Reducing Hot-Spot Contention in Shared Memory Multiprocessor Systems[1]

Sivarama P. Dandamudi
Centre for Parallel and Distributed Computing
School of Computer Science, Carleton University
Ottawa, Ontario K1S 5B6, Canada
sivarama@scs.carleton.ca

TECHNICAL REPORT TR-97-17

ABSTRACT

In parallel systems it is possible for several processors to request concurrent access to a shared data structure such as a synchronization variable. Such an access pattern causes what is known as hot-spot contention. In shared-memory multiprocessor systems that use a multistage interconnection network, hot-spot contention may result in "tree saturation" that degrades the system performance. It is important, therefore, to manage hot-spot contention properly. This paper reviews the existing strategies to reduce the effects of hot-spot contention. We first quantify the effects of hot-spot contention and identify the objectives of a hot-spot management strategy. We propose a taxonomy to categorize these strategies into one of *avoidance-based*, *prevention-based*, or *detection-based* methods. We then review and compare several representative strategies that have been proposed to reduce the effects of hot-spot contention. We conclude the paper by identifying several issues that need further research.

Index terms: Hot-spot contention, Memory contention, Message combining, Multistage interconnection networks, Shared-memory multiprocessors, Tree saturation.

1. INTRODUCTION

Asynchronous parallel systems can be implemented using either a shared-memory or a distributed-memory architecture. In shared-memory multiprocessor systems, all processors have access to a globally shared memory through an interconnection network as shown in Figure 1.1a. Shared-memory systems support a single shared address space and communication among the processors is through shared memory variables. In distributed-memory multicomputer systems, each processor has a local, private memory (see Figure 1.1b). In these systems, processors communicate by means of explicit message passing. Both types of systems are commercially successful. Example distributed-memory systems include the Intel Paragon and Teraflop systems, nCUBE3 from nCUBE, and Cray T3D and T3E [van96]. IBM RP3, HP-Convex Exemplar, and Sequent NUMA-Q 2000 belong to the shared-memory category [Alma94, van96].

We can divide shared-memory multiprocessors into either *uniform memory access* (UMA) systems or *non-uniform memory access* (NUMA) systems. In UMA multiprocessors, the cost of accessing a memory location by any processor in the system is the same. In NUMA

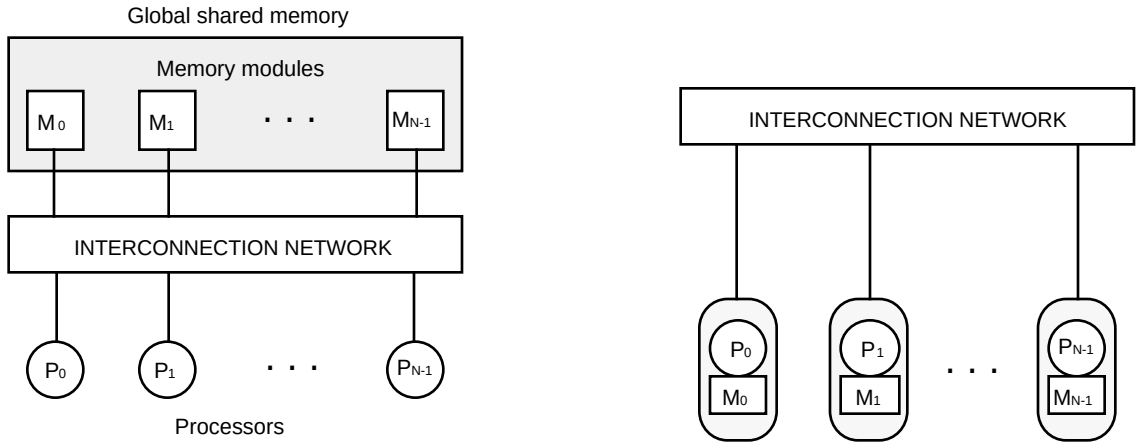
---

[1] An edited vesion of this paper will appear in *IEEE Concurrency*.
This report can be obtained from `http://www.scs.carleton.ca`

multiprocessors, the shared memory is physically distributed among the processors. Thus, some memory locations are closer to a processor and less expensive to access than others resulting in a non-uniform memory access cost.

In a UMA multiprocessor the shared memory is global to all processors. An interconnection network facilitates communication between the processors and the global shared memory. Typically, UMA multiprocessors use a single bus as the interconnection network. The Sequent Symmetry and Encore Multimax are examples of commercial bus-based UMA systems.

Using a common bus as an interconnection network severely limits system scalability (i.e., expandability of the system) because of bus bandwidth limitations. Furthermore, this type of interconnection network allows only one processor to communicate with the memory, leading to performance degradation. To avoid this problem, the shared memory is divided into *N* memory modules (Figure 1.1a). It is common to provide concurrent access to all memory modules, provided there are enough processors requesting access to memory modules and no two processors wish to access the same memory module. Large multiprocessor systems can use a multistage interconnection network. For example, the NYU Ultracomputer uses a multistage interconnection network similar to the one discussed in the appendix.



(a) Shared-memory multiprocessor **(b)** Distributed-memory multicomputer

Figure 1.1 Two parallel system architectures

Increasingly, the distinction between the shared-memory and distributed-memory architectures is getting blurred with the adaptation of hybrid architectures. For example, the Cray T3E uses the distributed memory architecture but logically supports a shared address space as in the shared-memory architecture.

In this paper, we primarily focus on large shared-memory systems that use a multistage interconnection network (details on multistage interconnection networks are given in the Appendix). IBM RP3, BBN Butterfly, and UI Cedar are examples of such systems (see [Alma94] for details on these systems). Typically, these systems have *N* processors and *N* memory modules. In such systems, a critical data structure located in one of the memory modules can create contention. Since each memory module can service one access request at a time, several (say, hundreds) processors requesting concurrent access to a particular memory module can create contention for that memory module. We give an example application that can cause such a

contention in Section 2.1. Note that memory contention occurs even if these requests are to different data items, all located in the same memory module. Such a memory module is called a *hot* memory module and the phenomenon is called *hot-spot contention*.

Contention for memory leads to contention for interconnection network buffers and links. In particular, as we will see later, memory contention leads to what is known as *tree saturation*. When this occurs, a congestion tree with the switch connected to the hot memory module as the root and extending to all switches connected to the processors forms. The buffers of the switches in the congestion tree saturate and therefore block all further inputs. Tree saturation causes severe performance degradation. Thus, it is important to reduce the adverse effects of hot-spot contention.

We quantify the effects of hot-spot contention due to tree saturation in Section 2. In multiuser systems, hot-spot contention caused by one application can potentially affect the whole system if such a contention is not managed properly. We identify the objectives of a hot-spot management strategy and propose a taxonomy for classifying the strategies to reduce the effects of hot-spot contention. Our taxonomy categorizes strategies into avoidance-based, prevention-based, and detection-based methods. We then review several representative strategies for alleviating hot-spot contention. We describe three avoidance-based, three prevention-based, and two detection-based strategies. We compare these strategies and identify several issues that need further research.

The focus of hot-spot contention studies has been on shared-memory systems that use a multistage interconnection network. The interconnection networks used in distributed-memory systems are often point-to-point networks such as mesh or hypercube. Hot-spot contention can also occur in distributed-memory systems. To give the reader an idea of the hot-spot contention problem in distributed-memory systems, we comment briefly on this problem.

2. HOT-SPOT CONTENTION

We characterize hot-spot access behaviour along two dimensions: hot-spot data size and type of access. The first refers to whether the access is to single or multiple data items or variables. Examples of a single hot-spot variable include a shared lock variable for process synchronization, or a loop index variable in a parallel loop. An example of multiple hot-spot variables is the request to access a complete row in a matrix application as illustrated in Section 2.1.

