

Quantifying Contention and Balancing Memory Load on Hardware DSM Multiprocessors¹

Dimitrios S. Nikolopoulos
Department of Computer Science
The College of William&Mary
McGlothlin Street Hall
Williamsburg, VA 23187-8795, U.S.A.
dsn@cs.wm.edu

Version:

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 overhead of contention on real parallel programs running on real hardware. The methodology uses as input the number of accesses from each node of the DSM to each page in memory. A trace of the memory accesses of the program obtained at runtime is used to compute a fairly 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 balances memory load using dynamic page migration. The algorithm attacks indirectly the problem of contention by balancing the remote memory access latency across the nodes of the system. Experiments with five irregular parallel codes on a 128-processor Origin2000 show that the algorithm yields significant performance improvements.

¹This work was partially supported by NSF grant No. EIA-99-75019 and a startup research grant from the College of William and Mary. Part of this work was conducted while the author was with the Coordinated Science Laboratory at the University of Illinois, Urbana-Champaign.

Biography

Dimitrios S. Nikolopoulos is an Assistant Professor of Computer Science at the College of William&Mary. Before joining William&Mary, he was a Visiting Research Assistant Professor of the Coordinated Science Laboratory at the University of Illinois, Urbana-Champaign. He received his Diploma Degree in Computer Engineering in 1996 and his Ph.D in Computer Engineering in 2000, both from the University of Patras, Greece. His research focuses on the development of programming environments for effective and effortless adaptation of programs to high-end computing platforms. He has authored around 40 papers on runtime optimization techniques and operating system support for high-performance and high-throughput computing. Three of his papers won “Best Paper Awards” at SC’2000, IPDPS’2002 and CCGrid’2002.

1. INTRODUCTION

It is well known that contention is one of the factors that limit the performance of parallel programs. Contention stems from simultaneous accesses of multiple processors to shared resources such as memory and network links. On cache-coherent distributed shared memory (DSM) multiprocessors, one of the most intrusive forms of contention occurs at the network interface of a node, due to simultaneous memory accesses from multiple nodes².

Although contention must be accounted for when programmers attempt to scale parallel programs on DSM multiprocessors, the hardware of these systems lacks the means to quantify directly the overhead of contention on the execution time of a program. The hardware counters of modern microprocessors do not reveal directly any information about contention. Carefully designed microbenchmarks [9] can measure the overhead 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 wastes due to contention. Useful information about contention can be collected by performance monitoring firmware running at the network interface [13]. This solution is appealing, but not portable, because it requires programmable network interfaces. Commercial hardware DSM systems do not offer this option. Due to the aforementioned limitations, performance studies that investigate the scalability of hardware DSM systems have to speculate on the overhead of contention by factoring out the overhead of all other parameters that affect performance [10].

In this paper we present a new methodology for quantifying remote memory access contention on 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, throughout the execution of the program. This information is collected in hardware page access counters, which are available in some commercial DSM systems, such as the SGI Origin2000 [12] and the Sun Wildfire [8]. The required information can also be collected with lightweight software instrumentation, if the operating system provides enough information on the mapping of virtual memory pages to nodes. The most valuable aspect of this methodology is that it is able to assess the overhead of contention on parallel programs while running them on real hardware, without simulating the hardware or modelling the behavior of the program at the source code level.

The paper presents also a new algorithm which balances the memory load across the nodes of a DSM multiprocessor using dynamic page migration. The algorithm attempts to indirectly resolve remote memory access contention, by identifying pages which are likely to contain hot spots and off-loading a significant number of remote memory accesses from these pages.

The algorithm differs from previously proposed page migration algorithms, both in its mechanics and in its purpose. Instead of on-line competitive analysis of page access frequencies, the algorithm uses off-line analysis of the complete memory access pattern of the program. On-line competitive page migration algorithms have been proposed for the optimization of memory access locality in applications with dynamic memory access patterns, which can only be detected at runtime [21]. Their purpose is to migrate pages which seem to be accessed more frequently by nodes other than their home nodes, based on a snapshot of the access counters of the pages taken periodically at runtime. The algorithm presented in this paper balances

memory load in terms of the accumulated latency of remote memory accesses to each node. Memory load balancing is performed using a complete trace of the memory accesses of the program. We use remote access latencies, instead of remote access frequencies, to compensate for the disparity of remote access latencies on cache-coherent DSM multiprocessors.

The proposed algorithm treats the target application as a black box and does not require programmer intervention or application-specific knowledge. On the negative side, the algorithm requires periodicity and time invariance in the behavior of the program. This means that the program should be either internally periodic (i.e. repeating the same computation for a number of iterations), or periodic across invocations. The algorithm performs whole program optimization of data placement. It does not correlate memory load imbalance and contention with specific parts of the program, neither does it restructure the program to alleviate contention via code transformations.

Our validation experiments show that we are able to estimate the execution time of programs with an error (expressed as a fraction of total execution time) that ranges between 1% and 6%. The mean error of the estimates, expressed as a fraction of the total execution time of the programs is 2.5%. The mean error of the estimates, expressed as a fraction of the actual overhead of contention in the execution time of the programs is 19.4%. Not surprisingly, the experiments with real parallel applications indicate that contention has a significant impact on performance. We find that in programs with irregular memory access patterns, severe memory load imbalance can occur either across the entire address space of the program, or in a narrow range of pages that concentrate disproportionate amounts of remote memory accesses. Our algorithm is able to alleviate contention and reduce significantly the parallel execution time of five such applications. The arithmetic mean of execution time improvement across all benchmarks and processor scales is 17.8%. On more than 100 processors, execution time is improved by 20–35%.

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

2. QUANTIFYING CONTENTION

2.1. Outline

The starting point for estimating the overhead 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 using microbenchmarks and is measured for varying degrees of contention. The second step is to collect $N + 1$ histograms, where N is the number of nodes on the multiprocessor. 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 capture the memory access pattern of the program throughout its lifetime. Given the 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 across the nodes that participate in the execution of the program. To isolate the overhead 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 served without contention.

2.2. Estimating the Overhead of Contention on a Single Memory Access

We use the methodology proposed by Hristea et.al. [9], to estimate the overhead of contention on a single memory access. This methodology measures accurately the back-to-back and restart latency of a memory access, on processors that allow multiple outstanding memory accesses to overlap with the execution of computation. The back-to-back latency is the non-overlapped latency of a cache miss which must be served from main memory, from the time the cache miss occurs, until the time the entire requested cache line is brought in the cache. The restart latency is the non-overlapped latency of a cache miss which must be served from main memory, from the time the cache miss occurs, until the requested word (instead of the entire cache line) is brought in the cache and the processor can restart execution.

The methodology uses a multithreaded microbenchmark, in which each thread executes the code shown in Figure 1, with different parameters for the number of iterations (`num_iter`) of the inner loop. The microbenchmark does pointer chasing on a linked list which is stored as an array. The work loop walks memory locations with a constant stride, which is passed to the program as a parameter. The outer loop of the microbenchmark chases the pointers along the list, so that each load of `j` is dependent on the previous load. This prevents the processor from overlapping two consecutive memory accesses in the pipeline. Furthermore, if the stride is set equal to the size of a cache line (measured in array elements), each load of `j` brings in a new cache line. The bandwidth is measured as the number of bytes transferred (`num_reads × cache_line_size`) divided by the elapsed time.

Between consecutive loads, each thread executes dummy work which is regulated by the variable `num_iters`. Note that the load uses a register allocated variable (`dummy`) and a cached variable (`j`), to avoid further cache misses. Note also that the dummy work loop is not overlapped with the subsequent load, because of the dependence on `j`. The amount of work inserted between the loads regulates the frequency of the loads. The more the work, the less frequent the loads. Consecutively, the more the work, the less the consumed bandwidth. The microbenchmark does not use an explicit parameter for the consumed bandwidth. The variable `num_iters` is empirically regulated with multiple executions of the microbenchmark, until the desired bandwidth consumption is reached. To measure the memory latency without contention, we run the microbenchmark without the dummy loop and calculate latency by inverting the product of the measured bandwidth and the cache line size. The calculation is correct since back to back memory accesses are dependent and can not be overlapped. This methodology assumes that the processor does not use hardware prefetching. This assumption is true for the processors of the system that we study in this paper.

The microbenchmark is executed simultaneously by multiple threads, one of which is designated as the master thread. The threads share the array that stores the linked list. The implementation ensures that the array is allocated in a single memory module of a single node of the multiprocessor. The `j` and `dummy` variables are private to each thread. The only difference between the master thread and the

slave threads is the number of iterations executed in the work loop (`num_iters`), which controls the memory bandwidth consumed by each thread. To measure the overhead of contention, we measure the reduction of the memory throughput of the master thread, with an increasing number of slave threads running concurrently. To measure the overhead on memory latency, we strip the dummy work loop off the master thread and calculate the back-to-back latency as described in the previous paragraph. We explain how we measure restart latency with the same microbenchmark in Section 2.3.

