RESTRICTED FETCH&Φ OPERATIONS FOR PARALLEL PROCESSING

by

Gurindar S. Sohi
James E. Smith
James R. Goodman

Computer Sciences Technical Report #922

March 1990

Appeared in the
International Conference on Supercomputing
Heraklion-Crete, Greece
June, 1989.

Restricted Fetch&Φ Operations for Parallel Processing

Gurindar S. Sohi
Computer Sciences Department

James E. Smith
Department of Electrical and Computer Engineering

James R. Goodman
Computer Sciences Department

University of Wisconsin-Madison
1210 W. Dayton Street
Madison, WI 53706

Abstract

This paper discusses a restricted form of the general Fetch&Φ operation and how the restricted form can be combined. In this restricted form, all processors participating in the combining have identical Fetch&Φ operations. Most applications of Fetch&Φ proposed in the literature satisfy the restrictions imposed. We show how this restricted form of Fetch&Φ allows an easy implementation of combining, especially in bus-based multiprocessors and multiprocessors with a separate synchronization memory. Applications of the proposed restricted Fetch&Φ operation are also considered.

1. Introduction

For scalable parallel processing, it is essential that systems be designed without serial bottlenecks. In proposed parallel processing systems, serial bottlenecks are often associated with access to shared resources. Because shared variables held in memory are typically used for controlling access to shared resources, the serial bottlenecks manifest themselves as contention for the same memory location, *i.e.*, a "hotspot" [11].

To avoid such serial bottlenecks due to memory hotspots, it has been proposed that the Fetch&Φ operation be used as a fundamental synchronization operation [1, 5]. The *Fetch&Φ*(*X*, *C*) operation indivisibly reads the value at memory location X, performs the binary Φ operation with the value at X and C, and places the result back into memory location X. If Φ is an associative and commutative operation, then multiple Fetch&Φ operations directed at the same memory location can be combined into a single request by the interconnection network [1]. An important special case of FetchΦ occurs when the Φ operation simply returns the value of X. This corresponds to a memory load operation. Combining operations of this type results in a broadcast of the value at X. For synchronization, the more useful Fetch&Φ operation is a Fetch&Add [1].

To combine multiple Fetch&Φ operations, hardware combining networks have been proposed, but none has been constructed for a large-scale system (64 processors or more). It was initially proposed that hardware combining be used in the IBM RP3 [12], but the technology used for the RP3 interconnection network made the implementation prohibitively expensive, and it was not completed.

Cheaper implementations have been proposed for SIMD environments but the applicability of the cheaper implementation to a MIMD environment is not clear [9]. By replacing a single synchronization variable with a logical hierarchy of variables stored in memory, combining requests through levels of the hierarchy can be performed by software. These software combining trees have been proposed for barrier synchronization operations in Cedar [16]; a more general software combining Fetch&Φ algorithm has been proposed for Multicube [4]. However, efficient software combining algorithms for general Fetch&Φ operations and their effectiveness are still open research questions.

Based on the existing literature, it is clear that an efficient, cost-effective implementation for a combining Fetch&Φ (or an equivalent) is highly desirable. We emphasize that the implementation of the selected primitive should be: i) applicable for a wide range of situations, *i.e.*, it should be able to eliminate most (or all) potential serial bottlenecks involving shared variables and ii) cost-effective, *i.e.*, the benefit should be commensurate with the complexity. In this paper, we present a solution that has both features. The main idea of the proposed solution is that by using a restricted form of the general Fetch&Φ primitive, we can implement the combining operations locally in the processors, without expensive combining hardware in the processor-memory interconnection network.

The rest of this paper is organized as follows. In section 2, we describe a restricted form of the general Fetch&Φ primitive that is the backbone of our paper.

We present the basic idea that facilitates a cheap hardware implementation, discuss potential implementations and the limitations of the restricted primitive. In section 3 we present and discuss some example application areas and in section 4 we present concluding remarks.

2. A Restricted Form of Fetch&Φ

2.1. Background

Consider the general form of the Fetch&Φ instruction, $F\&\Phi(X, e_i)$ [5]. To implement this instruction, we need to specify three things explicitly: i) the operation (Φ) to be carried out, ii) the variable X on which the operation is to be carried out and iii) the operand e_i. In order for two Fetch&Φ instructions to be combinable, they must operate on the same variable X and perform the same operation Φ. Furthermore, Φ must be an associative and commutative operation.

