Quantifying and Resolving Remote Memory Access Contention on Hardware DSM Multiprocessors*

Dimitrios S. Nikolopoulos
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
1308 West Main Street Urbana, IL 61801, U.S.A.
dsn@csrd.uiuc.edu

Abstract

This paper makes the following contributions: It proposes a new methodology for quantifying remote memory access contention on hardware DSM multiprocessors. The most valuable aspect of this methodology is that it assesses the impact of contention on real parallel programs running on real hardware. The methodology uses as input the number of accesses from each DSM node to each page in memory. A trace of the memory accesses of the program obtained at runtime from hardware counters is used to compute an accurate estimate of the fraction of execution time wasted due to contention. The paper presents also a new algorithm which detects potential hot spots in pages and resolves contention on them using dynamic page migration. The algorithm balances the remote memory accesses across the nodes of the system, while trying to improve memory access locality. Experiments with five parallel codes with irregular memory access patterns on a 128-processor Origin2000 show that our algorithm yields respectable reductions of execution time, averaging 27.7%.

1. Introduction

It is well known that contention is one of the factors that limits the performance of parallel programs. Contention stems from simultaneous accesses of multiple processors to critical resources such as memory banks and network links. On cache-coherent distributed shared memory (DSM) multiprocessors, one of the most intrusive forms of contention is the one that occurs at the network interface of a node, due to simultaneous requests for remote memory accesses, or other coherence protocol transactions directed to that node[1].

Although contention must be accounted for when attempting to scale parallel programs on DSM multiprocessors, the hardware of these systems lacks the means to quantify directly the impact of contention on the execution time of a program. The hardware counters of modern microprocessors do not reveal any information about contention. Carefully designed microbenchmarks [2] can measure the impact of contention on a single access to the memory hierarchy of a multiprocessor, however this measurement reflects the raw performance of the memory hierarchy and can not be used to estimate the fraction of time that a parallel program spends due to contention. Useful information about contention can be collected by performance monitoring firmware running at the network interface [10]. This solution is appealing, but not portable, because it requires programmable network interfaces. Commercial hardware DSM systems do not offer this option. As a consequence, performance studies that investigate the scalability of these systems have to speculate on the impact of contention by factoring out the impact of all other parameters that might affect performance [7].

In this paper, we present a new methodology for quantifying remote memory access contention on hardware cache-coherent DSM multiprocessors. This methodology estimates the fraction of execution time wasted due to contention, using as input the number of accesses from each node to each page in memory during the execution of the program. This information is collected conveniently in hardware page reference counters, which are available in commercial DSM systems such as the SGI Origin2000 [9] and the Sun Wildfire [6]. The most valuable aspect of this

*This work is supported by the National Science Foundation grant No. EIA-99-75019. The experiments were conducted on the NCSA Origin2000 at the University of Illinois at Urbana-Champaign.

[1]We use the term *node* to signify the basic building block of a hardware DSM multiprocessor. A node may have one or more processors. For the purposes of this study, we consider the accumulated remote memory transactions issued by the processors on the same node as a single set of remote memory accesses.

Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'02)
1530-2075/02 $17.00 © 2002 IEEE

IEEE COMPUTER SOCIETY

methodology is that it is able to assess the impact of contention on real parallel applications running on real hardware. The paper presents also a new algorithm that detects and resolves remote memory access contention, using dynamic page migration. Two important novelties of the algorithm are its ability to improve memory access locality while resolving contention and its transparency, since the algorithm does not require modifications to the programs.

Validation of our methodology under extreme conditions shows that we are able to predict the impact of remote memory access contention with an accuracy of 5.4%. Not surprisingly, our experiments with real parallel applications indicate that contention has a significant impact on performance, particularly in programs with irregular memory access patterns[2]. Our algorithm is able to alleviate contention and reduce the parallel execution time of six application benchmarks by 19–34%.

The rest of this paper is organized as follows. Section 2 presents our methodology for quantifying contention and its validation. Section 3 presents our contention resolution algorithm. Section 4 provides experimental evidence on the performance of the algorithm. Section 5 discusses related work and Section 6 concludes the paper.

2. Quantifying Contention

2.1. Outline

The starting point for estimating the impact of contention on a parallel program is to estimate the effect of contention on the latency of a single memory access to the target DSM's memory hierarchy. This estimate is obtained by microbenchmarking the target DSM and is measured for varying degrees of contention. The second step is to collect $N + 1$ histograms, where N is the number of nodes in the DSM. One of these histograms contains the number of memory accesses issued to each node, broken down into local and remote accesses. The other N histograms, one per node, contain the number of remote memory accesses issued to that particular node from each of the other nodes. These histograms are collected at runtime and plot the memory access pattern of the program throughout its lifetime. Given these histograms and an estimate of the latency of a single memory access under contention, we compute an estimate of the total memory access latency per node and the maximum memory access latency among all nodes that participate in the execution of the program. To isolate the impact of contention, we compare this latency against the latency of an ideal "contention-free,, version of the program, in which all memory accesses are assumed to be serviced without contention.

[2]In this context, the term *irregular* refers to the fact that the memory accesses of the programs are not balanced across the nodes of the DSM.

2.2. Estimating Contention on a Single Memory Access

We use the methodology proposed by Hristea et.al. [2], to estimate the impact of contention on a single memory access. We use a multithreaded microbenchmark, in which each thread alternates between a computing phase and a memory access phase. During the computing phase, each thread performs random computations using data available in registers and caches. During the memory access phase, each thread retrieves data from memory. One of the threads of the microbenchmark is designated as the master thread. During the memory access phase, the master generates memory references to a single memory module, to sustain a constant utilization of the memory bandwidth. The other threads serve as noise generators and issue memory accesses to the same memory module, at a rate which varies according to the degree of contention that the user desires to simulate. To estimate the impact of contention, the master thread measures the number of bytes transferred throughout the execution time of the microbenchmark (the master throughput) and the average latency per memory access, which is obtained by inverting the master throughput and dividing it by the cache line size.

