TOP-1: A Snoop-Cache-Based Multiprocessor

N. Oba  A. Moriwaki  S. Shimizu

IBM Research, Tokyo Research Laboratory,
IBM Japan, Ltd.
Tokyo, Japan 102

ABSTRACT
This paper presents a new cache coherence protocol and has performance analysis of a snoop-cache-based multiprocessor, TOP-1, which is tightly coupled and has pure shared memory. TOP-1 has two 64-bit buses with interleaved address access to provide a high data-transfer rate, and a large snoop cache to provide a high cache hit ratio. It also has a TOP-1 hybrid coherence protocol, which allows a write-update protocol and a write-invalidate protocol to coexist simultaneously. Furthermore, these protocols can be dynamically changed on the fly without any coherence problem. Each processor card has a statistics unit which collects various important statistical data without any hardware overhead. First, we give an overview of the TOP-1 architecture and its concepts. Next, we present the TOP-1 hybrid protocol and explain how it works. Then, we discuss the TOP-1 protocol and its performance by comparing the write-update and write-invalidate protocols.

1. Introduction
In the past several years, shared-memory shared-bus multiprocessors have been extensively analyzed and developed because of the simplicity of the parallel programming model. Most of them use the multi-cache mechanism to reduce the traffic on the shared bus. The problem of cache coherence is addressed by a snoop cache scheme; that is, each cache in the system constantly watches the transactions on the shared bus and takes appropriate actions to maintain coherence.

TOP-1 is a snoop-cache-based multiprocessor workstation. Figure 1 shows TOP-1's hardware block diagram. It consists of 10 processor cards, a 128-MByte (maximum) shared memory, a 1.2-GByte hard disk, and a MICROCHANNEL interface. Each processor card has an Intel 80386 for integer calculations and a WEITEK 1167 for floating point calculations. These components are interconnected by two 64-bit synchronous buses, which provide an 85-MByte/sec data transfer rate. Each processor has a 128-kByte private cache with snoop capability. The IBM PS/2 model 80 is in charge of I/O devices (e.g., display, keyboard, printer, and network).

Many snoop protocols have been proposed to date [1,2,3,4]. In view of the method of coherence control for shared data, they could be categorized into two types, write-update type and write-invalidate type. Let the cache block that is held by more than one cache be a shared-block, and the block that is held by only one cache in the system be a private-block. In the write-update protocol, when a processor needs to modify a shared-block, all copies of it in other caches are simultaneously updated. In the write-invalidate protocol, on the other hand, all copies in other caches are invalidated. In both protocols, the shared bus is used whenever cache coherence control is required. Therefore, the system performance is affected by the behavior of the shared data. We will therefore explain these in the light of our experimental work.
We developed the **TOP-1 hybrid protocol**, which allows these two types of protocol to coexist. Furthermore, it allows them to be changed on the fly. In Chapter 2 we will describe how the TOP-1 protocol works without causing a coherence problem. Then, in Chapter 3 we will give the performance model of TOP-1, which can be applied to the model of a general snoop-cache-based multiprocessor.

In Chapter 3 we will give the performance model of TOP-1, which can be applied to the model of a general snoop-cache-based multiprocessor. In Chapter 4 we will briefly describe the hardware event monitor called the **statistics unit**. After that, we will give the TOP-1 performance measurement results obtained by using that event monitor. All the data presented in this document were obtained on the bare TOP-1 hardware. The TOP-1 Operating System has been already implemented on TOP-1, but we intentionally refrained from using it, because we wanted to avoid any kind of OS effects or overheads. Finally, we will discuss the performance of the snoop-cache-based multiprocessor.

### 2. TOP-1 Protocol

#### 2.1. TOP-1 Hybrid Protocol

The TOP-1 protocol is a write-back- and write-allocate-type snoop cache protocol. The key to the TOP-1 protocol is that it allows two types of coherence control (the write-update protocol and the write-invalidate protocol) to coexist. Furthermore, they can be dynamically changed on the fly without causing a coherence problem.

TOP-1 protocol works as follows.

Each processor's cache has two cache modes, that is, update-mode and invalidate-mode. The cache mode is specified by a cache mode register, in TOP-1's cache bus controller. Since this register is I/O mapped, each CPU can change the content of the cache mode register by an I/O instruction. Whenever the snoop write comes into the cache, the cache controller refers to the cache mode register and takes a specified action.