We ran this microbenchmark on the target DSM platform, a 128-processor SGI Origin2000. This system is organized in a fat hypercube topology with 32 vertices, 2 nodes per vertex 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 per processor. The system has 64 Gigabytes of DRAM, distributed uniformly across the 64 nodes. We assumed a maximum memory bandwidth of 667 Megabytes/s (as suggested in [9]).

To measure memory throughput, we configured the microbenchmark so that the master thread fetches approximately 200 Megabytes/s from memory and each slave thread fetches approximately 10 Megabytes/s from memory. We ran the microbenchmark with up to 63 slaves, each of them executing on a different node³. The measured master throughput is shown in the top chart of Figure 2. To measure memory latency, we repeated the same experiment, but set the master thread to issue only back-to-back memory accesses without executing work. The result is shown in the bottom chart of Figure 2. The master throughput appears to thrash when more than 45 nodes issue accesses to the same memory module simultaneously. The result is expected. 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) almost matches the maximum available memory bandwidth. In fact, the memory bandwidth is saturated due to contention between threads. In the second configuration of the microbenchmark, the sustained master throughput is 267 Megabytes/s. The back-to-back latency without contention is measured as 478 ns. Latency (shown in the bottom chart of Figure 2) starts increasing exponentially at the same point where throughput seems to thrash, i.e. when 45 or more nodes contend to access the same memory module.

2.3. Computing the Overhead of Contention in Execution Time

The first step to estimate the overhead of contention in the execution time of a program is to collect $N + 1$ histograms: one histogram with the memory accesses to each node, divided into local and remote memory accesses; and N histograms, each with the remote memory accesses issued to a node by each of the other $N - 1$ nodes.

Assume that the system has N nodes, denoted as $n_i, i = 1 \dots N$. 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 remote processors, throughout the lifetime of the program. The number of remote memory accesses to n_i is the sum of accesses from nodes $n_k, k \neq i$, 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 and R'_{ki} signifies the number of remote accesses from n_k to n_i .

The following analysis makes two assumptions: First, we assume that the latency of consecutive memory accesses is not overlapped with computation. This assumption is not valid on most modern superscalar microprocessors. On the Origin2000, the processors can have up to four outstanding memory accesses. In fact, previous studies [22] have shown that the processors of the Origin2000 can mask up to 50% of the memory access latency in real applications. We investigate whether memory latency overlap affects the accuracy of the estimates in Section 2.4.

The second assumption is that contention occurs at the memory queue only between local and remote accesses. More specifically, the analysis estimates only the potential delay of local accesses due to intervening remote accesses in the memory queue. We do not assume any other overlap of memory accesses by the memory controller. This assumption is valid for the Origin2000, but may need to be revisited in newer systems.

The analysis calculates probabilities of one or more remote memory accesses arriving at a memory module while a local memory access is being served. The time window examined equals the non-overlapped restart latency of a local memory access. This is assumed to be the smallest time window during which contention can occur. The restart latency of a cache miss is measured with the microbenchmark shown in Figure 1, using a slightly modified configuration. The key modification is to overlap a memory access with independent computation and keep increasing the amount of computation as long as the overall execution time of the microbenchmark does not increase. Up to that point, the computation is completely overlapped with the memory access. The length of the maximum amount of work that the processor can execute in the inner loop without increasing the execution time of the microbenchmark gives the restart latency of the processor. More details on this methodology can be found in [9].

Contention depends on the number of memory accesses that arrive for service at a node while the node is already serving a memory access. 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} be the restart latency of a local memory access without contention.

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

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

This corresponds to the mean number of memory accesses to n_i , during an interval 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 compute the latency of a memory access to n_i , either local or remote, when this access contends against 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 $p_r(i, 1)$) is:

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

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

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

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

$$p_r(i, m) = \sum_{\substack{k_1=1, \\ k_1 \neq i}}^N \sum_{\substack{k_2=1, \\ k_2 \neq i, k_1}}^N \dots \sum_{\substack{k_m=1, \\ k_m \neq i, k_1, \dots, k_{m-1}}}^N \Pi_r(i, m) \\ \Pi_r(i, m) = \frac{R'_{k_1 i}}{L_i + R_i} \frac{R'_{k_2 i}}{L_i + R_i - 1} \dots \frac{R'_{k_m i}}{L_i + R_i - m + 1} \quad (8)$$

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

$$p_{lr}(i, m) = \frac{L_i}{L_i + R_i} \sum_{\substack{k_1=1, \\ k_1 \neq i}}^N \dots \sum_{\substack{k_{m-1}=1, \\ k_{m-1} \neq i, k_1, \dots, k_{m-2}}}^N \Pi_{lr}(i, m) \\ \Pi_{lr}(i, m) = \frac{R'_{k_1 i}}{L_i + R_i} \dots \frac{R'_{k_{m-1} i}}{L_i + R_i - m + 2} \quad (9)$$

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

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

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

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

The left factor accounts for the latency without contention, while the right factor accounts for the latency with 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 against 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 shown in Figure 1. Note that l_{cont} depends on m .

The non-overlapped remote memory access latency from n_k to n_i with contention is derived similarly as follows:

$$r_{ki} = \left(1 - \sum_{m=1}^N p_{cont}(i, m)\right) r_{ki, nocont} + \sum_{m=1}^N p_{cont}(i, m) r_{ki, cont}(m) \quad (12)$$

In Equation 12, $r_{ki, nocont}$ is the remote access latency from node k to node i without contention. This latency is calculated by adding the local latency with no contention (l_{nocont}) and the latency of traversing the interconnection network from k to i , which is denoted as d_{ki} . On the Origin2000, this latency is approximately 100 ns per hop, assuming no contention on the network switches. The remote access latency from k to i with contention ($r_{ki, cont}$) is calculated with the microbenchmark shown in Figure 1, by measuring the execution time of the reads performed by one slave to the shared array, when the slaves execute as much work as possible, without delaying the back-to-back memory accesses. The location of the slave and the number of peer slaves that compete to access the shared array are parameters and can be modified to measure the latency of memory accesses from different distances, under varying degrees of contention. Note that the microbenchmark factors in contention in the interconnection network, since multiple remote nodes compete to access the same memory module. The remote memory access latency under contention depends on m (the number of contending nodes) and the distance between k and i , which can be set in the microbenchmark.

The accumulated latency of memory accesses to n_i is estimated as $(L_i l_i + \sum_{k=1, k \neq i}^N R'_{ki}) r_{ki}$. The overhead of remote memory access contention to execution time is estimated as:

$$\max_i \left(L_i (l_i - l_{nocont}) + \sum_{k=1, k \neq i}^N R'_{ki} (r_{ki} - r_{ki, nocont}) \right) \quad (13)$$

where the maximum is computed across all nodes $n_i, i = 1 \dots N$. The left term measures the overhead of contention on local memory accesses, whereas the right term measures the overhead of contention on remote memory accesses.

2.4. Validation

To validate the methodology, we ran experiments with five parallel benchmarks. We used MG from the NAS benchmarks [11], LG, SL and TS, from the Integrated Forecasts System of the European Center of Medium Range Weather Forecasts [23], and SPECclimate, a benchmark included in the SPECchpc96 benchmark suite [20]. All benchmarks are implemented in OpenMP and optimized for efficient execution

on shared-memory architectures by third parties. More details on the benchmarks are provided in Section 4.

We executed the benchmarks in two configurations. In the first configuration, each benchmark uses the automatic page placement algorithm of the operating system to distribute data across nodes. The user can select between first-touch page placement and round-robin page placement by setting an environment variable. We executed the benchmarks with first-touch page placement, which resulted to better overall performance. In the second configuration, each benchmark was executed with an artificial page placement algorithm which maximizes contention. This theoretically worst algorithm places all data in the memory of a single node. Contention is maximized by forcing all processors except the ones on the designated node to access the same remote memory, upon every secondary cache miss.

The worst algorithm is easy to implement in iterative codes. The user sets two environment variables, one for activating first-touch page placement and another for instructing the runtime system to change the number of processors on which the program executes on-the-fly. The first iteration of the benchmark is executed on one processor and the rest of the iterations are executed on the maximum number of processors. These modifications force the data of the benchmark to be mapped to the memory of the node on which the first iteration is executed.

