Node Architecture and Performance Evaluation of the Hitachi Super Technical Server SR8000

**Yoshiko Tamaki, Naonobu Sukegawa, Masanao Ito, Yoshikazu Tanaka,
Masakazu Fukagawa*, Tsutomu Sumimoto*, and Nobuhiro Ioki****

**Central Research Laboratory, Hitachi, Ltd.
1-280 Higashi-Koigakubo, Kokubunji, Tokyo, 185-8601, Japan.
Tel: +81-423-23-1111, Fax: +81-423-27-7743
e-mail: tamaki@crl.hitachi.co.jp
* General Purpose Computer Division, Hitachi, Ltd.
1 Horiyamashita, Hadano, Kanagawa, Japan.
** Software Division, Hitachi, Ltd.
549-6 Shinanochou, Totsuka-ku, Yokohama, Kanagawa, Japan.**

Abstract

A new architecture for the Hitachi super technical server SR8000 has been developed. The performance of the SR8000 is 8 GFLOPS in each node, which is a RISC-based SMP, and 1 TFLOPS when up to 128 nodes are connected with an interconnect network. A node of the SR8000 provides a new COMPAS (CO-operative Micro-Processors in single Address Space) architecture and improved PVP (Pseudo Vector Processing) architecture. COMPAS provides rapid simultaneous start up of all microprocessors in a node. And PVP provides stable and high data-reference throughput even when the node processes a larger data set than the cache size. These architectures result in the node performance equivalent to that of a vector processor. The new features of COMPAS and PVP are inter-processor communication and multiple outstanding prefetching. We evaluate the node performance of the SR8000, and compare it with that of the Hitachi predecessor vector processor, S-3800. And we demonstrate that the SR8000 has superior node performance than that of the S-3800.

Keywords: SR8000, SMP, parallel, pseudo-vector

1. Introduction

Over the last twenty years or so, vector processors have been used[1]-[4] for scientific and engineering applications requiring very high performance. The characteristics of vector processors is high single-processor performance when processing large-size data.

RISC-based SMP systems have recently been used[5] for scientific and engineering applications requiring middle-to-high performance. The characteristics of these systems are high performance when processing smaller size data than the cache size, necessity of source tuning, such as blocking to achieve high performance when processing large-size data, and necessity of large granularity of DO-loops to achieve efficient parallel execution.

In developing the super technical server SR8000, we aimed to adopt RISC-based SMP nodes and make them achieve equivalent or higher performance than that of a vector processor when processing not only smaller but also larger size data than the cache size. We therefore have to construct a node to satisfy the following conditions:

- High peak performance equivalent to that of a vector single processor.
- High main-storage throughput equivalent to that of a vector single processor.
- Equivalent or higher cache throughput than the main-storage throughput of a vector single processor.
- Rapid simultaneous start-up of processors in the node to achieve high performance even when the node processes DO-loops with small granularity.
- Stable and high data-reference throughput, which is achieved by accessing the cache when data is in the cache and by accessing the main storage in a pseudo-vector processing manner when data is not in the cache.

First, we explain why rapid simultaneous start-up is required. Figure 1.1 plots parallelization effect against granularity of DO-loops and start-up overhead of parallel execution. It shows that a smaller start-up overhead is required when DO-loops with smaller granularity are executed. Vector processors can achieve high performance when they process 1- dimensional DO-loops with long

loop length and multiple dimensional DO-loops with any loop length. In order to achieve high performance when processing these DO-loops, we have to achieve small start-up overhead and developed COMPAS architecture.