We ran this microbenchmark on our target DSM platform, a 128-processor cluster of the NCSA SGI Origin2000. This cluster is organized in a fat hypercube topology with 64 nodes and 2 processors per node. The processors are MIPS R10000 running at 250 MHz, with 32 Kilobytes of split L1 cache and 4 Megabytes of unified L2 cache each. The cluster has 64 Gigabytes of DRAM memory, distributed uniformly across the 64 nodes. We assumed a maximum memory bandwidth of 667 Megabytes/s (as suggested in [2]) and configured the microbenchmark so that the master thread generates memory accesses at a constant rate of 200 Megabytes/s and each noise generator generates memory accesses at a constant rate of 10 Megabytes/s. We ran the microbenchmark with up to 63 noise generators, each of them running on a different node[3]. The result of this experiment (average memory access latency and master throughput vs. the number of nodes contending to access memory) is illustrated in Figure 1. The memory system appears to thrash when 46 or more nodes contend to access the same memory module. The result is intuitive, because with 46 noise generators, the accumulated rate of memory accesses from the master and the noise generators (200 Megabytes/s from the master and 460 Megabytes/s from the noise generators) matches the maximum available memory bandwidth.

[3]We are interested in contention incurred from remote memory accesses, therefore we restricted the number of threads per node to one, instead of two, which is the number of processors per node on the Origin. The latter would assess the impact of sharing the bus of the node between two processors. Since this is an architecture-specific feature of the Origin, we did not investigate it further for the purposes of this paper.

Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'02)
1530-2075/02 $17.00 © 2002 IEEE
IEEE COMPUTER SOCIETY

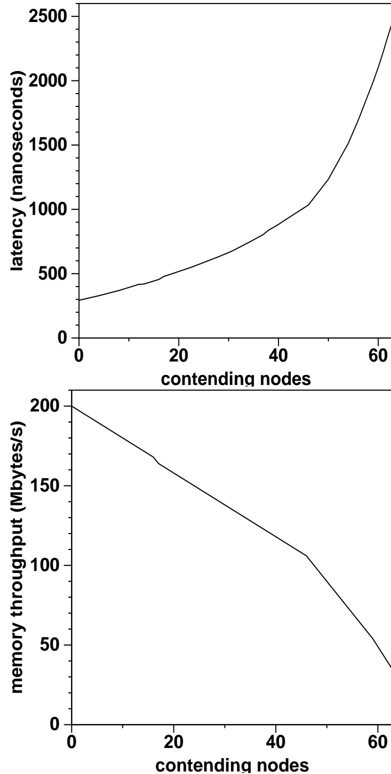

Figure 1. *Impact of contention on the latency of a single memory access and memory throughput on a 128-processor Origin2000. The master throughput without contention is fixed to 200 Megabytes/s.*

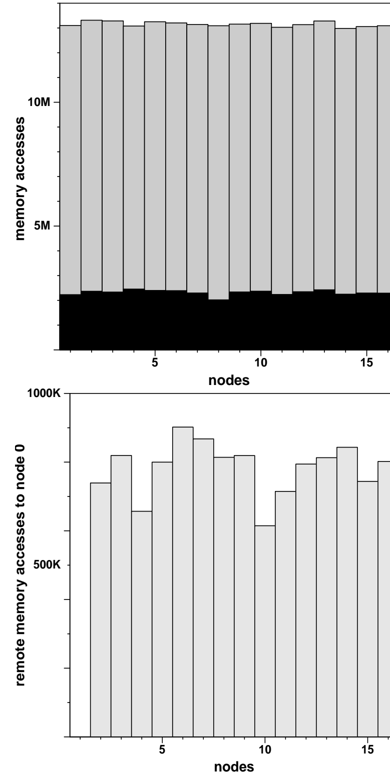

Figure 2. *Histogram of memory accesses per node, divided into local (gray) and remote (black) accesses and histogram of remote accesses from nodes 1 . . . 15 to node 0, collected during the execution of NAS BT on 32 processors of the NCSA Origin2000.*

2.3. Computing the Contribution of Contention to Execution Time

The first step to estimate the contribution of contention to the execution time of a program is to collect a histogram with the memory accesses to each node, divided into local and remote memory accesses, and N more histograms, each with the remote memory accesses issued to a node by each of the other nodes. Figure 2 shows two such histograms, obtained from executions of the NAS BT benchmark on 32 processors (16 nodes) of the NCSA Origin2000.

Assume that the system has N nodes, denoted as $n_i, i = 1 \ldots N$. Let us fix a node n_i, the memory of which is accessed L_i times from local processors (i.e. the processors on n_i) and R_i times from processors in remote nodes throughout the lifetime of the program. The number of remote memory accesses to n_i is the sum of accesses from each node $n_k, k \neq i$, i.e.

$$R_i = \sum_{k=1, k \neq i}^{N} R'_{ki} \quad (1)$$

Note that R_i signifies the number of remote accesses to n_i, while R'_{ki} signifies the number of remote accesses issued from n_k to n_i. The following analysis makes the assumption that consecutive memory accesses issued from the same source to the same destination do not contend with each other. In other words, contention is assumed to occur only between memory accesses issued from different nodes. The analysis can be easily extended to account for contention between accesses from the same node. We skip the relevant details due to space limitations.