We obtain the memory access histograms of the executions of the benchmarks with first-touch page placement and the worst page placement algorithm, using the process described in Section 2.3. For brevity, we name the first version of the benchmark FT and the second version of the benchmark W . Without loss of generality, we assume that in the W version, all pages are placed in node 1. Let t_{ft} be the execution time of the FT version. Let $l_{i,ft}, i = 1 \dots N$ be the latency of local accesses in the FT version, and $r_{ij,ft}, i = 1 \dots N, j = 1 \dots N, j \neq i$ be the latency of remote accesses in the FT version. Let $l_{i,w}, r_{ij,w}$ be the same latencies in the W version. These latencies are calculated directly from equations 11 and 12. Let L_i and R'_{ij} denote the number of local and remote accesses from node i in the FT version. We estimate the execution time of the benchmark with the worst-case page placement as:

$$t_{est} = t_{ft} + \max \left(L_1(l_{1,w} - l_{1,ft}) + \sum_{i=2}^N R'_{1i}(l_{1,w} - r_{1i,ft}), \right. \\ \left. \max_{i \neq 1} (L_i(r_{i1,w} - l_{i,ft}) + \sum_{j=1, j \neq i}^N R'_{ij}(r_{i1,w} - r_{ij,ft})) \right) \quad (14)$$

Briefly, equation 14 gives the additional latency due to contention in the W version. The overhead of contention is obtained by taking the maximum of the additional latencies of all nodes of the system in the W version, due to the placement of all data pages in node 1. The additional latency of node 1 is composed of two terms, separated by comma between the two lines in equation 14. The first term accounts for the additional latency of node 1 and the second for the additional latency of the other nodes. The maximum of these additional latencies gives an estimate of the net increase of execution time in the W version, compared to the FT version. The additional latency of node 1 is itself the sum of two terms. The first term, $(L_1(l_{1,w} - l_{1,ft}))$, accounts for the additional latency of local accesses in the FT version, which remain local in the W version, but with higher latency ($l_{1,w} >$

$l_{1,ft}$) due to contention. The second term ($\sum_{i=2}^N R'_{1i}(l_{1,w} - r_{1i,ft})$) is the additional latency of remote accesses from node 1 to the other nodes in the *FT* version, which are converted to local accesses in the *W* version. Again due to contention, the latency of these converted-to-local accesses may actually be higher than the latency of the same accesses, when they were issued remotely in the *FT* version. The additional latency of the accesses from remote nodes to node 1 in the *W* version is also composed of two terms. The first term ($L_i(r_{i1,w} - l_{i,ft})$), is the additional latency of the local accesses of each node $i, i \neq 1$, which are converted to remote accesses to node 1 in the *W* version. The second term ($\sum_{j=1, j \neq i}^N R'_{ij}(r_{i1,w} - r_{ij,ft})$), is the additional latency of the remote accesses from any node $i, i \neq 1$ to any node $j, j \neq i$, when these accesses are redirected to node 1 in the *W* version.

The aforementioned analysis assumes that the work distribution between processors is balanced, which is true for the benchmarks that we study. It also assumes that the latency of memory accesses is not overlapped. We calculated one more estimate of the execution time of the *W* version, assuming that the latency of memory accesses is overlapped.

We used the methodology proposed by Wasserman et.al [22] to estimate how much of the memory latency is actually overlapped in the benchmarks. This methodology calculates the ideal CPI (cycles per instruction) of a program and the additional CPI consumed due to memory stalls, using hardware counters. The number of committed instructions, the number of CPU cycles consumed by the program, and the number of L1 and L2 cache misses can all be measured with hardware counters. The MIPS R10000 has only two hardware counters that can count different events in parallel, so multiple executions of the benchmarks with different configurations of the hardware counters are needed to collect all the data.

For each program, we ran a version with a small data set that fits entirely in the L1 cache (the data cache size is 16 Kilobytes on the MIPS R10000) and calculated the ideal CPI. The number of stall cycles per L2 cache miss was calculated with linear curve fitting using the least squares method. For the five benchmarks that we study (MG, LG, SL, TS and SPECclimate) the number of memory stall cycles calculated with this methodology was 75, 73, 70, 77 and 71 CPU cycles respectively. These numbers are arithmetic means of the L2 cache miss latencies from the executions of each benchmark on various numbers of processors from 1 to 128.

The non-overlapped latency of L2 cache misses was measured as 85 CPU cycles. The result indicates that 84–91% of the memory latency is not overlapped for the benchmarks that we studied. This ratio is significantly higher compared to the ratios reported in [22], however it is not unjustified, since the benchmarks have irregular memory access patterns. This implies relatively high cache miss rates and more memory latency that needs to be overlapped. The experiments show also that there is significant latency overlap at low processor scales. The processors overlap 40–55% of the latency on executions using up to 8 processors (for MG and SPECclimate) and 16 processors (for LG, SL and TS). On these low processor scales, contention is insignificant. On larger processor scales, where contention becomes significant, there is only barely measurable overlap of memory latency.

We compared the values of t_{est} of the *W* versions of the benchmarks with and without accounting for overlapped memory latency, to the actual execution times of the *W* versions of the benchmarks. The results are shown in Figures 3 through

7. The top chart in each figure shows the normalized execution time of different configurations on various processor scales. Execution times are normalized to the execution time of the *FT* versions. The second chart in each figure shows the error of the estimates, as a fraction of the difference in execution time between the *FT* version and the *W* version, with and without accounting for overlapped memory latency.

The results show that the estimates of execution times of the *W* versions exceed the actual execution times of the *W* versions by 1%–6%. The error of the estimates as a fraction of the difference in execution time between the *FT* version and the *W* version ranges between 8.3% and 80%, assuming no overlapped memory latency. The arithmetic mean of the error is 19.4% and the standard deviation is 15.3.

In general, the method overestimates the overhead of contention and the margin between the estimated and the actual overhead tends to widen when the number of processors is increased. One of two explanations could be given for this result. The first is that we do not take into account overlapped memory latency. As shown in the charts though, taking into account latency overlap seems to improve the accuracy of the estimate only marginally. This is by no means a general conclusion. The potential for latency overlap depends on the benchmark and the benchmarks that we use do not exhibit this property. Since we do have a simple method to estimate latency overlap, it is straightforward to extend the model to take into account this factor.

A second possible explanation for the inaccuracy of the contention estimates is the initial assumption of exponential arrivals of memory accesses to nodes. In many programs, remote memory accesses tend to occur in a bursty pattern, at the beginning of specific phases of the parallel computation. We investigated a few more alternatives to improve the accuracy of the contention estimation methodology in this aspect. The first was to sample memory accesses at a finer granularity. Instead of calculating estimates of inter-arrival times of memory accesses using the complete execution trace of the program, we calculate estimates of inter-arrival times for different phases of the program. Phases can be identified using program-specific constructs. In shared-memory codes, phases are naturally separated by loop boundaries. The intuition is that the communication intensity of a parallel program changes at different stages of the computation. Sampling memory access traces at different phases of execution can help us obtain more accurate characterizations and models of the remote memory access pattern. This will help us isolate the overhead of contention within specific regions of parallel code.

We applied phased sampling of memory accesses in the codes. We set loop boundaries as phase boundaries. Consecutive loops with the same access pattern and the same accessed data set were coalesced in a single phase. We measured the execution time of each phase, calculated a separate Poisson process per phase and used this process to estimate the overhead of contention on a phase-by-phase basis. Our preliminary evaluation shows that by calculating estimates of contention in this manner, we can improve the accuracy of the overall estimate of contention (measured as overhead in execution time) by 4–13%. The average error of the prediction of contention, estimated as a fraction of the difference in execution times between the *FT* and the *W* versions was reduced from 19.4% to 18.2%.

Another idea which we plan to explore is to use a different process to model the arrivals of memory accesses. In particular, we are considering the use of Markov-modulated Poisson Processes (MMPP) [15]. These processes are doubly stochastic Poisson processes, with arrival rates controlled by a two-state Markov process.

MMPP have proven to be quite successful in modelling bursty traffic patterns in communication systems, because they can capture important correlations between inter-arrival times. MMPP can be used as an alternative to identify implicitly phased behavior in parallel programs.

3. BALANCING MEMORY LOAD AND RESOLVING CONTENTION

We argue that the problem of resolving remote memory access contention on hardware DSM multiprocessors can be indirectly addressed by balancing the remote memory accesses across the nodes of the system, so that the traffic of coherence messages for accessing remotely located data is distributed evenly across the inter-connection network. Contention is an artifact of having multiple processors from different nodes accessing the same page in memory. By balancing the remote memory accesses, we reduce the likelihood of contention in two ways. First, if we reduce the number of remote accesses to nodes that concentrate disproportionately large fractions of remote accesses, we reduce the probability of contention on these nodes. Second, if we balance the remote accesses across the nodes of the system, we minimize the average number of nodes that contend to access the same page. Note that if an algorithm which balances memory load improves the performance of the program, it does not necessarily do so by resolving contention. The improvement may come from reducing the number of remote memory accesses to pages that concentrate relatively high remote memory access latency. A better correlation between the effects of the algorithm and contention can be obtained by estimating contention using the methodology described in Section 2.