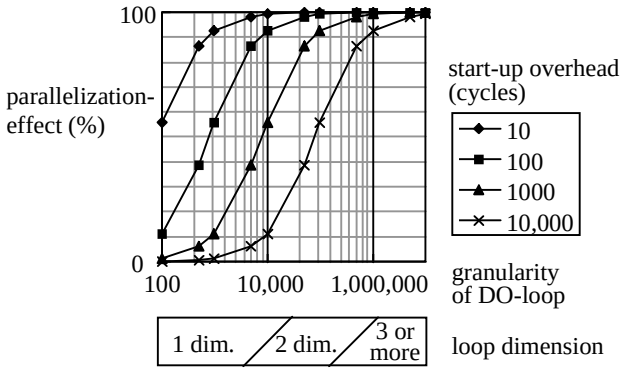

Figure 1.1 Parallelization effect

Second, we discuss high data-reference throughput (last condition). In order to achieve high data-reference throughput when processing large-size data, the PVP architecture is already provided in the Hitachi parallel supercomputer SR2201.[6] Before arithmetic execution, it pre-loads data from the main storage not to the cache but to registers in a pipelined manner. In order to achieve stable and high data-reference throughput when processing any size data, we have to improve the PVP architecture.

2. System Overview of SR8000

Figure 2.1 shows the system configuration of the SR8000. In this configuration, up to 128 nodes are connected in an inter-node multi-dimensional crossbar network[7] and the resulting performance is 1 TFLOPS. Each node is configured with multiple RISC-based micro instruction-processors with cache, and the total performance of all the processors in a node is 8 GFLOPS. This performance is equivalent to that of a single vector processor of Hitachi's predecessor system, the S-3800. The main storage configured with multiple banks and the storage controller configured with a multi-stage crossbar provide high main-storage throughput, equivalent to that of a vector single processor. A copy of cache-tags of all processors in a node is in the storage controller, and it achieves high speed cache-coherency-processing.

3. Node architecture of SR8000

3.1 Parallel execution model

The parallel structure of the DO-loops is a fork join, which consists of a serial-execution part and parallel-execution parts appearing one after another as shown in Figure 3.1. The serial-execution part is assigned to one thread and executed on one processor, and parallel-execution parts are assigned to multiple threads and executed on multiple processors. In general, the run-time scheduler schedules the serial- and parallel-execution parts to threads, then the operating system schedules threads to processors.

When the operating system schedules threads, the execution of a program is interrupted and this causes an overhead of several-thousand-cycles. On the other hand, as described in section 1, the start-up overhead of parallelization must be small. The thread scheduling by the operating system to assign or de-assign processors must thus be avoided during execution of a fork sequence or a join sequence. This requires that all the number of threads required for parallel execution must always be scheduled to processors. We therefore implement the operating system to schedule threads in a so-called “gang scheduling” manner, in which the operating system simultaneously assigns all processors required by a program for parallel execution and simultaneously de-assigns all the processors from the program.