The type of access refers to whether the access is read-only or update (i.e., read/write). Even though it is possible on multiprogrammed systems to cause contention for a memory module if several applications request access to the same memory module, we will not consider such inter-application memory contention. Next we look at an example to illustrate hot-spot contention behaviour.

2.1. An Example: Gaussian Elimination

We use Gaussian elimination as an example to explain two types of memory access behaviour that can result in hot-spot contention. The first type of access is the result of several processes attempting to read the same row (multiple-variables/read-only type) while the second type of access involves updating a synchronization variable (single-variable/update type).

As an example of an application that can cause hot-spot contention, we give the parallel implementation of Gaussian elimination. Gaussian elimination is a standard method for solving a system of linear equations $\mathbf{Ax} = \mathbf{b}$ where $\mathbf{A}$ is an $M \times M$ matrix, $\mathbf{b}$ is a $M \times 1$ vector, and $\mathbf{x}$ is a $M \times 1$ vector of unknowns. Gaussian elimination transform the given system of linear equations into the equivalent form $\mathbf{Ux} = \mathbf{c}$, where $\mathbf{U}$ is an $M \times M$ upper triangular matrix in which all elements below the diagonal are zero. The Gaussian elimination code for an $M \times M$ matrix is as follows [Bian93]:

```
for pivot ← 1 to M-1
   forall row ← pivot+1 to M
      temp ← A[row][pivot]/A[pivot][pivot]
      for col ← pivot to M
         A[row][col] ←A[row][col] - A[pivot][col] × temp
      end_for
   end_forall
end_for
```

On each iteration, the outer sequential **for** loop selects the pivot row and creates (M – `pivot`) processes using **forall**. Each such process eliminates the entries in a single row of the input matrix. For example, if M is 513, the first iteration of the outer sequential **for** loop creates 512 processes. Each such process requires almost concurrent access to the same pivot row as all processes begin execution at approximately the same time. As most languages use row-major ordering to store matrices, let us suppose that each row of the matrix is stored in a memory module. Then when pivot is 1, for instance, row 1 of matrix **A** is accessed by 512 processes and can potentially cause contention for the memory module that contains the row (i.e., multiple-variables/read-only type of hot-spot access).

Barrier Synchronization

As an example of a single-variable/update type of hot-spot access, we look at barrier synchronization that is required by a large number of parallel applications to synchronize computations. In our Gaussian elimination example, the outer sequential **for** loop starts the next iteration only when the previous iteration is completed. This means all processes created during the current iteration will have to complete their computation. For example, if M is 513, during the first iteration there are 512 processes in progress and the second iteration of the outer **for** loop will be initiated only when all 512 processes are done.

One way of achieving this synchronization is to keep a global shared variable and initialize it 512. Each time a process completes its computation (this process is said to have reached the barrier point), it decrements the synchronization variable (see Figure 2.1). When the value of the variable is zero, it indicates that all processes have completed the computation and the next iteration can be initiated. Clearly, mutually exclusive access is required for the synchronization variable. Also note that once a process has completed its computation, it decrements the synchronization variable and then busy waits on this variable (i.e., repeatedly reads the value of the synchronization variable until it becomes zero). This busy waiting causes additional contention but there are techniques that can be used to reduce this type of access contention. For instance, busy-waiting can be done on a local cached copy of the synchronization variable [Mell91]. Such techniques are outside the scope of this paper and will not be discussed further.

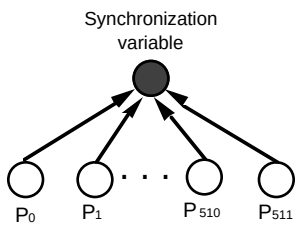


Figure 2.1 Hot-spot contention due to access to a synchronization variable by 512 processes

2.2. Memory Contention Due to Hot-Spot Access

What are the effects of hot-spot contention? To see this let us consider a system with N processors. Assume that a fraction f of these N processors (the hot processors) make requests to a hot memory module. The remaining $(1-f)N$ processors (i.e, the normal processors) issue normal memory requests that are uniformly distributed over the N memory modules. The hot processors issue memory requests at rate r_1 ($0 \le r_1 \le 1$) per network cycle, of which a fraction h are directed to a hot memory module. We assume that a processor continues to issue hot-spot requests even if there are pending ones. The normal processors issue memory requests at rate r_2 ($0 \le r_2 \le 1$) per network cycle. Thus the number of memory requests directed at the hot memory module R_{hot} is

$$R_{hot} = f N r_1 h + f r_1 (1-h) + (1-f) r_2 \quad \text{requests/network cycle}$$

The first term represents the hot-spot requests from the hot processors and the second term represents the normal requests directed at the hot-spot from the hot processors. The last term represents the normal requests directed at the hot-spot from the normal processors. Note that there are $(1-f)N$ normal processors with each normal processor generating memory requests at rate r_2, which are uniformly distributed over the N memory modules. The last term represents the fact that $1/N$ of these $(1-f)N\, r_2$ requests are targeted for the hot memory module.

Since the hot memory module can service only one request per network cycle (i.e., maximum $R_{hot} = 1$), we get the following by equating R_{hot} to 1:

$$r_1 = \frac{1-(1-f)\, r_2}{f\,(1+h\,(N-1))}$$

Since there are fN hot processors with a request rate of r_1 and the remaining processors with a request rate of r_2, the maximum bandwidth per processor b is

$$b = \frac{f\,N\,r_1 + (1-f)\,N\,r_2}{N} = \frac{1 + (1-f)\,h\,(N-1)\,r_2}{1 + h\,(N-1)} \tag{1}$$

We will continue this analysis later. For now, let us assume that all N processors are hot (i.e., $f = 1$). Then, the bandwidth per processor b reduces to

$$b = \frac{1}{1 + h\,(N-1)} \text{ requests/cycle} \tag{2}$$

The total network bandwidth B is

$$B = \frac{N}{1+h\,(N-1)} \text{ requests/cycle} \tag{3}$$

Figure 2.2 shows how the network bandwidth B is limited by hot-spot traffic as a function of system size N. For example, in a 1000-processor system, even a small fraction of traffic (i.e, h = 0.125%) directed towards the hot memory module can limit the total bandwidth to less than 50% of the bandwidth available when there is no hot-spot traffic (i.e., when h = 0%).

In practice, many applications may allow only one pending hot-spot request. In order to see how performance degrades due to hot-spot contention, we present data on the Gaussian elimination application in Table 2.1 [Bian93]. These data were obtained by simulation under the assumption that a cache line size of 64 bytes, a fixed network latency of 36 processor cycles, and a local memory latency of 10 processor cycles. A remote memory access cycle involves sending a request

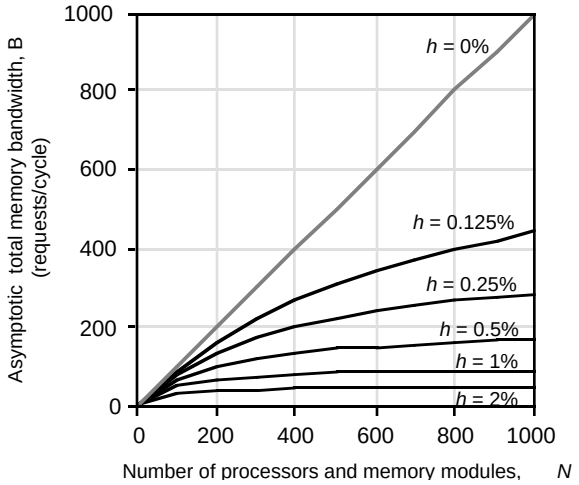


Figure 2.2 Impact of hot spot contention on total network bandwidth. Total bandwidth is calculated from Eq. 3. (from [Pfis85])

message, a local memory access, and receiving a reply message. Thus, if there is no memory contention, a remote memory access takes a total of 36+10+36 = 82 processor cycles.