The cache block is controlled by five states: invalid, clean-private, clean-shared, dirty-private, and dirty-shared. The state transition diagram of the TOP-1 protocol is shown in Figure 2. On the shared bus, there is a signal line Cache Hit (CH) that indicates a snoop hit. The CH is driven by an open-collector driver, to allow it to be driven by more than one cache controller simultaneously. When a processor causes a read miss, the cache controller takes the shared bus and puts the read command onto the shared bus. If another cache has a copy, it activates the CH. The reading cache controller can recognize that the cache block is shared or not shared by the CH. When a processor writes to the shared block, the cache controller gets the shared bus and supplies the address and data. When the snoop write comes in, the cache controller checks the corresponding tag of the cache block. In the write-update mode, if the snoop hits in the cache, the cache controller updates the cache block by the latest value and does not activate the CH. The cache block state of the CPU, which does the shared block write, is changed to private if the CH is not activated, or to shared if the CH is activated, as shown in Figure 2.

Note that each processor can take either of these modes arbitrarily. Figure 3 shows the situation of two protocols coexisting. In Figure 3, processors 2 and 4 are set in the write-update mode and processor 3 is set in the write-invalidate mode. When the snoop write comes in, the corresponding data in processors 2 and 4 are simultaneously updated, while the data in processor 3 are invalidated. The cache controllers of processors 2 and 4 activate the CH but that of processor 3 does not. In this case, the state of the cache block in processor 1 is still shared because the CH is asserted.

To avoid a conflict of the access to the cache mode register, register reference by the snoops always has a higher priority than CPU access. This means that the snoop operation referring to the cache mode register makes CPU access to it wait for a cycle. Therefore, CPU can change the cache mode at any time without creating a consistency problem.
2.2. Write-update vs. Write-invalidate

When we were designing the snooping protocol for TOP-1, we discussed the advantages and disadvantages of various protocols. The main question was which, the write-update type or the write-invalidate type, could provide better performance, and thus which type we should use. These protocols are equivalent from the viewpoint of coherence. However, both of them have advantages and disadvantages with respect to the performance.

The write-update protocol would be suited to tightly shared data, which are frequently used by a lot of processors. If the write-invalidate protocol is used in such a case, every time a processor modifies the shared data, all copies in other caches are invalidated. Other processors cause cache misses when they access those shared data. In the write-update protocol, however, other processors can read data without cache misses, because all copies are simultaneously updated. The write-update protocol would provide better performance for a semaphore, or a buffer used in a producer-consumer model.

On the other hand, the write-invalidate protocol would provide better performance for data exclusively used by one processor, such as local variables, and stack data. Data that are exclusively used by one processor are possibly shared by more than one processor when they are migrated to another processor. We discuss such a situation later in this section. A write-invalidate protocol would be also suited to shared data that are potentially shared by more than one processor but are exclusively used by one processor for part of the time.

Although we have discussed the advantages and disadvantages of the two protocols above, the practical situation is more complicated. We discuss the practical situation with respect to the memory read/write sequence and to the cache configuration. Let us consider a simple producer-consumer model to estimate the cache performance.

Although read and write operations on the bus have a different effect on the system performance, we just focus on the number of bus operations to estimate the system performance. Let \( N_{bus} \) be the number of bus operations to maintain the cache coherence. We assume that the number of data is \( N_{data} \) and the data set occupies a consecutive address space. \( N_{data} \) is assumed to be a large number to ignore the effect of synchronization.

First, assume that the cache line size and the data bus width are equal to the processor's access-width. We assume that the producer is implemented on processor \( A \) and the consumer is on processor \( B \).

The first iteration of the producer-consumer model has different cache operations from the second and later iterations, because data writes cause write misses. In the first iteration, \( N_{data} \) writes by processor \( A \) cause \( N_{data} \) write misses, which are practically executed by \( N_{data} \) read misses followed by \( N_{data} \) write hits to the private data. Note that processor \( B \) has not had copies yet. Processor \( B \) reads \( N_{data} \) data that are held only in processor \( A \)'s cache, so that processor \( B \) causes \( N_{data} \) read misses. Therefore, \( N_{bus} = 2 \times N_{data} \).

