Design choices for the TOP-1 multiprocessor workstation

by S. Shimizu
N. Oba
T. Nakada
M. Ohara
A. Moriwaki

A snoopy-cache-based multiprocessor workstation called TOP-1 (TOkyo research Parallel processor-1) was developed to evaluate multiprocessor architecture design choices as well as to conduct research on operating systems, compilers, and applications for multiprocessor workstations. TOP-1 is a ten-way multiprocessor using the Intel 80386™ microprocessor chip and the Weitek WTL 1167™ floating-point coprocessor chip. It is currently running under a multiprocessor version of AIX®, which was also developed at the IBM Tokyo Research Laboratory. Our research interest was focused on the design of an effective snoopy cache (all caches monitor all memory-cache traffic) system and the quantitative evaluation of its performance. One of the unique aspects of the TOP-1 design is that the cache supports four different, original snoopy protocols, which may coexist in the system. To evaluate the performance, we implemented a hardware statistics monitor that gathers statistical data. This paper focuses mainly on the TOP-1 cache design—its protocol, and its evaluation by means of the statistics monitor. Besides its cache design, TOP-1 has three other unique architectural features: two independently arbitrated 64-bit buses supported by two snoopy-cache controllers per processor, a communication and interruption mechanism for notifying other processors of asynchronous events, and an efficient arbitration mechanism to allow prioritized quasi-round-robin service with distributed control. These features are also described in detail.

1. Introduction

Multiprocessor workstations are a very promising solution to the continuously increasing need for computing power in workstations. In the past several years, shared-memory, shared-bus multiprocessors have been extensively analyzed and developed because of the simplicity of the hardware and the flexibility with respect to various parallel-programming models. Most such multiprocessors use a snoopy-cache mechanism (in which all caches monitor all bus traffic between caches and shared storage) to reduce bus traffic and to maintain multicache coherency [1–4].

Historically, the first implementation of cache memory for a commercial computer was the IBM System/360™ Model 85 computer [5]. Multicache coherency mechanisms were implemented in the IBM System/370™ Model 168 [6] and 3033 [7] systems with a so-called "broadcast write-invalidation" scheme. Also, the IBM 308X system [8] included a so-called "directory" scheme for maintaining multicache coherency.

The IBM Tokyo Research Laboratory has been working on a multiprocessor workstation since 1986. As a research

vehicle, we developed a snoopy-cache-based multiprocessor workstation to evaluate the multiprocessor architecture design and to do research on operating systems, compilers, and applications for multiprocessor workstations. An overview of the project is given in [9]. The result, TOP-1 (TOkyo research Parallel processor-1), is a snoopy-cache-based ten-way multiprocessor with shared-memory architecture.

The unique architectural features of TOP-1 are the following: a mechanism that allows several different cache-coherency protocols to coexist in the system and to be changed by software from one memory operation to the next; two-way-interleaved dual 64-bit buses, which are supported by two snoopy-cache controllers per processor card; a communication and interruption mechanism for notifying processors of asynchronous events; and an efficient arbitration mechanism that provides prioritized quasi-round-robin service with distributed control. Moreover, TOP-1 has a hardware statistics monitor that enables us to obtain various statistics on cache-algorithm performance, operating systems, and applications.

In this paper, these important architectural features of TOP-1 are described. Performance results measured by the statistics monitor are also described. In Section 2, the hardware organization of TOP-1 is described. Sections 3 and 4 discuss considerations in the design of the memory system and the interprocessor communication mechanisms. In Section 5, we briefly discuss the performance results that were measured by using the statistics monitor.

2. System organization

One major objective of the TOP-1 project is to study hardware and software trade-offs. We wanted to obtain run-time statistics, analyze them, and reflect the results in the design of future multiprocessors. We also wanted to study the effectiveness of using run-time statistics to change the hardware algorithm (by the compiler or the operating system) for dynamic system optimization. We therefore implemented multiple snoopy-cache protocols and variable-priority arbitration mechanisms.

As shown in **Figure 1**, TOP-1 is organized around a shared bus. Attached to the bus are up to 128 megabytes (MB) of system memory and up to 10 identical processing units (although the hardware implementation limit to the number of processors is 11, the standard configuration of TOP-1 hardware consists of 10 processing units), each of which comprises an Intel 80386™ microprocessor and a Weitek WTL 1167™ floating-point coprocessor, a 128-kilobyte (KB) snoopy cache, and a system-bus interface. (The 10-processor limit comes from various factors such as an effective performance target of 30 MIPS, the back-plane physical dimension, and a reasonable box size.) Also on the bus is a Micro Channel® interface adapter card to connect TOP-1 to a PS/2®, which is used as the I/O processor of TOP-1, through the Micro Channel interface standard. The adapter card also has a 128KB snoopy cache, through which the PS/2 can directly access the TOP-1 shared bus, hence the system memory. This allows various I/O devices on the PS/2 to be used as the I/O devices of TOP-1. We built a special hard-disk controller to provide a hard-disk subsystem with high performance and large capacity. One of the ten processors is connected to the hard-disk controller via the local bus extension.

3. Memory system

The memory system uses a so-called snoopy-cache mechanism, like those in other small-scale, tightly coupled multiprocessors [1–4], to reduce the bus traffic and to maintain multicache coherency. Its unique feature is that it supports four different protocols.