Table 2.1 gives three statistics: number of remote memory accesses delayed by memory contention, the average latency of remote accesses, and the execution time. On a 50 processor system, 20% percent of all remote memory accesses suffer delays due to memory contention, which causes an increase in the average latency for remote memory accesses. The average remote memory latency increases to 164 as opposed to the minimum possible value of 82 processor cycles. If we consider a 200 processor system, the number of delayed remote memory accesses increases to 84%. The latency of remote memory accesses increases approximately by a factor of 10 when the number of processors increases from 50 to 200. The memory contention manifests itself in increased execution time as shown in Table 2.1. This application exhibits a *slowdown* by a factor of 2 when the system size increases from 50 to 200. Note that this data alone is not sufficient to show that the performance degradation is due to contention for memory. However, we will show in Section 3.1.1 that this indeed is the case.

Table 2.1 Effects of memory contention on the Gaussian elimination application for a 512x512 matrix (from [Bian93])

	50 processors	100 processors	200 processors
Percent of delayed remote memory accesses	20%	56%	84%
Average remote memory latency (in cycles)	164	572	1546
Execution time (in millions of cycles)	7.4	8.5	15.6

Tree saturation

In systems using multistage interconnection networks, hot-spot contention causes what is known as the *tree saturation* phenomenon (see Figure 2.3). When the hot memory module becomes the bottleneck due to hot-spot contention, the buffers in the *n*x*n* switch connected to it become full. (See the appendix for details on routing in the multistage interconnection network.) These full

buffers in turn cause the n buffers in the switches that feed this switch to be full which in turn causes the n^2 switches that feed these n switches to be full, and so on. Note that we have used 2x2 switches (i.e., $n = 2$) in Figure 2.3. Eventually, a tree of saturated switches results with the stage 0 switch connected to the hot memory module as the root as shown in Figure 2.3. The tree thus formed is called the *congestion tree*. Tree saturation degrades performance due to the following:

1. Since all switches connected to the processors are saturated, even the normal message traffic (i.e., non-hot spot traffic) gets affected by the hot-spot congestion. This effect leads to degraded performance of the application that is causing the hot-spot contention.

2. Even worse than the effect just described is that if the system is multiprogrammed, all other applications, even if their message traffic involves no hot-spots, are adversely affected due to hot-spot contention created by one application. We should keep this in mind even though our focus is on the effects of hot-spot contention on a single application.

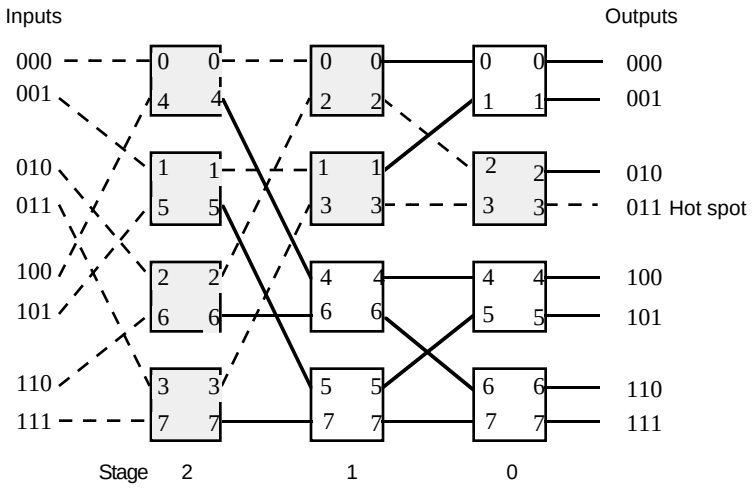


Figure 2.3 Tree saturation caused by hot-spot contention for the hot memory module 3 Saturated switches are shown shaded with the corresponding links shown with dashed lines. The congestion tree, rooted at output port 011, is shown by the dashed links.

Benefit of controlling tree saturation

Let us continue our analysis from Eq. (1) with the assumption that only a fraction f of N processors are hot. We know that, if there were no interference due to tree saturation, the normal processors can proceed with their memory requests. Thus, the normal processors can achieve the maximum rate r_2 of 1 per network cycle. Therefore, from Eq. (1) we obtain the bandwidth per processor with no tree saturation b_{nts} as

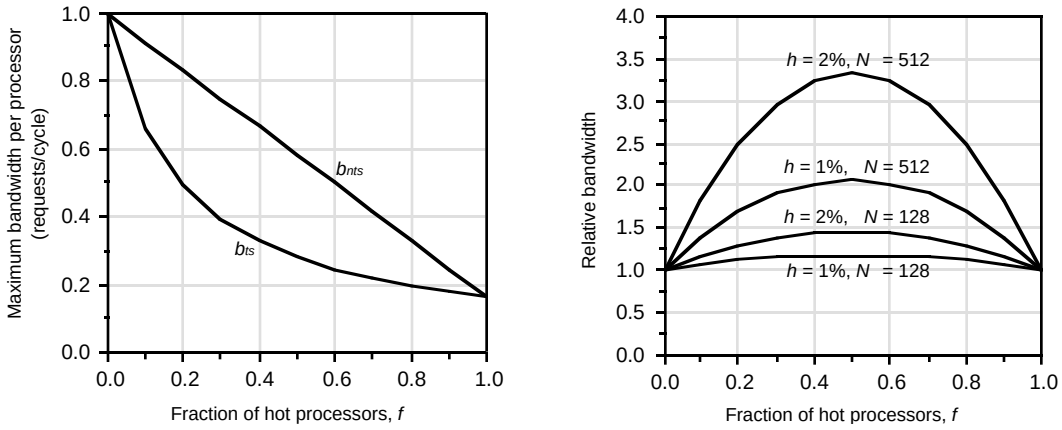
$$b_{nts} = \frac{1 + (1 - f)\ h\ \ (N - 1)}{1 + h\ \ (N - 1)} \tag{4}$$

The request rate of hot processors r_1 is, however, limited to

$$r_1 = \frac{1 - (1 - f)\ r_2}{f\ (1 + h\ \ (N - 1))} = \frac{1}{1 + h\ \ (N - 1)}$$

When there is tree saturation, the system behaves as if all, not just *fN*, processors are participating in the hot-spot activity. It has been conjectured in [Scot90] that the system behaviour under tree saturation can be modelled as if *all N* processors are generating hot-spot requests at a smaller fraction *fh*. Note that in reality only *fN* processors are generating hot-spot requests at a fraction *h*. This conjecture has been validated by simulation experiments [Scot90]. Then, the maximum bandwidth per processor b_{ts} is obtained by substituting *hf* for *h* in Eq. (2) as

$$b_{ts} = \frac{1}{1 + h\,f\,(N-1)} \tag{5}$$



(a) h = 1%, N = 512 **(b)** relative bandwidth = b_{nts}/b_{ts}

Figure 2.4 Relative per-processor bandwidths with and without tree saturation (from Eqs. (4) and (5))

Figure 2.4a shows the difference between the two bandwidths b_{ts} and b_{nts} for a 512-processor network with a hot-spot rate *h* of 1%. If the side effects of tree saturation are ignored, the bandwidth per processor decreases linearly with the fraction of hot processors *f*. The difference between the two curves reflects the impact of tree saturation on the normal traffic. This difference increases with the hot-spot rate *h* and system size *N* as shown in Figure 2.4b. We will next discuss several strategies to reduce the effects of hot-spot contention.

3. STRATEGIES TO REDUCE HOT-SPOT CONTENTION

From the previous discussion, we identify the following objective that any strategy devised to reduce hot-spot contention should meet.

> **Objective 1:** The primary objective of any strategy intended to reduce the effects of hot-spot contention should be to eliminate tree saturation. That is, while it is reasonable to expect hot-spot requests to suffer delays due to memory contention created by them, the normal non-hot-spot traffic should not be adversely affected due to the presence of hot-spots.