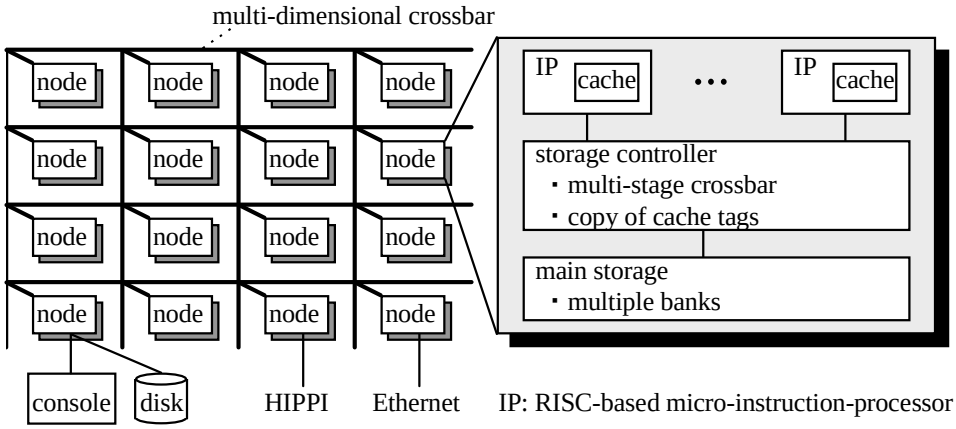

Figure 2.1 SR8000 System Configuration

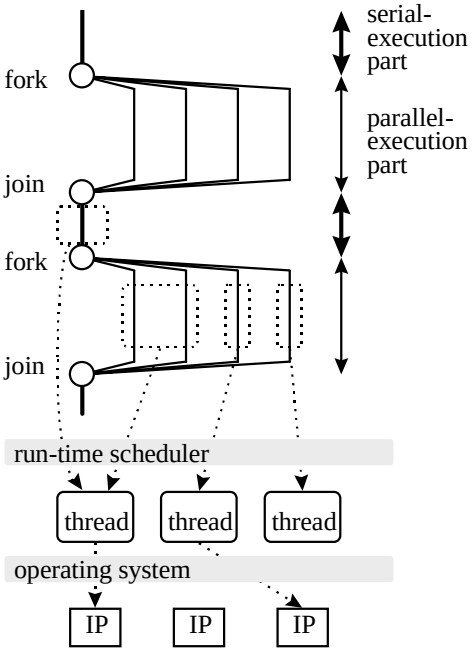

Figure 3.1 Structure of parallel DO-loop and two-level scheduling

In most cases, by distributing the loop index, the compiler can distribute the DO-loops to the same number of parallel-execution parts as the number of processors. And in most cases, these parts have equal granularity with each other. The gang scheduling also ensures that all processors are assigned. Therefore, to achieve the most rapid scheduling, the run-time scheduler does not have to schedule dynamically but has to schedule the parallel-execution parts to threads statically in a one-to-one manner.

Figure 3.2 shows the model of parallel execution on the node of the SR8000. The operating system generates the same number of threads as the number of processors and schedules them to the processors simultaneously. We call the thread executing the serial-execution part the parent thread, and the threads executing the parallel execution-parts the child threads. The compiler generates the instruction code of the serial-execution parts and the instruction code of the parallel-execution parts of the DO-loop, which is divided into the number of processors. At first, the parent thread executes one serial-execution part, and the child threads wait for the fork in a spin-loop-waiting manner. The compiler does not know which DO-loop is executed after the serial-execution part because there are subroutines, if-statements, or goto-statements in a program. Only the parent thread therefore knows the succeeding DO-loop when it executes the serial-execution part. And the parent thread must signal the instruction address of the DO-loops to the child threads. Child threads branch to the instruction address and execute the parallel-execution parts. The parent thread also executes a parallel-execution part so that all the processors process the parallel-execution part. After execution of the parallel-execution part, each child thread signals to the parent thread, and then the parent thread executes the next serial-execution part.

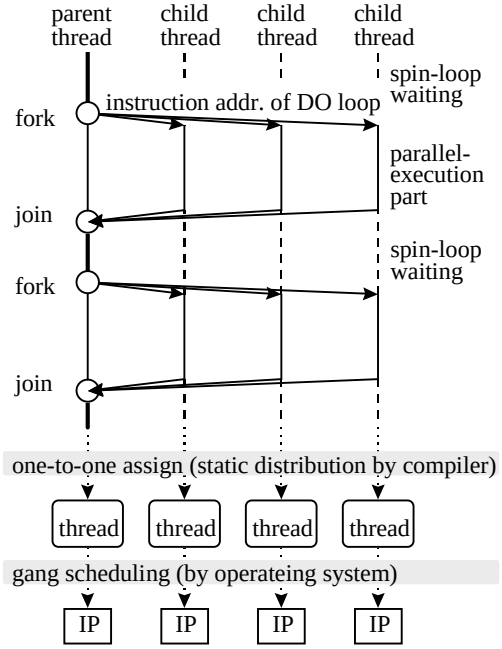

Figure 3.2 Model of Parallel execution on the node of SR8000

3.2 COMPAS architecture

The COMPAS architecture provides rapid fork and join sequences. When a fork or a join sequence is executed, the caches must be made coherent because the data stored by a processor might be referred to by another processor executing a succeeding part. As described in section 2, the caches become coherent using the copy of cache tags in the storage controller. From now on, for easier explanation, we call the processor assigned to a parent thread the parent processor, and the processors assigned to child threads the child processors.