3.1. Outline of the Algorithm

The main idea of the algorithm is to identify *spikes* of remote memory accesses in the memory access histograms. These spikes imply that some nodes concentrate disproportionately large numbers of remote memory accesses. These nodes are more likely to contain hot spots. The algorithm attempts to cut down the height of the spikes by redistributing remote memory accesses. We assume that a histogram of the memory accesses per node is available either at runtime, or after a full execution of the program.

The tool for balancing remote memory accesses in the algorithm is dynamic page migration [16]. The algorithm searches for hot spots at page-level granularity and migrates pages so that hot spots are distributed, rather than concentrated in few 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. These algorithms use competitive migration criteria, which compare the number of accesses from each node to each page in memory and migrate pages to the nodes which access them more frequently [21]. Using the notation developed in Section 2, a competitive migration algorithm would migrate a page from n_i to n_j if for the specific page, $R'_{ji} > L_i$. Recall that R'_{ji} denotes the number of remote accesses from n_j to n_i and L_i is the number of local accesses from n_i to the page.

There are two important differences between the algorithm and previously proposed competitive page migration algorithms [21]. The first is that the algorithm does not migrate pages based on the number of accesses to individual pages. It migrates pages based on the accumulated latency of accesses to each node. The algorithm identifies the nodes in which the accumulated latency of remote memory

accesses is higher than the accumulated latency of local memory accesses. It migrates out of these nodes pages which have higher remote rather than local memory access latency, even if these pages are placed together with the processor which accesses them more frequently. These pages are migrated to nodes with less memory load (in terms of remote memory access latency) than the original home nodes of the pages. Migrating these pages poses a trade-off between locality and memory load. Locality of accesses to isolated pages may be sacrificed to improve locality of accesses to nodes as a unit.

We account for differences between the latencies of remote accesses, by weighing memory access frequencies with memory access latencies. On ccNUMA multiprocessors, the latency of a remote access depends on the distance between the accessing and the accessed node in the interconnection network. Since more costly remote memory accesses are more important to localize, the algorithm multiplies the access frequency for each pair of source-target nodes ($R'_{ki}, k \neq i$) with the latency of a memory access from k to i (r_{ki}), which is given by equation 12.

The second difference between the algorithm and competitive migration algorithms is that the latter are on-line algorithms, while ours is an off-line algorithm. Competitive page migration algorithms retrieve snapshots of page access counters at random points of execution⁴ and migrate pages using access traces obtained from these snapshots. Our algorithm uses a complete trace of the memory accesses of the program, instead of snapshots. If the program is periodic and time-invariant (i.e. it repeats the same computation for a number of iterations), the trace can be obtained at runtime, after executing one period. Otherwise, the trace is obtained from a full execution of the program. The trace is used to produce the histograms of memory accesses on a per node basis, after accumulating the local/remote memory accesses to each page on each node. In both cases, the algorithm attempts to balance memory load for the program as a whole, rather than for specific phases in the program. It is possible to apply the algorithm for on-line optimization, if the program is periodic and the granularity of the computation is coarse enough to amortize the cost of the algorithm.

3.2. The Algorithm

The algorithm checks if the accumulated latency of remote memory accesses to each node exceeds the accumulated latency of local memory accesses, i.e. if

$\sum_{k=1, k \neq i}^N R'_{ki} r_{ki} > L_i l_i$. The idea behind this heuristic is that migrating pages from a node which concentrates excessive remote access latency to a node which concentrates low remote access latency will increase the local/remote access ratio of the former and improve memory load balance. This heuristic is the key for resolving potential hot spots. Even if a page is actually placed together with the processor which accesses it more frequently, the accumulated latency of remote memory accesses to the node that hosts the page may exceed the latency of local memory accesses.

The algorithm (Figure 8) sorts the nodes in descending order of latency of remote memory accesses and selects the node which concentrates the highest remote memory access latency (line 1). It migrates out of this node each page for which there exists at least one node with higher latency of remote accesses to the page (lines 3–5). If there are more than one such nodes, the page is migrated to the node with the highest latency of remote accesses to the page (line 4). Note that this part

of the algorithm is identical to a competitive page migration algorithm and will improve the memory access locality of the program if there are pages that satisfy the criterion $R'_{ji,p}r_{ji,p} > L_{i,p}l_{i,p}$, where p denotes the page number. Priority is given to these pages, since migrating them achieves simultaneously two goals, balancing memory load and improving locality. Unfortunately, this criterion in itself neglects pages that concentrate large numbers of remote accesses from multiple processors, even though none of these processors sees higher latency than the processors on the home node. In the experiments presented in Section 4, we show that the criterion does not suffice to balance memory load in irregular codes.

After considering the pages with remote sharers that see higher latency than the home nodes, the algorithm considers pages for which the accumulated latency of remote accesses is higher than the accumulated latency of local accesses. These pages are sorted in descending order of accumulated remote access latency (line 8). The pages are checked from the top to the bottom of the sorted list and for each page, the algorithm finds the node with the minimum accumulated latency of remote accesses (lines 10–11). It then checks if migrating the page to that node will increase the latency of remote accesses beyond the latency of remote accesses to the node that hosted the page before the migration (lines 12–14).

We explain the last step in more detail. Recall that $R'_{qi}r_{qi}$ denotes the latency of remote accesses from n_q to n_i . If a page p is migrated from n_i to n_k , the local memory accesses from n_i to the page ($L_{i,p}$) will be converted to remote accesses from n_i to n_k . The latency of these $L_{i,p}$ accesses will be $L_{i,p}r_{ik}$. The remote accesses from other nodes to the same page, while the page was on n_i ($R'_{qi,p}$, $q \neq i, k$) will remain remote, but they will be redirected to node n_k , therefore their latency will be $\sum_{q=1, q \neq i, k}^N R'_{qi,p}r_{qk}$. The previously remote accesses from n_k ($R'_{ki,p}$) to the page will be converted to local, therefore their latency must not be accounted for as remote memory access latency. Finally, the rest of the remote memory accesses to node n_k before the migration ($\sum_{q=1, q \neq k}^N R'_{qk}$) and their latency (r_{qk}) will remain unchanged. Therefore, the new latency of remote accesses to node n_k will be $\sum_{q=1, q \neq k}^N R'_{qk,p}r_{qk} + L_{i,p}r_{ik} + \sum_{q=1, q \neq k, i}^N R'_{qi,p}r_{qk}$, which must be less than the latency of remote accesses to n_i , the home node of the page before the migration, which is given by $\sum_{q=1, q \neq i}^N R'_{qi}r_{qi}$.

The algorithm is invoked in an iterative manner on each node, in order of descending remote memory access latency. Iterations of the algorithm are executed until the maximum latency of remote memory accesses can not be reduced further.

3.3. Implementation Issues

There are two important implementation issues concerning the algorithm, namely the collection of memory access histograms and the invocation points.

Memory access histograms can be collected from hardware counters, if the multiprocessor's memory system has hardware vectors of access counters attached to physical pages. We are aware of at least two commercial systems which provide page access counters in hardware, the SGI Origin2000 [12] and the Sun Wildfire [8]. These counters count the number of accesses from each node to each page in physical memory and are dumped to buffers maintained by the operating system

either upon overflow, or upon exceeding a programmable threshold. In both cases, the counters interrupt a CPU on the local node and the operating system's ISR stores the counters in local buffers. On the Origin2000, the user can access both the software and the hardware page access counters at user-level via the */proc* interface.

On systems without hardware page access counters, it is possible to implement software access counters by instrumenting memory accesses. The idea is to instrument all loads and stores of the program or a representative subset of them at the binary level, record their virtual addresses, identify their mapping to nodes and obtain the histograms from these mappings. Identifying the mapping of virtual addresses to nodes is possible if the system provides the means to control and query the mappings of virtual pages to nodes. All modern DSM multiprocessors with hardware cache coherence that we are aware of provide a NUMA-aware data placement interface to the user, so that the user can explicitly place pages on specific nodes. For example, on the Compaq GS320 [7], a system which does not have hardware page access counters, it is possible to use the NUMA APIs of Tru64 UNIX and obtain accurate information on the placement of ranges of virtual addresses in physical memory. Given this information, software instrumentation for access tracing is possible. Note that the programmer does not need to have access to TLBs. The mappings can be obtained with system calls. The drawback of this solution is that it incurs significant overhead and it is not correct if the programmer needs to use memory access tracing for runtime data redistribution.

Memory access histograms are collected at the end of execution. If the program has an iterative structure (i.e. it repeats the exact same parallel computation for a number of iterations), it suffices to collect the histograms at the end of the first iteration of the program. The notion of a period can be generalized to capture a chain of basic blocks which is executed repeatedly in the program and accounts for most of the execution time. Program periods can be identified using static or dynamic forms of basic block analysis [19]. In many practical cases, particularly in numerical codes, a period is simply an outer iteration which encapsulates the main body of the computation performed in the program. Previous studies have shown that many scientific applications have a strong periodic behavior [19]. The memory access trace of a period can in most cases be a very good approximation of the memory access trace of the program as a whole.