Now consider the operand e_i of the $F\&\Phi(X, e_i)$ issued by processor P_i. In its complete generality, the operand e_i can be different from the operand e_j of instruction $F\&\Phi(X, e_j)$ issued by processor P_j , and the two can be combined used a hardware combining network. However, in most practical applications of the Fetch&Φ instruction in parallel processing, $e_i = e_j$ for arbitrary P_i and P_j [1,6]. For example, the Fetch&Φ instructions commonly used for parallel queue management are *Fetch&Add*(*QueueHead*, *EntrySize*) and *Fetch&Add*(*QueueTail*, *EntrySize*), where *EntrySize* is a constant, and those for barrier synchronization are *Fetch&Add*(*Var*, −1). This observation that e_i for all processors P_i executing $F\&\Phi(X, e_i)$ is the same is a key to a cheaper implementation of a combining Fetch&Φ instruction in a restricted (though widely applicable) form. In this restricted form of the Fetch&Φ instruction, $RF\&\Phi(X, C)$, all processors perform the same operation with the same operand (C) on the variable X. Thus, a restricted *Fetch&Add*(X, C) operation is the same as a Fetch&Increment by a constant (C) instruction.

2.2. Operation of Restricted Fetch&Φ (X, C)

Let us suppose that the processing system consists of N processors, and $M (M \leq N)$ processors issue a $RF\&\Phi(X, C)$ instruction. To combine the M $RF\&\Phi(X, C)$ instructions directed at a variable X, the only information we need to know is *which M* processors have issued the $RF\&\Phi(X, C)$ instruction since all processors have the same operand C. Thus, if *every* processor knows which M processors have issued the $RF\&\Phi(X, C)$ instruction, then using a preassigned priority scheme[1], each one of the participating M processors can carry out an appropriate number of $RF\&\Phi(X, C)$ instructions locally and obtain a unique value of X just as if the $RF\&\Phi(X, C)$ operation had been carried out using a combining network. This process is best illustrated by an example.

Consider a multiprocessor with 4 processors, P_1, P_2, P_3 and P_4. Suppose that 3 processors are ready to execute a $RF\&\Phi(X, C)$ instruction. The three values of X that represent a serialization of the instructions are:

$$V_1 = X\Phi(C)$$

$$V_2 = X\Phi(C\Phi C)$$

$$V_3 = X\Phi(C\Phi C\Phi C)$$

These values must now be assigned to the three processors to complete the implementation of the 3 $RF\&\Phi(X, C)$ instructions. Our general strategy for implementing these operations, without a combining network, is the following. By interacting with one another (via mechanisms to be described later) all the processors inform the others of their intention to execute the same $RF\&\Phi(X, C)$ instruction. Once a processor knows which other processors are also executing the same instruction, it can determine how many times it must apply the Φ operation to obtain a unique value (C, $C\Phi C$ or $C\Phi C\Phi C$ in the above example) by determining its position in the overall priority chain. One processor takes the responsibility of fetching X from the shared memory (or wherever it exists), and/or updating X with the final value. The memory could also be responsible for this instead. Once X has been fetched, it can be broadcast to all processors participating in the combining operation which in turn can carry out the final phase of the $RF\&\Phi(X, C)$ instruction locally. Thus, the variable X is accessed and updated only once and each processor gets a unique value of X, just as might occur in a combining network.

In the above example, suppose that processors P_1, P_3 and P_4 initiated the $RF\&\Phi(X, C)$ instructions and suppose that the processors are prioritized as P_1, P_2, P_3 and P_4. All processors realize that P_1, P_3, and P_4 are participating in a combinable $RF\&\Phi(X, C)$ instruction. Since P_1 has no processors with "a higher priority" also carrying the $RF\&\Phi(X, C)$ instruction, it takes responsibility for fetching X for all processors (if it does

[1]The use of the word "priority" may be a bit misleading. All that we need do is define and implement a serial ordering of the processors that are participating in the combining operation. In this paper, by prioritization we mean the enforcing of an arbitrary serial order, and by priorities we mean the relative positions in the serial order. Note that the ordering need not be static.