Figure 3.3 shows the fork sequence constructed by existing instructions, such as a “sync” instruction, which ensures that the cache of a processor becomes coherent when the sync instruction is completed on the processor. In this figure, the following four steps are executed sequentially. At the end of the serial-execution part, the parent processor executes a sync instruction, and this execution is signaled to the copy of cache tags (1). And the cache of the parent processor becomes coherent, it is signaled to the parent processor (2). Then the parent processor stores the instruction address of the DO-loop in the main storage (3), and all child processors load the instruction address from the main storage (4).

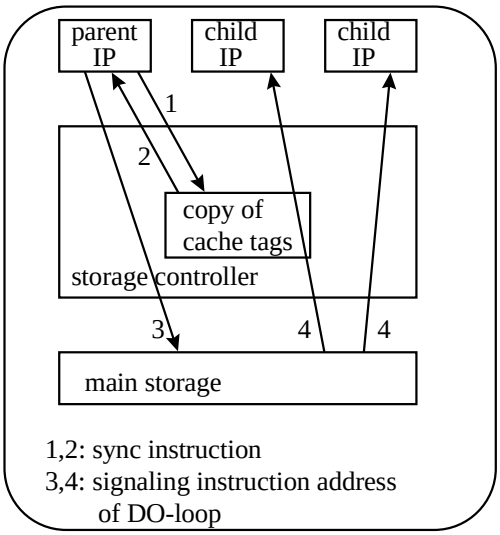

Figure 3.3 Fork sequence using sync instruction

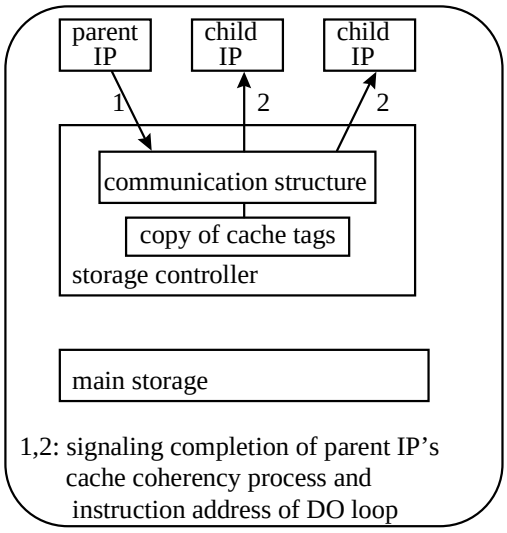

Figure 3.4 Fork sequence using communication structure

The instruction address is usually in the cache of the parent processor when it executes the fork sequence. So the signaling of the instruction address via the storage controller will provide a more rapid fork sequence than that via the main storage. Waiting completion of cache coherency process (1,2) and signaling the instruction address (3,4) are executed sequentially. This sequential execution is apparently useless and makes the fork sequence larger. We therefore developed a communication structure in the storage controller. This structure waits for completion of cache coherency process and signals the instruction address to the child processors simultaneously.

Figure 3.4 shows the fork sequence using the communication structure. At the end of the serial-execution part, the parent processor signals the instruction

address to the communication structure (1). And the cache coherency process of the parent processor is completed, then the instruction address is signaled to all child processors (2).

Figure 3.5 shows the join sequence using a sync instruction. In this figure, the following four steps are executed sequentially. At the end of the parallel-execution part, each child processor executes a sync instruction, and this execution is signaled to the copy of cache tags (1). The cache coherency process of each child processor is completed, then this completion is signaled to the child processor (2). Each child processor stores flag data that indicates this completion to the main storage (3), and the parent processor loads all the flag data stored by each child processor (4).