In the second and later iterations, data writes by processor \( A \) are executed as writes to the shared data, because processor \( B \) has already had the copy. In the write-update mode, there are \( N_{data} \) writes to the shared data by processor \( A \), and \( N_{data} \) read hits by processor \( B \). Therefore, \( N_{bus} = N_{data} \). In the write-invalidate mode, on the other hand, there are \( N_{data} \) writes to the shared data by processor \( A \), but data in processor \( B \) are all invalidated, so that processor \( B \) causes \( N_{data} \) read misses. Therefore, \( N_{bus} = 2 \times N_{data} \). In this case, the write-update mode provides a better performance, because the write-update mode uses the shared bus \( N_{data} \) times, whereas the write-invalidate mode does \( 2 \times N_{data} \) times.

Next, let us consider the situation where the cache line size and the data bus width are wider than the processor's access-width. Assume that the cache line size and the data bus width are eight times wider than the processor's access-width.

In the first iteration, processor \( A \) writes the first data of a cache line and causes a write miss, but succeeding seven writes are write hits to the private data, because the first write miss causes the cache to read one cache line consisting of eight data. Like processor \( A \), processor \( B \) reads the first data of a cache line and causes a read miss, but succeeding seven reads are read hit. Therefore, \( N_{bus} = 2 \times N_{data} + 8 - N_{data} \).

Operations in the second and later iterations are interesting. In the write-update mode, all writes by processor \( A \) are write hits to the shared data, because processor \( B \) has already had copies. Processor \( A \)'s cache uses the shared bus to update copies in processor \( B \)'s cache. All data reads by processor \( B \) are read hits, so that processor \( B \) doesn't use the shared bus. Therefore, \( N_{bus} = N_{data} \). In the write-invalidate mode, the write to the first data of a cache line is a write hit and uses the shared bus to invalidate the copy in processor \( B \)'s cache. However, succeeding seven writes are write hits to the private data because processor \( B \) no longer has copies. The first read by processor \( B \) is a read miss, but succeeding seven reads are read hits. Therefore, \( N_{bus} = N_{data} + 8 - N_{data} \).

In this case, the write-invalidate mode provides a better performance, because the write-update mode uses the shared bus \( N_{data} \) times, whereas the write-invalidate mode does \( N_{data} / 4 \) times. Even in such a simple model, these two modes behave differently.

We next discuss the property of the shared data. There are two kinds of shared data real shared data and accidental shared data. Real shared data means data that are actually referenced by more than one processor. For example,
processor_A produces a data structure and passes it to processor_B, who processes that data structure. In a snoop-cache-based multiprocessor, such a data structure is realized as shared data. Accidental shared data, on the other hand, means data that are referenced by just one processor at a time, but by more than one processor over a period. For example, suppose that a process is running on processor_A and is then migrated to processor_B for some reason. The working set of this process is moved from processor_A to processor_B, so that the working set is not used on processor_A anymore. However, all or part of the working set should still remain in processor_A’s cache. This is possible in TOP-1, because each processor has a 128K-Byte cache, which can contain thirty two 4K-Byte pages. We call this accidental shared data.

Generally speaking, the write-update mode is suited to the situation where the number of read shared data is large. The write-invalidate mode is suited to the situation where accidental shared data are often created. However, it depends on the memory read/write pattern. In the producer-consumer model, the write-update mode provides a better performance if the consumer reads data right after the producer writes data. The write-update mode, however, does not work well if the consumer reads data after the producer writes data many times, because the overhead of writes to the shared data, that are not used by the consumer immediately, breaks out.

Consequently, we decided to implement these two protocols on TOP-1 and to study their behavior with real applications.

3. Performance Model

3.1. Performance Model of TOP-1

There are many possible ways to represent system performance, such as MIPS (Million Instructions Per Second) and other benchmarks. However, it is said that these kinds of evaluation sometimes lead to ambiguous results. We will focus on the cache hit ratio and the data sharing rate to estimate the system performance.

In terms of cache, it goes without saying that a high cache hit ratio provides high performance, but as mentioned before, the data sharing rate affects the performance, too.

To evaluate the cache/bus performance, we should take account of the following three parameters.

• Cache Hit Ratio
• Data Sharing Rate
• Replace Ratio (if the cache block to be replaced is dirty, it must be written back to the shared memory.)

The data transfer rate of cache/bus system is another factor that we should take into account, but it is assumed to be implied by the execution cycle here. Therefore, we can calculate the turn-around time (TAT) of memory access by the following formula:

$$TAT = (c_{cache\&bus} + c_{bus\_wait}) \times \tau .$$

(1)

$c_{cache\&bus}$ is the number of cycles executed in the cache and bus system. $c_{bus\_wait}$ is the number of cycles spent waiting to get the bus when it is occupied by another processor. $\tau$ is cycle time (in seconds).