If identifying periods in a program is not possible, or the program does not exhibit periodic behavior at runtime, a memory access trace of the whole program must be used. As stated earlier, we assume that the program has a time-invariant behavior as far as the memory access pattern is concerned. This means that even if the program is not periodic in its internal structure, it has periodic behavior across executions, regardless of the input. Overcoming this restriction is a subject of further investigation.

Using the algorithm for runtime optimization is viable only if the cost of the algorithm does not exceed the gains from balancing the memory load. Although this trade-off can be investigated analytically, we do not discuss the relevant details in this paper. It suffices to mention that the execution time of a period in the program should be in the order of hundreds of milliseconds, to amortize the cost of dynamic page migration.

4. EXPERIMENTAL RESULTS

We evaluate the algorithm with five application benchmarks selected from three benchmark suites. The selected benchmarks are MG from the NAS benchmarks [11], the LG, SL and TS kernels from the Integrated Forecasts System of the European Center for Medium Range Weather Forecasts [23], and SPECclimate, a benchmark included in the SPECchpc96 benchmark suite [20]. MG uses a V-cycle multigrid algorithm for solving discrete Poisson equations on three-dimensional grids of different resolutions. The algorithm starts with an approximate solution in the finest grids, which is projected to progressively coarser grids. The three kernels from the IFS system perform data transpositions between quasi-regular grids that model layers of the earth’s 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. All benchmarks are implemented in OpenMP by third parties and are manually optimized for efficient execution on shared-memory architectures. We tried to isolate the impact of contention and the contention resolution algorithm on these benchmarks. Therefore, we did not apply any additional optimizations for better cache locality or load balancing.

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 the methodology for quantifying contention, we obtained the results shown in Figure 9. The estimates were obtained by executing the benchmarks with first-touch page placement. On the maximum number of processors (128 for MG and SPECclimate, 121 for LG, SL and TS). Note that the overhead of contention is significantly less than the one estimated with the worst page placement, but it is still significant enough to worth improvement. Contention is estimated to account for at least 14% and at most 30% of execution time. First-touch does not achieve a balanced distribution of memory accesses in the codes, but is still the best-performing automatic page placement algorithm on the Origin2000⁵.

We applied the contention resolution algorithm to the benchmarks by linking them with *UPMlib* [17], a runtime system developed to optimize transparently the memory access locality of OpenMP programs running on ccNUMA multiprocessors. *UPMlib* provides intelligent page migration algorithms which can be used to localize memory accesses without placing the burden of manual data distribution on the programmer. In the experiments presented in this section, we used two algorithms implemented in *UPMlib*. The first algorithm uses only competitive page migration, by comparing the local memory access latency to the remote memory access latency from each node to each page in memory. This is equivalent to running lines 1 – 7 in the algorithm shown in Figure 8. The second algorithm executes the full sequence shown in Figure 8. It uses the latency of remote memory accesses per node and the accumulated latency of remote memory accesses to make decisions for migrating pages. Note that no other page migration algorithm of *UPMlib* is activated while the programs are running. Therefore, any performance changes observed in the experiments can only be attributed to the two variants of the contention resolution algorithm.

All programs with which we experimented are iterative. The basic period corresponds to a time step of the parallel computation. The contention resolution algorithm is applied by instrumenting the source code to collect the memory access

histograms after the execution of the first period. The algorithm runs off-line and the optimized versions of the benchmarks are recompiled with the improved page allocation scheme hard-coded in their initialization phases. We do not investigate runtime detection and resolution of contention in this paper.

For each benchmark, we show the execution time of the benchmarks versus the number of processors (Figures 10, 12, 14, 16 and 18) and the memory access histograms of the benchmarks obtained from their execution on 64 processors (Figures 11, 13, 15, 17 and 19). The first version is not linked with *UPMlib* and uses the first-touch page placement algorithm of the Origin2000 to distribute data automatically. The second version is linked with *UPMlib* and uses the page migration algorithm only for competitive page migration based on the memory access latencies from each node to each page in memory. The third version is also linked with *UPMlib* and uses the complete algorithm shown in Figure 8.

The charts show the execution time of the programs, normalized to the execution time of the *FT* versions. The IFS kernels require a square number of processors. Although the speedup of the programs is not relevant for this comparison, we note that all codes scale reasonably well up to 64 processors. MG scales very well up to 128 processors, but the speedup of the other four programs seems to level off beyond 64 processors.

Looking at the memory access histograms when the benchmarks are executed without the contention resolution algorithm, we can easily observe the spikes of remote memory accesses and the irregularity of the memory access patterns. In the three codes from IFS and in SPECclimate, the irregularity of memory accesses is attributed to the structure of the modelled grids. The problem in MG is that first-touch achieves 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 the maximum number of processors, after applying the contention resolution algorithm in full is as much as 28.7% for MG, 24.1% for SPECclimate, 19.9% for LG, 28.4% for SL, and 27.3% for TS. It appears that the algorithm achieves less than the expected improvements only in MG and matches or exceeds the expected improvements in LG, SL, TS and SPECclimate. The improvements from using only competitive page migration are marginal. On the maximum number of processors, execution time is reduced only by 6.1% in MG, 5.2% in LG, 4.2% in SL, 9.8% in TS and 3.9% in SPECclimate.

The results show that competitive page migration is not enough to balance the memory load. Furthermore, they show that memory load balancing is an important performance factor that needs to be addressed in conjunction with memory access locality optimizations. For the irregular codes used in this paper, although locality optimizations for better reuse of the processor caches are crucial for performance, locality optimizations for better clustering of memory accesses upon cache misses are not as effective due to memory load imbalance.

The histograms of memory accesses show that different benchmarks exhibit different forms of memory load imbalance. In MG, imbalance seems to occur in a very narrow range of the address space, causing contention in two nodes. In SPECclimate, most remote memory accesses are scattered across two thirds of the nodes. In the three irregular kernels, approximately half of the nodes concentrate more than 90% of remote memory accesses.

The histograms of memory accesses obtained after applying the memory load balancing algorithm 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 home node. Consequently, the algorithm increases the fraction of local accesses by 23% in TS, 32% in LG and 54% 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 11). The improvement comes from balancing memory load. In SPECclimate, the algorithm does not reduce significantly the number of remote memory accesses. The improvements come from balancing the memory load 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. We conducted a few preliminary experiments in which we applied the phased contention estimation process outlined in Section 2.4. In these experiments, we found that the overhead of contention on execution time can vary as much as one order of magnitude across different phases of the program. We also found that the actual placement of pages derived if we apply the algorithm for local optimization of the phases of high contention differs significantly from the placement of pages derived if we apply the algorithm for the program as a whole. Unfortunately, the cost of redistributing pages for local optimization of page placement across phases seems to be too high to afford in this benchmark. Therefore, we were not able to improve the execution time and the memory access pattern of SPECclimate further with phase-based optimization.

The competitive page migration algorithm shows similar behavior in the memory accesses of all five benchmarks. The algorithm appears to reduce significantly the number of remote memory accesses per node. The reductions range between 33% and 74%. However, it does not actually change the memory access pattern, in the sense that memory accesses remain highly unbalanced. Correlating this observation with the effect of reducing remote memory accesses on execution time, we conclude that remote memory access contention and memory load imbalance can have a more significant impact on performance than remote memory access frequency. Optimizing data placement for the former is often more important than optimizing data placement for the latter. We also point out that the contention resolution algorithm which uses accumulated memory access latencies per page does not overlook memory access locality. The memory access traces show that the total number of remote memory accesses tends to reduce as a side-effect of the algorithm.

Finally, to investigate whether the algorithm imposes high overhead when contention resolution and memory load balancing are not needed, we ran experiments with two regular applications from the NAS benchmarks, namely BT and SP. These two benchmarks solve Navier-Stokes equations in three dimensions, using different factorization methods. The benchmarks use regular sequential and constant-stride accesses to shared arrays. Figure 20 shows the memory access patterns of the two programs, taken from executions on 64 processors of the Origin2000. The access patterns of the benchmarks are very well balanced and the benchmarks are not expected to see any benefit from the contention resolution algorithm. Therefore, they can serve as tests for the overhead of the algorithm on performance. Figure 21 shows the execution times of the benchmarks with first-touch, without linking with *UPMlib* and with first-touch after linking with *UPMlib* and activating the library

to migrate the pages selected by the contention resolution algorithm. The overhead of *UPMlib* is minimal and adds no more than 2.5% to the execution time of the benchmarks. These experiments demonstrate that the algorithm is not intrusive for regular codes.

5. RELATED WORK

Several papers have pinpointed contention as one of the most important problems for the scalability of parallel programs (e.g. [1, 4, 6]). However, most of these works quantified contention either by detailed simulation or with analytical modelling. Bletloch 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. They extended the BSP model to account for contention and validated their extension on the Cray C90 and J90 systems. Frank et.al. [6] extended the LogP model to account for contention in both distributed and shared memory multiprocessors. Dai and Panda [4] 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 overhead of contention on real programs by directly executing them on real hardware.