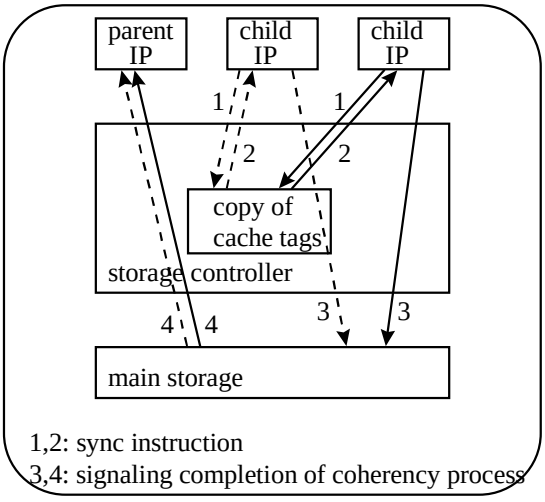

Figure 3.5 Join sequence using sync instruction

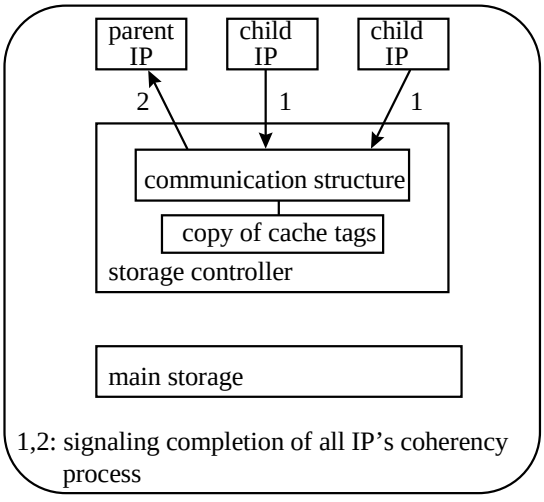

Figure 3.6 Join sequence using communication structure

The signaling the completion of the cache coherency process via the child processors and via the main storage (2, 3, and 4) is apparently useless because the storage controller knows when cache coherency process is completed. We therefore made the communication structure signal the completion of cache coherency process of all child processors to the parent processor.

Figure 3.6 shows the join sequence using the communication structure. At the end of the parallel-execution part, each child processor signals to the communication structure (1). And all the caches of child processors are become coherent, then the communication structure signals to the parent processor (2).

3.3 PVP architecture

By using the PVP architecture, before arithmetic execution, the SR8000 performs pre-refer requests of data transfer from the main storage to a processor in pipelined manner. This results in non-blocking arithmetic execution such as that performed by a vector processor.

Two types of pre-refer requests are studied: prefetch and pre-load. Prefetch transfers pre-requested line data from the main storage to the cache, and pre-load transfers pre-requested element data from the main storage to the registers. When the accessed data size is large, both of them achieve high performance. When the accessed data size is small, data prefetched in a DO-loop may be in the cache when the data is accessed in a succeeding DO-loop, but data pre-loaded in a DO-loop must be re-loaded from the main storage in any succeeding DO-loop. we therefore perform PVP by using prefetch. Figure 3.7 shows the instruction code of PVP.

However, prefetch is not efficient when the main storage is accessed non-continuously because the line transfer causes useless data access. On the other hand, pre-load is efficient even with non-continuous access because it transfers element data. We therefore perform PVP by using both prefetch and pre-load as shown in Figure 3.8.