In a snoop-cache-based multiprocessors, $c_{cache\&bus}$ can be calculated as:

$$c_{cache\&bus} = Prh \times C_{rh} + Prm \times C_{rm} + Pwmp \times C_{wmp} + Prms \times C_{wms} + P_{r} \times C_{r}.$$  

(2)

Table 1 listed the meaning of each term in Formula (2) as well as TOP-1’s actual execution cycles.

<table>
<thead>
<tr>
<th>Execution Cycle</th>
<th>TOP-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c_{rh}$</td>
<td>2</td>
</tr>
<tr>
<td>$c_{rm}$</td>
<td>8</td>
</tr>
<tr>
<td>$c_{wmp}$</td>
<td>2</td>
</tr>
<tr>
<td>$c_{wms}$</td>
<td>8</td>
</tr>
<tr>
<td>$c_{r}$</td>
<td>13</td>
</tr>
<tr>
<td>$c_{bus_wait}$</td>
<td>4</td>
</tr>
</tbody>
</table>

We can estimate the system performance with Formula (2) if the probability parameters are given. Let the cache hit ratio and the data sharing rate be $h$ and $S$, respectively, and let the read access rate of all memory accesses and the replacing rate be $R_{read}$ and $R_{replace}$, then Formula (2) is rewritten as:

$$c_{cache\&bus} = Prh \times C_{rh} + Prm \times C_{rm} + Pwmp \times C_{wmp} + Prms \times C_{wms} + P_{r} \times C_{r}.$$  

(2)

1 A write miss is performed by the sequence of a read miss and a write hit. Write miss to the private block means that a read miss occurs first and a write hit to the private block follows. In the same way, write miss to the shared block means that a read miss occurs first and a write hit to the shared block follows.
Formula (3) shows that the execution cycle decreases when the cache hit ratio increases or the shared rate decreases. However, it should be noted that the hit ratio and the shared rate are not given independently. Let us consider an extreme example of a multiprocessor system that does not allow the existence of shared data. In such a system, a copy of shared memory must be held in only one cache. It is clear that the data sharing rate $S$ is zero, but the hit ratio $h$ may be very low.

4. Actual Evaluation of TOP-I

4.1. TOP-I Statistics Unit

Before going into the evaluation results, we will explain the statistics unit. This is a hardware event monitor built into TOP-I cache, bus controller chip, which consists of 20-bit counters to capture the number of events of, for example:

- Instruction fetch (hit miss)
- Data read (hit miss)
- Data write (hit miss, private shared)
- Dirty line replace
- Snoop read (hit miss)
- Snoop write (hit miss)
- CPU/snoop access conflict
- Bus wait cycle.

There is no hardware overhead in collecting these numbers. If one of the counters overflows, the cache bus controller stops counting up and sends an interrupt to the processor. Moreover, the address range of statistics gathering can be specified. Event counters are counted up only if the access address is in the range between address range registers. Probabilities in Table 1 can be obtained by using the statistics unit.

4.2. Evaluation on TOP-I

In order to evaluate the TOP-I performance, we ran an example program to obtain various statistical data. The characteristics required for the programs are as follows:

- The algorithm is so simple that the computation procedure can be easily controlled.
- The data sharing rate is controllable according to the matrix decomposition.

We chose the Laplace equation on a torus as an example. This torus is coordinated and discretized so that it is represented by the set of points

$\{(i, j) \mid 0 \leq i < m, 0 \leq j < n; i, j, m, n : \text{integer}\}$. (4)

Then the solution is represented by an $m \times n$ matrix,

$U = (u_{i,j})$, whose element is the value of the solution on the corresponding point. As the torus wraps round, we can specify that $u_{i+m,j} = u_{i,j}$ and $u_{i,j+n} = u_{i,j}$ so that no boundary condition exists. The finite-difference approximation we used here was the five-point form of the iterative scheme we adopted is the simple Jacobi method. Therefore, the scheme is defined by

$u_{i,j}^{k+1} = (u_{i-1,j}^{k} + u_{i+1,j}^{k} + u_{i,j-1}^{k} + u_{i,j+1}^{k}) / 4$. (5)

To compute this scheme in parallel, we divide the torus $U$ into $p$ subsets and assign each of them to one of $v$ processors. In an iteration, each processor calculates the assigned $u_{i,j}^{k+1}$ by Formula (5). In an iteration, each processor is in charge of calculating $m \times n$ elements, that is to say, it reads $m \times n \times 4$ elements and writes $m \times n$ elements. All processors calculate the assigned sub-matrices in parallel and take a synchronization at the end of each iteration.