not exist locally), and calculating V_1. P_3 knows that one other processor with "a higher priority" has issued a $RF\&\Phi(X, C)$ instruction and, therefore, it is responsible for computing V_2 once X is known and using V_2 as its result of the $RF\&\Phi(X, C)$ instruction. Finally, P_4 computes V_3 locally, and updates X with the new value V_3. If the three processors generating $RF\&\Phi(X, C)$ instructions were P_1, P_2 and P_3 then, with the above prioritization scheme, P_1, P_2 and P_3 would compute and receive V_1, V_2 and V_3, respectively with P_1 fetching X from its remote location and distributing the value to all participating processors and, for example, P_3 updating the remote value of X.

In general, to implement the $RF\&\Phi(X, C)$ instruction with combining, each processor needs to determine which other processors are participating in the combining operation and each processor needs to know the old value of X (it could, of course, compute the old value from the new). Then, each processor can compute and use a unique value of X locally, based upon its position (with respect to others participating in the combining operation) in a preassigned priority chain (or serial order). Given the basic principle of having each processor determine its position in the serial order by determining how many higher priority processors are participating, let us now consider some hardware implementations of $RF\&\Phi(X, C)$.

2.3. Implementations of Restricted Fetch&Φ (X, C)

To implement a combining $RF\&\Phi(X, C)$, we need to solve two problems: i) determining which processors are executing the $RF\&\Phi(X, C)$ instruction and ii) conveying this information to all the participating processors.

To solve the first problem, some encoding of the processors participating in the operation is needed. Since there are a total of N processors, each of which could be participating in the operation, there are 2^N possible combinations and a minimum of N bits is needed[2]. A simple N-bit encoding is the unary encoding in which bit i is associated with processor P_i. P_i sets bit i if it is participating in the operation and resets it if it is not.

The second problem is easily solved by a broadcast interconnection network such as a bus. In fact, an N-bit-wide bus provides an easy solution to both problems; let us illustrate with an example.

Consider the 4-processor system of Figure 1. The processors are interconnected by a 4-bit wide bus with lines S_0, S_1, S_2 and S_3. For purpose of illustration, we shall assume that this bus is used solely for the purpose of implementing combinable $RF\&\Phi(X, C)$ operations. In general, this could be a separate synchronization bus, or even the primary interprocessor interconnect (as in a bus-based multiprocessor). Each processor monitors all lines of the bus and only processor P_i can write on line i of the bus. When processor P_i wants to carry out a $RF\&\Phi(X, C)$ operation, it grabs the bus and informs the other processors of its intentions. In the process, it also informs the other processors about the operation Φ and values of X and C (the broadcast of C may not be needed if the possible values of C that could be used are restricted). After this step, all processors interested in participating in the combining $RF\&\Phi(X, C)$ operation raise their respective line on the bus. In the next step, by determining which processors have stated their

[2]Note that theoretical encodings (such as a Huffman encoding) that require less than N bits on the average might be possible if particular combinations of processors participating in combining operations are more frequent than other combinations. However, we shall not consider them in this paper.

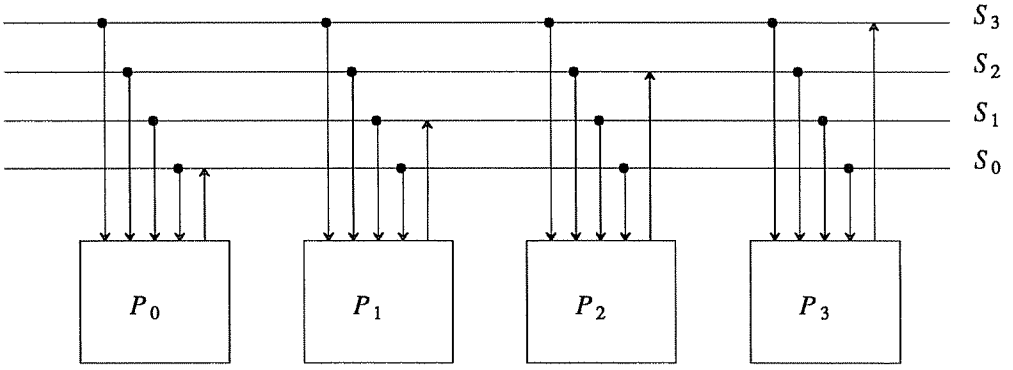

Figure 1: A 4-Processor System with a Broadcast Combining Bus

intentions about participating in the combining operation, each processor can carry out its $RF\&\Phi(X, C)$ operation locally as mentioned earlier. In this manner, up to N $RF\&\Phi(X, C)$ operations can be combined in a single bus cycle, where N is the smaller of the number of lines in the bus or the number of processors that are monitoring the bus. As we shall see in section 3.1, most of the hardware needed to implement these operations already exists in commercial bus-based multiprocessors, and this scheme can be used in such processors with little hardware modifications[3].