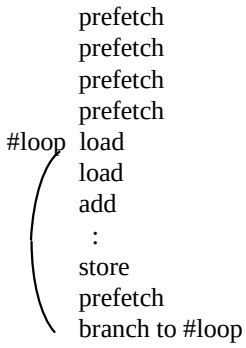

Figure 3.7 Structure of PVP code using prefetch

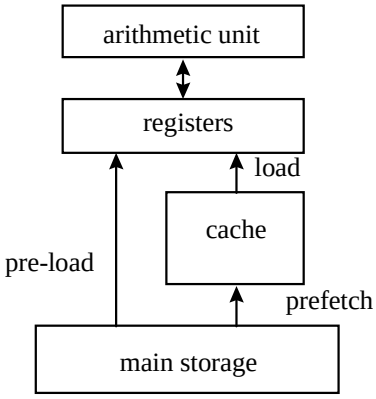

Figure 3.8 Improved PVP architecture

To achieve high main-storage throughput, we have to determine the number of prefetches executable in parallel, the so-called number of outstanding prefetches. Figure 3.9 shows the execution timing of PVP code using prefetch instructions. In this figure, four prefetches before arithmetic instructions are balanced with the main storage latency, the size of line transfer, and the data reference throughput (see the black prefetch request and the black

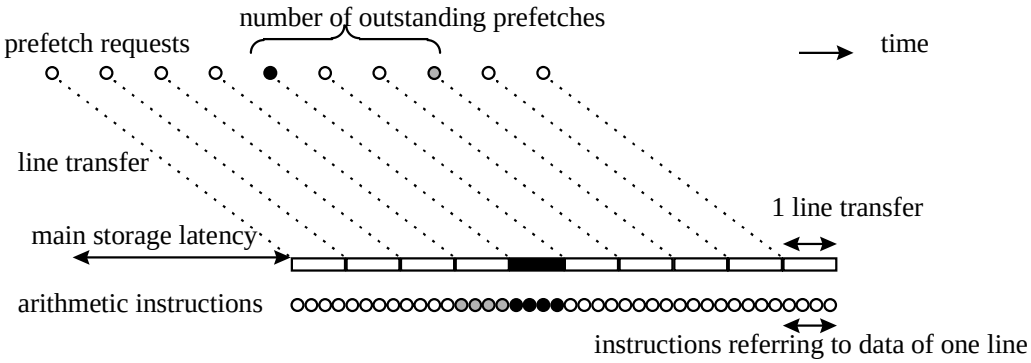

Figure 3.9 Execution timing of PVP code

arithmetic instructions). If we assume the processor can execute only three outstanding prefetches, the hatched prefetch request cannot be issued until the black line transfer is completed. And it causes execution stalling of not only the hatched prefetch but also the hatched arithmetic instructions because the hatched prefetch instruction and the hatched arithmetic instructions are in the same loop iteration (see Figure 3.7). Therefore, in order to achieve high performance, the processor has to be able to execute enough outstanding prefetches that satisfy the following rule: (data reference throughput [GB/s]) = (size of line transfer [B]) x (number of outstanding prefetches) / (main storage latency [ns]). And we implement the processor to satisfy this rule.

The above-mentioned improved PVP architecture provides stable and high data-reference throughput, which is achieved by accessing the cache when data is in the cache but accessing the main storage in a pseudo-vector processing manner when data is not in the cache.

4. Performance evaluation

We evaluated the node performance of a core-loop for Cholesky factorization and a loop for matrix multiplication, which are both typical computations of linear algebra. Figure 4.1 shows the example loops for processing Cholesky factorization and matrix multiplication. We used the compiler under development to evaluate the performance of these loops. The core-loop for Cholesky factorization is a three-dimensional loop, which is parallelized in the second inner loop because of dependency in the outer-most loop. The loop for matrix

multiplication is a three-dimensional loop, which is parallelized in the outer-most loop.

```
(a) Cholesky factrization

      do 10 k=1,n
      do 10 j=k,n
      do 10 i=1,k-1
        a(k,j)=a(k,j)-a(i,j)*a(i,k)
10    continue

(b) matrix multiplication

      do 10 j=1,n
      do 10 k=1,n
      do 10 i=1,n
        a(i,j)=a(i,j)+b(i,k)*c(k,j)
10    continue
```

Figure 4.1 Example loops

Figure 4.2 shows the performance of the core-loop for Cholesky factorization, and it shows performances of a node of SR8000, this node when it does not execute a prefetch instruction, and a single vector processor of the S-3800. The single vector processor of the S-3800 has the same peak performance as the node of the SR8000.