Because the calculation of $u_{i,j}^{k+1}$ is divided among $v$ processors, the data referenced during calculation is shared if it is on the boundary of sub-matrices. Thus, the ratio of shared data depends on the choice of the number of sub-matrices $p$ into which the torus is divided. We control the data sharing rate by the value of $p$. In our computation scheme, the Laplace matrix is horizontally divided into $p$ sub-matrices, as shown in Figure 4. The total number of data shared between processors is $n \times 2 \times p = 2np$, which is the number of boundary data. If we assume a steady-state model and an infinite cache size, there is no cache miss in the write-update protocol, because no data is flushed out from the cache. In the write-invalidate protocol, on the other hand, the shared data are invalidated when shared writes occur. Therefore, there are $2np$ cache misses in the invalidate-type protocol, because the number of boundary data is $2n$ for each sub-matrix.

Figure 4: Sub-Matrix Allocation on Processors
The data sharing rate $S$ is defined as:

$$S = \frac{N_{\text{shared write}}}{N_{\text{write}}}$$  \hspace{1cm} (6)

where $N_{\text{shared write}}$ is the number of writes to the shared data, and $N_{\text{write}}$ is the total number of writes. $S$ does not mean the amount of space occupied by the shared data.

In this section, we first discuss the execution model, and then show the evaluation data.

4.3. Evaluation Results

We implemented the Laplace equation discussed in the previous section. The Laplace matrix $U$ is constructed by 1280 rows of 64 columns, i.e., $m = 1280$ and $n = 64$. Each element consists of two integers: $u_{i,j}$, which corresponds to 8 bytes. Note that a row occupies continuous 512 bytes on memory. $p$ is varied from 10 to 640 in order to vary the data sharing rate. The number of processors $v$ is ten. All sub-matrices are periodically allocated to processors, as shown in Figure 4. This allocation is chosen not to cause cache replacements by accesses of the Laplace matrix. The effect of the synchronization is very small, because the time taken to calculate one iteration is much longer than that to synchronize. Data on the sub-matrix boundary are shared by only two processors. An element $u_{i,j}$ is written only once in an iteration, before it is referred to calculate four adjacent elements, $u_{i+1,j}$, $u_{i,j+1}$, $u_{i-1,j}$, and $u_{i,j+1}$ in the next iteration.

In order to simulate real shared data and accidental shared data, we performed the evaluation in two ways: fixed allocation and alternating allocation. Fixed allocation means that every processor is in charge of pre-defined sub-matrices. For instance, from the beginning to the end of program, processor 1 calculates sub-matrix 1 and processor 2 does sub-matrix 2. This is used to evaluate the effect of real shared data on the system performance. On the other hand, alternating allocation means that every processor is in charge of different sub-matrices at every $L$th iteration. For instance, first, processor 1 calculates sub-matrix 1 and processor 2 does sub-matrix 2. After the next $L$ iterations, processor 1 does sub-matrix 2 and processor 2 does sub-matrix 1. This simulates the worst case for process migration, and this is used to evaluate the effect of accidental shared data on the system performance. The program for fixed allocation is called $\text{LAPf}$ and that for alternating allocation is called $\text{LAPA}$.

Table 2 shows the number of read hits/misses in the fixed allocation and the alternating allocation. Table 3 shows the number of write hits/misses in the fixed allocation, and Table 4 shows that in the alternating allocation.

4.3.1. Fixed Allocation

To see the effect of the snoop protocol on the system performance, we ran $\text{LAPf}$ in seven $p$ cases and collected statistical data. Figure 5 shows $\text{TAT}$, which is calculated by Formula (1). The minimum value of $\text{TAT}$ is 2 (cycle). When $p$ is increasing, the cache hit ratio is decreasing in the write-invalidate mode as shown in Figure 6. On the other hand, the cache hit ratio is almost 100% in the write-update mode. In fact the cache hit ratio is over 99% but not just 100%. It is because that code and stack are also allocated on the shared memory which possibly occupy the same cache block, so that a few cache replacements occur. The write-update mode provides a better performance in this case, because the write invalidate mode invalidates the shared data which are on the boundary of sub-matrices. The cache invalidation incurring cache misses degrades the performance in this program.