It is also desirable (but not necessary) to satisfy the following secondary objectives:

> **Objective 2:** Optionally, it is desirable to reduce the delays associated with hot-spot requests as well.

Objective 3: The strategy should not impose any performance penalty in the absence of hot-spots (i.e., during normal operation).

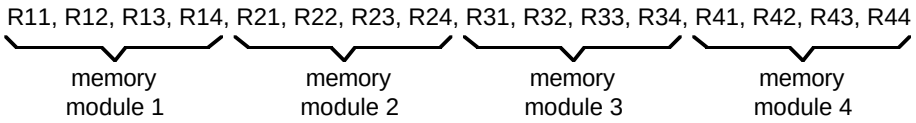
In this section we look at several strategies to handle hot-spot contention. We divide them into three classes: avoidance-based, prevention-based, and detection-based strategies. This classification is analogous to the schemes used for handling deadlocks [Islo80]. Avoidance-based strategies involve pre-planning so that tree saturation is effectively controlled. For example, this might involve a different way of distributing data so that hot-spots are not created in the first place. We discuss three avoidance-based strategies in Section 3.1. In prevention-based strategies, decisions are taken to control message traffic at run time so that tree saturation can be prevented. We present three such strategies in Section 3.2. Detection-based strategies may allow the formation of tree saturation but such contention is detected and congestion-control mechanisms are invoked to control contention. These strategies typically use some form of feedback and we discuss two feedback schemes in Section 3.3

3.1. Avoidance-Based Strategies

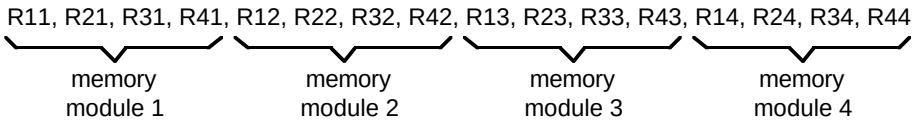
We discuss three avoidance-based strategies: the first two are software-based and the last one is hardware-oriented.

3.1.1. Data redistribution

To see how an application can distribute data to avoid hot-spot contention, we use the Gaussian elimination example discussed in Section 2.1. We have identified that contention for the pivot row creates memory contention and causes severe performance degradation. The reason that a row is mapped to a memory module is that most major languages, except Fortran, use row-major order for storing matrices. In row-major order, elements of a matrix are stored on a row-by-row basis. For example, assuming that we have four memory modules in the system, a 4X4 matrix **R** could be stored in the row-major order as follows:



To avoid contention for a row, we could distribute the elements rows to different memory modules. For example, we could store them in column-major order, in which the matrix is stored on a column-by-column basis as shown below:

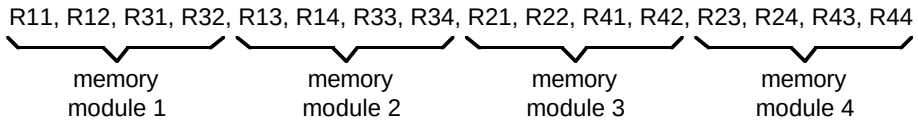


What guarantee is there that such a distribution eliminates contention? It could be that we have replaced the original hot-spot corresponding to a row by several hot-spots (equal to the number of elements in a row) distributed over all memory modules. The experimental data in [Bian93] indicates that we have indeed eliminated memory contention problem by the column-major order distribution. With column-major distribution, on a 50-processor system, the number of delayed remote memory accesses has reduced from 20% (given in Table 2.1) to about 1.5% and the remote memory latency from 164 to 83 processor cycles. Recall that the minimum remote memory latency is 82 cycles. However, the execution time of the application increases by fourfold from 7.4M

cycles to 30.2M cycles! While successfully avoiding memory contention we have introduced other problems that adversely impact the application performance even more.

The reason for the performance deterioration is that the column-major distribution introduces 15 times as many cache misses due to false sharing. To explain this phenomenon, let us assume that our cache block is two elements in size. Let us further assume that two processors - one working on row 1 and the other on row 2 - load their cache with elements R11 and R21 that constitute a cache block. Even though the two processors are working on two different elements, at cache block level, they appear to share the data. This is called false sharing and, in our example, causes unnecessary cache misses that result in the performance deterioration mentioned before.

To alleviate this problem, a block-column data allocation method has been introduced [Bian93]. In this method, each row is divided into cache blocks and the column-major allocation is done at the cache block level rather than at the element level. The block-column allocation method for a cache block size of 2 is shown below.



Notice that R11 and R12 are allocated to memory module 1, R13 and R14 are allocated to memory module 2, and so on. The performance of the block-column allocation is shown in Table 3.1 and indicates that the method is effective in handling both the cache miss and memory contention probl-ems. The average remote memory latency is 82 cycles, which is the minimum possible value. As opposed to the slowdown experienced due to memory contention (see Table 2.1), the application actually realizes a speedup of 2.6 when the number of processors increases from 50 to 200.

Table 3.1 Effect of block-column allocation in alleviating memory contention on the Gaussian elimination application (from [Bian93])

	50 processors	100 processors	200 processors
Percent of delayed remote memory accesses	0.16%	0.12%	0.27%
Average remote memory latency (in cycles)	82	82	82
Running time (in millions of cycles)	7.7	4.5	2.96

3.1.2. Software combining

Software combining has been proposed and studied in detail by Yew et al. [Yew87]. To illustrate the principle, we use the barrier synchronization example discussed in Section 2.1. With software combining, a tree of data items is created as shown in Figure 3.1. The original hot-spot acts as the root of the tree. If N=512 and assuming a branching factor of the tree K of 8, there are 73 data items each initialized with a value of 8. These 73 data items are distributed as follows: 64 at level 2, 8 at level 1 and 1 at level 0. The nodes are partitioned into 64 groups of eight, with each group sharing one of the data items corresponding to the leaves of the tree. When the last node in each group decrements its data item to zero, this node then decrements its parent data item. This process is repeated until the data item corresponding to the root of the tree is finally decremented to zero. Software combining reduces hot-spot contention in this example because there are 73 data items

each being accessed by only 8 nodes instead of one data item being accessed by 512 nodes (compare Figures 2.1 and 3.1).

We should note that if we map all the combining tree data items to the same memory module, even though we eliminate contention for the original data item, contention for the memory module still exists at the same level! What we have to do in order to avoid memory module level contention is to distribute the tree data items over the *N* memory modules. We can see that the combining tree creates the maximum number of tree data items when the branching factor *K* is 2. When *K* = 2, the number of tree data items, including the original hot-spot, is equal to *N*–1, all of which can be distributed one data item per memory module as we have *N* memory modules. This successfully eliminates the module level contention mentioned above.

As shown in Figure 2.1, in the original scheme, concurrent access of the hot-spot data item by 512 processors causes contention. The software combining tree creates several intermediate data items to reduce contention but increases the number of messages compared to the original scheme. For example, the original scheme in Figure 2.1 requires only 512 messages to update the synchronization variable. In contrast, the combining tree in Figure 3.1 requires 512 messages to update the level 1 data items, 64 (=512/8) messages to update the level 2 data items, and 8 (= $512/8^2$) messages to update the original data item at level 0. Thus, the combining tree requires a total of 584 messages.

In general, assuming that a hot-spot rate of rh from each processor as in Section 2.2, the increase in hot-spot traffic is given by [Yew87]

$$\sum_{i=1}^{\log_K N - 1} \frac{rh}{K^i} = rh\left(\frac{1 - K/N}{K - 1}\right)$$

where *K* is the branching factor of the combining tree. Note that the maximum increase occurs when *K* = 2 and this is equal to $rh\,(1-2/N)$. For large *N*, increased hot-spot traffic due to the use of software combining tree is upper bounded by *rh* (i.e., the hot-spot traffic at most doubles due to the combining tree). Despite this increase in hot-spot traffic, the simulation results in [Yew87] indicate that the software combining can successfully prevent tree saturation and provide performance improvements. It has been shown that software combining tree effectively relieves the normal traffic from the adverse effects of tree saturation. In addition, the latency associated with hot-spot traffic also decreases.