Since this loop is parallelized in the second outer loop, large start-up overhead of parallelization causes low performance when the matrix size N is small. However, Figure 4.2 shows that the SR8000 achieves higher performance than a vector processor at any matrix size. This SR8000 performance shows that the start-up overhead is small enough to achieve high performance when processing DO-loops with small granularity.

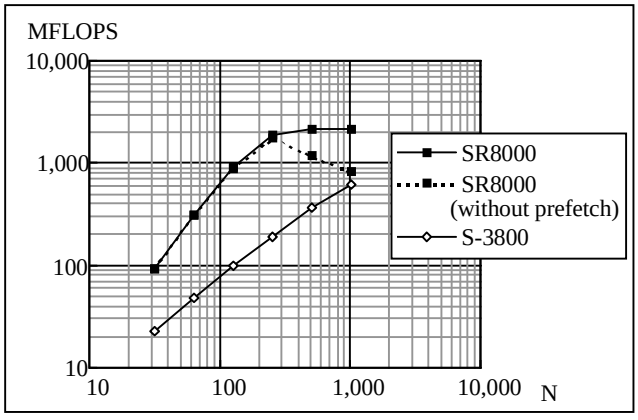

Figure 4.2 Performance of core-loop for Cholesky factorization

The working data set, which is accessed in the second inner loops, is larger than the cache size when the matrix size N is larger than 512. Figure 4.2 shows that, when the N is larger than 512, the performance of the SR8000 is stable and high, but the performance of the SR8000 without using prefetch instructions becomes lower. This shows that the SR8000 with PVP architecture is efficient and can achieve high main-storage throughput even when the data size is larger than the cache size.

Figure 4.3 shows the matrix multiplication performances of a node of SR8000, a node of SR8000 when it does not execute a prefetch instruction, and a single vector processor of the S-3800. The matrix multiplication is parallelized in the outer-most loop. So the start-up overhead of parallelization is negligible, and the performance depends on the data reference throughput. In the matrix multiplication, the data size accessed in the loop is larger than the cache size when the matrix size N is larger than 128. Figure 4.3 shows that the SR8000 achieves stable and higher performance than a vector processor when N is smaller than 128. This performance shows that the SR8000 can achieves stable and high cache throughput. Figure 4.3 also shows that, when the N is larger than 128, the performance of the SR8000 is stable and high, but the performance of the SR8000 when it does not use prefetch instructions becomes lower. This result shows that the SR8000 with PVP architecture is efficient and can achieve high main-storage throughput .

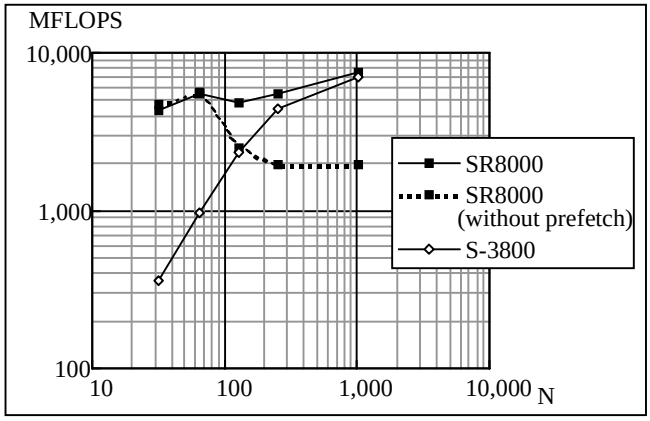

Figure 4.3 Performance of matrix multiplication

5. Conclusions

We developed new node architectures, called COMPAS (CO-operative Micro-Processors in single Address Space) and PVP (Pseudo Vector Processing), for the Hitachi super technical server SR8000. COMPAS provides rapid simultaneous start-up of processors. And PVP provides stable and high data-reference throughput by accessing the cache when data is in the cache and by accessing the main storage when data is not in the cache. Moreover, we evaluated the node performance of the