With the above implementation, the number of lines on the bus needs to be as large as the total number of processors participating to achieve a single-cycle combining operation. If the width of the bus is smaller that the number of processors connected to it, we can combine $RF\&\Phi(X, C)$ operations generated by up to $P \times D$ processors by breaking up the combining operation into P phases, where D is the number of lines in the broadcast bus. This is easily done by partitioning the processors into P groups of D processors each and by having processors in different groups state their intentions to combine using the same D lines, but on different phases of the broadcast-and-combine operation. Of course, additional control logic is needed to keep track of the different phases.

The broadcast capabilities of a single bus provide an elegant implementation of a combining $RF\&\Phi(X, C)$ operation since all processors can determine which others are participating simply by monitoring the bus. If combining is to be done with more processors than the number that can be connected to a single bus, we need to construct an alternate subsystem that achieves the purpose of informing each processor about its relative position in the overall serial ordering. This may be achieved through a combination of the bus-based implementation of $RF\&\Phi(X, C)$ for M processors in a cluster connected to a single bus and a conventional combining network that connects several clusters. The conventional combining network would treat each $RF\&\Phi(X, C)$ with M combined operations from a single cluster as a single $F\&\Phi(X, MC)$, where MC is the value that would have the same effect as M applications of the restricted Φ operation, and combine the different $F\&\Phi(X, MC)$ requests from the different clusters to determine the number of participants in the combining operation and consequently determine the serial ordering. The task of determining the serial ordering can also be achieved, possibly more simply, by the use of parallel prefix computation networks [3, 7]. Such networks are currently under study.

Once the serial ordering has been established, the computation done by a "processor" might be done explicitly in software, in hardware or both. Furthermore, because of the constant operand C of the $RF\&\Phi(X, C)$ operation, the actual operations carried out by the processor may be different from the Φ operation to improve the efficiency of computation. For example, the ith processor carrying out and $F\&Add(X, C)$ would not add C to X i times but rather would add the result of multiplying i and C to X.

2.4. Limitations of Restricted Fetch&Φ (X, C)

The $RF\&\Phi(X, C)$ operation described above suffers from two limitations when compared to a full-fledged $F\&\Phi(X, Y)$ operation implemented with a general hardware combining network: a) it is of limited use unless all the processors have the same value of Y and b) the use of bandwidth-limited buses may restrict its use for general-purpose combining. Let us elaborate on these limitations.

As mentioned earlier, most existing examples of $F\&\Phi(X, e_i)$ are actually $RF\&\Phi(X, C)$ [1]. Some applications, such as updates to a database, clearly are not [15]. Of the examples that are not, it is possible that some of them could be altered to exploit a $RF\&\Phi(X, C)$ operation. For the remaining, the proposed implementation will not be as effective as a general combining network. However, if many processors have the same value of Y, some combining can be carried out in the "average" case by using the $RF\&\Phi(X, C)$ implementation for the operation $F\&\Phi(X, Y)$, combining all requests with the same value of Y and repeating the process until all processors have completed their $F\&\Phi(X, Y)$ operation. Of course, in the worst case in which all processors have different values of Y, no combining can be carried out using the proposed $RF\&\Phi(X, C)$ implementation whereas it could be carried out in a more general combining network. Many times, however, it may be possible to reformulate the program to exploit an efficient $RF\&\Phi(X, C)$.

The second limitation is the bandwidth of the network that implements the $RF\&\Phi(X, C)$. Unlike a full-fledged combining network embedded in the regular data transfer network, the proposed network can not combine requests "on the fly" as they pass through the data transfer network (however, see the special case of bus-based multiprocessors in section 3.1). Combinable requests must be submitted explicitly to this network and other non-combinable requests should not because of the limited bandwidth of the network. This means that an *a priori* distinction must be made between the two types of requests. We do not perceive this to be a problem since many hot-spot requests that benefit from Fetch&Φ operations are typically to synchronization variables, queue pointers, etc., that are known at compile time.

