Parallel Processing Architecture for the Hitachi S-3800 Shared-Memory Vector Multiprocessor

Katsuyoshi Kitai, Tadaaki Isobe(*), Yoshikazu Tanaka, Yoshiko Tamaki, Masakazu Fukagawa(*), Teruo Tanaka, and Yasuhiro Inagami

Central Research Laboratory, Hitachi, Ltd.
1-280, Higashi-koigakubo, Kokubunji, Tokyo 185, Japan.
(*) General Purpose Computer Division, Hitachi, Ltd.
1, Horiyamashita, Hadano, Kanagawa 259-13, Japan.

Abstract

This paper discusses the architecture of the new Hitachi supercomputer series, which is capable of achieving 8 GFLOPS in each of up to four processors. This architecture provides high-performance processing for fine-grain parallelism, and it allows efficient parallel processing even in an undedicated environment. It also features the newly-developed time-limited spin-loop synchronization, which combines spin-loop synchronization with operating system primitives, and a communication buffer (CB) which caches shared variables for synchronization, thus allowing them to be accessed faster. Three new instructions take advantage of the CB in order to reduce the parallel overhead. The results of performance measurements confirm the effectiveness of the CB and the new instructions.

1. Introduction

Several supercomputers with vector processors have been introduced over the last decade. Since vector multiprocessors first appeared in the Cray X-MP[1], more powerful systems, like the Cray Y-MP/8E[2], Y-MP C90[3], and NEC SX-3[4] have evolved. The Hitachi supercomputer S-3800[5] series is upwardly compatible with its single-processor predecessors, the S-810[6] and S-820[7,15-18] series.

ICS-7/93 Tokyo, Japan

The four-processor S-3800 model is capable of a state-of-the-art performance of 32 GFLOPS, and its computing power is appropriate for the central machine of networked computing environments. Its system throughput is high not only because it can execute many programs concurrently, but because its automatic vectorizing and parallelizing facilities reduce the execution time of each single program.

In developing the multiprocessor system of the S-3800, we had two major goals. One was to produce a high-performance parallel processing architecture capable of supporting fine-grain parallelism down to the level of two-dimensional loop structures (where each dimension is about 100 elements and the inner loop is vectorizable), while maintaining binary compatibility with the S-810 and S-820 series. Our other goal was to achieve efficient parallel processing — even in an undedicated environment — without wasting vector processor cycles in synchronization.

Figure 1 shows the theoretical effects of running four processors in parallel. The data plotted here were derived under the following three assumptions: (1) the granularity of each parallel operation is neither vectorized nor parallelized; (2) the vectorizable part is also parallelizable; and (3) vector execution is ten times as fast as scalar execution. This figure clearly shows that for the same granularity, vector multiprocessor systems need more efficient synchronization and a higher parallelization ratio than scalar multiprocessor systems. The finer a program's grain parallelism, the more opportunity there is to parallelize the program — either automatically or directly. Furthermore, the finer the granularity, the less the parallel overhead. We therefore

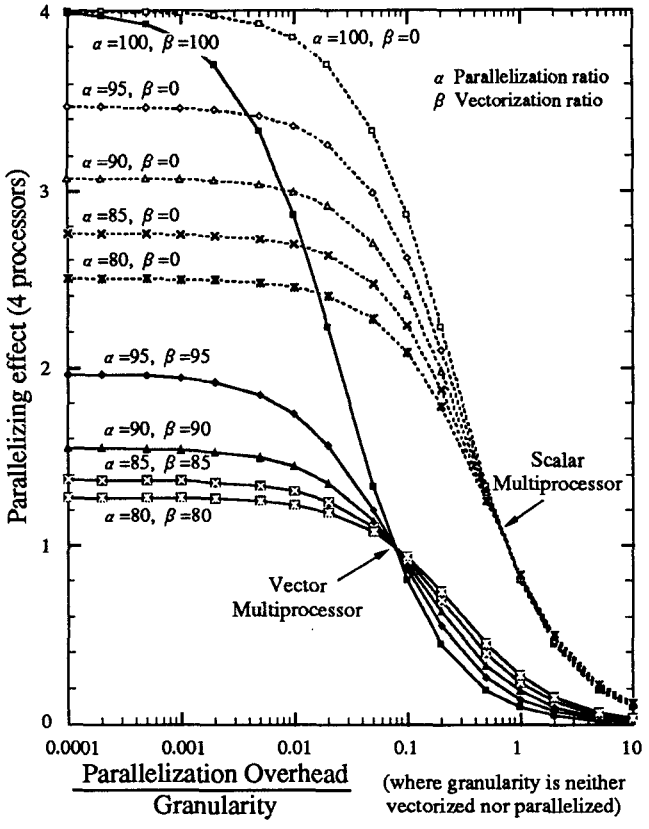


Fig. 1. Parallelization effect on scalar and vector multiprocessor systems.