- *Cache organization*

The main specifications of the TOP-1 cache are as follows:

- *128KB cache size*: In general, a larger cache size results in a higher hit ratio, hence better performance and lower system-bus traffic. On the other hand, the cache size should not be so large that it represents an expense out of proportion to the added performance, nor should it occupy an unreasonable fraction of the physical space. Moreover, in a multiprocessor system with an update-type snoopy cache (described below), a very large cache may increase the number of shared-state cache lines, thus requiring unnecessary updating of the cache lines. Considering these trade-offs, we adopted a fairly large cache size of 128 KB.
- *64-bit line size*: In general, a larger line size results in a higher hit ratio (even for a fixed cache size), hence better performance. On the other hand, since a large cache line is transferred over the bus by using a so-called burst-transfer mode, a larger cache line requires a longer time to complete the transfer. If we assume extra time for transferring a larger cache line (this extra delay seen by the processor can be eliminated by using a fetch-bypass or load-through mechanism), the average memory-access time is not necessarily decreased, even if the cache-hit ratio is slightly improved. Furthermore, the extra bus cycles needed to transfer a longer cache line may cancel out the benefit of a small improvement in the hit ratio, even if the fetch-bypass mechanism can be used. Moreover, in a multiprocessor with a snoopy cache, a larger line size may increase the number of shared lines in the cache, since an entire cache line has to be in the *shared* state, even if only one byte in the cache line is actually shared. In fact, our simulations show that the frequency of occurrence of shared cache lines for a 128-bit line is about 1.5 times that for a 64-bit line if we do not use any software techniques for gathering shared

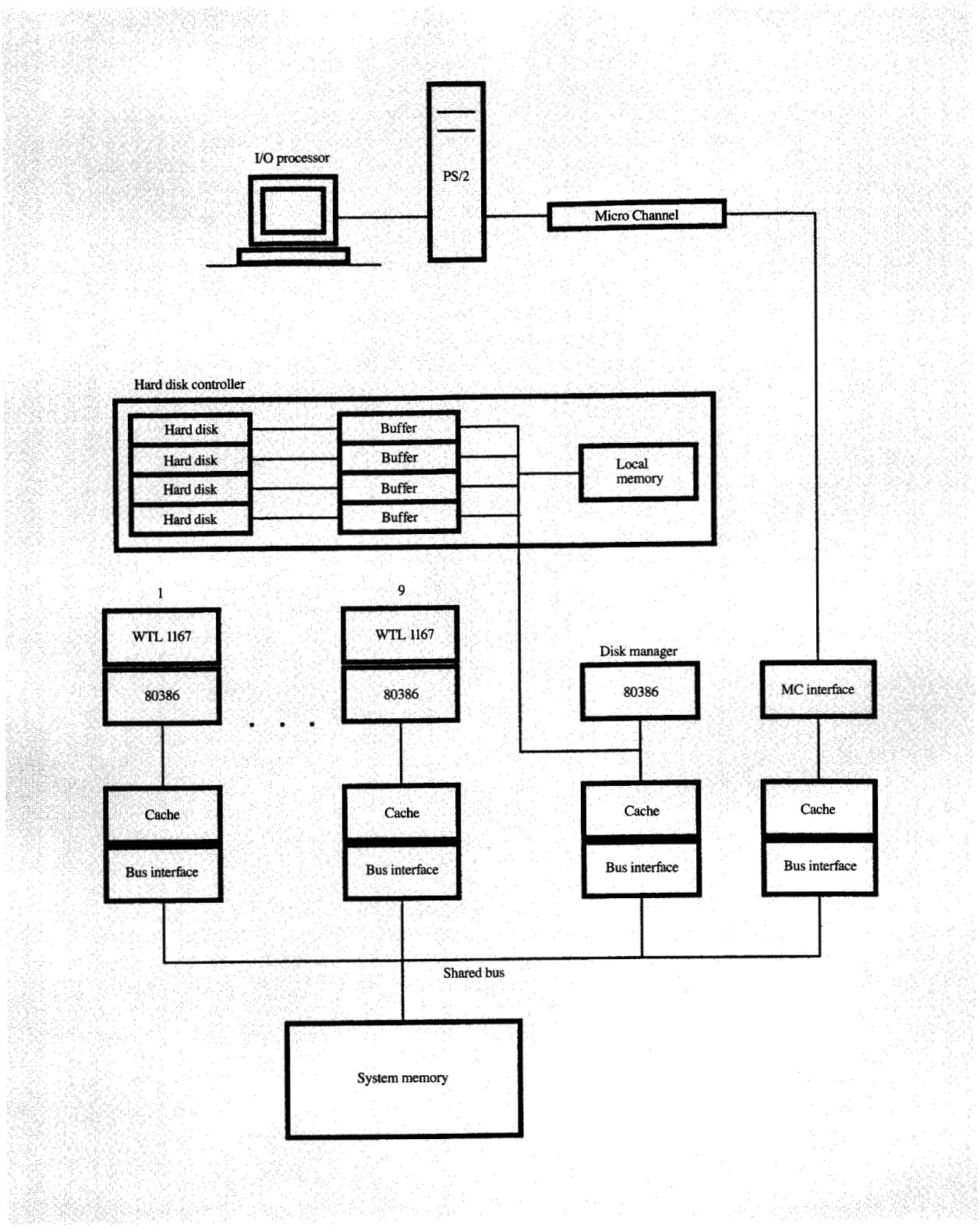

Figure 1

TOP-1 multiprocessor.

data within a limited memory area. In view of these criteria, we adopted a 64-bit line size, which is the same as the data width of the system bus. (We also believe that a much longer cache line gives better performance, at least for a single processor, under conditions in which the fetch-bypass mechanism can be used, and that the burst transfer can be realized very efficiently in practice.)