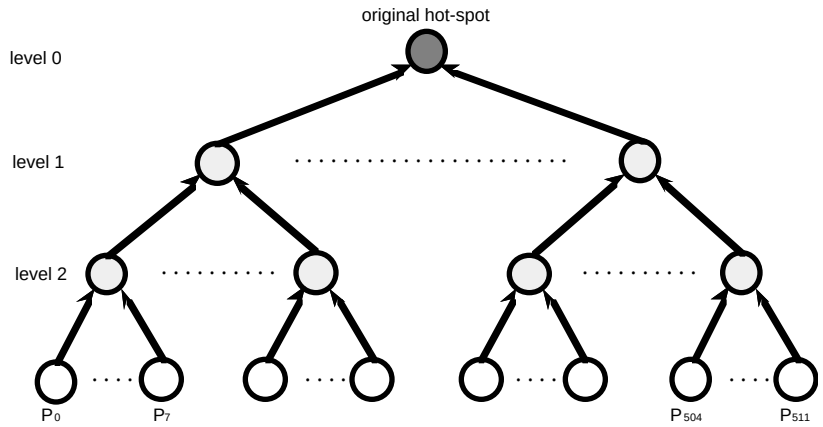


Figure 3.1 A software combining tree with a branching factor *K* of 8. Original hot spot data item is shown shaded dark and the intermediate data items are shown lightly shaded. (from [Yew87])

3.1.3. Using Multipath Networks

Another way to avoid hot-spot contention is to add resources to the interconnection network. Specifically, the interconnection network shown in the appendix has only a single path between an input-output pair. Because of this unique path, normal message traffic gets affected due to hot-spot contention. The basic idea of this scheme is that, if we provide multiple paths, regular message traffic can be diverted around the hot-spot area. Figure 3.2 shows how a multipath interconnection network can be derived by adding an extra stage to the input side of the network along with multiplexers and demultiplexers at the input and output stages, respectively [Wang95]. In addition, dual I/O links to and from the processors and memory modules are required. This modified network is called the extra stage cube network for obvious reasons.

The input (extra) stage can be dynamically enabled or disabled by using the multiplexers as shown in Figures 3.2(b) and (c). Similarly, the demultiplexers can be used to enable or disable the output stage (see Figures 3.2(d) and (e)). Note that the original network shown in Figure A1 results if we disable the input stage and enable the output stage. However, enabling both input and output stage provides two disjoint paths between any input and output pair.

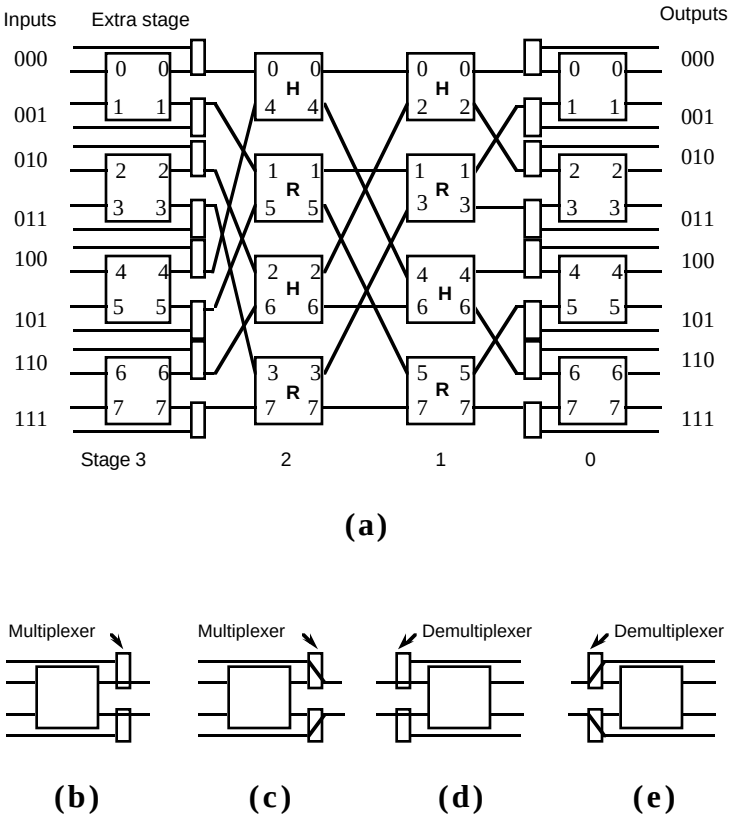


Figure 3.2 The extra stage cube network with 8 inputs and outputs using 2X2 switches (b) input stage enabled (c) input stage disabled (d) output stage enabled (e) output stage disabled (from [Wang95])

The basic idea in using this type of network for reducing hot-spot contention is to separate the hot-spot and normal traffic, form each source, by using one dedicated path for each type of traffic. For example, when there is a steady state hot-spot and regular traffic, we can set the switches in the extra stage to direct all hot-spot traffic to upper outputs and the normal traffic to lower outputs

of this stage. Then only switches labelled **H** in Figure 3.2 would handle the hot-spot traffic and the switches labelled **R** would handle the normal traffic. Such a separation in routing can be done in a distributed manner. For details on this and other improved routing strategies, see [Wang95]. The simulation results presented in [Wang95] show that this type of network is able to control tree saturation and reduce the delay of memory request that are not directed to the hot memory module. This strategy, however, assumes that each memory request can be categorized as either a hot-spot request or a normal request.

3.2. Prevention-Based Strategies

We discuss three prevention-based strategies. The first and the last strategies require hardware modifications while the second strategy can be implemented either in hardware or in software.

3.2.1. Hardware combining

As in software combining, this strategy uses combining as the basic means of preventing tree saturation. However, as its name implies, combining is accomplished in hardware by enhancing the switch to support combining. Hardware combining is used in the Ultracomputer and IBM RP3 systems. Figure 3.3 shows the network switch used in the Ultracomputer that can perform combining [Alma94].

Hardware combining works as follows. At each switch, memory request messages directed at the same memory locations are combined into a single message. A record of the messages combined is kept in a wait buffer in each switch. When the reply to a combined message reaches the switch that performed combining, the switch generates multiple replies to all combined messages using the information in its wait buffer. Since combined messages can themselves be combined, the generation of multiple replies has an effect similar to that in the software combining. The combining scheme of the Ultracomputer and IBM RP3 systems is called *pairwise* combining as two messages are combined into one.

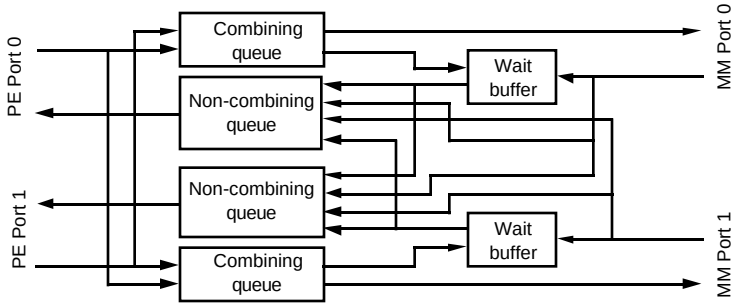


Figure 3.3 The Ultracomputer switch incorporating combining capability [from [Alma94])

Hardware combining involves increased cost for enhancing the network to support combining. It has been estimated that such an enhancement would increase the size/cost of the switches in the RP3 by a factor of at least 6 [Pfis85]. The estimate for the Ultracomputer is much more modest: the pin count increases by 50% and the number of gates double to enhance the switch to support combining [Alma94].