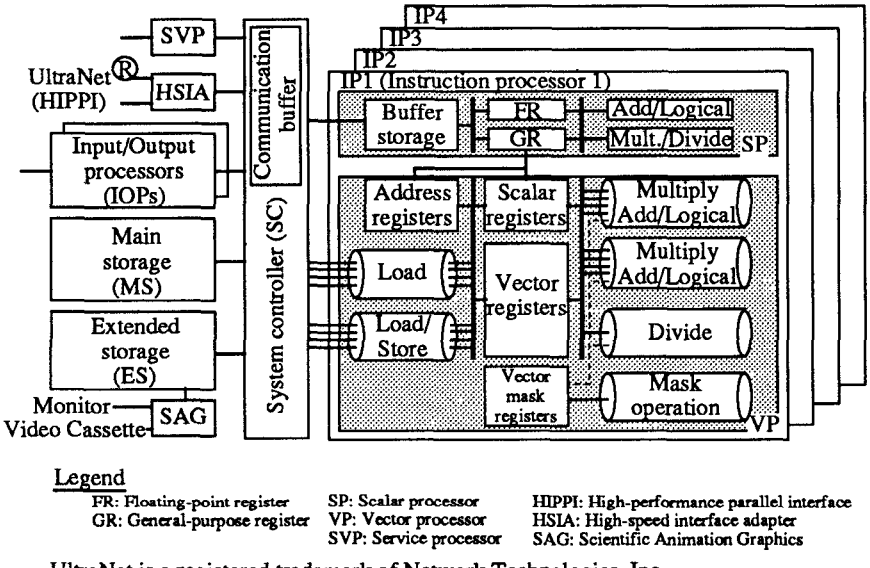
decided to use a new hardware mechanism in order to provide faster event[10] synchronization (by a new spin-loop method), and shorter hardware-lock[10] time for mutual exclusion.

A parallelized program may be executed on the single-processor model of the S-3800 series, and it may also be executed in an undedicated environment (especially when debugging or testing a parallel program). In an undedicated environment, the number of processors assigned by the operating system depends on the program's priority, so the execution time of a parallel program and the waiting time for event synchronization would vary every time the program is executed. When the waiting time is long, the processor cycles for the other programs are wasted. This drawback calls for a new synchronization feature that does not lose cycles, thus stabilizing the CPU time and account time.

We begin this paper with a brief overview of processor organization in the S-3800 series, and then describe the parallel processing model. We then propose a new synchronization feature (time-limited spin-loop), a new instruction (compare-and-wait), and a communication buffer (CB) which keeps the variables for synchronization. Finally, we describe the experimental performance of the two- and four-processor models.

2. Processor Organization of the S-3800

The processor organization of the S-3800 (Fig. 2) consists of four scalar and vector processors (SPs and VPs), a system controller (SC), main storage (MS), extended storage (ES), input/output processors (IOPs), and a service processor (SVP). Each scalar processor is compatible with Hitachi's general purpose main frame M-series, supports three new instructions for multiprocessing, and thirty-one instructions for controlling the vector processing, and also contains 256-KB store-through buffer storage (BS). Each vector processor has two add/multiply pipelines, one divide pipeline, one mask pipeline, 128 KB of vector registers, and 1 KB of vector mask registers. It also has one load/store pipeline and one load pipeline to and from the main storage. The communication buffer (CB), which enables efficient parallel processing, will be discussed in Section 4.



UltraNet is a registered trademark of Network Technologies, Inc.

Fig. 2. Processor organization of the S-3800.

3. Parallel Processing Model

Before discussing the multiprocessing architecture of the S-3800, we will briefly overview its parallel processing scheme, with reference to the development goals — especially those related to synchronization primitives — described in Section 1. These goals can only be reached if the interaction with the operating system is reduced and flexible parallel processing control is provided. As a result, we adopted a two-level scheme (Fig. 3) for scheduling parallel sub-tasks (or parallel operations), similar to the two-level schemes used in IBM's Parallel FORTRAN[8] and PCF FORTRAN for advanced machines[9]. The automatic parallelizing compiler and the compiler directives identify parallel sub-tasks, a run-time scheduler dispatches the sub-task to one of the logical processors, and the operating system assigns the logical processor to a real processor.

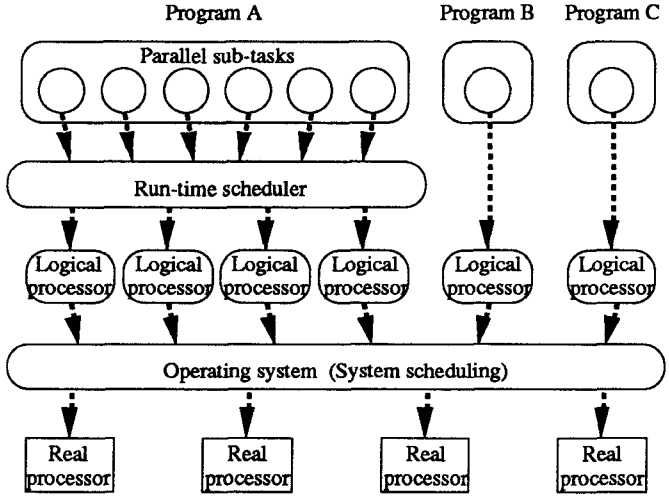


Fig. 3. Two-level scheduling scheme.

The parallelizing compiler extracts three types of parallel structure: loop structures, case structures, and inter-subroutines or inter-function structures. Serial structures and parallel structures appear in the program one after the other (Fig. 4). A fork primitive (at the beginning of parallel structures) creates the parallel sub-tasks, and the join primitive (at the termination of parallel structures) provides barrier synchronization for the sub-tasks.