- *Direct mapping*: There are also various trade-offs among large-set associativity, small-set associativity, and direct mapping as the cache placement algorithm. The most important factor is again the cache-hit ratio, as it is for the cache-line size. In general, larger-set associativity gives a higher hit ratio, hence better performance, since the mechanism can easily handle several access streams (such as code fetch, source data access, and destination data access) with a number of processes running at a time, each with its access locality. This tendency is definitely true for a large system that executes a variety of very large jobs. For a large cache, however, the improvement in the hit ratio achieved by increasing associativity is small [10], since the probability of conflicts on the same cache line is very small if the cache is large enough. On the other hand, in practice, larger-set associativity results in larger circuit delays for the tag comparator, since more address tags must be compared with the requested address. Furthermore, for a read hit, since the output from the data memory array of the set-associative cache must be selected after the comparison of the tag fields has been completed, an extra delay is incurred before the data become available to the processor. For these reasons (mainly for practical implementation reasons), we compromised on a direct-mapping cache.
- *Duplicated tag memory*: The cache tag memory is duplicated so that the processor and the bus-snooping hardware can access the cache tag memory in parallel. Cache-access conflicts occur only when the processor or the bus-snooping hardware requires the tag memory to be updated and the other component has to read the tag memory in the same cycle, or when both the processor and the bus-snooping hardware request the cache data memory to be accessed in the same cycle. A read access to the tag memory for a snoopy address comparison does not interfere with a cache access induced by the processor, as long as the processor cache access does not require the tag memory to be updated with a new value. When cache access conflicts between the processor and the bus-snooping hardware occur, the bus-snooping hardware has a higher priority than the processor. Therefore, the processor access is delayed at least one cycle, until the conflict condition ends. This arbitration policy ensures that the snoopy access is completed in a fixed period, simplifying the system-bus protocol.

- *Snoopy protocol*

Each cache controller is implemented with an IBM CMOS gate array with 16 000 gates. Both the cache tag and data memories are implemented by means of the IBM fast SRAM, with a 30-ns access time and an 8K × 18-bit configuration. The controller manages the TOP-1 snoopy-cache protocols as well as the shared-bus interface, including the arbitration logic. It also includes the monitoring hardware for collecting statistics.

A number of snoopy-cache protocols have been proposed, but none of them is considered to be ideal in all situations [11]. Coherency protocols can generally be classified as update or invalidate protocols, each of which is suitable for different situations. A novel aspect of TOP-1 is that it supports both update and invalidate protocols. Furthermore, each type of protocol can have two modes: standard and block-I/O. In total, there are four protocol combinations, all of which can exist in the system at the same time. On each processor, the protocol can be dynamically changed by program, from one memory operation to the next.

In the TOP-1 protocols, each cache line is in one of five states: *clean-private* (this is the only copy in the caches, and it has not been modified), *dirty-private* (this is the only copy in the caches, but it has been modified), *clean-shared* (other caches may have a copy; this may or may not be a modified copy, but write-back is not needed at replacement), *dirty-shared* (write-back is required at replacement; other caches may have a copy), or *invalid* (this does not contain valid memory data).

A special bus line used to detect sharing is referred to as the CH (cache hit) line. The CH line is an open-collector wired-OR line that is activated by one or more snooping cache controllers when it holds a copy of the memory address broadcast by a requesting cache. The requesting cache controller can recognize whether any other cache has a copy of the requested memory address by examining the CH line at the end of a bus cycle.

Each cache controller has a "cache-mode register," which uses two bits to specify the snoopy protocol type (update or invalidate) and the protocol mode (standard or block-I/O). The first bit indicates "inv" (invalidate protocol) when "1." The second bit indicates "I/O" (block-I/O mode) when "1."

Figure 2 is a state-transition diagram of the TOP-1 protocols, including both update and invalidate protocols. (For simplicity of discussion, only the standard mode is depicted in the figure. We discuss the block-I/O mode later in this section.) Both the state of the CH line and the protocol-type bit control state transitions.

The TOP-1 protocol can be described as follows:

- When one processor experiences a read miss: If another cache has a *dirty-private* or *dirty-shared* copy, that cache