Hristea and Lenoski [9] used microbenchmarks to measure the overhead 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. As discussed in Section 2, these microbenchmarks are effective in measuring contention as an architectural parameter, but can not quantify contention in real programs.

The works of De Lara et.al. [5] and Lu et.al. [14] appear to be more closely related to the contribution of this paper. They present means to quantify contention in parallel programs running on shared virtual memory 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. [5] 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 which 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. [14] propose a method which replicates sequential code that precedes parallel sections, so that contention at the beginning of parallel sections is avoided. Our work differs in that it is applicable to hardware cache-coherent DSM multiprocessors, which have some fundamental architectural differences compared to clusters with shared virtual memory.

There is a significant body of work on dynamic page migration for NUMA multiprocessors [2, 3, 18, 21]. Previous work has considered page migration as a tool to optimize the locality of memory accesses, so that the overall number of remote memory accesses in the program is reduced. Real implementations of page

migration engines use an interrupt based-mechanism which periodically checks the access counters of one page and assesses if the page should migrate or not. The counters are decayed progressively so that the page migration engine is not biased by obsolete page access history. We use page migration to balance the remote memory accesses on a per-node rather than a per-page basis. We also employ page migration as a whole-program optimization technique, rather than a runtime optimization method with temporal scope.

6. CONCLUSIONS AND FUTURE WORK

We have presented a simple methodology for quantifying the overhead 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 or software page access counters. Driven by this methodology, we proposed an algorithm that balances memory load and alleviates contention. The algorithm integrates a locality-sensitive page migration criterion with a criterion which balances the number of remote memory accesses and the associated communication traffic across DSM nodes. We have validated experimentally the accuracy of the methodology and have shown that contention has significant overhead, which may account for as much as 34% of execution time in programs with irregular memory access patterns. We have also presented experiments that prove the effectiveness of the algorithm.

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

ACKNOWLEDGMENT

The author would like to thank the referees for their constructive comments on earlier versions of this paper.

REFERENCES

- [1] G. Blleloch, 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] W. Bolosky, R. Fitzgerald, and M. Scott. Simple but Effective Techniques for NUMA Memory Management. In *Proc. of the 12th ACM Symposium on Operating System Principles (SOSP'89)*, pages 19–31, Litchfield Park, Arizona, Dec. 1989.
- [3] R. Chandra, S. Devine, A. Gupta, and M. Rosenblum. Scheduling and Page Migration for Multiprocessor Compute Servers. In *Proc. of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, pages 12–24, San Jose, California, Oct. 1994.

- [4] 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.
- [5] 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.
- [6] 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.
- [7] K. Gharachorloo, M. Sharma, S. Steely, and S. V. Doren. Architecture and Design of AlphaServer GS320. In *Proc. of the 9th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'IX)*, pages 13–24, Cambridge, Massachusetts, Nov. 2000.
- [8] 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.
- [9] C. Hristea, D. Lenoski, 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.
- [10] 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.
- [11] 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.
- [12] 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.
- [13] 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.
- [14] 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.

- [15] K. Meier-Hellstern. A Fitting Algorithm for Markov-modulated Poisson Processes Having Two Arrival Rates. *European Journal of Operational Research*, 29:370–377, 1987.
- [16] 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.
- [17] 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.
- [18] C. Scheurich and M. Dubois. Dynamic Page Migration in Multiprocessors with Distributed Global Memory. *IEEE Transactions on Computers*, 38(8):1154–1163, Aug. 1989.
- [19] T. Sherwood, E. Perelman, and B. Calder. Basic Block Analysis to Find Periodic Behavior and Simulation Points in Applications. In *Proc. of the 2001 International Conference on Parallel Architectures and Compilation Techniques (PACT'2001)*, pages 3–14, Barcelona, Spain, Sept. 2001.
- [20] Standard Performance Evaluation Corporation. SPEC HPC96 Documentation. <http://www.spec.org/hpg>, Dec. 2001.
- [21] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum. Operating System Support for Improving Data Locality on CC-NUMA Compute Servers. In *Proc. of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 279–289, Cambridge, Massachusetts, Oct. 1996.
- [22] H. Wasserman, O. Lubeck, L. Yuo, and F. Bassetti. Performance Evaluation of the Origin2000: A Memory-Centric Characterization of the LANL ASCI Programs. In *Proc. of ACM/IEEE Supercomputing'97: High Performance Networking and Computing Conference (SC'97)*, San Jose, California, Nov. 1997.
- [23] P. White. IFS Documentation: Part VI, Technical and Computational Procedures. Technical Report CY21R4, European Centre for Medium-Range Forecasts, Feb. 2000.

```

1  #define work(dummy,j) ((j) += (dummy))
2
3  void initialize_a(size) {
4    for (i = 0; i < size; i += stride)
5      a[i] = &a[i + stride];
6  }
7
8  void microbench () {
9    s = start_time();
10   j = a;
11   for (i = 0; i < num_reads; i++) {
12     j = (int *)*j;
13     for (k = 1; k < num_iters; k++)
14       work(dummy,j);
15   }
16   e = end_time();
17   return (num_reads*cache_line_size)/(e - s)
18 }

```

FIG. 1 *Simplified version of the microbenchmark used for measuring memory latencies under different levels of contention. Variable declarations are omitted for brevity. The variable `dummy` is set to 0, so that the pointers chased in line 12 are not corrupted. Compiler optimizations are deactivated to avoid squashing the instructions that update `j`.*

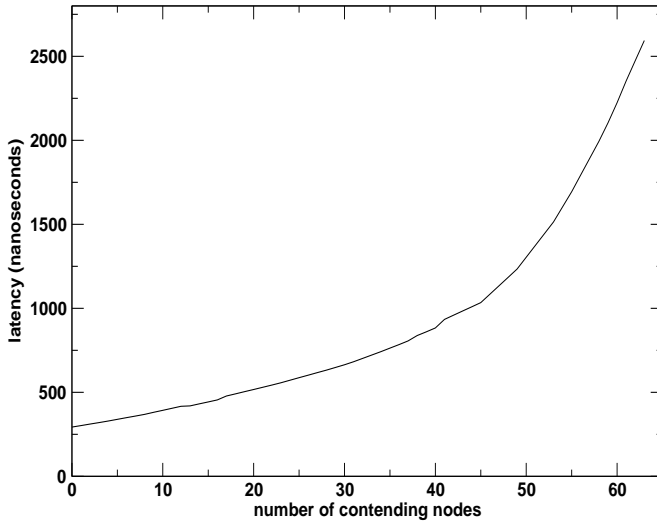
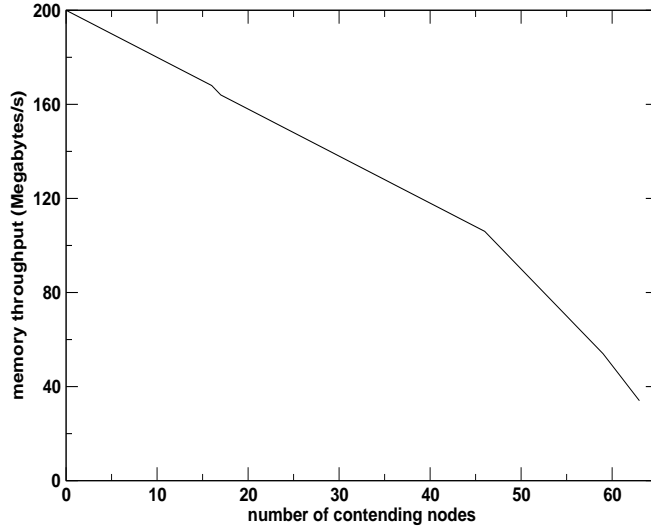


FIG. 2 *Impact of contention on memory throughput and memory latency on a 128-processor Origin2000. To measure memory throughput, the master throughput without contention is set to 200 Megabytes/s. The slave throughput is set to 10 Megabytes/s. To measure latency, the master executes dependent back-to-back memory accesses without executing work in the dummy loop. The horizontal axis shows the number of contending nodes.*