A copy of the run-time scheduler is allocated to each program, and each copy manages three resources to execute a program in parallel: the logical processors that execute sub-tasks, the dispatching queue in which sub-tasks are set, and the free list of the work memory used as temporary variable storage by the sub-tasks. The fork primitive creates children sub-tasks. The run-time scheduler inserts these sub-tasks into the dispatching queue and signals the event to the logical processors in order to start parallel execution. Immediately after receiving the signal, the logical processors take a sub-task from the dispatching queue, get the necessary work memory, and start executing the sub-task. When they have finished, they take another sub-task from the dispatching queue. This means that as long as there are sub-tasks in the queue, the processors do not need to interact with the operating system. In addition, this scheme enables parallel processing to be achieved independent of the number of real processors assigned by the operating system. During the execution of the join primitive, the parent task waits until all the sub-tasks have finished their work, this event being signalled from the last sub-task to the parent task.

There are two types of synchronization primitives: locks and events[10]. Locks may be used to ensure mutual exclusion in accessing a critical region in the following cases: when a sub-task is inserted into or deleted from the

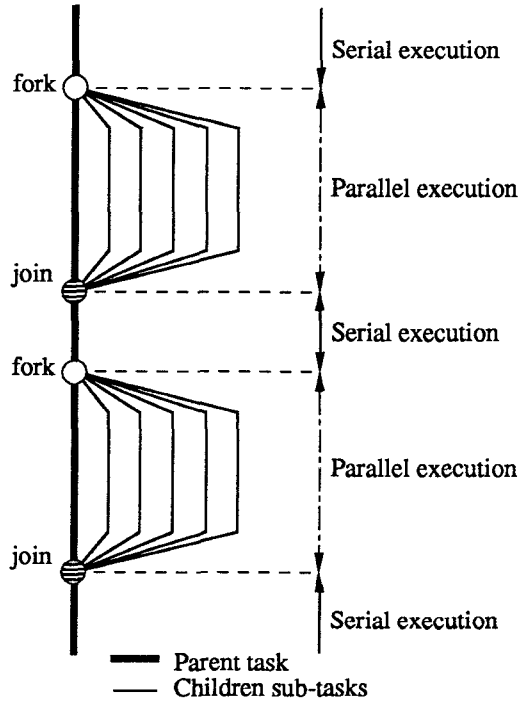
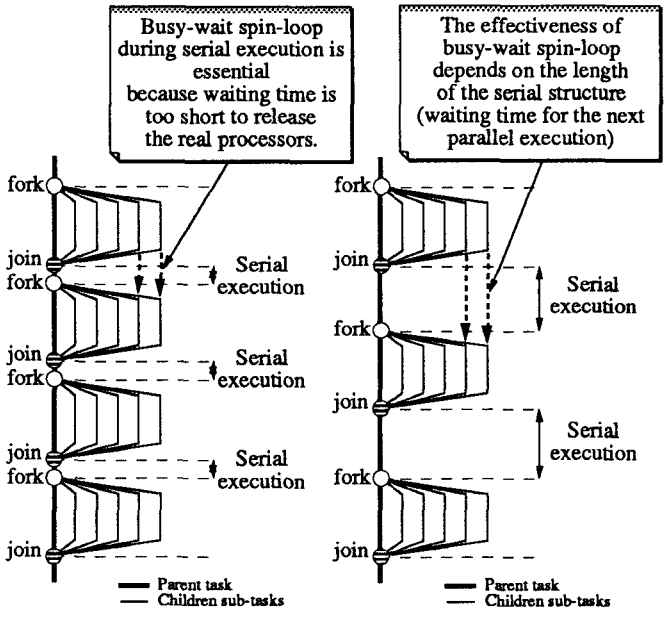


Fig. 4. Structure of the parallel program.

dispatching queue, and when increasing or decreasing the counters for parallel loops or for identifying the last sub-task. The compare-and-swap operation[10] is appropriate for the former case, whereas the fetch-and-add operation[10] is appropriate for the latter case. Notice that resolving lock contention is the main feature of parallel processing. Lock contention generally occurs in two cases. The first is when the header of the dispatching queue is accessed by all the logical processors immediately after the sub-tasks are created, and the second occurs if the granularity of each sub-task is nearly the same in which case the counter in the parent task which is used to identify the last sub-task may be updated by several sub-tasks at about the same time. It is therefore important to reduce the hardware-lock time for mutual exclusion.

The other synchronization primitives, events, may be used to force a sub-task to wait until another sub-task has reached a certain point. They are used in starting logical processors or in waiting for all the sub-tasks to finish their work.



(a) high parallelization ratio (short serial structures)

(b) intermediate parallelization ratio (long serial structures)

Fig. 5. Waiting times before fork primitives.

4. Multiprocessing Architecture of the S-3800

The S-3800 multiprocessing architecture incorporates four new features: a time-limited spin-loop function used when waiting for events, a communication buffer (CB) for faster synchronization and shorter hardware-lock time, a spin-loop timer for measuring the total spin-loop time of single programs, and three new instructions for synchronization — compare-and-wait (CW), fetch-and-add (FA), and store-and-set CB (STSCB). In addition, the M-series synchronization instructions are also supported: compare-and-swap (CS), compare-double-and-swap (CDS), and test-and-set (TS).

4.1 Time-Limited Spin-Loop

Two methods are generally available for event synchronization: operating system (OS) primitives (post/wait macros); and the busy-wait spin-loop method. The spin-loop method is required for loop-level fine-grain parallelism on the vector multiprocessor system because the OS primitives take 10 to 100 times longer[11]. The disadvantage of this method, however, is that a long busy-wait time might reduce the system throughput in an undedicated environment. However, in a dedicated environment, we can be fairly certain that the spin-loop method causes no problems.