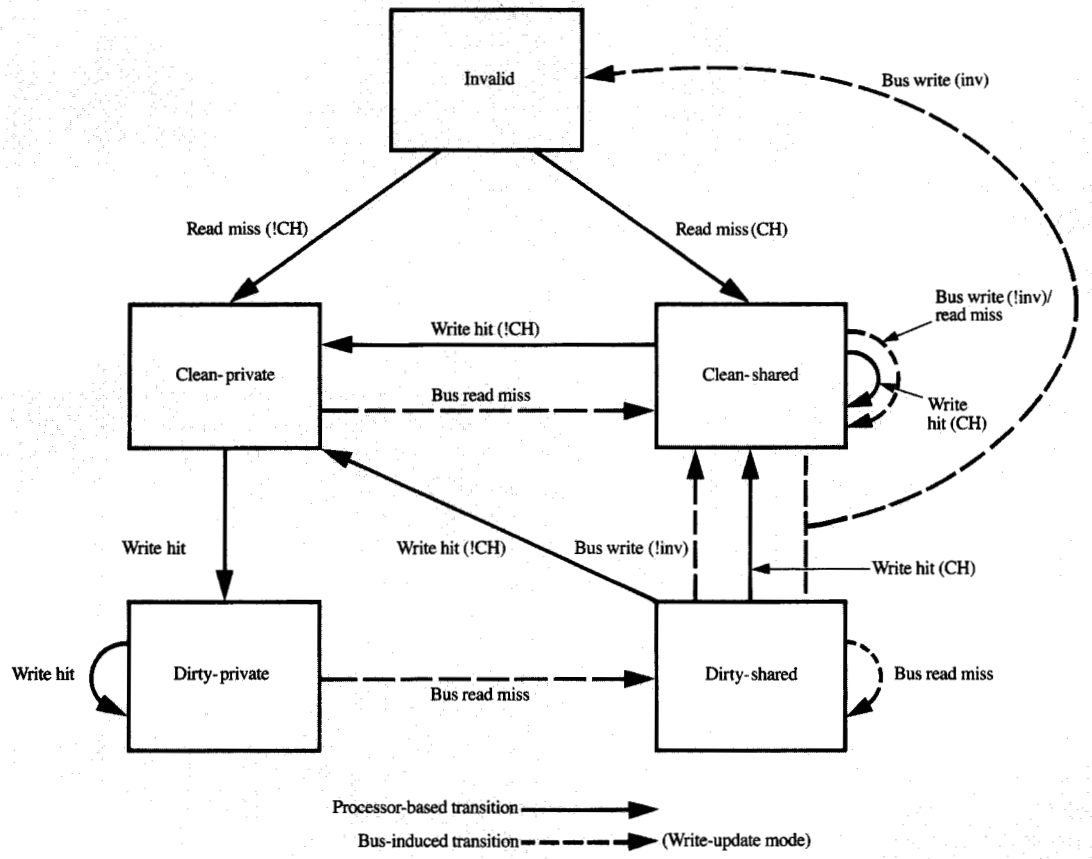

Figure 2

State-transition diagram for TOP-1 snoopy protocol.

supplies the data, raises the CH line, and sets the cache-line state (the state of the cache line being read) to *dirty-shared*, because main memory is not updated with the dirty data. (In the TOP-1 protocol, the *dirty* state implies responsibility for write-back to the main memory; hence, only one exists in the system at a time.) Otherwise, the data come from main store, even if they can be found in another cache (see the discussion below). Any cache with a *clean-private* or *clean-shared* copy raises the CH line and sets its local state of the cache line to *clean-shared*. The requesting cache loads the cache line in the *clean-shared* state when the CH line is active. Otherwise, it loads the cache line in the *clean-private* state.

- When one processor experiences a cache hit for a write access: If the cache line is *dirty-private*, the write can be completed locally without using the system bus, and no state change occurs. If the cache line is *clean-private*, the write can also be completed without using the shared bus, but the state is changed to *dirty-private*. If the cache line is in either the *clean-shared* or *dirty-shared* state, the entire cache line is broadcast over the system bus. Other caches with copies observe the bus. If the snoopy-protocol bit in the cache-mode register specifies the update protocol (not inv), those caches update their own copies with the new data on the bus and activate the CH line. On the other hand, if the snoopy-protocol bit specifies the invalidate protocol (inv), those snooping caches invalidate their own copies and do not activate the CH line. The requesting cache can determine whether or not the cache line is still shared by examining the CH line. If the CH line is not activated, the requesting cache changes the cache-line state to *clean-private*, since no other cache has a copy. If the CH line

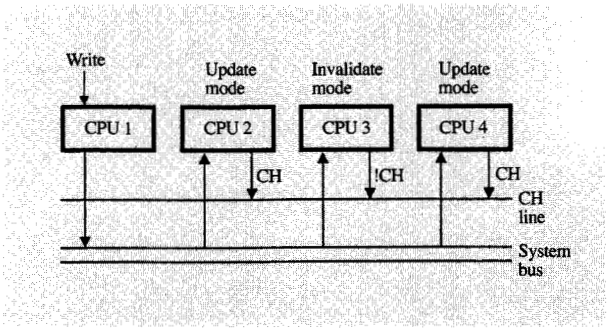

Figure 3

Example of coexistence of update and invalidate protocols.

is activated, the cache-line state remains in the *clean-shared* state, since sharing continues.

- When one processor experiences a cache miss for a write access: In the TOP-1 protocol, a write miss is performed as a combination of a read miss and a write hit. The requesting cache controller generates a read-miss access on the bus and immediately generates a write access if the CH line was raised during the read-miss access. Otherwise, the write is performed locally without using the bus, because there is no sharing.

The TOP-1 protocol is designed to optimize the cache performance by taking account of the difference between the cycle times of the cache memory and the main memory in the following ways:

- In the TOP-1 protocol, clean data come from the main memory rather than another cache that has a clean copy. Since the main memory can provide the data in three 16MHz clock cycles, the same time as for the data response from a snooping cache, there is no difference in access time between main memory and a snooping cache. Of course, a cache can provide data to its own processor much faster than main memory can (more specifically, a processor needs no wait cycle to access its cache, but it needs three wait cycles to access main memory). From the bus-access point of view, however, the cache cannot provide the data so rapidly, because the cache data memory is accessed only after the cache tag is matched to the bus address, in order to reduce unnecessary cache data-array accesses because of bus snooping. Thus, the snooping cache requires at least two cache cycles to respond with the data (signal flight time on the bus should also be added). This data-reply policy can reduce the interference between the processor-induced access and the bus-induced access on a snooping cache. (Interference would occur if another cache replied with a clean copy and delayed a cache access induced by the processor until the completion of the snoopy data-reply cycle.)
- When a write access occurs on the bus, the main memory is always updated with the new data, since main memory can be written in three bus-clock cycles, the same as the snooping cache-update time. This policy reduces the number of *dirty* lines in the cache that must be written back to main memory at the time of replacement, hence reducing the bus traffic induced by write-back activities.
- When a dirty copy is sent from the owning cache to a requesting cache, main memory is not updated with the new data. This policy simplifies the memory- and cache-control logic, since every bus activity can be completed within a fixed number of clock cycles. (On the other hand, if the cache owning the *dirty* cache line updates the main memory as well as the requesting cache, as in the Firefly protocol [2], extra cycles are needed to update the main memory.) Although this policy may increase the number of *dirty* copies in the system, thus increasing the bus traffic caused by write-back activities, the effect can be relaxed, since in the TOP-1 protocol, every bus write updates the main memory and changes the cache-line state from *dirty* to *clean*, as described above.