The essence of the analysis is to estimate how much a memory access to a node may be dilated due to contention. We define contention as the number of memory accesses that arrive for service at a node while the node is already servicing a memory access, either local or remote. We assume that the memory accesses to n_i follow a Poisson arrival process. Let t_{ex} be the execution time of the program and l_{nocont} the latency of a memory access without contention. l_{nocont} is measured with the microbenchmark described in Section 2.2. For the Origin2000, $l_{nocont} \approx 300ns$. We

Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'02)
1530-2075/02 $17.00 © 2002 IEEE

IEEE COMPUTER SOCIETY

model the memory accesses issued to n_i as a Poisson process with mean equal to:

$$\mu_i = \frac{(L_i + R_i) l_{nocont}}{t_{ex}} \quad (2)$$

which corresponds to the mean number of memory accesses to n_i, during an interval of length equal to the latency of a memory access to n_i without contention. The probability that m memory accesses will be issued to n_i during this interval is:

$$p_i(m) = \frac{e^{-\mu_i} \mu_i^m}{m!} \quad (3)$$

We wish to compute the latency of a memory access to n_i, either local or remote, when this access contends with m accesses to n_i issued from m distinct nodes. The probability that a memory access to n_i is local is:

$$\frac{L_i}{R_i + L_i} \quad (4)$$

The probability that a memory access to n_i is a remote access issued from $n_k, k \neq i$ is:

$$\frac{R'_{ki}}{L_i + R_i} \quad (5)$$

The probability that a memory access to n_i is a remote access from any other DSM node (denoted as $p_{i,r}(1)$) is:

$$p_{i,r}(1) = \sum_{k=1, k \neq i}^{N-1} \frac{R'_{ki}}{L_i + R_i} \quad (6)$$

The probability that two consecutive remote memory accesses to n_i are issued from two distinct nodes n_{k_1} and n_{k_2} (denoted as $p_{i,r}(2)$) is:

$$p_{i,r}(2) = \sum_{\substack{k_1=1,\\ k_1 \neq i}}^{N-1} \sum_{\substack{k_2=k_1+1,\\ k_2 \neq i}}^{N} \frac{R'_{k_1 i}}{L_i + R_i} \frac{R'_{k_2 i}}{L_i + R_i - 1} \quad (7)$$

Proceeding in the same manner, we can compute $p_{i,r}(m), m = 3 \ldots N-1$, i.e. the probability that m consecutive remote memory accesses to n_i are issued from m distinct nodes as:

$$p_{i,r}(m) = \sum_{\substack{k_1=1,\\ k_1 \neq i}}^{N-m+1} \sum_{\substack{k_2=k_1+1,\\ k_2 \neq i}}^{N-m+2} \cdots \sum_{\substack{k_m=k_{m-1}+1,\\ k_m \neq i}}^{N} \Pi_{i,r}(m)$$

$$\Pi_{i,r}(m) = \frac{R'_{k_1 i}}{L_i + R_i} \frac{R'_{k_2 i}}{L_i + R_i - 1} \cdots \frac{R'_{k_m i}}{L_i + R_i - m + 1} \quad (8)$$

Similarly, the probability that m consecutive memory accesses to n_i include a local access form n_i and $m-1$ remote memory accesses from $m-1$ distinct nodes (denoted as $p_{i,lr}(m)$) is:

$$p_{i,lr}(m) = \frac{L_i}{L_i + R_i} \sum_{\substack{k_1=1,\\ k_1 \neq i}}^{N-m+1} \cdots \sum_{\substack{k_{m-1}=k_{m-2}+1,\\ k_{m-1} \neq i}}^{N-1} \Pi_{i,lr}(m)$$

$$\Pi_{i,lr}(m) = \frac{R'_{k_1 i}}{L_i + R_i} \cdots \frac{R'_{k_{m-1} i}}{L_i + R_i - m + 2} \quad (9)$$

From equations (3), (8) and (9), we can compute the probability that a memory access to n_i will contend with m memory accesses from m distinct nodes as:

$$p_i(cont, m) = p_i(m)(p_{i,r}(m) + p_{i,lr}(m)) \quad (10)$$

The expected latency of a memory access to n_i is computed as:

$$l_i = (1 - \sum_{m=1}^{N} p_i(cont, m)) l_{nocont} + \sum_{m=1}^{N} p_i(cont, m) l_{cont}(m) \quad (11)$$

The left factor accounts for the latency without contention, while the right factor accounts for the latency under varying degrees of contention. The sum in the right factor depends on the probability that a memory access from some node to n_i will be contending with one or more memory accesses from other nodes to n_i. $l_{cont}(m)$ is the latency of a single memory access under contention, measured with the microbenchmark presented in Section 2.2. Note that l_{cont} depends on m.

The accumulated latency of memory accesses to n_i is estimated as $(L_i + R_i) l_i$. The contribution of memory access contention to execution time is estimated as:

$$\max_i ((L_i + R_i)(l_i - l_{nocont})) \quad (12)$$

where the maximum is computed among all nodes $n_i, i = 1 \ldots N$. Note that the term $(L_i + R_i) l_{nocont}$ denotes the ideal memory latency on n_i, that is, the accumulated latency of memory accesses to n_i, if all these accesses were serviced without contention.

2.4. Validation

To validate our methodology, we ran a simple experiment that models extreme contention. We used the BT benchmark from the NAS benchmark suite [8]. BT is a complete CFD simulation that solves three-dimensional Navier-Stokes equations using an iterative method. The

Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'02)
1530-2075/02 $17.00 © 2002 IEEE

IEEE COMPUTER SOCIETY

benchmark is implemented in OpenMP and is carefully optimized for better cache and memory access locality on the Origin2000. More specifically, the benchmark uses the first-touch page placement algorithm, by executing one *warm-up* parallel iteration before executing the timed parallel iterations. The first-touch algorithm places each page locally to the processor that reads or writes in the page first during the course of execution.

As shown in Figure 2, first-touch page placement produces an almost perfectly balanced memory access pattern (in terms of local and remote memory accesses per node) for BT. Estimation of contention with our methodology indicates that when pages are placed with the first-touch algorithm, contention accounts for no more than 0.2% of execution time. Since this measurement alone does not give us any indication on the accuracy of our methodology, we use it only to establish a working assumption that BT does not suffer from contention.