Here we shall look closely at which method is appropriate for two-level scheduling in undedicated systems. Wait operations are used either when waiting for the creation of new sub-tasks, or when waiting for the completion of all children sub-tasks. In the first case, the waiting time depends on the parallelization ratio, which is related to the length of the serial structures between one parallel structure and the next. When the parallelization ratio is high enough, the serial structures will be so short that the spin-loop method is essential — and, better, the waiting time is expected to be short enough so as not to spoil the system throughput [Fig. 5(a)]. At intermediate parallelization ratios, when the program benefits from parallelizing only if the logical processors are assigned to the same real processors during the execution of serial structures, the spin-loop method is effective [Fig. 5(b)]. If the program does not benefit from parallelizing, the logical processors should release their real processors by using OS primitives. When the ratio is low, the program should not be parallelized at all, but if this is

unavoidable, the OS primitives are used in order not to spoil the system throughput.

The waiting time for the completion of all children sub-tasks depends on the granularity of each parallel sub-task. As long as we use the two-level scheduling scheme described in Section 3, the waiting time of the parent task is less than the execution time of the last sub-task. With fine-grain parallelism, the spin-loop method is essential in order to reduce the synchronization time [Fig. 6(a)], but with coarse-grain parallelism the short synchronization time achieved by the spin-loop method is not essential [Fig. 6(b)]. OS primitives should therefore be used in order to avoid losing too many cycles.

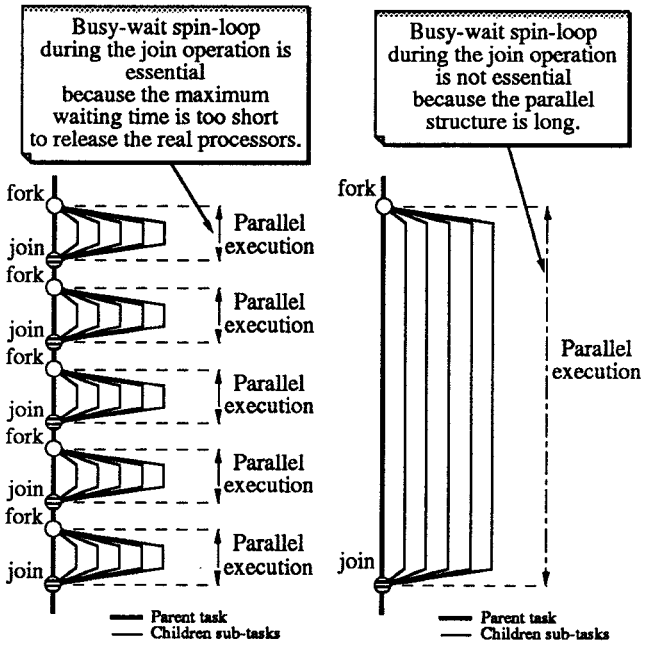


Fig. 6. Waiting times before join primitives.

We therefore decided to combine the spin-loop method with OS primitives to create what we call the time-limited spin-loop method. The first characteristic of this method is that the busy-wait spin-loop method is used initially to achieve fast synchronization, and then after a predetermined time the OS primitives are used in order to keep the system throughput high. The second characteristic is that the busy-wait spin-loop is not terminated by the hardware (like the Cray X-MP/Y-MP deadlock interruption[2,3]) but by software. The time-limited spin-loop method is therefore effective not only for fine-grain parallelism but also for parallel processing in an undedicated environment.

The time-limited spin-loop method involves the following procedures:

(1) The time limit of the busy-wait spin-loop is decided.

(2) Until the limit time is reached, a busy-wait spin-loop is used to wait for an event.

(3) After the time limit, the run-time scheduler sets a flag to indicate that the task is waiting in the OS primitive (wait macro) mode instead of the spin-loop mode, and releases the real processor. In dispatching another program, the operating system will consider the system balance. This flag is used to judge whether a store instruction or an OS primitive (post macro) should be used to send the event to the waiting sub-task.

The limit time of the spin-loop is the key issue for this feature, and it should meet two conditions. One is that in fine-grain parallelism, the limit time should be more than the granularity. The other is that at intermediate parallelization ratios, the limit time should be as long as the serial structure. Deciding the limit time, however, is an issue that still needs to be discussed.

4.2 Spin-Loop Timer and the Compare-and-Wait Instruction

The time-limited spin-loop feature can be implemented by using the existing compare and conditional branch instructions. Although this has the advantage that the scalar architecture does not have to be expanded or changed, we did not use this technique for two reasons.

(1) Inter-processor communication via the main storage is slow [Fig. 7(a)]. The event is first stored in the main storage, and the inconsistency with the buffer storage is detected in every scalar processor that was waiting for the event. This results in a block transfer from the main storage to the buffer storage, and only then is the condition of the synchronization met.

(2) An interrupt might occur during the busy-wait spin-loop. Without interaction with the operating system, there would then be no way to measure the busy-waiting time. In addition, the busy-waiting time would vary from

one execution to the next in an undedicated system. It is meaningless to interact with the operating system while busy-waiting to achieve fast synchronization.

For these two reasons, we decided to introduce a new instruction, compare-and-wait (CW), and a spin-loop timer. The CW instruction operates on the communication buffer (CB), which is shared by all the scalar processors and which follows a store-through protocol. The CB will be discussed in detail in the next subsection.