[3] In Figure 1, we have only shown 4 lines of the bus. Since our scheme uses this bus to broadcast values of X and C, the bus should be wide enough to accommodate both.

This distinction between combinable and non-combinable requests with separate networks for each has also been made in the IBM RP3 [12].

3. Applications of Restricted Fetch&Φ

In this section, we consider some applications of the $RF\&\Phi(X, C)$ operation and its proposed implementation. In sections 3.1 and 3.2, we shall illustrate the use of $RF\&\Phi(X, C)$ in two commercial multiprocessor systems and in sections 3.3 and 3.4, we shall illustrate its use for the operations of barrier synchronization and parallel queue management, respectively.

3.1. Single-Bus, Snooping Cache Multiprocessors

An application for which $RF\&\Phi(X, C)$ is most easily adapted is the single-bus, shared-memory multiprocessor employing a "snooping" cache coherence protocol. Nearly all the hardware necessary to implement $RF\&\Phi(X, C)$ is already present in such a system. In a snooping protocol, all processors observe every memory operation, and intervene or accept data as necessary to maintain a single, consistent view of the shared memory. This means that every cache controller observes every memory operation, checking its tag memory for each bus operation. It is only a small extension to recognize that a local pending bus operation is a $RF\&\Phi(X, C)$ to the same address as the one appearing on the bus. When such a coincidence is recognized, the controller must intervene to effect the combining.

The combining can be accomplished by defining a new operation, different than either a read or a write, *i.e.*, there is data delivered to memory, and data returned. The data delivered is a unary representation of M, the number of processors involved in the combined operation. The data returned is X, the old value from memory, (it could also be the new value).

In fact, neither the old or the new value need be broadcast during the operation, since the participating processors may be assumed to have the old value, and they can compute the new value. However, other controllers that are not participating in the current operation must also perform the computation and update their local cache line in order to participate in a later operation. Transmitting the new value of the line would not only obviate such local computations among non-participants, but would also allow processors to participate even if they didn't already have the current value of X in their cache.

3.2. CRAY X-MP- and Y-MP-Style Parallel Processing

An effective technique for coordinating and synchronizing multiprocessor systems is used in the CRAY X-MP and Y-MP systems. Such a mechanism is indispensable if fine- to medium-grain parallelism at the loop level is to be exploited [8], *i.e.*, if parallelism is to be exploited by having each processor execute a loop iteration (or some number of them). After each processor has executed its task, it must access and obtain a new value for the loop iteration variable and begin executing a new loop iteration.

To facilitate the exploitation of fine- and medium-grain parallelism, the CRAY X-MP and Y-MP multiprocessors use a small memory organized as a shared register file. Data and control information can be passed through this register file. Registers can contain semaphore bits and can be operated upon with indivisible read-modify-write operations. In the actual implementations, the processors have their own identical copies of the shared register file [14]. If one processor wishes to modify the file, the change must be broadcast to the others. By having their own copies, busy-waits on control variables can be performed without using any system resources. Arbitration logic is needed to determine the winning processor when more than one processor tries to update the same register at the same time.

Access to a loop index variable (stored in the shared register file) is, at best, an $O(N)$ operation (where N is the number of processors) even with a complex implementation that prevents all processors accessing a lock for the register when it is released. If an implementation does not prevent all waiting processors from attempting to access a lock when it is released, $O(N^2)$ traffic can be generated on the system resources (see explanation for a similar phenomenon in section 3.4). This contention can increase the number of cycles spent in the synchronization phase and consequently decrease the benefit of fine-grain parallel processing [13].

To minimize the synchronization overhead (that is, implement the synchronization in $O(1)$ time) and still carry out fine-grain parallel processing, a combining $RF\&\Phi(X, C)$ implementation can be used for the shared register file (in fact, the proposed combining $RF\&\Phi(X, C)$ implementation is ideally suited for this situation). After the processors have used the bus (that connects the various copies of the shared register file) to determine which processors and how many want to perform a specific operation, they can all simultaneously read, modify and write their own copy of the variable in parallel. Hence, no broadcast of the modified variable is needed and, furthermore, only a single cycle (or a small constant number of cycles) is needed for each processor to obtain a unique copy of the loop index variable (as opposed to a best-case of N serial cycles and complex arbitration hardware that is needed for serializing

simultaneous writes without a combining operation).