One can conclude from the above description of the protocol that a requesting cache need not know which snoopy protocols (update or invalidate) the other caches use. Furthermore, the snoopy protocol bit in the cache-mode register of a requesting cache has no influence on the operation of the cache controller, but the bit affects the operation of a snooping cache.

The requesting cache has only an indirect influence on its operation through the response on the CH line from other caches. Therefore, it should be noted that each cache controller can arbitrarily use either an update or an invalidate protocol at any time. **Figure 3** shows an example in which two protocols coexist in the system. In the figure, processors 2 and 4 are using the update protocol, and processor 3 is using the invalidate protocol. The protocol used by processor 1 is irrelevant in this case, since it is a requesting processor. When a write access occurs on the bus, the corresponding cache lines in processors 2 and 4 are simultaneously updated with the new data on the bus, while the cache line in processor 3 is invalidated. The cache controllers of processors 2 and 4 activate the CH line, but that of processor 3 does not. In this case, the cache-line state in the processor 1 cache remains *shared* because the CH line is activated.

If all cache controllers in the system are using the update protocol, the resultant protocol is similar to that of the Xerox Dragon [1] and the Digital Equipment

Corporation Firefly [2]. If all cache controllers in the system are using the invalidate protocol, all of the shared cache lines are invalidated in other caches when a write occurs, as in the Berkeley SPUR [3].

The block-I/O mode is used for consecutive block data transfers to and from external devices, in order to remove the cache lines that are no longer needed. Data transfer to and from I/O devices must be carefully designed in order to maintain data consistency in the multicache system. In TOP-1, we solved the problem by using the snoopy-cache mechanism for the I/O subsystems (I/O Processor and Disk Manager). This method, however, creates a new problem, since the amount of shared data in the system increases after data transfer. This is especially true after paging from a hard disk. The data read into the shared memory continue to reside in the Disk Manager cache, even after the cache line has been used by a requesting processor. The writing overhead for shared cache lines adds extra bus traffic and degrades system performance. The block-I/O mode of cache protocol forces the cache line of the requesting cache to be written back to main memory and to be invalidated after the last word of the cache line has been written from the I/O device. The second bit (mode-modification bit) of the cache-mode register of each cache controller has no influence on the operation of the cache controller when it is a snooping cache, but it affects the operation when it is a requesting cache. This control mechanism is orthogonal to the protocol-type control. Hence, it should be noted that each cache controller can be in either standard or block-I/O mode without regard to the protocol type. Although the block-I/O mode can also maintain cache coherency, even if it is used by the ordinary processors, it is intended to be used only by the I/O Processor and the Disk Manager.

- *System bus*

As mentioned previously, the shared bus is a resource that limits system performance, even though a private cache is added to each processor to reduce the bus traffic. The usual goals in system-bus design are to maximize the bus bandwidth and to minimize the delay, with a reasonable cost and complexity of hardware.

There are various design choices for the bus-transfer mechanism, to increase the bus bandwidth. An example is a so-called block-transferred bus, which can provide a higher bus bandwidth if the memory cycle time is much larger than the bus clock cycle, because one memory cycle is required for the first word but only one bus cycle for each consecutive word. With current technology, however, the memory access time is only a few times larger than the bus cycle, because the bus cycle time is limited by delays, signal-flight time, signal reflection, and clock skews. In the case of TOP-1, we designed the bus cycle time to be 62.5 ns (16 MHz), the same as that of the processor clock, and the main memory access time to be 187.5 ns (three cycles of the bus clock) including error checking and correction. This being so, the data width of the bus and the shared memory should usually be as wide as the cache line size, since it is desirable to transmit an entire line in one transfer-cycle time.

Instead of using a block-transferred bus, TOP-1 uses two buses, arbitrated independently, each with a 64-bit data width. The shared memory and the cache are two-way interleaved on 8-byte address boundaries. In particular, when a processor accesses an "even address" (an even multiple of eight), the access goes to the "even cache." If the access misses in the "even cache," the "even cache controller" issues a memory-access request to the "even main memory" through the "even bus." For an "odd address" (an odd multiple of eight) access, the control mechanism is similar. Consequently, the shared bus can provide an 85MB/s effective data-transfer rate for both read and write operations, since both require three 16MHz bus cycles.

Another novel aspect of the shared-bus design is the high-speed and fair-service arbitration mechanism. With a unique arbitration mechanism, which we named "modified back-off arbitration," the shared-bus arbitration is performed in one bus cycle (62.5 ns), using distributed control (the arbitration-control logic is found in all processors). Instead of an encoded arbitration code, as used in so-called back-off mechanisms, each requester issues a decoded code on the bus according to its current request priority. This mechanism allows high-speed arbitration, because no feedback loop is required to settle the arbitration bus signal. After a requester has obtained the right to access the bus as a result of the arbitration, each requester increments its priority, so that a quasi-round-robin service is realized. Moreover, each requester can set its priority range so that the arbitration code is constrained to the specified range. Since this mechanism allows processors to be classified into several groups with regard to the arbitration priority range, it is useful for process scheduling. For instance, some processors used for high-priority processes can run faster if a high-priority range is assigned. In such an application, however, we must be careful to ensure that all processors have some bus access.

4. Multiprocessor synchronization

In general, multiprocessor synchronization can be implemented by shared variables in the shared memory or by hardware message-passing mechanisms. TOP-1 implements both mechanisms.

- *Interprocessor signaling*

Although TOP-1 is a shared-memory multiprocessor, a message-passing hardware mechanism is also provided in order to allow the processors to communicate

asynchronous events or to interrupt one another. Messages are broadcast to processors specified as destinations, and any number of processors can be specified. The messages are received or discarded by the destination processors, according to the state of the receivers' buffers. We provide two kinds of message-passing protocols: Everybody messaging and Anybody messaging.