The CW instruction operates as follows:

(1) There are three operands: (a) maximum spin-loop count, (b) location of the event synchronization variable, and (c) expected value to establish synchronization.

(2) The expected value is compared with the contents of the location, which are fetched directly from the CB, bypassing the buffer storage and decrementing the spin-loop count.

(3) If the value and the contents are equal, the instruction is completed with condition code 0. Otherwise steps (2) and (3) are repeated until the spin-loop count is zero.

(4) If the spin-loop count is zero, the instruction is completed with condition code 2.

(5) The CW instruction is interruptible in step (3).

Because the CW instruction uses the CB for event synchronization, and since the CB is the cache shared by all the processors, this instruction causes no inconsistency and it does not require block transfers [Fig. 7(b)]. We can therefore achieve fast event synchronization by combining the new CW instruction and the CB. The weak point of the CW instruction is that the synchronization points are more sparse than they are when the existing compare and conditional branch instructions are used. This is because the CB is farther away from the processors than the buffer storage.

The spin-loop timer is used to time the busy-wait spin-loops. This timer is incremented whenever a CW instruction is executed. The operating system can both read from and write to the spin-loop timer, and its value is used to optimize the performance of programs executed on dedicated systems. In undedicated systems, the total spin-loop time would be different in every execution, but to enable a stable CPU time or account time to be provided, the operating system can use the spin-loop timer value to calculate the account time and to carry out enforced job termination when excessive CPU time has been used.

4.3 Communication Buffer, Store-and-Set CB Instruction

In a shared-memory multiprocessor system, parallel sub-tasks execute their own codes and synchronize with each other when using shared variables. Such shared variables are usually copied into the private buffer storage of each scalar processor, but when one of the processor updates a variable, all the copies in the buffer storages of the other processors should reflect this change. This is carried out by block

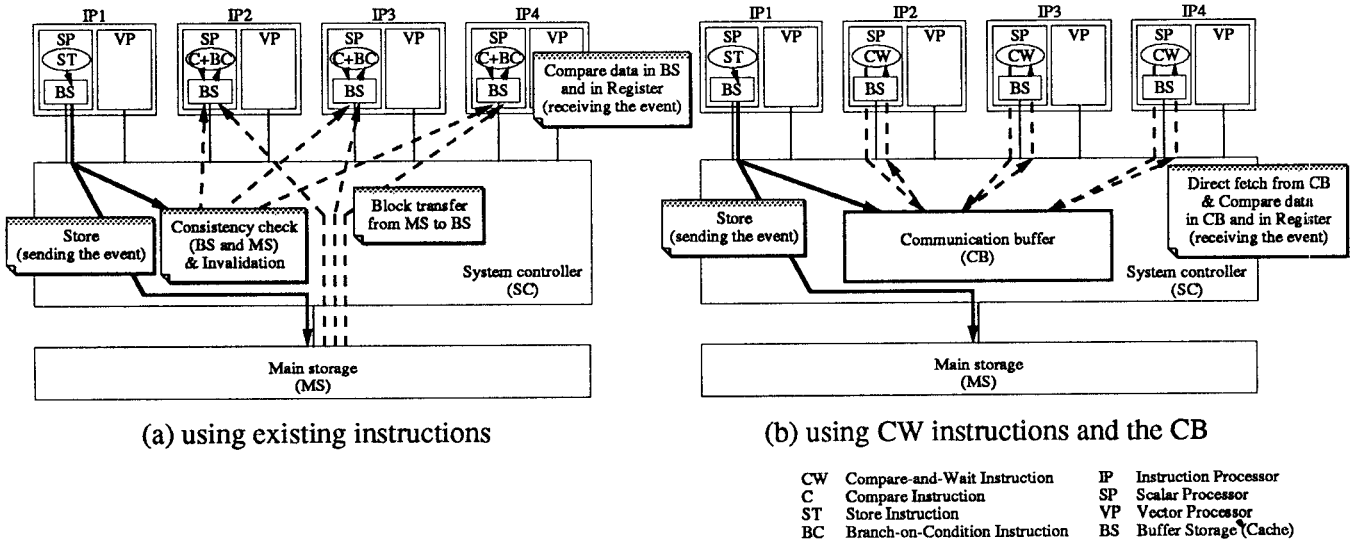


Fig. 7. Two methods for event synchronization.

transfers from the main storage to all the buffer storages, causing some degradation of performance.

The communication buffer (CB) is a store-through buffer storage (cache) which is shared by all the scalar processors, and which follows a set-associative mapping replacement protocol. This buffer only caches the data used for synchronization, namely the operands for these five instructions: compare-and-wait (CW), fetch-and-add (FA), compare-and-swap (CS), compare-double-and-swap (CDS), and test-and-set (TS). The first two instructions are new features of the S-3800, and the last three already exist in the M-series architecture*. The operands of these instructions are fetched directly from the CB, bypassing the buffer storage. By restricting the data held in the CB to synchronization data, the performance of vector store operations is not degraded. Because the synchronization data is never accessed by the vector processors, there is no need for block transfer from the main storage to the CB.

Although the CB is primarily transparent to the software, it is essential for the run-time scheduler to take full advantage of the CB in order to provide high-performance parallel processing. We therefore created a new instruction, store-and-set CB (STSCB), that initializes the synchronization variables and sets them to the CB. In this instruction, four bytes from a general-purpose register are stored into the main storage, and then copied to the CB with a block transfer operation.