In the validation experiment, we model extreme conditions by forcing the warm-up iteration of the benchmarks to run on one processor. This modification instructs the operating system to place the complete data set of the programs on a single designated node. Therefore, all processors but the ones of the designated node have to access the memory of that node upon every secondary cache miss. In this experiment, contention is purposely maximized. Our idea for measuring the accuracy of our methodology is to compare an estimate of contention using the memory access histograms of this *worst-case* execution, add it to the execution time of the benchmarks with first-touch page placement (which is considered to be the best case as far as contention is concerned) and compare the result against the actual worst-case execution time of the benchmark in the experiment.

We obtained the memory access histograms of this execution and estimated the impact of contention using the procedure described in Section 2.3. Without loss of generality, we assume that data is placed in node 0. Let t_{ft} be the execution time of the benchmark when its data is placed across nodes with the first-touch algorithm. We estimate the execution time of the benchmark with maximum contention as:

$$t_{est} = t_{ft} + (L_0 + R_0)(l_0 - l_{nocont}) + \sum_{i=1}^{N} R'_{i0} dist(i, 0) \quad (13)$$

The second factor of the sum in equation (13) is derived from equation (12), given that all data is placed in one node. The third factor of the sum is added to compensate for the fact that the local memory accesses of nodes $1 \ldots N$ during the execution of the benchmark with first-touch page placement are converted to remote memory accesses to node 0, since all data is placed in node 0. These

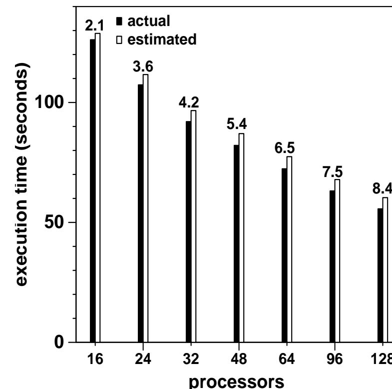

Figure 3. *Estimated and actual execution times of BT, with data placed in one node. The numbers on top of the bars show the difference (in percent) between the estimated and the actual times.*

memory accesses incur additional latency for traversing the interconnection network and this latency has to be added to t_{est}, since it is not accounted for in t_{ft}. This latency is signified by $dist(i, 0)$ and depends on the distance in hops between the accessing and the accessed node. We estimated the values of $dist(i, 0)$ using microbenchmarks [2]. In the Origin2000, each hop accounts for approximately 100 ns of remote memory access latency.

We compared t_{est} to the actual execution time of the benchmark when its data is placed in one node. The result (Figure 3) shows that the estimations are at most 8.4% off the actual execution time. The average error of the estimation is 5.4%, which we view as satisfactory. In general, our method tends to overestimate the impact of contention. We attribute the overestimation to two reasons. First, we do not model several architectural features of the Origin2000 that reduce contention, such as multiple memory ports. This is done to keep our model as simple and architecture-independent as possible. Second, the initial assumption of exponential arrivals of memory accesses to nodes may not be realistic enough. Remote memory accesses in real programs tend to occur in a bursty pattern, at the beginning of specific phases of the computation. Further research is required to explore alternative models.

3. Resolving Contention

We view the problem of resolving remote memory access contention on hardware DSM multiprocessors as a problem of balancing the remote memory accesses across the nodes of the system, so that the traffic of messages for accessing remotely located data is distributed evenly across the interconnection network. The main idea of our contention resolution algorithm is to identify *spikes* of re-

Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'02)
1530-2075/02 $17.00 © 2002 IEEE

IEEE COMPUTER SOCIETY

(1) find n_i such that $\forall n_j, j = 1 \ldots N, j \neq i, R_i > R_j$
(2) for each page g located in n_i
(3) if $\exists j, j \neq i, R'_{ji,g} > L_{i,g}$
(4) select $n_j, \forall n_k, k \neq i, k \neq j,$
(5) $(R'_{ki,g} > L_{i,g}) \wedge (R'_{ji,g} > R'_{ki,g})$
(6) migrate the page to n_j
(7) else if $R_{i,g} > L_{i,g}$
(8) find n_k so that $\forall n_j, j = 1 \ldots N, j \neq i, j \neq k,$
(9) $R_k < R_j$
(10) if $R_k + L_{i,g} + R_{i,g} - R'_{ki,g} < R_i$
(11) migrate g to n_k
(12) endif
(13) endif
(14) end for each

Figure 4. *Page migration algorithm for resolving contention. This algorithm runs repeatedly, until* $max_i R_i$ *cannot be further reduced.*

mote memory accesses in the memory access histograms. These spikes imply that some nodes concentrate disproportionately more remote memory accesses than others, hence they are likely to contain hot spots. The algorithm attempts to cut down the height of the spikes by redistributing remote memory accesses.

The vehicle for balancing remote memory accesses in our algorithm is dynamic page migration [12]. The algorithm searches for hot spots at page-level granularity and migrates pages so that hot spots are distributed, rather than concentrated in individual nodes. Page migration algorithms have been used before to improve the locality of memory accesses, by moving pages that incur frequent remote accesses from a single node. Our algorithm behaves similarly for pages that concentrate more remote rather than local memory accesses from a single node. Using the notation developed in Section 2, a page located in node n_i is migrated to node n_j if $R'_{ji,g} > L_{i,g}$, where g denotes the page number. However, in addition to this criterion, the algorithm checks if the accumulated number of remote memory accesses to the page exceeds the number of local memory accesses, i.e. if $R_{i,g} > L_{i,g}$. These pages are identified as candidates for migration. The intuition behind this heuristic is that migrating such pages from a node with excessive remote accesses to a node with few remote accesses will improve the local/remote access ratio of the former and balance the remote accesses better.