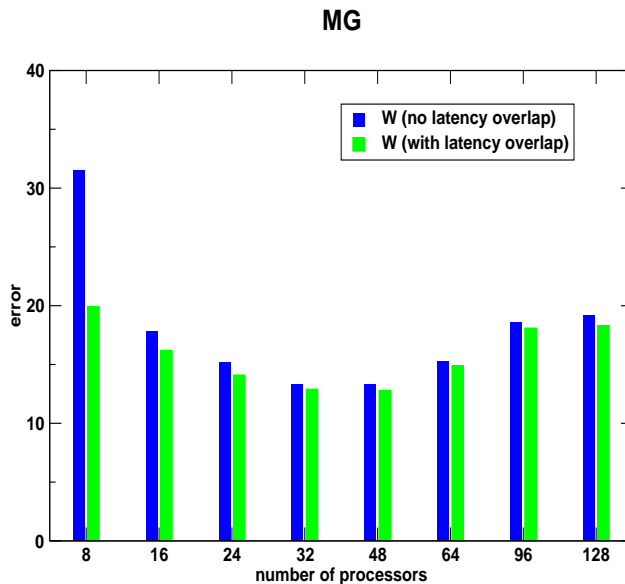
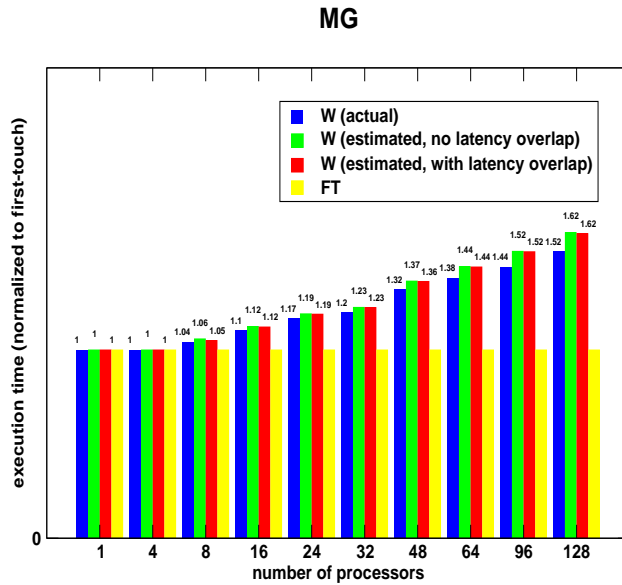


FIG. 3 The top chart shows the estimated execution time of the W version of MG. The chart shows also the actual execution time of the W version and the estimated execution time of the W version, if the overlapped memory latency is subtracted from the overhead of contention. All times are normalized to the execution time of the FT version. The bottom chart shows the error in the estimates, as a fraction of the difference in execution time between the FT and the W versions.

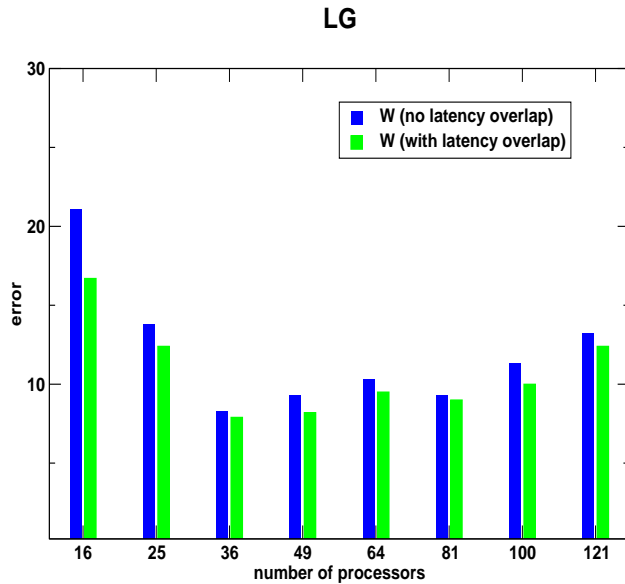
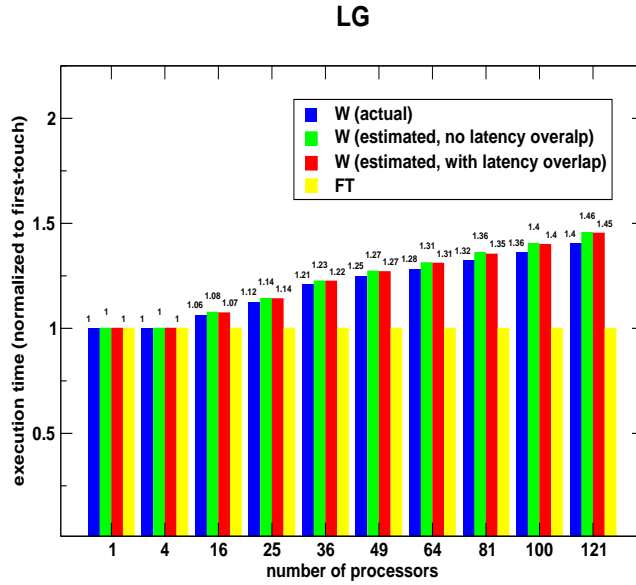


FIG. 4 The top chart shows the estimated execution time of the W version of LG. The chart shows also the actual execution time of the W version and the estimated execution time of the W version, if the overlapped memory latency is subtracted from the overhead of contention. All times are normalized to the execution time of the FT version. The bottom chart shows the error in the estimates, as a fraction of the difference in execution time between the FT and the W versions.

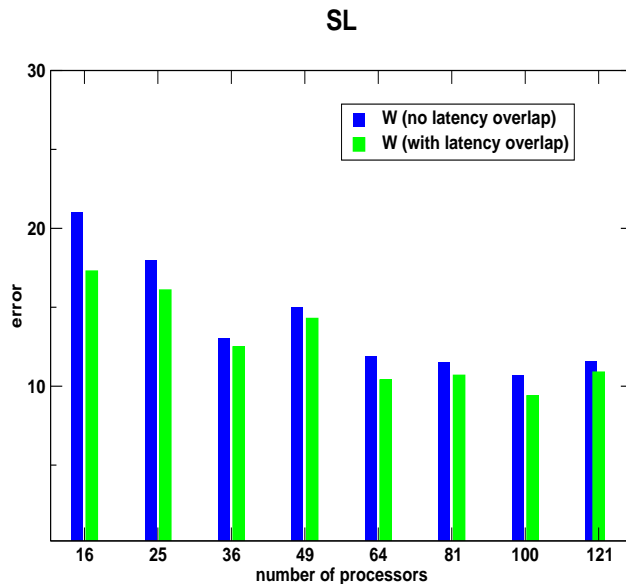
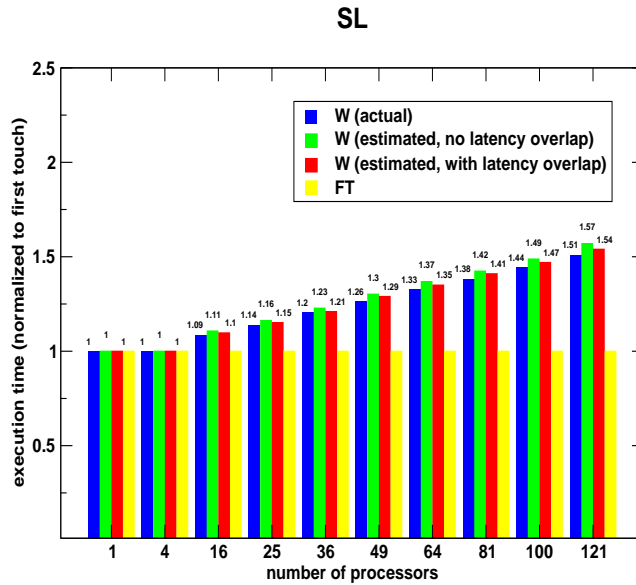


FIG. 5 The top chart shows the estimated execution time of the *W* version of *SL*. The chart shows also the actual execution time of the *W* version and the estimated execution time of the *W* version, if the overlapped memory latency is subtracted from the overhead of contention. All times are normalized to the execution time of the *FT* version. The bottom chart shows the error in the estimates, as a fraction of the difference in execution time between the *FT* and the *W* versions.