In implementing the CB, we had the alternatives of the shared-registers approach or the CB approach. The shared-registers approach has been adopted in several vector multiprocessor systems, such as the Cray X-MP/Y-MP series, the NEC SX-3 series, and the Convex C2/C3 series[12]. This approach has the advantages of a shorter access time and simpler implementation, but the CB approach has more advantages for the S-3800 series.

First of all, there are no restrictions of the number of synchronization variables. In addition, it is impossible to save a shared register into the main storage temporarily, because these registers are shared by other processors, some of which might be in use. This limitation of the shared-registers approach restricts the programming freedom, whereas the CB is primarily transparent to the software and thus offers more flexibility.

Secondly, the M-series architecture already has CS, CDS, and TS instructions for achieving mutual exclusion from critical regions. In executing these instructions, the hardware lock is set while data in the main storage is updated. In the CB approach, these instructions can also take advantage of the CB because the access time of the CB is shorter than that of the main storage.

Third, the shared registers must be saved and restored when the program is swapping out or swapping in.

Finally, the CB approach is more reliable because the CB only stores a copy of part of the main storage. Even if the CB hardware is damaged, it can be disconnected without bringing the entire system to a halt.

4.4 Fetch-and-Add Instruction

The compare-and-swap (CS) instruction modifies the data only if it has not been changed by any other processors and its value is what the CS instruction expects. This means that if lock contention has occurred and another processor has changed the data, it is necessary to loop back in preparation for updating and try again (Fig. 8). It follows that a long hardware lock time and a long execution time would be required to complete the lock synchronization for mutual exclusion.

```
        L     GR1, COUNTER
RETRY   LR    GR0, GR1
        A     GR0, "1"
        CS    GR1, GR0, COUNTER
        BNE   RETRY
```

Fig. 8. Synchronization using the CS instruction (Increment by '1' operation).

The fetch-and-add operation, as used in the NYU Ultracomputer[13,14], is particularly effective for updating counters in critical regions, since this operation is capable of waiting without looping back. When increasing or decreasing the counters for parallel loops or for identifying the last sub-task, this operation would therefore result in much less lock time and execution time than would result from using the CS instruction. Furthermore, parallel loops

* The scalar architecture of the S-3800 is based on Hitachi's main frame M-series architecture.

tend to appear in a fine-grain parallelism. We therefore decided to introduce a fetch-and-add (FA) instruction in which an integer variable in the CB is increased according to a variable in a general-purpose register, and the increased variable is loaded into the general-purpose register.

Because almost all the counters used in parallel processing are either incremented or decremented by one, we have classified the FA instruction into two types. When the increment value is '1' or '-1', the operation is carried out inside the CB with a little additional hardware. Otherwise, the operation is carried out in the scalar processor after fetching the counter value from the CB. Operating this increment or decrement in the CB further reduces both the lock time and the execution time.

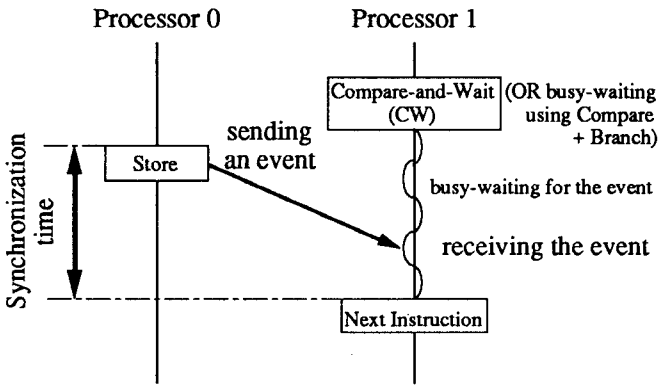


Fig. 9. Synchronization time for an event primitive.

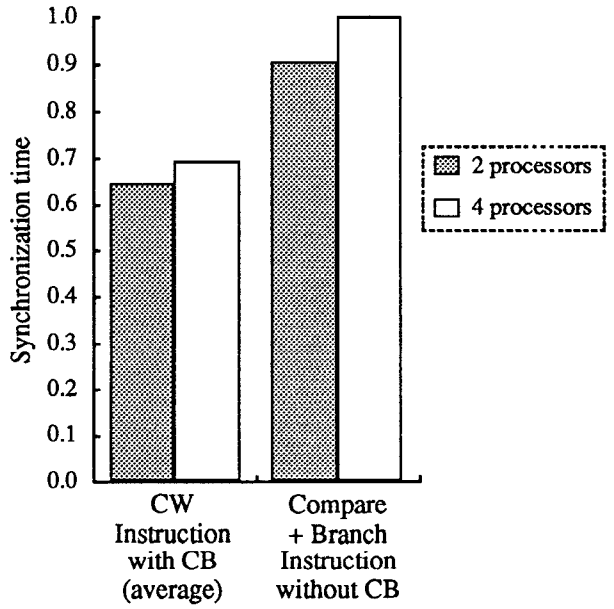


Fig. 10. The effect of the CW instruction and the CB on event synchronization.

5. Performance Evaluation

We measured the performance of the event and lock synchronization primitives on the two-processor and four-processor models. To precisely evaluate the ways in which the synchronization primitives are affected by the CB and the new instructions, we simulated the logical processes of the S-3800.

The event synchronization time can be defined as the total time from when a processor sends an event until all the other processors have received the event and started executing the next instruction (Fig. 9). We used the store instruction for sending an event, and we used the CW instruction with the CB for receiving an event. For comparison, we also used a combination of compare and branch instructions for receiving an event. By eliminating block transfers, the former process reduced the event synchronization time by an average of 30% (Fig. 10).