Simulation results in [Pfis85] suggest that pairwise combining can effectively eliminate tree saturation in small networks (say, up to 64-input networks). However, other studies have shown that pairwise combining cannot eliminate tree saturation in large networks (for example, in a 1024-

input network) [Lee94]. The problem with pairwise combining is that two messages will be combined only if they arrive at a switch at the same time. However, the probability of such arrivals occurring is low. As an alternative, Lee et al. [Lee94] propose k-way combining. In this scheme, a total of up to *k* messages can be combined. This k-way combining can be achieved by combining an arriving message, if it can be combined, with other messages waiting in the queue at the switch. Obviously the larger the *k* value the more complex the combining network and more contention for the wait buffers. The simulation results in [Lee94] suggest that 3-way combining is sufficient for reasonably large systems that use 2X2 switches.

From performance point of view, software combining tends to perform better than hardware combining. The main reason is that software combining can be tailored to achieve a higher degree of combining. For example, in Figure 3.1, software combining achieves 8-way combining. Even from the implementation point of view, software combining is more flexible in implementing k-way combining for large k values. Hardware combining is relatively more rigid and implementations in systems such as Ultracomputer and IBM RP3 have used only 2-way combining.

3.2.2. Discard strategy

The discard strategy [Ho89] prevents tree saturation by forwarding only one request and discarding all others when more than one message are destined to the same memory module. In hardware combining, such messages would have been combined into a single message. If we focus on a network with 2X2 switches, when two packets arrive at a switch contending for the same output port, one randomly selected message is discarded and the other is forwarded to the output buffer. Of course, the discarded message has to be reinitiated. This reinitiation can be done either by the switch (switch-initiated) or by the source processor (processor-initiated). In the switch-initiated strategy, the switch which discarded the packet would have to notify the source processor such that the processor can reinitiate the request. This strategy, therefore, requires hardware modifications to the network. The processor-initiated strategy, on the other hand, uses a time-out mechanism. If the processor has not received a reply within a predetermined time-out period, it reinitiates the request which is assumed to have been discarded by a switch. In this case it is important to select an appropriate time-out period, which is a function of network load. Too small a time-out period may unnecessarily reinitiate duplicate requests and too large a value can increase the memory latency.

The simulation results in [Ho89] used the switch-initiated strategy. These results show the discard strategy to be useful in eliminating the adverse effects of hot-spot contention on the normal traffic. Latency of normal memory requests under the discard strategy is similar to the pairwise hardware combining scheme. However, since the discard strategy looks at requests going to the same memory module, this strategy works for multiple hot-spots which are mapped to the same memory module whereas hardware combining is suitable for a single hot-spot.

3.2.3. Limiting requests

Another way of preventing tree saturation is to balance the request generation rate to the service rate of the memory modules. That is, during each network cycle, limit the number of requests directed to each memory module entering the network to that serviced by the memory modules [Scot90]. Requests that cannot enter during a particular network cycle are blocked until a later cycle.

To implement this scheme, we need an arbiter that takes *N* inputs from the *N* processors and decides which requests are to be blocked, if any, for a later submission during each network cycle. Thus, the arbiter should be capable of performing *N* arbitrations -- one to each memory module --

during each network cycle. The hardware cost makes such a scheme not suitable for large systems. If we limit the number of memory modules to a single memory module that is identified as the hot module, we can reduce the number arbitrations from N to 1.

This scheme suffers from two basic problems.

- It may unnecessarily restrict the available bandwidth when there is no hot-spot traffic in the system. This is because two normal requests issued during the same network cycle are constrained to enter the network one at a time.
- Since this scheme depends on a global arbiter, it is not scalable to large systems. Furthermore, the arbiter may be too costly to implement for large systems even for a single hot memory module.

Simulation results in [Scot90] show that limiting is beneficial when the hot-spot rate is high. At low hot-spot rate, the limiting scheme restricts the available bandwidth as pointed out before.

3.3. Detection-Based Strategies

We discuss two detection-based strategies that use feedback to activate a congestion-control mechanism.

3.2.1. Threshold-based feedback

The feedback scheme proposed in [Scot90] monitors the size of the queue at each memory module (i.e., at the output of the network). If the size is at or above threshold T_h, we assume that the memory module is hot and notify the processors. The processors respond by holding back requests to the hot module. When the queue size falls below threshold T_l ($\leq T_h$), the module is considered normal and processors can resume requests to the memory module. The use of two thresholds introduces hysteresis for the feedback system. This scheme is referred to as the *feedback* scheme.

The original proposal uses $T_l = T_h = T$ and used a damping mechanism similar to the limiting discussed before to reduce overshoot, undershoot and oscillation problems associated with the feedback scheme. When a memory module is detected as hot, the limiting mechanism is invoked. The limiting then causes only one hot memory request to enter the network from all N processors as discussed before. This scheme is referred to as the *feedback with limit-damping* scheme. The main difference between this scheme and the feedback scheme is that the processors under the feedback scheme stop submitting requests to the hot module while it is hot whereas the feedback with limit-damping scheme allows submitting at most one hot memory request per cycle. This feedback mechanism, therefore, eliminates the bandwidth restrictions imposed by the conservative limiting scheme discussed earlier.

Simulations on a 256-node network that uses 2X2 switches with four output buffers indicate that the feedback scheme by itself performs worse than than the limiting scheme if we are interested in the maximum bandwidth obtainable for high hot-spot rates. The main reason for this relative performance is that the feedback scheme is sensitive to the threshold value used. The threshold value is limited by the number of output buffers available at a switch. However, if the threshold is less than the number of stages in the network, the hot module may become temporarily "idle" for sometime.

We now illustrate why the hot memory module becomes idle with an example. Assume that the threshold T is set at four. A 256-node network that uses 2X2 switches will have $\log_2 256 = 8$

stages. In this case, a memory module is considered hot if the number of queued requests is four. Once a module is detected hot, the feedback scheme will not allow any processor to submit a request to that memory module. When the number of queued requests at the hot module goes down to three, processors will resume requests to this module. However, these newly submitted requests take 8 cycles to reach the memory module, which leaves the memory module idle for up to 5 cycles. Feedback with limit-damping, which eliminates the drawback discussed, performs much better than the limiting scheme.

A problem with this scheme is that a finite delay exists between the time a memory module becomes hot and the time the processors stop submitting requests to the hot module. Furthermore, there may be several memory requests already in transit. These factors may cause a temporary tree saturation. The next scheme avoids this drawback by using buffer occupancy of the switches close to the network input for early detection of congestion caused by hot-spots.

3.2.2. Buffer occupancy-based feedback

This scheme uses buffer occupancy to detect hot-spot contention before the tree saturation occurs [Liu95]. Liu et al. have observed through simulations that, when a congestion tree is being formed, the buffer occupancy of switches in the congestion tree is sharply different from those of their adjacent switches that are not part of the congestion tree in a very short period of time. Thus, the difference in buffer occupancies in selected adjacent switches can be used to detect congestion early. Note that any switch that is part of the congestion tree is also connected to a switch that is not part of the congestion tree through one of its output ports (e.g., see Figure 2.3). To implement this scheme, hardware modifications to the switch are required. Liu et al. suggest the use of occupancy monitors embedded into switches of selected stages. This strategy also needs additional wires for feedback purposes. Once the congestion is detected, the processors are notified of the switches involved such that the processors can block memory requests that use the congested switches.

The question now is: How do we know when the congestion is stopped such that the processors can resume the blocked requests? The problem is when the congestion-control mechanism is activated, the difference in buffer occupancy of adjacent switches is not as dramatic as before the activation of the congestion-control mechanism. Thus, during the time when congestion-control mechanisms are activated, the individual switch buffer occupancy threshold is used for congestion control. Finally, to avoid problems such as oscillation associated with the feedback system, the hot-spot is deemed to have ceased if no congestion is detected for a given period of time.