3.3. Barrier Synchronization

Clearly, barrier synchronization can be implemented with $RF\&\Phi(X, C)$ by initializing X with a count of the number of processors to be synchronized and having each processor execute $RF\&\Phi(X, -1)$. When the count reaches zero, the processors are synchronized.

If barrier synchronization is the only form of synchronization to be done, then an even more restricted form of Fetch&Φ can be used, with a correspondingly more efficient implementation. This is precisely what was done in the Burroughs Flow Model Processor (FMP) [2, 10].

The proposed FMP had a "coordinator" unit with connections to all the processors. Interrupts and small diagnostic control messages could be sent from the coordinator, but its primary function was to combine synchronization information quickly from the 512 processors. The hardware combining could be made very efficient because the type of operation was given as an instruction to the coordinator, and all the enabled processors involved in synchronization performed the same operation.

The FMP provides an example of simple, fast hardware combining due to restricting generality and flexibility of synchronization mechanisms; the proposed implementation of $RF\&\Phi(X, C)$ can also be viewed as a generalization of this approach.

3.4. Parallel Queue Management

While barrier synchronization can also be implemented with synchronization primitives other than Fetch&Φ (since barrier synchronization does not require any intermediate results), scalable, parallel queue management is one application for which a combining Fetch&Add is currently the only known solution [1]. With a combining Fetch&Add operation, parallel access to queue elements can be carried out with $O(1)$ accesses to the queue pointers. Without a combining Fetch&Add, access to the queue pointers *can* result in $O(N^2)$ traffic on the processor-memory interconnect[4]. Even in the best case, $O(N)$ serial read-modify-write cycles have to be performed on the queue pointers.

As mentioned earlier, a $RF\&\Phi(X, C)$ operation could be used in situations where each processor takes from and adds to the queue, fixed-size portions of work. We believe that this is, by far, the most common case. When variable-sized portions of work are added and deleted, partial combining (as in section 2.4) might be possible.

A surprising (and pleasant) side-effect of using $RF\&\Phi(X, C)$ and the proposed combining hardware for parallel queue management is that the resulting code for inserting and deleting uniform-sized tasks into and from the queue is *actually simpler* than the equivalent code using a general Fetch&Add operation with a general combining network. To illustrate this, we consider the `Insert` procedure from [1].

```
Insert

If #Q_U≥ SIZE: Full

If F&A(#Q_U,1)≥SIZE:
   F&A(#Q_U,-1)
   Full

MyI <- Mod(F&A(#I,-1), SIZE)

Q[MyI] <- Data

F&A(#Q_L,1)
```

For simplicity, we have ignored the semaphores needed to prevent the access of meaningless data.

In the above code, the first four lines (with 2 F&A operations) are needed to prevent queue overflow. With the $RF\&\Phi(X, C)$ implementation suggested in this paper, the additional F&A operations to check for queue overflow are not necessary. This is because each processor can locally determine how many entries are being inserted by processors higher up in the serial order and, by comparing `SIZE` with the return value of the $RF\&\Phi(X, C)$ locally, insertions that would cause overflow can be converted to NOPs and resubmitted when `SIZE` changes. Of course, each processor would have to maintain a valid local copy of `SIZE`. Maintaining up-to-date local copies of `SIZE` is straightforward if the $RF\&\Phi(X, C)$ implementation also returns the total number of processors participating in a $RF\&\Phi(X, C)$ along with the number of higher priority processors so that updates to `SIZE` can be carried out locally.

[4] $O(N^2)$ traffic results when a lock must be set and released to update the queue pointers. When a lock is released, $O(N)$ processors that are waiting to grab the lock attempt to get it, thereby causing $O(N)$ traffic each time the lock is released. Only 1 processor can get the lock and the remaining processors have to try again when the lock is released. This leads to $O(N^2)$ accesses to the lock variable and $O(N^2)$ traffic on the interconnect. Note that if any processors are spinning over the interconnect the traffic can be much worse than $O(N^2)$.