The maximum lock synchronization time can be expressed as total time needed for all the processors to serially update the lock word (Fig. 11). We evaluated the time taken to increase the counter by one using an FA instruction with the CB, which we compared with CS instructions both with and without the CB. We inserted some instructions in the program which made all the processors start executing the FA or the CS instructions at the same time. The difference between the performance of the FA instruction and that of the CS instruction clearly shows the advantage of the former technique (Fig. 12). This difference includes the effects of executing unity increment operations in the CB in case of the FA instruction. As a result, the execution time of unity increments and decrements was reduced by 75% in the two-processor model and by 86% in the four-processor model. Even without the FA instruction, the effect of the CB on speeding up CS instructions is also significant: the execution time of the CS instruction was reduced by 39% in the two-processor model and by 42% in the four-processor model.

Figure 13 shows the overall effect of introducing the CB and the new instructions. The vertical axis shows the parallelization effect with four processors in executing a DAXPY operation ("V=S*V+V"), where V is the vector data (or one-dimensional matrix data) and S is the scalar data. This figure shows the result of a preliminary evaluation assuming the following conditions: (1) The program has a

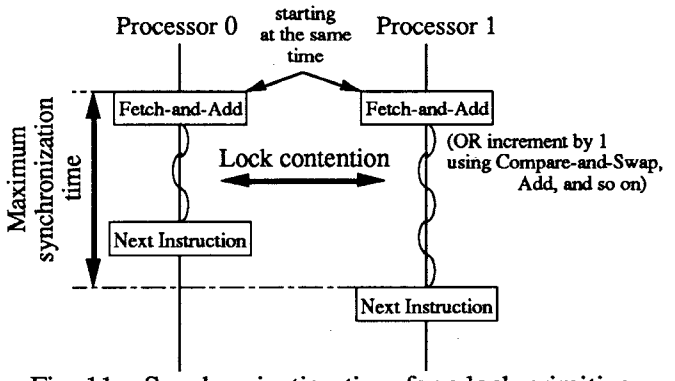


Fig. 11. Synchronization time for a lock primitive.

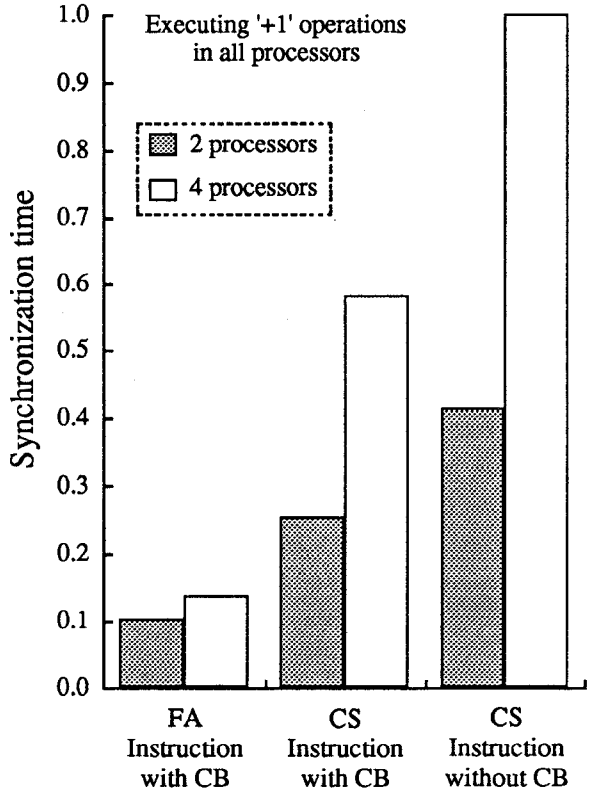


Fig. 12. The effect of the FA instruction and the CB on lock synchronization.

two-dimensional loop structure, where inner loop ("V=S*V+V") is vectorized and the outer loop is divided into four parallel segments. The horizontal axis shows the size of each dimension. (2) During the whole execution of the parallel loop structure, the event wait operation (CW instruction) and the lock contention (FA instruction) each appear twice. The difference between the two lines represents the advantages of the CB and the new instructions introduced in the S-3800.

6. Conclusions

We have presented the parallel processing architecture of the Hitachi supercomputer S-3800 series, which features a time-limited spin-loop using a new compare-and-wait (CW) instruction and a spin-loop timer, a communication buffer (CB), a store-and-set CB (STSCB) instruction, and a fetch-and-add (FA) instruction. The run-time scheduler entirely controls parallel processing, taking full advantage of these features.

The time-limited spin-loop feature was shown to be essential for practical parallel processing in an undedicated environment, as well as for extremely high-speed parallel processing with fine-grain parallelism. Performance evaluations have demonstrated that the CB and the three new instructions greatly improve the performance of parallel processing.