Simulations performed on a 512-processor network show that both the feedback schemes discussed behave similarly when the hot-spot rate is low. However, for high hot-spot rates, the buffer occupancy-based feedback scheme performs significantly better than the threshold-based feedback scheme. The reason is that the threshold-based feedback scheme is relatively slow in detecting hot spots. As a result, this scheme allows a temporary tree saturation. The buffer occupancy-based feedback scheme, on the other hand, detects the hot-spot contention prior to the occurrence of tree saturation.

4. DISCUSSION

We have discussed eight strategies to handle the hot-spot contention problem. A comparison of these strategies is given in Table 3.2. Most strategies except the first two need some hardware implementation requiring enhancements to the interconnection network. The discard strategy can be implemented either in hardware or in software depending on the reinitiation strategy selection.

Most strategies proposed can handle multiple hot-spots except the two that are based on the combining principle.

A precise quantitative cost comparison of these schemes is difficult because some strategies require a software implementation while others require hardware modifications to the network. For this reason, we do not give the relative costs of the three avoidance-based strategies. For the remaining two types of strategies, we provide a qualitative cost comparison. Of the three prevention-based strategies, discard is the least expensive strategy. Both hardware combining and limiting requests are more expensive strategies. In hardware combining, each switch has to have wait buffers, combining logic, and reply generation logic. Implementing the limiting requests strategy requires an arbiter that can perform N arbitrations during each network cycle, where N is the number of memory modules. Among the two detection-based strategies, the buffer occupancy-based feedback scheme is more expensive as occupancy monitors are required.

Since the primary objective of these strategies is the elimination of tree saturation, it is implied that all of them satisfy this requirement. For this reason, we have not included this objective in Table 3.2. However, we should note that hardware combining, if restricted to pairwise combining, may not eliminate tree saturation in large systems. For large systems, k-way combining may be required. For reasonably large systems, it has been suggested that 3-way combining is sufficient [Lee94]. The threshold-based feedback strategy may also cause a temporary tree saturation as explained in Section 3.2.1.

We now look at whether these strategies attempt to improve latencies of hot-spot requests as stated in our second objective. Since data redistribution avoids hot-spots at the application level, hot-spot requests are not generated in this strategy. The two combining-based strategies tend to improve the latencies of hot-spot requests. The other strategies make no attempt to improve hot-spot latency. If such an improvement is obtained in these strategies, it is purely a side effect. For this reason, we give “no” for the corresponding entries in Table 3.2. However, since memory contention is effectively controlled by these strategies, they do provide improvements in hot-spot latencies for the case of mixed hot-spot and normal requests.

Finally, we compare these eight hot-spot management strategies based on whether or not they induce overhead in normal operation. That is, when the system does not have hot-spots, we desire that there be no penalty for incorporating a specific hot-spot control mechanism as stated in our third objective. In the data redistribution strategy, it is application dependent. As we have demonstrated in Section 3.1.1, a simple element-based column-wise distribution of data eliminates the contention problem. But normal behaviour of the system changes so much that performance deteriorates substantially. The block-column method improves the situation but it marginally increases run time for a 50-processor system (compare Tables 2.1 and 3.1).

In software combining, we introduce the combining tree only for hot-spots. For the multipath network, we can obtain the original network by enabling/disabling the input and output stages as discussed. Therefore, for both these strategies, there is no overhead in the absence of hot-spots.

In hardware combining, there is an additional delay introduced due to switch enhancements to support combining. Thus, even normal requests suffer a small amount of delay. However, this can be minimized by using one regular network for the normal traffic and a separate combining network for hot-spot traffic.

The discard strategy introduces an additional delay. If two requests are contending for the same output port, the conflict must be resolved. In the limiting requests strategy, the arbiter restricts the

two normal requests directed at the same memory module to enter the network one at a time. This strategy unnecessarily restricts the available bandwidth due to the overly cautious approach. In both feedback strategies, if the feedback indicates no contention, the normal system behaviour is not affected.

Table 3.2 Comparison of hot-spot management strategies

Strategy	Implementation needs modification in	Single or multiple data items accessed	Accomplishes objective 2	Accomplishes objective 3
Avoidance-based strategies				
Data redistribution	software	multiple	not applicable	application dependent
Software combining	software	single	yes	yes
Multipath networks	hardware	multiple	no	yes
Prevention-based strategies				
Hardware combining	hardware	single	yes	no
Discard	hardware/software	multiple	no	no
Limiting requests	hardware	multiple	no	no
Detection-based strategies				
Threshold-based feedback	hardware	multiple	no	yes
Buffer occupancy-based feedback	hardware	multiple	no	yes

Hot-spot contention in distributed-memory systems

Hot-spot contention can also occur in distributed-memory multicomputer systems. For example, in barrier synchronization, the processor coordinating the synchronization activity may become a "hot-spot". As in shared-memory architectures, hot-spot contention has been shown to have an impact on both normal as well as hot-spot traffic in distributed-memory systems [Dand92,Dand91]. Studies have shown that both hardware and software combining can be used to alleviate hot-spot contention [Dand89,Dand91].

With an unbounded message buffering capacity at each processor, both combining schemes tend to provide similar performance improvements and completely eliminate the adverse effects of hot-spot contention on both the normal and hot-spot messages.

With bounded message buffering capacity, hardware combining could eliminate only the influence of hot-spot contention on normal messages. Hot-spot messages experience performance degradation whether or not hardware combining is used. On the other hand, software combining is more effective than hardware combining. Although normal messages experience similar delays with both hardware and software combining, only the software combining scheme provides substantial improvements in the throughput capacity and in the delays associated with hot-spot messages.

Further research

The focus of hot-spot contention studies has been on shared-memory systems that use a multistage interconnection network. Except for a few of studies [Abra89,Dand92,Dand91], hot-spot contention has not received as much attention in distributed-memory multicomputer systems. Since

several large parallel systems (e.g., 9000+ node Pentium Pro-based Teraflop system) from Intel and nCUBE are based on the distributed-memory architecture, it is important to study the hot-spot contention problems in these type of systems.

In addition, the concentration of previous studies has been mostly on managing hot-spot contention caused by a single application in shared-memory systems. However, in a multiprogrammed system, it is possible that each application may not cause any contention. But these applications collectively may cause memory contention problems similar to the hot-spot contention we have discussed. All the techniques proposed for the traditional hot-spots might not apply to this problem and further study is needed.

It is unlikely that large systems are based purely on a shared-memory architecture. There is a trend towards a hybrid architecture for building large parallel systems. For example, shared-memory clusters can be interconnected as in the distributed-memory architecture. The hot-spot contention problem needs to be studied in such architectures.

5. CONCLUDING REMARKS

As the system size increases, hot-spot contention becomes a more serious problem affecting the performance of the whole system. In particular, as parallel systems evolve toward general-purpose, multiprogrammed systems, it is imperative that systems effectively manage system bottlenecks such as hot-spots. Otherwise, inter-application dependence can result in unpredictable execution times for individual applications. For example, if hot-spots are not managed properly, a single application that creates a hot-spot can adversely affect the execution times of all other applications running on a multiuser system. This type of interference among applications is clearly undesirable. Some of the strategies or a combination of them can be used to reduce the adverse effects of hot-spots in large parallel systems.

ACKNOWLEDGEMENTS

I am grateful to the financial support provided by NSERC and Carleton University in carrying out this research. The analysis presented in Section 2.2 is largely based on [Scot90].

REFERENCES

[Abra89] S. Abraham and K. Padmanabhan, "Performance of the Direct Binary n-Cube Network for Multiprocessors," IEEE Trans. Computers, Vol. C-38, No. 7, July 1989, pp. 1000-1011.