In Everybody messaging, all destination processors receive the message, and an interrupt signal to all receivers is generated only when all the specified receivers receive the message. If any of the destination processors cannot receive the message because of a full receive buffer, none of them receives the message, and no interrupt signal is generated to any destination processor. This mechanism is implemented by using a special bus line of wired-OR logic. Each processor specified as a destination activates the special bus line when it cannot receive the message because of a full receive buffer. At the end of cycle, each destination processor examines the state of the special line, and receives the message only when the line is inactive, indicating that all of the specified processors can receive the message. The sender also examines the state of the special line at the end of cycle and recognizes whether or not the message was successfully sent. Everybody messaging is very useful, since it allows an operating system to maintain TLB (translation lookaside buffer) consistency among multiple processors and also allows multiprocessing programming languages to cause interruption of the execution of processors.

In Anybody messaging, each destination receives a message and generates an interrupt signal to its processor if it can. Whenever one or more destinations receive the message, the sender detects that the message was successfully received. This is implemented by using the special bus line of wired-OR logic in a mode opposite to that of Everybody messaging. Each processor specified as a destination activates the special bus line when it can receive the message because its receive buffer is empty. At the end of the cycle, the sender examines the state of the special line and recognizes that the message was successfully sent when the special line is active. The mechanism is useful for requesting a service of any processor that is ready to serve.

- *Bus locking*

Shared-memory multiprocessors must implement atomic "read-modify-and-write" operations. TOP-1 implemented atomic operations with a so-called "bus-locking" mechanism in which the system bus is completely owned during an execution of the atomic read-modify-and-write operation. When a processor issues a locked instruction such as XCHG, the bus is locked without regard to cache hit or miss, and the effects of the memory modification are broadcast instantaneously. This allows a spin lock on a semaphore to be effectively implemented by using cacheable shared-memory space with a "read-read-modify-and-write" technique. (A semaphore is examined by a read loop of nonlocked instructions, which can be handled locally in the cache without issuing any bus access and affecting other caches. When the read loop detects that the contents of the semaphore are changed, the semaphore is examined again with a locked instruction to ensure mutual exclusion.)

TOP-1 also provides a mechanism to lock the bus for a sequence of instructions. Two I/O instructions, called Lock and Unlock, lock and unlock the bus. A sequence of instructions enclosed by Lock and Unlock is executed atomically, since the bus is locked during the period. This mechanism provides a means of implementing high-level synchronization operations, such as enqueue and dequeue, in a very effective way.

5. Statistics measurements

The cache behavior of uniprocessors has been extensively discussed in the literature [10], and some papers have reported on the cache performance of tightly coupled multiprocessors [11–14]. However, most such reports used software simulators driven by statistical parameters or traces, as did our preliminary performance evaluation.

In contrast, we used the hardware statistics monitor built into the TOP-1 cache/bus controller to evaluate the actual cache and bus performance of TOP-1. This monitor allows us to gather statistical data from the machine without any overhead. We can accurately count the number of fetches, including instructions that are prefetched but not actually executed owing to prefetch queue flushes. When a program has a tight loop that is intensively executed, software simulations usually give incorrect results, because the effect of the prefetch queue cannot be taken into account. The statistical events captured by our monitor are as follows:

- Number of memory accesses (instruction fetch, data read, and data write, separately).
- Cache hit ratio for each type of memory access.
- Number of write accesses to *private*-state cache lines and to *shared*-state cache lines.
- Number of write-backs for dirty cache lines.
- Number of cache updates/invalidations from the snoopy mechanism.
- Number of dirty-data replies to the requester.
- Number of bus cycles during which a processor waits for the bus.

In addition, we can dynamically specify the address range (by page boundaries) for which the statistics monitor counts the numbers shown above. Thus, if the program mapping is previously known or can be controlled by the

Table 1 Memory access rates for three benchmark programs.

Application		*Instruction fetch* (%)	*Data read* (%)	*Data write* (%)
Parallel MAKE	Kernel processor	86	10	4
	User processor	75	17	8
Edge detection	Kernel processor	84	12	4
	User processor	69	23	8
Ray tracing	Kernel processor	88	9	3
	User processor	60	30	10

Table 2 Cache-hit ratios for three benchmark programs.

Application		*Instruction fetch* (%)	*Data read* (%)	*Data write* (%)
Parallel MAKE	Kernel processor	99.99	99.99	99.99
	User processor	99.40	99.69	97.28
Edge detection	Kernel processor	99.87	99.89	99.71
	User processor	99.95	99.42	99.44
Ray tracing	Kernel processor	99.99	99.99	99.93
	User processor	99.54	99.19	99.67

operating system, we can selectively gather the statistics for each memory area. For example, we can get the cache hit ratio for the kernel only, for the synchronization variables only, or for processing data only.

We gathered statistics for three benchmark programs, "parallel MAKE," image-edge detection, and ray tracing. In "parallel MAKE," each processor performs MAKE (C compile and bind) independently for the same source code. Thus, the measurement is not for parallel processing but for multiprocessing. The parallel edge-detection program detects edges in images of 1024 × 768 pixels with 256 levels of gray scale, by using two orthogonal 3 × 3 masks called Sobel masks. The entire image is spatially divided into eight equal subareas. Each task detects the image edges in one subarea. Therefore, data sharing occurs only on the boundaries of the subareas. The ray-tracing program is parallelized so that one task is executed for each horizontal scan line.