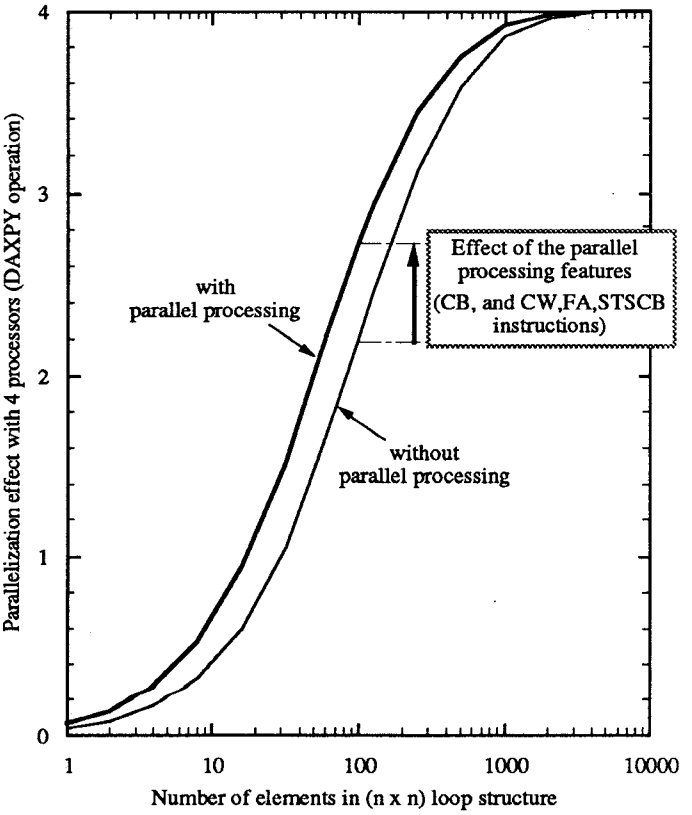


Fig. 13. Effect of the parallel processing features in the S-3800.

[Conditions: a) 2-dimensional loop divided into 4 segments, b) CW and FA instructions appear twice each (for loop parallel).]

References

[1] M.C. August, G.M. Brost, C.C. Hsiung, and A.J. Schiffleger, "Cray X-MP: The Birth of a Supercomputer," *IEEE Computer*, pp. 45-52, Jan., 1989.

[2] U. Detert, and G. Hofemann, "Cray X-MP and Y-MP memory performance," *Parallel Computing 17*, North-Holland, pp. 579-590, 1991.

[3] W. Oed, "Cray Y-MP C90: System features and early benchmark results," *Parallel Computing 18*, North-Holland, pp. 947-954, 1992.

[4] A. Iwaya, and T. Watanabe, "The Parallel Processing Feature of the NEC SX-3 Supercomputer System," *International Journal of High Speed Computing*, Vol. 3, Number 3 & 4, 1991.

[5] K. Ishii, H. Abe, S. Kawabe, and M. Hirai, "An Overview of the Hitachi S-3800 Series Supercomputer," *Proc. of Supercomputer '92*, pp. 65-81, Jun., 1992.

[6] S. Nagashima, Y. Inagami, T. Odaka, and S. Kawabe, "Design Consideration for a High-Speed Vector Processor: The Hitachi S-810," *ICCD '84*, pp. 238-243, Oct., 1984.

[7] T. Odaka, S. Kawabe, and H. Wada, "Development of Hitachi Supercomputer S-820 System," *Proc. of the third International Conf. on Supercomputing*, pp. 71-77, 1988.

[8] L.J. Toomey, E.C. Plachy, R.G. Scarborough, R.J. Sahulka, J.F. Shaw, and A.W. Shannon, "IBM Parallel FORTRAN," *IBM Systems Journal*, Vol. 27, No. 4, pp. 416-435, 1988.

[9] The Parallel Computing Forum, "PCF FORTRAN: Language Definition," Aug., 1988.

[10] K. Hwang, and F.A. Briggs, "Computer Architecture and Parallel Processing," *McGraw-Hill Inc.*, 1985.

[11] P. Carnevali, P. Sguazzero, and V. Zecca, "Microtasking on IBM Multiprocessors," *IBM Journal of Research and Development*, Vol. 30, No. 6, pp. 574-582, Nov., 1986.

[12] T. Jones, "Engineering Design of the Convex C2," *IEEE Computer*, pp. 36-44, Jan., 1989.

[13] A. Gottlieb, R. Grishman, C.P. Kruskal, K.P. McAuliffe, L. Rudolph, and M. Snir, "The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer," *IEEE Trans. Computers*, pp. 175-189, Feb., 1983.

[14] G.F. Pfister, W.C. Brantley, D.A. George, S.L. Harvey, W.J. Kleinfelder, K.P. McAuliffe, E.A. Melton, V.A. Norton, and J. Weiss, "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture," *Proc. of International Conf. on Parallel Processing*, pp. 764-771, 1985.

[15] H. Wada, S. Kawabe, and T. Odaka, "Hitachi supercomputer S-820 overview," *Proc. of Supercomputing Europe '89*, pp. 139-147, 1989.

[16] H. Wada, K. Ishii, S. Yazawa, and S. Kawabe, "High-speed Vector Instruction Execution Schemes of Hitachi Supercomputer S-820 System," *Proc. of International Conf. on Parallel Processing*, pp. 291-298, 1988.

[17] H. Wada, K. Ishii, M. Fukagawa, H. Murayama, and S. Kawabe, "High-speed Processing Schemes for Summation Type and Iteration Type Vector Instructions on Hitachi Supercomputer S-820 System," *Proc. of International Conf. on Supercomputing*, pp. 197-206, 1988.

[18] H. Wada, T. Isobe, M. Furukawa, and S. Kawabe, "High-speed Storage Control Schemes of Hitachi Supercomputer S-820 System," *Proc. of International Conf. on Supercomputing*, pp. 341-350, 1989.