SR8000 with a core loop for Cholesky factorization and another loop for matrix multiplication. The evaluation shows that, when processing DO-loops with small granularity, the SR8000 achieves equivalent or higher performance than that of a vector single processor because the start-up overhead is small. It also shows that, when processing working set data of small size, the SR8000 achieves superior performance to that of a vector single processor because the cache throughput is high. The evaluation also shows that, when processing working set data of large size, the SR8000 achieves equivalent performance to that of a vector single processor because the main storage throughput is high. When processing the matrix multiplication of small size, the node of the SR8000 achieves stable and high performance of 5 GFLOPS. And when processing the matrix multiplication of large size (matrix size of 1000), it achieves higher performance than 7 GFLOPS.

References

[1] Olaf Lubeck, James Moore, and Raul Mendez, "A Benchmark Comparison of Three Supercomputers: Fujitsu VP-200, Hitachi S810/20, and Cray X-MP/2", IEEE Computer, December 1985, pp. 10-24.

[2] Kouichi Ishii, Hitoshi Abe, Shun Kawabe, and Michihiro Hirai, "An Overview of the Hitachi S-3800 Series Supercomputer", Proceedings of Supercomputing '92, 1992, pp. 65-81.

[3] Steven W. Hammomd, Richard D. Loft, and Philip D. Tannenbaum, "Architecture and Application: The Performance of the NEC SX-4 on the NCAR Benchmark Suit", Proceedings of Supercomputing '96, 1996.

[4] Kenichi Miura, "Vector-Parallel Processing and Fujitsu's Approach to High Performance Computing, VPP 300/VPP 700 Systems", Proceedings of the 5th International School/Symposium for Space Simulations, 1997, pp. 448-450.

[5] http://www.netlib.org/benchmark/top500/top500.list.html

[6] K. Saito, M. Hashimoto, H. Sawamoto, R. Yamagata, T. Kumagai, E. Kamsda, K. Matsubara, T. Isobe, T. Hotta, T. Nakano, T. Shimizu, and K. Nakazawa, "A 150 MHz Superscalar RISC Processor with Pseudo Vector Processing Feature", Proceedings Notebook for Hot Chips VII, 1995, pp. 197-205.

[7] Hiroaki Fujii, Yoshiko Yasuda, Hideya Akashi, Yasuhiro, Inagami, Makoto Koga, Osamu Ishihara, Masamori Kashiyama, Hideo Wada, and Tsutomu Sumimoto, "Architecture and Performance of the Hitachi SR2201 Massively Parallel Processor System", Proceedings of the 11th International Parallel Processing Symposium, 1997, pp. 233-241.

Authors

Yoshiko Tamaki, Naonobu Sukegawa, Masanao Ito, and Yoshikazu Tanaka

Central Research Laboratory, Hitachi, Ltd.

1-280 Higashi-Koigakubo, Kokubunji, Tokyo, 185-8601, Japan.

Tel: +81-423-23-1111, Fax: +81-423-27-7743

e-mail: {tamaki, sukegawa, m-itoh, ytanaka}@crl.hitachi.co.jp

Masakazu Fukagawa and Tsutomu Sumimoto

General Purpose Computer Division, Hitachi, Ltd.

1 Horiyamashita, Hadano, Kanagawa, Japan.

Nobuhiro Ioki

Software Division, Hitachi, Ltd.

549-6 Shinanochou, Totsuka-ku, Yokohama, Kanagawa, Japan.