4. Conclusions

In this paper, we have discussed a restricted form, $RF\&\Phi(X, C)$, of the general Fetch&Φ operation and have shown how this restricted form can lead to simpler and cheaper hardware combining implementations than in the general case. In this restricted form, all participating processors have the same value of both operands of the Fetch&Φ operation. We presented hardware implementations for $RF\&\Phi(X, C)$ and considered some practical situations in which a $RF\&\Phi(X, C)$ operation can perform the same functional tasks as the general Fetch&Φ operation. The proposed implementation is particularly suitable for bus-based snooping cache multiprocessors and other multiprocessors that have a small, shared synchronization memory. While a $RF\&\Phi(X, C)$ operation does not have the complete functionality of the general combining Fetch&Φ operation, we believe that it constitutes a good engineering solution in many situations and should be considered in systems where hardware combining (especially to a synchronization memory) is desirable but implementing a general combining Fetch&Φ operation is too expensive.

Acknowledgments

This work was supported in part by NSF grants CCR-8706722 and MIP-8604224. The authors would also like to thank the anonymous reviewers for their helpful comments.

References

[1] G. S. Almasi and A. Gottlieb, *Highly Parallel Computing*. Redwood City, CA: Benjamin/Cummings Publishing Company, Inc., 1989.

[2] Burroughs Corporation, "Final Report Numerical Aerodynamic Simulation Facility Feasibility Study," March 1979.

[3] F. E. Fich, "New Bounds for Parallel Prefix Circuits," in *Proc. 15th Annual ACM Symposium on Theory of Computing*, Boston, MA, pp. 100-109, April, 1983.

[4] J. R. Goodman, M. K. Vernon, and P. J. Woest, "A Set of Efficient Synchronization Primitives for a Large-Scale Shared-Memory Multiprocessor," in *Proc. ASPLOS-III*, Boston, MA, April 1989.

[5] A. Gottlieb, et al, "The NYU Ultracomputer -- Designing a MIMD, Shared Memory Parallel Machine," *IEEE Transactions on Computers*, vol. C-32, pp. 175-189, February 1983.

[6] A. Gottlieb, B. D. Lubachevsky, and L. Rudolph, "Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors," *ACM TOPLAS*, vol. 5, pp. 164-189, April 1983.

[7] R. E. Ladner and M. J. Fischer, "Parallel Prefix Computation," *JACM*, vol. 27, pp. 831-838, October 1980.

[8] J. L. Larson, "Multitasking on the Cray X-MP-2 Multiprocessor," *IEEE Computer*, pp. 62-69, 1984.

[9] G. J. Lipovski and P. Vaughn, "A Fetch-and-Op Implementation for Parallel Computers," in *Proc. 15th Annual Symposium on Computer Architecture*, Honolulu, HI, pp. 384-392, June 1988.

[10] S. F. Lundstrom, "Applications Considerations in the System Design of Highly Concurrent Multiprocessors," *IEEE Transactions on Computers*, vol. C-36, pp. 1292-1309, November 1987.

[11] G. F. Pfister and V. A. Norton, "'Hot-Spot' Contention and Combining in Multistage Interconnection Networks," *IEEE Transactions on Computers*, vol. C-34, pp. 943-948, October 1985.

[12] G. F. Pfister, et al, "The IBM Research Parallel Processor Prototype (RP3): introduction and architecture," *Proceedings 1985 International Conference on Parallel Processing*, pp. 764-771, August 1985.

[13] C. D. Polychronopoulos, "The Impact of Run-Time Overhead on Usable Parallelism," *Proceedings 1988 International Conference on Parallel Processing*, August 1988.

[14] R. D. Pribnow, "System for Multiprocessor Communication Using Local and Common Semaphore and Information Registers," *United States Patent 4,754,398*, June 1988.

[15] H. S. Stone, "Database Applications of the FETCH-AND-ADD Instruction," *IEEE Transactions on Computers*, vol. C-33, pp. 604-612, July 1984.

[16] P.-C. Yew, N.-F. Tzeng, and D. H. Lawrie, "Distributing Hot-Spot Addressing in Large Scale Multiprocessors," *IEEE Transactions on Computers*, vol. C-36, pp. 388-395, April 1987.