The algorithm (Figure 4) identifies first the node that concentrates the largest number of remote memory accesses (line 1). It migrates out of this node each page for which there exists at least one node that accesses it more frequently (lines 3–6). If there are more than one such nodes, the page is migrated to the node with the largest number of accesses (lines 4–5). Subsequently, the algorithm checks if the accumulated number of remote accesses to the page is more than the number of local accesses (line 7). If this is true, the algorithm finds the node with the minimum number of accumulated remote accesses (lines 8–9) and checks if migrating the page to that node will increase the number of remote accesses beyond the number of remote accesses to the node that hosted the page before the migration (lines 10–12).

We explain the last step in more detail. Recall that R_i denotes the accumulated number of remote accesses to n_i, while R'_{ki} denotes the number of remote accesses from n_k to n_i. If g is migrated from n_i to n_k, the local memory accesses from n_i to g ($L_{i,g}$) will be converted to remote accesses from n_i to n_k. The remote accesses from other nodes to g while g was on n_i ($R_{i,g}$) will be the same, except from the accesses from n_k ($R'_{ki,g}$), which will be converted to local. Therefore, the number of remote accesses to n_k (R_k) will be increased by $L_{i,g} + R_{i,g} - R'_{ki,g}$. Moving the page to n_k will be effective, if the accumulated number of remote accesses to n_k after the migration does not exceed the accumulated number of remote accesses to n_i before the migration. This implies that the maximum number of remote memory accesses to the nodes of the system will be reduced after migrating the page.

The algorithm is invoked repeatedly, until the maximum number of remote memory accesses among all nodes can not be further reduced. Note that the algorithm is straightforward to parallelize, by running an instance of it on each node and having a processor forward the pages that need to be migrated to other nodes.

4. Experimental Results

We evaluate our algorithm with five application benchmarks selected from three benchmark suites. The selected benchmarks are MG from the NAS benchmarks [8], the LG, SL and TS kernels from the Integrated Forecasts Model of the European Center for Medium Range Weather Forecasts [15], and SPECclimate, a benchmark included in the SPEChpc96 benchmark suite [14]. The benchmarks were selected after obtaining their memory access histograms from parallel executions on the Origin2000 and observing that contention has a significant impact on their parallel execution time. Using our methodology for quantifying contention, we estimated that on the maximum number of processors (128), contention accounts for 33.3% of execution time in MG, 14.5% of execution time in LG, 20.9% of execution time in SL, 27.9% of execution time in TS, and 23.3% of execution time in SPECclimate.

The criterion for selecting the benchmarks was the occurrence of spikes and irregularities in their memory access patterns. We executed the complete sets of benchmarks in

Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'02)
1530-2075/02 $17.00 © 2002 IEEE
IEEE COMPUTER SOCIETY

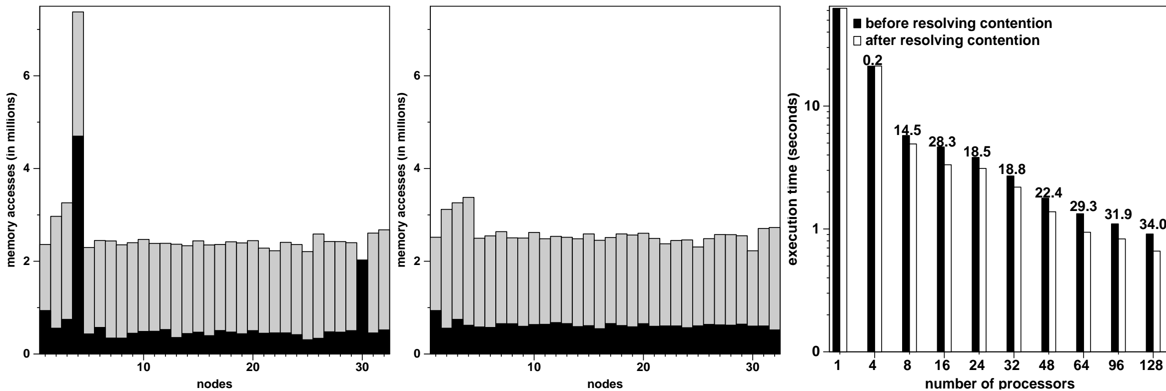

Figure 5. *Access traces before (left) and after (middle) resolving contention and execution times of MG.*

the three benchmark suites (NAS, SPEC, IFS) using the first-touch page placement algorithm of the Origin2000. The NAS codes are already tuned to use first-touch, by executing the warm-up parallel iteration (see Section 2.4). We applied the same modification to the IFS and the SPEC codes. The warm-up iteration ensures that processors touch data in the order they access it during the main part of the parallel computation. This is likely to increase cache reuse and improve memory access locality in codes with regular and balanced memory access patterns. Unfortunately, this is not the case in codes with irregular memory access patterns. After running the benchmarks and obtaining their memory access histograms, we identified the codes were few nodes concentrate excessively large fractions of remote accesses.

We applied our contention resolution algorithm to the benchmarks by linking them with *UPMlib* [13], a runtime system developed to optimize transparently memory access locality in OpenMP programs running on NUMA multiprocessors. The contention resolution algorithm is applied by instrumenting the source code to invoke the algorithm after the execution of the first iteration of the parallel computation. Since the selected benchmarks repeat the same parallel computation for a number of iterations, a snapshot of their memory access patterns after the first iteration is sufficient to obtain their memory access histograms. The algorithm runs once after the execution of the first iteration and in parallel, across the nodes of the system. The execution times presented in the following charts include the overhead of the algorithm. In this way, we show that the algorithm can be used to resolve memory access contention dynamically, at runtime.

For each benchmark, we report the memory access histograms from the execution of the benchmark on 64 processors (32 nodes) and the execution time of the benchmark on 1 to 128 processors, before and after applying the algorithm (Figures 5 through 9). The numbers on top of the execution time bars show the improvement from alleviating contention in percent. The IFS kernels require a square number of processors to implement appropriate grid decompositions, therefore we measure their execution time up to 121 processors. All codes scale reasonably well up to 128 processors, although the speedups of all benchmarks except MG seem to level off at certain points, due to either the granularity of parallelism (SL, TS and LG), or limited coverage, i.e. a significant fraction of sequential work which is not parallelized (SPECclimate).