Before presenting the results, we mention the structure of the operating system TOP-1 OS, since the benchmark programs executed on TOP-1 OS (hence their statistical data) are greatly affected by the OS structure. TOP-1 OS, implemented at the Tokyo Research Laboratory, is a multiprocessor extension of the AIX® PS/2® operating system. In TOP-1 OS, a modified AIX kernel runs on the Disk Manager, attached to the hard-disk controller (see Figure 1). In Tables 1–4, we use the notation "kernel processor" to specify the Disk Manager, dedicated to execution of the kernel, and "user processor" to denote processors dedicated to user-mode execution. For the measurements discussed in this paper, the cache-mode register of all processors, including the Disk Manager, was set to the update-type protocol with standard mode. For evaluations with other combinations of cache protocol and mode, refer to [15–17].

Table 1 presents the memory access rate statistics for instruction fetch, data read, and data write. Memory access rate is the percentage of all memory accesses by a processor that fall in a given category. In Table 1, for example, 86% of the memory accesses of the kernel processor were for instruction fetches when the parallel MAKE benchmark was run. It can be seen that the instruction fetch rate is very large. We believe the major reason for this is the extra prefetches, instructions that are fetched but not actually executed, due to the prefetch queue flush.

Table 2 presents the cache-hit ratio for each type of memory access, and shows that the hit ratio is very high in all cases. Although the major reason for this must be the small size of the working sets for all three programs, another important reason can be offered. We surmised that stack accesses drastically increased the data hit ratio in all cases, since most programs written in C include many stack accesses, and stack accesses have tight locality, both in time and space. We verified this effect by gathering statistics for data areas including and excluding the stack area (these data are not shown in the table), and we observed that very large percentages of data accesses are

Table 3 Ratios of shared-data write and copy-back for three benchmark programs.

Application		*Rate of shared-data write* (%)	*Rate of dirty-line copy-back* (%)
Parallel MAKE	Kernel processor	80	64
	User processor	27	13
Edge detection	Kernel processor	69	45
	User processor	21	4
Ray tracing	Kernel processor	90	12
	User processor	36	1

Table 4 Snoopy characteristics for three benchmark programs.

Application		*Snoopy read hit ratio* (%)	*Snoopy write hit ratio* (%)
Parallel MAKE	Kernel processor	3.3	78.4
	User processor	1.9	38.4
Edge detection	Kernel processor	1.0	76.3
	User processor	1.4	12.5
Ray tracing	Kernel processor	7.8	68.5
	User processor	4.8	7.4

generated by stack operations [15]. This is probably because C passes parameters through the stack, local variables are allocated to the stack, and the Intel 80386 does not have enough registers, so programs must often save and restore the register values on the stack.

Table 3 shows the statistics for shared-data write and dirty-line copy-back, which also generate bus traffic. The rate of shared-data write in the table is the percentage of the total number of write accesses that resulted in data writes to shared data. The rate of dirty-line copy-back is the percentage of the total number of cache misses that result in copy-back to the main storage. It can be seen that the rate of shared writes is very high for the kernel processor. This result depends strongly on the structure of TOP-1 OS and the cache protocol used, since the hard disks are directly connected to the kernel processor (Disk Manager); hence the data and code read from the hard disk are always transferred via the cache of the kernel processor. Another reason is the essential data sharing between the kernel processor and the user processors.

Table 4 shows the read hit ratios and the write hit ratios for the bus-induced snoopy accesses. The snoopy read hit interferes with the processor access only when the corresponding snoopy data are in the *dirty* state, because of the need for a data reply. The snoopy write hit always interferes with the processor access, since it always requires a cache-data update (in these measurements, all caches use the update-type protocol). It can be seen that the snoopy read hit ratio is very small, from 1% to 8%. This result indicates that most of the read accesses induced on the bus are for local data read misses but not for shared data. In contrast, it can also be seen that the snoopy write hit ratio is very large, from 69% to 78% for the kernel processor and from 7% to 38% for the user processors. This is because the bus write access occurs only for shared-state cache lines and never for private-state cache lines.

In this section, we have presented and briefly analyzed the measurement results of the TOP-1 memory system performance. The situation is drastically changed if the configuration is different from the one described above. Refer to [15] and [17], which include a more detailed evaluation for the update and invalidate protocols, and [16], which evaluates more extensively the TOP-1 memory system under TOP-1 OS for various configurations.

6. Summary

We have described the unique hardware features of the TOP-1 multiprocessor workstation. The unique features are the mechanism that allows several different snoopy protocols to coexist, the two independent buses and the efficient and fair arbitration mechanism for them, and the message-broadcasting mechanism for effective asynchronous communication. The statistics monitor

hardware is very useful for precisely measuring the frequency of various hardware events. The hardware is now fully operational, and several machines have been built.

Using several TOP-1s, we are evaluating the multiprocessor hardware design more intensively, studying more effective operating systems, and designing and implementing parallel-processing languages and applications.

Acknowledgments

The authors would like to thank Dr. Norihisa Suzuki, Dr. Tsutomu Kamimura, and Mr. Toshiaki Kurokawa, who initiated and managed the TOP-1 research project. We also wish to thank the members of the Workstation Systems and Parallel Systems groups at the Tokyo Research Laboratory of IBM Japan, Ltd. for technical discussions.