The data sharing rates in both protocols were almost equal. Nevertheless, we found that the total data sharing rate is very much lower than we expected. The total data sharing rate is only 10% even when $p = 640$. There are two kinds of data, C global data and C local data. The former are allocated to the 80386 DATA segment, and the latter to the 80386 STACK segment. In our environment, the data of matrices are allocated to the 80386 DATA segment, while the stack data are allocated to the 80386 STACK segment. Note that each processor has its own STACK segment, which is never shared between processors. When $p = 640$, the data sharing rate of the Laplace matrix reaches 100%. However, needless to say, stack data are not shared. Figure 7 shows the data sharing rate in two cases, that is to say, including stack read/write and excluding stack read/write. Even when the Laplace matrix is fully shared, the total data sharing rate is only 10%. This result implies that 90% of data write is stack write and 10% is matrix data write. We have not investigated this result in detail, but we can conjecture why the number of stack operations is so large. First, C passes parameters through the stack. Second, it allocates local variables to the stack. Third, the 80386 does not have...
enough registers, so the program must often save and restore the register values on the stack.

4.3.2. Alternating Allocation
To simulate the worst case for process migration, odd-numbered processors (1, 3, 5, 7 and 9) and even-numbered processors (2, 4, 6, 8 and 10) swap the target sub-matrices after every 20 iterations, i.e. $L = 20$. To be precise, processors 1 - 2, 3 - 4, 5 - 6, and 7 - 8 are coupled to swap the target sub-matrices with their partner. In the practical situation, stack data as well as matrix data will be migrated to another processor. However, we did not move stack data in this work. Therefore, the Laplace matrix is always shared between those processor couples, irrespective of the value of $p$. In this environment, the write-update mode updates all shared data, including accidental shared data. Accidental shared data survives till the end of program, because we allocate the matrix data not to cause the cache replacement. On the other hand, the write-invalidate mode invalidates shared data so that it works better than the write-update mode if the overhead of accidental shared data is larger than the decrease of the cache hit ratio.

Figure 5 shows the TAT of LAPa. The write-invalidate mode provides better performance when $p$ is small, because it invalidates unnecessary accidental shared data which survive, however, in the write-update mode. Figure 10 shows the data sharing rate of two modes. The write-update mode has about 10% shared data, irrespective of the value of $p$. Figure 9 shows the hit ratios. The write-invalidate mode has a lower hit ratio comparing to the write-update mode.

When the number of real shared data is small, the write-invalidate protocol achieves better performance than the write-update protocol. Although the write-invalidate protocol causes more read-misses than the write-update protocol, the write-invalidate protocol reduces the write access to the accidental shared data. On the other hand, if the number of real shared data is larger, write-invalidate protocol is no longer superior to the write-update protocol due to the penalty of higher miss ratio.

5. Conclusion
We described the TOP-1 hybrid protocol, which allows the write-update and the write-invalidate protocols to co-exist. We also showed that these protocols can be changed on the fly without coherence problem. Next, we discussed situations where one of the two protocols works well or does not work well. We should take account of the hardware architecture, such as the processor data width, the cache line size and the data bus width to design the cache coherence protocol.

In Chapter 3, we introduced the performance model of TOP-1. This model could be extended to the general case of snoopy-cache-based multiprocessors. In Chapter 4, we showed actual measurement data derived from TOP-1. The Laplace equation was used to study how the hit ratio...
and the data sharing rate affect the system performance. In order to simulate real shared data and accidental shared data, we introduced a fixed allocation model and an alternating allocation model. We showed the effect of the data sharing rate and hit ratio on the system performance.

The write-update protocol works well if the number of accidental shared data is small in our experimental work. As far as accidental shared data is concerned, the writeinvalidate protocol provide a better performance than the write-update protocol. In the write-update protocol, accidental shared data survive in the cache until they are replaced. We think that if the write-update protocol is used, we need some mechanism to erase accidental shared data. This encourages us to emphasize the usefulness of TOP-1 hybrid protocol which has the capability of dynamic protocol change. Using write-invalidate protocol only when accidental shared rate is large reduces the penalty of the write-update protocol.

TOP-1 allows the two protocols to be changed on the fly and also allows them to coexist, but the next problem is when to change the two protocols. Our future plan is to use statistical data to select protocol. We are also gathering data under TOP-1 OS.

Bibliography