Looking at the memory access histograms when the benchmarks are executed without the contention resolution algorithm (leftmost charts in Figures 5 through 9), observe the spikes of remote memory accesses and the irregularity of the memory access patterns. In all benchmarks, the irregularity of memory accesses is attributed to the irregular structure of the modelled grids. The three kernels from the IFS system use quasi-regular grids that model layers of the atmosphere and are densely populated by grid points towards the equatorial and sparsely populated by grid points towards the poles. Similar grids are used in SPECclimate, which predicts mesoscale and regional-scale atmospheric circulation. MG uses a V-cycle multigrid algorithm for solving discrete Poisson equations on 3D grids of different resolutions. The algorithm starts with an approximate solution in the finest grids, which is projected to progressively coarser grids. The problem in MG is that first-touch page placement implements a balanced blocked distribution of the fine-grain grids, which progressively collapses to an unbalanced distribution of the coarser-grain grids.

The reduction of execution time on 128 processors after applying our contention resolution algorithm is as much as 34.0% for MG, 18.7% for LG, 28.5% for SL, 34.1% for TS and 23.1% for SPECclimate. It appears that the algorithm achieves the expected improvement in MG and SPECclimate and more than the expected improvement in LG, TS and SL.

The histograms of memory accesses obtained after ap-

Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'02)
1530-2075/02 $17.00 © 2002 IEEE

IEEE COMPUTER SOCIETY

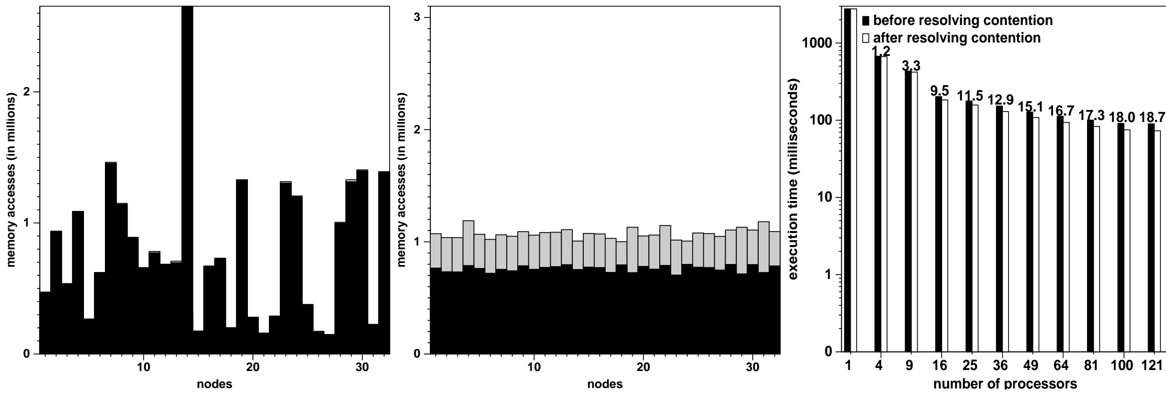

Figure 6. *Access traces before (left) and after (middle) resolving contention and execution times of LG.*

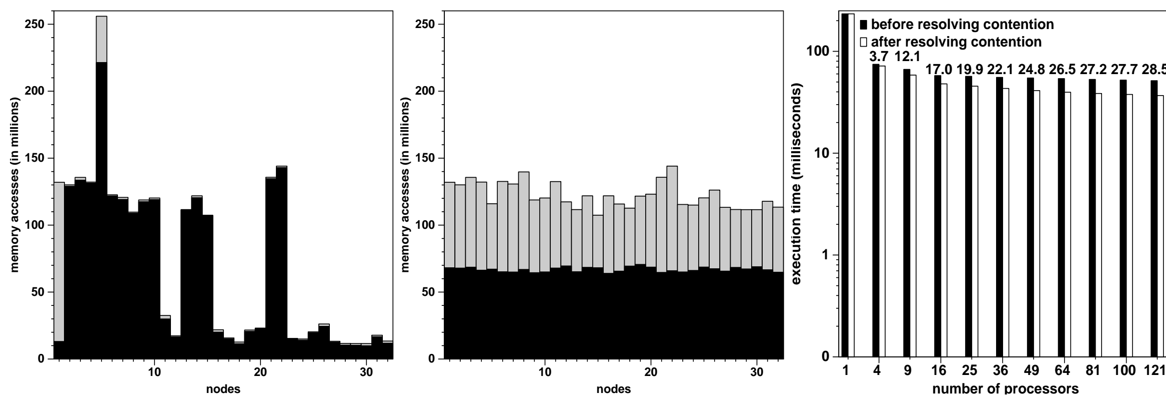

Figure 7. *Access traces before (left) and after (middle) resolving contention and execution times of SL.*

plying the contention resolution algorithm (charts in the middle of Figures 5 through 9) show that the algorithm is quite effective in balancing remote memory accesses across nodes, particularly in MG, LG, SL and SPECclimate. In LG, SL and TS, the algorithm has a profound side-effect on memory access locality. Dynamic page migration converts remote memory accesses to local memory accesses, when a single remote node accesses a page more frequently than the local node. Consequently, the algorithm increases the fraction of local accesses by approximately 20% in TS, 30% in LG and 50% in SL. The maximum number of remote memory accesses is cut down by a factor of 3 in LG, a factor of 4 in SL and a factor of 2 in TS. This benefit comes in addition to alleviating contention and explains the difference between the estimated and the actual improvement.