References

1. R. R. Atkinson and E. M. McCreight, "The Dragon Processor," *Proceedings of the Second International Conference on ASPLOS*, IEEE, Palo Alto, 1987, pp. 65–69.
2. C. P. Thacker and L. C. Stewart, "Firefly: A Multiprocessor Workstation," *Proceedings of the Second International Conference on ASPLOS*, IEEE, Palo Alto, 1987, pp. 164–172.
3. R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkins, and R. G. Sheldon, "Implementing a Cache Consistency Protocol," *Proceedings of the 12th International Symposium on Computer Architecture*, ACM, Boston, 1985, pp. 276–283.
4. J. R. Goodman, "Using Cache Memory to Reduce Processor-Memory Traffic," *Proceedings of the 10th International Symposium on Computer Architecture*, ACM, Stockholm, Sweden, 1983, pp. 124–131.
5. J. S. Liptay, "Structural Aspects of the System/360 Model 85," *IBM Syst. J.* **7**, No. 1, 15–21 (1968).
6. *IBM System/370 Model 168, Theory of Operation/ Diagrams Manual—Processor Storage Control Function (PSCF)*, Vol. 4, Order No. SY22-6934-01, available through IBM branch offices.
7. *IBM 3033 Processor Complex, Theory of Operation/ Diagrams Manual—Processor Storage Control Function (PSCF)*, Vol. 4, Order No. SY22-7004-01, available through IBM branch offices.
8. R. N. Gustafson and F. J. Sparacio, "IBM 3081 Processor Unit: Design Considerations and Design Process," *IBM J. Res. Develop.* **26**, No. 1, 12–21 (1982).
9. N. Suzuki, *Parallel LISP: Languages and Systems, Lecture Notes in Computer Science 441*, Springer-Verlag, New York, 1990, pp. 353–363.
10. A. J. Smith, "Cache Memories," *ACM Computing Surv.* **14**, No. 3, 473–530 (1982).
11. J. Archibald and J. L. Baer, "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model," *ACM Trans. Computer Syst.* **4**, No. 4, 273–298 (1986).
12. R. L. Lee, P. C. Yew, and D. H. Lawrie, "Multiprocessor Cache Design Considerations," *Proceedings of the 14th International Symposium on Computer Architecture*, Pittsburgh, 1987, pp. 253–262.
13. S. J. Eggers and R. H. Katz, "A Characterization of Sharing in Parallel Programs and Its Application to Coherency Protocol Evaluation," *Proceedings of the 15th Annual International Symposium on Computer Architecture*, ACM, Honolulu, 1988, pp. 373–382.
14. S. J. Eggers and R. H. Katz, "Evaluating the Performance of Four Snooping Cache Coherence Protocols," *Proceedings of the 16th Annual International Symposium on Computer Architecture*, ACM, Jerusalem, Israel, 1989, pp. 2–15.
15. N. Oba, A. Moriwaki, and S. Shimizu, "TOP-1: A Snoop-Cache-Based Multiprocessor," *Proceedings of the 9th IEEE International Phoenix Conference on Computers and Communications*, IEEE, Phoenix, 1990, pp. 101–108.
16. N. Yamanouchi, "Performance Effects of Program Structures on a Snoop-Cached Multiprocessor System," *Proceedings of the International Conference on InfoJapan '90*, IPSJ, Tokyo, Japan, 1990, pp. 339–346.
17. S. Shimizu, N. Oba, A. Moriwaki, and T. Nakada, "Design and Evaluation of Snoop-Cache-Based Multiprocessor, TOP-1," *Proceedings of the International Symposium on Shared Memory Multiprocessors*, IPSJ, Tokyo, Japan, 1991, pp. 209–217.

Received October 9, 1990; accepted for publication February 20, 1991

Shigenori Shimizu *IBM Research Division, Tokyo Research Laboratory, IBM Japan, Ltd., 5-19, Sanban-cho, Chiyoda-ku, Tokyo 102, Japan.* Dr. Shimizu received the B.E. in 1977, the M.S. in 1979, and the Ph.D in 1983, all in instrumentation engineering, from Keio University, Japan. In 1984, he joined the research staff of the Japan Science Institute (now called the Tokyo Research Laboratory) of IBM Japan in Tokyo, where he has specialized in the areas of VLSI architecture and parallel processing.

Nobuyuki Oba *IBM Research Division, Tokyo Research Laboratory, IBM Japan, Ltd., 5-19, Sanban-cho, Chiyoda-ku, Tokyo 102, Japan.* Dr. Oba is an advisory researcher in the Parallel Systems Hardware group at the Tokyo Research Laboratory, IBM Japan. He has worked on the TOP-1 hardware design and performance evaluation. His research interests include parallel architectures and performance evaluation for parallel systems. Dr. Oba received the B.E. and Ph.D degrees from Tohoku University, Japan, in 1981 and 1986, respectively.

Takeo Nakada *IBM Research Division, Tokyo Research Laboratory, IBM Japan, Ltd., 5-19, Sanban-cho, Chiyoda-ku, Tokyo 102, Japan.* Mr. Nakada received the B.E. degree in communication engineering in 1984 and the M.E. degree in information engineering in 1986 from the Faculty of Engineering at Tohoku University, Japan. He joined IBM Japan at the Tokyo Research Laboratory in 1986. Currently he is a researcher in the Parallel Systems Hardware group. Mr. Nakada's research interests include parallel computer systems, performance evaluation, and parallel-processing algorithms.

Moriyoshi Ohara *IBM Research Division, Tokyo Research Laboratory, IBM Japan, Ltd., 5-19, Sanban-cho, Chiyoda-ku, Tokyo 102, Japan.* Mr. Ohara is a researcher in the Parallel Systems Hardware group at the Tokyo Research Laboratory. He received the B.E. from the University of Tokyo in 1986,

joining IBM Japan that same year. Mr. Ohara's research interests include processor architecture, parallel architecture, and compiler technology.

Atsushi Moriwaki *IBM Research Division, Tokyo Research Laboratory, IBM Japan, Ltd., 5-19, Sanban-cho, Chiyoda-ku, Tokyo 102, Japan.* Mr. Moriwaki received the B.E. and M.E. in applied mathematics and physics in 1982 and 1984, respectively, from Kyoto University, Japan. In 1984 he joined IBM Japan at the Japan Science Institute, where he has worked on CAD system maintenance and logic design support.