[Alma94] G. S. Almasi and A. Gottlieb, *Highly Parallel Computing*, Second Edition, Benjamin/Cummings, Redwood City, CA, 1994.

[Bian93] R. Bianchini, M. E. Crovella, L. Kontothanassis, and T. LeBlanc, *Alleviating Memory Contention in Matrix Computations on Large-Scale Shared-Memory Multiprocessors*, Technical Report 449, Computer Science Department, University of Rochester, Rochester, New York 14627. (Available from http://www.cs.rochester.edu/trs/systems-trs.html).

[Dand89] S. P. Dandamudi and D. L. Eager, "The Effectiveness of Combining in Reducing Hot-Spot Contention in Hypercube Multicomputers," *Int. Conf Parallel Processing* , St. Charles, Illinois, Vol. I, pp. 291-295.

[Dand91] S. P. Dandamudi, *Hierarchical Hypercube Multicomputer Interconnection Networks*, Ellis Horwood, Chichester, England, 1991.

[Dand92] S. P. Dandamudi and D. L. Eager, "Hot Spot Contention in Binary Hypercube Networks," *IEEE Trans. Computers*, Vol. C-41, No. 2, February 1992, pp. 239-244.

[Ho89] W. S. Ho and D. L. Eager, "A Novel Strategy for Controlling Hot Spot Contention," *Int. Conf. Parallel Processing,* 1989, Vol. I, pp. 14-18.

[Islo80] S. S. Isloor and T. A. Marsland, "The Deadlock Problem: An Overview," Computer, September 1980, pp. 58-77.

[Lee94] G. Lee, C. P. Kruskal, and D. J. Kuck, "On the Effectiveness of Combining in Resolving Hot Spot Contention," J. *Parallel and Distributed Computing,* Vol. 20, 1994, pp. 136-144.

[Liu95] J. C. Liu, K. G. Shin, and C. C. Chang, "Prevention of Congestion in Packet-switched Multistage Interconnection Networks," *IEEE Trans. Parallel and Distributed Systems,* Viol. 6, No. 5, May 1995, pp. 535-541.

[Mell91] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors," *ACM Trans. Comput. Syst.*, vol. 9, no. 1, February 1991, pp. 21-65.

[Pfis85] G. F. Pfister and V. A Norton, "Hot-Spot Contention and Combining in Multistage Interconnection Networks," *IEEE Trans. Computers*, Vol. C-34, No. 10, October 1985, pp. 943-948.

[Scot90] S. L. Scott and S. S. Sohi, "The Use of Feedback in Multiprocessors and Its Application to Tree Saturation Control ," *IEEE Trans. Parallel and Distributed Systems,* Viol. 1, No. 4, October 1990, pp. 385-398.

[Sieg90] H. J. Siegel, *Interconnection Networks for Large-Scale Parallel Processing,* Second Edition, McGraw-Hill, New York, 1990.

[van96] A. J. van der Steen and J. J. Dongarra, "Overview of Recent Supercomputers," *NHSC Review,* 1996 Volume, First Issue, July 1996 (This paper can be obtained from the URL: `http://www.crpc.rice.edu/NHSEreview`).

[Wang95] M. C. Wang, H. J. Seigel, M. A. Nicols, S. Abraham, "Using Multipath Network for Reducing the Effects of Hot Spots," *IEEE Trans. Parallel and Distributed Systems,* Viol. 6, No. 3, March 1995, pp. 252-268.

[Yew87] P. C. Yew, N. F. Tzeng, and D. H. Lawrie, "Distributing Hot-Spot Addressing in Large-Scale Multiprocessors," *IEEE Trans. Computers*, Vol. C-36, No. 4, April 1987, pp. 388-395.

APPENDIX -- MULTISTAGE INTERCONNECTION NETWORKS

Multistage interconnection networks (MINs) use *n*x*n* switches consisting of *n* inputs and *n* outputs. Here we focus on 2x2 switches. A typical 2x2 switch, shown in Figure A1(a), is a crossbar with output message buffers. A 2x2 switch can be in one of two positions -- straight-through or exchange -- as shown in Figure A1(b). The switch settings are controlled by two control input bits. If the control bit of an input is 0, the input is connected to the upper output port. If the control bit is 1, the input is connected to the lower output port. Thus a switch is set in the straight-through position if the upper input has a control bit 0 and the lower input 1. The switch is set in the exchange position if the upper input control bit is 1 and the lower input 0. If both inputs have either 0 or 1 as their control input, a conflict will arise.

We can construct a N-input MIN by using $\log_2 N$ stages, with each stage consisting of $N/2$ switches. Figure A1 shows the indirect cube MIN for N=8 inputs. Output ports of switches have message buffers as shown in Figure A1(a). We, however, do not show these buffers explicitly in an interconnection network for the sake of clarity. The inter-stage connections are governed by the following rule: the two inputs of a switch in stage i differ by 2^i. For example, the top switch in stage 0 has inputs 0 and 1, in stage 1 that inputs are 0 and 2, and in stage 2 the inputs are 0 and 4. Typically, the inputs are connected to processors and the outputs to memory modules.

Message routing in a MIN can be done in a distributed manner by using the destination address. If we express the destination address as a binary number, we can use the *i*th bit as the control bit to set the switch in stage *i*. Figure A1(c) shows routing of a message from input 5 to output 1. Expressing 1 in binary gives a 001. So we will use the most significant bit 0 as the control bit in stage 2. Thus, the lower input of the second switch in stage 2 is connected to upper output as shown. Similarly, the middle 0 bit is used to control the switch in the middle stage and the least significant bit 1 is used to set the top switch in stage 0. For more details on MINs, see [Sieg90, Alma94].

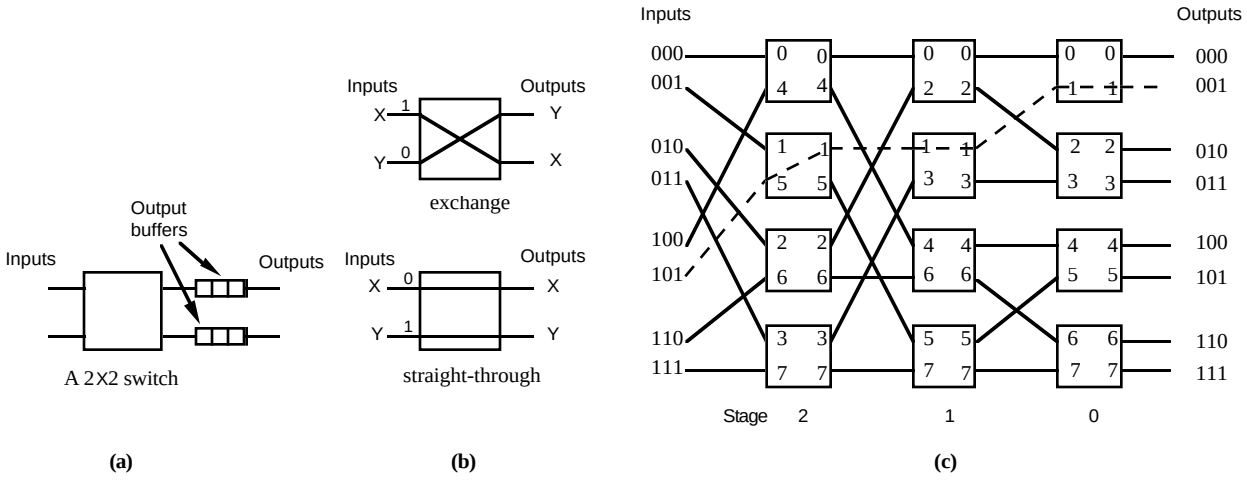


Figure A1 Multistage indirect cube network