MG has two hot spots (on nodes 4 and 30, as illustrated in the histograms in Figure 5). This case resembles our extreme experiment (Section 2.4), in which we forced the processors to access always the same node upon secondary cache misses. Redistributing the remote memory accesses of the two hot spots improves the execution time of MG by as much as 34% on 128 processors. The improvement stems solely from resolving contention.

In SPECclimate, the algorithm does not reduce significantly the number of remote memory accesses. The improvements stem from alleviating contention by redistributing remote memory accesses and appear to be related to the data access pattern of the program in certain communication-intensive phases. Tracking the sources of contention inside the code might help us investigate the phased behavior of this program.

To conclude, we verified that contention is an important performance factor that has to be accounted for in hardware DSM multiprocessors. Our contention resolution algorithm improves the performance of five benchmarks with irregular memory access patterns by 27.7% on average. This is accomplished by distributing evenly the remote memory accesses across the system and, in most cases, reducing significantly the number of remote memory accesses.

5. Related Work

Several papers have pinpointed contention as one of the most important impediments of scalability in parallel pro-

Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'02)
1530-2075/02 $17.00 © 2002 IEEE
IEEE COMPUTER SOCIETY

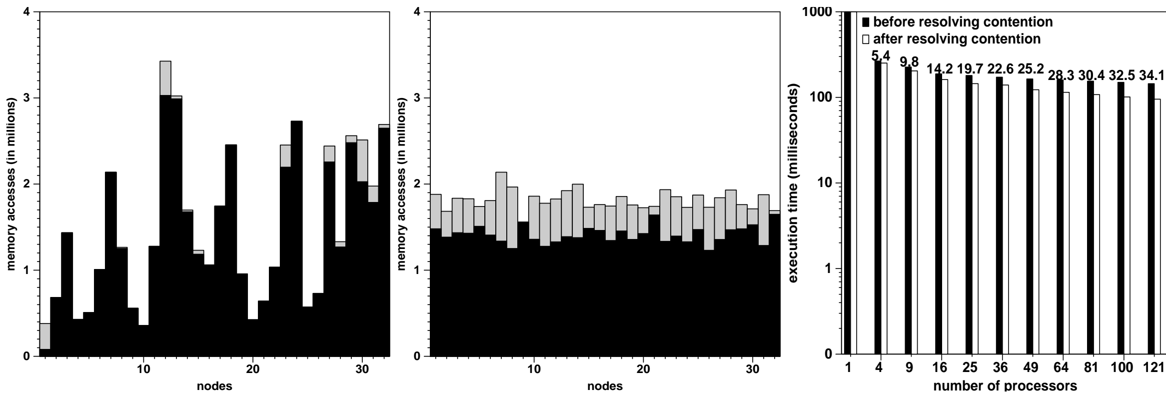

Figure 8. *Access traces before (left) and after (middle) resolving contention and execution times of TS.*

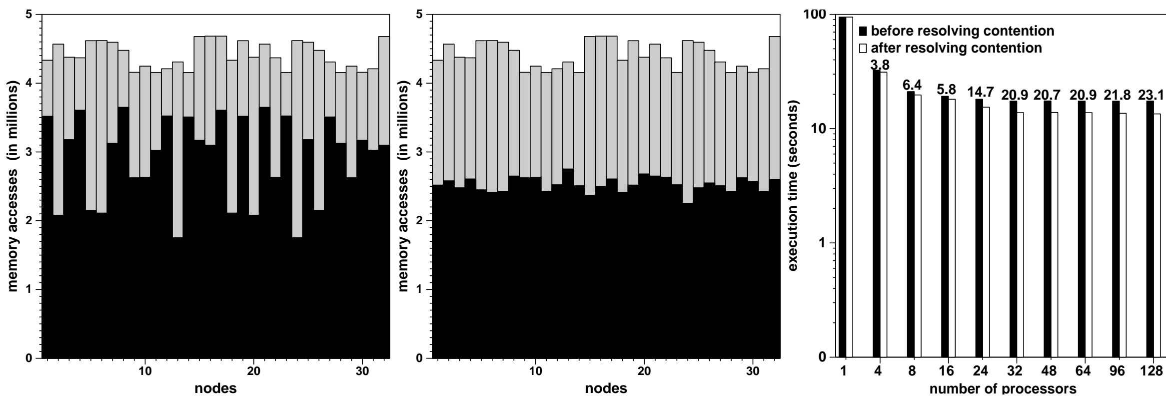

Figure 9. *Access traces before (left) and after (middle) resolving contention and execution times of SPECclimate.*

grams (e.g. [1, 3, 5]). However, most of these works quantified contention either by detailed simulation or with analytical modelling. Blelloch et.al. [1] developed a formal model for analyzing memory bank contention on shared memory multiprocessors in which processors access memory via a switching network. Frank et.al. [5] extended the LogP model to account for contention in both distributed and shared memory multiprocessors. Dai and Panda [3] simulated a detailed network model that captures all types of contention in every part of the network of a hardware DSM multiprocessor. They have shown that network contention can have a significant impact on performance and conducted a sensitivity analysis of architectural parameters that might affect contention, such as the design of caches, CPU speed and network speed. The fundamental difference between these works and ours is that these works either predict or simulate the effect of contention using a number of architectural and algorithmic parameters, while we quantify the impact of contention on real programs by directly executing them on real hardware.

Hristea and Lenoski [2] used microbenchmarks to measure the impact of contention on the latency of memory accesses on the Origin2000. We used the same methodology to measure memory access latency as a function of the degree of contention.

The works of De Lara et.al. [4] and Lu et.al. [11] appear to be more closely related to our contribution, in the sense that they present means to quantify contention in parallel programs running on shared virtual memory systems (also known as software DSM systems), and propose techniques to resolve contention. These techniques can be implemented either in the DSM protocol or at the application layer. De Lara et.al. [4] show that contention in shared virtual memory systems can be quantified by counting the requests for page updates that arrive at a node while another page update request is already being processed. Along with some application-specific optimizations for reducing contention, they propose a technique that identifies data structures with one writer and multiple readers as hot spots and redistributes these data structures dynamically, to balance the coherence protocol load for keeping the data structures up to date. Lu et.al. [11] propose a method that replicates sequential code that precedes parallel sections, so that contention at the beginning of parallel sections is avoided.

Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'02)
1530-2075/02 $17.00 © 2002 IEEE
IEEE COMPUTER SOCIETY

Our work differs in that it is applicable to hardware cache-coherent DSM multiprocessors, which have fundamental architectural differences compared to clusters with shared virtual memory.

6. Conclusion

We have presented a simple methodology for quantifying the impact of contention on real programs running on real hardware DSM multiprocessors. Our methodology estimates the overhead of contention as a function of the number of memory accesses from each node to each page in memory. This information can be collected in hardware page reference counters, which are available in several commercial systems. Driven by this methodology, we proposed an algorithm that alleviates contention and, as a side-effect, improves memory access locality using dynamic page migration. The algorithm integrates a locality-sensitive page migration criterion with a criterion that balances the number of remote memory accesses and the associated protocol traffic across the DSM nodes. We have validated experimentally the accuracy of our methodology and have shown that contention has a significant impact, which may account for as much as 34% of execution time in programs with irregular memory access patterns. Finally, we have presented experiments that prove the effectiveness of our contention resolution algorithm.

Future work will hopefully address the problem of correlating contention with specific parts of the code and attempt to resolve contention that stems from application-specific phenomena, such as false sharing, phased behavior in the memory access pattern and bursty communication that occurs between sequential and parallel sections. Investigating the effects of contention in systems with more aggressive coherence protocols is also within our plans.

References

[1] G. Blelloch, P. Gibbons, Y. Matias, and M. Zagha. Accounting for Memory Bank Contention and Delay in High-Bandwidth Multiprocessors. In *Proc. of the 7th ACM Symposium on Parallel Algorithms and Architectures (SPAA'95)*, pages 84–94, Santa Barbara, California, June 1995.

[2] D. L. C. Hristea and J. Keen. Measuring Memory Hierarchy Performance on Cache-Coherent Multiprocessors Using Microbenchmarks. In *Proc. of the ACM/IEEE Supercomputing'97: High Performance Networking and Computing Conference (SC'97)*, San Jose, California, Nov. 1997.

[3] D. Dai and D. Panda. How Much Does Network Contention Affect Distributed Shared Memory Performance? In *Proc. of the 1997 International Conference on Parallel Processing (ICPP'97)*, pages 454–461, Bloomingdale, Illinois, Aug. 1997.

[4] E. de Lara, Y. Hu, H. Lu, A. Cox, and W. Zwaenepoel. The Effect of Contention on the Scalability of Page-Based Software Shared Memory Systems. In *Proc. of the 5th ACM Workshop on Languages, Compilers and Runtime Systems for Scalable Computers (LCR'2000), LNCS Vol. 1915*, pages 155–169, Rochester, New York, May 2000.

[5] M. Frank, A. Agarwal, and M. Vernon. LoPC: Modeling Contention in Parallel Algorithms. In *Proc. of the 6th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'97)*, pages 276–287, Las Vegas, Nevada, June 1997.

[6] E. Hagersten and M. Koster. WildFire: A Scalable Path for SMPs. In *Proc. of the 5th International Symposium on High Performance Computer Architecture (HPCA-5)*, pages 171–181, Orlando, Florida, Jan. 1999.

[7] D. Jiang and J. P. Singh. A Methodology and an Evaluation of the SGI Origin2000. In *Proc. of the 1998 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'98)*, pages 171–181, Madison, Wisconsin, June 1998.

[8] H. Jin, M. Frumkin, and J. Yan. The OpenMP Implementation of the NAS Parallel Benchmarks and its Performance. Technical Report NAS-99-011, NASA Ames Research Center, Oct. 1999.

[9] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th International Symposium on Computer Architecture (ISCA'97)*, pages 241–251, Denver, Colorado, June 1997.

[10] C. Liao, D. Jiang, L. Iftode, M. Martonosi, and D. Clark. Monitoring Shared Virtual Memory Performance on a Myrinet-based PC Cluster. In *Proc. of the 12th ACM International Conference on Supercomputing (ICS'98)*, pages 251–258, Melbourne, Australia, July 1998.

[11] H. Lu, A. Cox, and W. Zwaenepoel. Contention Elimination by Replication of Sequential Sections in Distributed Shared Memory Systems. In *Proc. of the 8th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'01)*, pages 53–61, Snowbird, Utah, June 2001.

[12] D. Nikolopoulos, T. Papatheodorou, C. Polychronopoulos, J. Labarta, and E. Ayguadé. A Case for User-Level Dynamic Page Migration. In *Proc. of the 14th ACM International Conference on Supercomputing (ICS'2000)*, pages 119–130, Santa Fe, New Mexico, May 2000.

[13] D. Nikolopoulos, T. Papatheodorou, C. Polychronopoulos, J. Labarta, and E. Ayguadé. UPMlib: A Runtime System for Tuning the Memory Performance of OpenMP Programs on Scalable Shared-Memory Multiprocessors. In *Proc. of the 5th ACM Workshop on Languages, Compilers and Runtime Systems for Scalable Computers (LCR'2000), LNCS Vol. 1915*, pages 85–99, Rochester, New York, May 2000.

[14] Standard Performance Evaluation Corporation. SPEC HPC96 Documentation. http://www.spec.org/hpg, Dec. 2001.

[15] P. White. IFS Documentation: Part VI, Technical and Computational Procedures. Technical Report CY21R4, European Centre for Medium-Range Forecasts, Feb. 2000.

Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'02)
1530-2075/02 $17.00 © 2002 **IEEE**

IEEE COMPUTER SOCIETY