit that corresponds to that processor in the global result register to reflect the local branch-test result, perform a barrier synchronization, and then perform a multiway branch using the bit vector that was accumulated in the global result register. There was no prototype hardware implementation and there are several minor flaws in the presentation given in [BrN90], but PAPERS can correctly implement this model using a single `p_waitvec()` operation followed by a multiway branch based on the value returned.

Consider the example program fragment in Listing 5. Listing 6 gives the VLIW code sequence showing how this example is VLIW parallelized so that A, B, C, and D will be executed in parallel on a 4-processor workstation cluster using PAPERS. (For brevity, we have omitted the usual repair code implied by this speculative execution.) Further improvement can be made because most C compilers would compile the `switch` statement's sparse list of `case` values into code implementing a binary search. Instead, the compiler should create a nearly perfect hash function to perform the `switch` without any comparison operations. For example, our hash function generator [Die92] suggests that the mapping from Listing 6 can be implemented directly using a 4-element jump table indexed by `(((-t)>>2)&3)`. Using a 4-processor 486DX/33 PAPERS workstation cluster under Linux, the complete VLIW branch code sequence executes in about 4μs.

```
A if (B) { C if (D) { a: E } b: F } c: G
```

Listing 5: Example Program Fragment For VLIW Branching

Processor 0	Processor 1	Processor 2	Processor 3
A	t = B	C	t = D
t = p_waitvec(0)	t = p_waitvec(t)	t = p_waitvec(0)	t = p_waitvec(t)
switch (t) {	switch (t) {	switch (t) {	switch (t) {
case 0: goto c;	case 0: goto c;	case 0: goto c;	case 0: goto c;
case 2: goto b;	case 2: goto b;	case 2: goto b;	case 2: goto b;
case 8: goto c;	case 8: goto c;	case 8: goto c;	case 8: goto c;
case 10: goto a;	case 10: goto a;	case 10: goto a;	case 10: goto a;
}	}	}	}

Listing 6: PAPERS VLIW Code For Multiway Branch

In addition to the fundamental optimizations using `p_waitvec()` and a hash function generator, there are a variety of special cases that can be readily optimized by the compiler. A very common special case occurs when a single branch instruction is executed within a VLIW instruction. In this case, it is not necessary to use `p_waitvec()`; a single `p_any()` will suffice. Likewise, in this case, it is not necessary to compute a hash function to index a jump table — an ordinary conditional branch instruction can test the value returned by `p_any()`. The usual techniques for eliminating unnecessary barrier synchronizations also can be applied [BrN90].

4. Conclusion

Many parallel programs are based on aggregate operations, for which traditional workstation clusters have proven to be ill-suited. In this paper, we have presented a variety of new opportunities for compilers to optimize parallel program execution using a conventional UNIX workstation cluster augmented by fast hardware for aggregate operations.

Using the low-latency PAPERS hardware to implement aggregate operations, a workstation cluster can function as a fine-grain reconfigurable parallel machine, rather than a loosely-connected set of autonomous computers. Thus, the compiler issues focus on statically scheduling communication and directly implementing optimized fine-grain SIMD and VLIW execution models. The techniques and performance improvements quoted in this paper represent only the most cursory exploration of this new view of workstation clusters.

To encourage others to pursue this new point of view, the PAPERS hardware design is fully public domain and is optimized to be easily and cheaply replicated. Likewise, as the compiler technology discussed in this paper becomes more stable, we plan to distribute it in the form of both optimization phases that can be used in compilers constructed using PCCTS [PaD92] and as complete public domain compilers for a subset of MPL and a small Fortran 77 dialect called Fortran-P.

References

[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.

[BrG94] U. Bruening, W. K. Giloi, and W. Schroeder-Preikschat, "Latency Hiding in Message-Passing Architectures," *8th International Parallel Processing Symposium*, Cancun, Mexico, April, 1994, pp. 704-709.

[BrK94] E. A. Brewer and B. C. Kuszmaul, "How to Get Good Performance from the CM-5 Data Network," *8th International Parallel Processing Symposium*, Cancun, Mexico, April 1994, pp. 858-867.

[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.

[CoD94] W. E. Cohen, H. G. Dietz, and J. B. Sponaugle, "Dynamic Barrier Architecture For Multi-Mode Fine-Grain Parallelism Using Conventional Processors," To appear, *Int'l Conf. on Parallel Processing*, August 1994.

[CoN88] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman, "A VLIW Architecture for a Trace Scheduling Compiler," *IEEE Trans. on Computers,* vol. C-37, no. 8, pp. 967-979, Aug. 1988.

[Die92] H. G. Dietz, *Coding Multiway Branches Using Customized Hash Functions*, Purdue University School of Electrical Engineering Technical Report TR-EE 92-31, July 1992.

[DiM94] H. G. Dietz, T. Muhammad, J. B. Sponaugle, and T. Mattox, *PAPERS: Purdue's Adapter for Parallel Execution and Rapid Synchronization*, Purdue University School of Electrical Engineering Technical Report TR-EE 94-11, March 1994.

[DiO92] H. G. Dietz, M.T. O'Keefe, and A. Zaafrani, "Static Scheduling for Barrier MIMD Architectures," *The Journal of Supercomputing*, vol. 5, pp. 263-289, 1992.

[Fis84] J. A. Fisher, "The VLIW Machine: A Multiprocessor for Compiling Scientific Code," *IEEE Computer*, July 1984, pp. 45-53.

[Gup89] R. Gupta, "The Fuzzy Barrier: A Mechanism for the High Speed Synchronization of Processors," *Third Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, April 1989, pp. 54-63.

[HaL91] P. J. Hatcher, A. J. Lapadula, R. R. Jones, M. J. Quinn, and R. J. Anderson, "A Production-Quality C* Compiler for Hypercube Multicomputers" *Third ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, Williamsburg, Virginia, April 1991, pp. 73-82.

[KeP93] R. Keryell and N. Paris. "Activity Counter: New Optimization for the Dynamic Scheduling of SIMD Control Flow," *Proc. Int'l Conf. Parallel Processing*, pp. II 184-187, August 1993.

[LiC94] L. T. Liu, and D. E. Culler, "Measurements of Active Messages Performance on the CM-5," University of California Berkeley, CS csd-94-807(dir), UCB//CSD-94-807, May 1994. 36 pages.

[MCC91] MasPar Computer Corporation, *MasPar Programming Language (ANSI C compatible MPL) Reference Manual, Software Version 2.2*, Document Number 9302-0001, Sunnyvale, California, November 1991.

[PaD92] T. J. Parr, H. G. Dietz, and W. E. Cohen, "PCCTS Reference Manual (version 1.00)," *ACM SIGPLAN Notices*, Feb. 1992, pp. 88-165.

[Pol88] C. D. Polychronopolous, "Compiler Optimizations for Enhancing Parallelism and Their Impact on Architecture Design," *IEEE Trans. Comput.,* vol. C-37, no. 8, pp. 991-1004, Aug. 1989.

[TMC90] Thinking Machines Corporation, *C* Programming Guide*, Thinking Machines Corporation, Cambridge, Massachusetts, November 1990.

[Wat93] D. W. Watson, *Compile-Time Selection of Parallel Modes in an SIMD/SPMD Heterogeneous Parallel Environment*, Ph.D. Dissertation, Purdue University School of Electrical Engineering, August 1993.

[Wie93] J. Wiegand, "Cooperative development of Linux," *Proceedings of the 1993 IEEE International Professional Communication Conference*, Philadelphia, PA, October 1993, pp. 386-390.