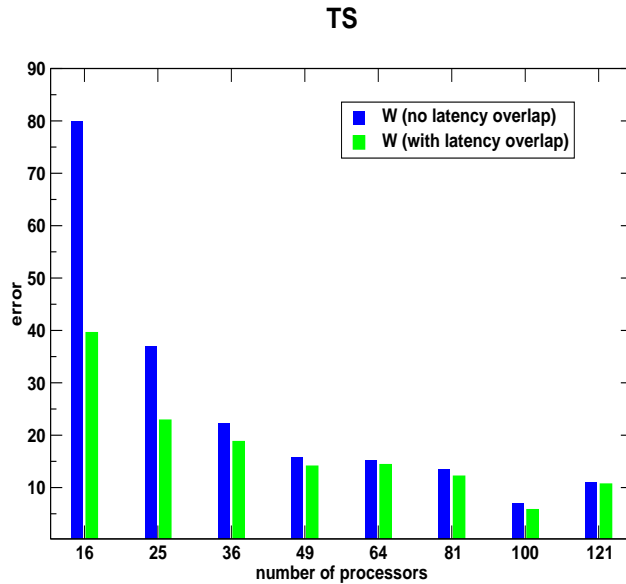
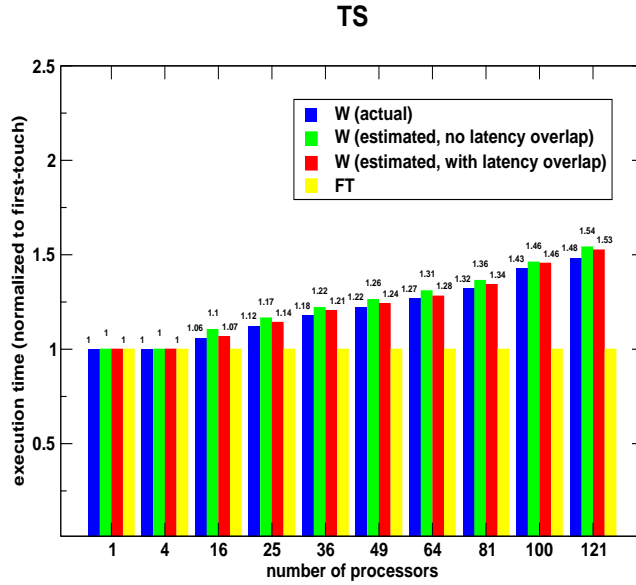


FIG. 6 The top chart shows the estimated execution time of the *W* version of *TS*. The chart shows also the actual execution time of the *W* version and the estimated execution time of the *W* version, if the overlapped memory latency is subtracted from the overhead of contention. All times are normalized to the execution time of the *FT* version. The bottom chart shows the error in the estimates, as a fraction of the difference in execution time between the *FT* and the *W* versions.

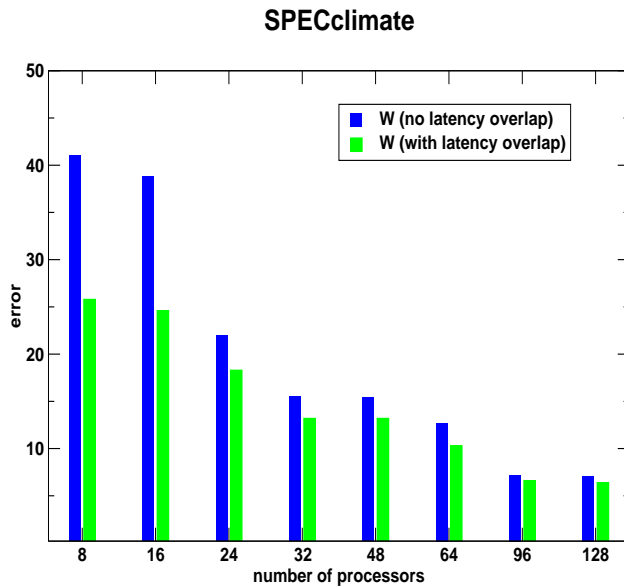
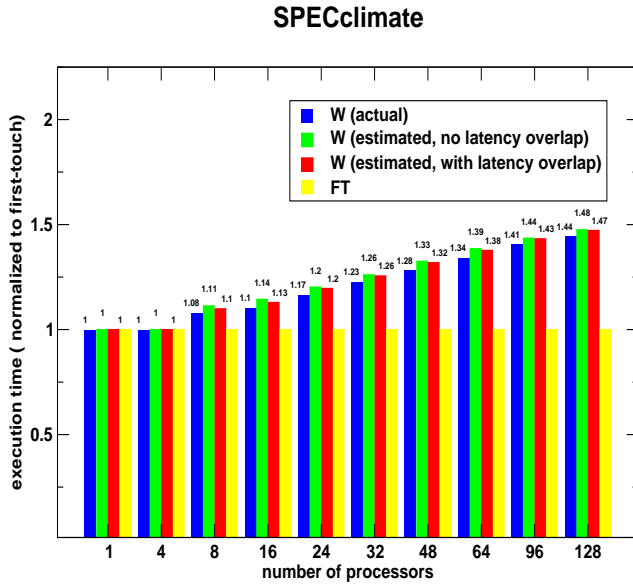


FIG. 7 The top chart shows the estimated execution time of the *W* version of SPECclimate. The chart shows also the actual execution time of the *W* version and the estimated execution time of the *W* version, if the overlapped memory latency is subtracted from the overhead of contention. All times are normalized to the execution time of the *FT* version. The bottom chart shows the error in the estimates, as a fraction of the difference in execution time between the *FT* and the *W* versions.

- (1) find n_i such that $\forall n_j, j = 1 \dots N, j \neq i, \sum_{q=1, q \neq i}^N R'_{qi} r_{qi} > \sum_{q=1, q \neq j}^N R'_{qj} r_{qj}$
- (2) for each page p located in n_i
- (3) if $\exists j, j \neq i, R'_{ji,p} r_{ji} > L_{i,p} l_i$
- (4) select $n_j, \forall n_k, k \neq i, k \neq j, (R'_{ji,p} r_{ji} > R'_{ki,p} r_{ki})$
- (5) migrate the page to n_j
- (6) endif
- (7) end foreach
- (8) sort the pages in n_i in descending order of $\sum_{j=1, j \neq i}^N R'_{ji,p} r_{ji}, j = 1 \dots N$
- (9) for each page p located in n_i from the top of the sorted list
- (10) if $\sum_{q=1, q \neq i}^N R'_{qi,p} r_{qi} > L_{i,p} l_i$
- (11) find n_k so that $\forall n_j, j = 1 \dots N, j \neq i, j \neq k, \sum_{q=1, q \neq k}^N R'_{qk} r_{qk} < \sum_{q=1, q \neq k}^N R'_{qj} r_{qj}$
- (12) if $\sum_{q=1, q \neq k}^N R'_{qk} r_{qk} + L_{i,p} r_{ik} + \sum_{q=1, q \neq i, k}^N R'_{qi,p} r_{qk} < \sum_{q=1, q \neq i}^N R'_{qi} r_{qi}$
- (13) migrate p to n_k
- (14) endif
- (15) endif
- (16) end for each

FIG. 8 The page migration algorithm for resolving contention. This algorithm runs repeatedly, until $\max_i \sum_{q=1, q \neq i}^N R'_{qi} r_{qi}$ cannot be reduced further. $L_{i,p}$ denotes the number of local accesses from node i to page p (assuming that the page is located in node i) and $R'_{j,i,p}$ denotes the number of remote accesses from node j to page p .

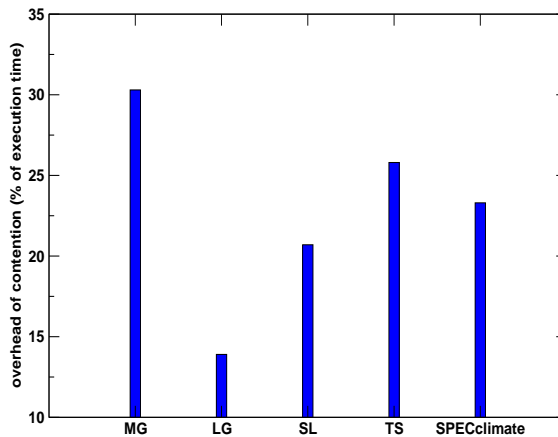


FIG. 9 *Estimated overhead of contention on the parallel execution time of the benchmarks on the Origin2000. The contention overhead is calculated from executions of MG and SPECClimate on 128 processors and LG, SL and TS on 121 processors. The estimates are obtained from equation 13 and are plotted as percentage of execution time of the FT versions of the benchmarks.*

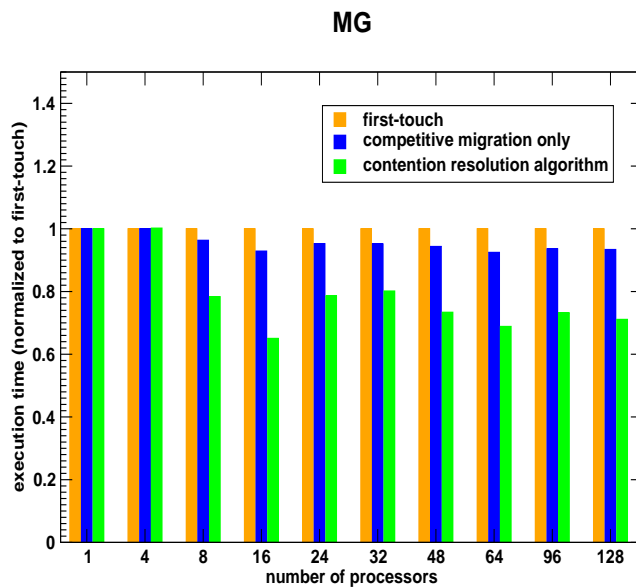


FIG. 10: Execution time of MG with competitive page migration and the complete contention resolution algorithm. The execution times are normalized to the execution time of the FT version.

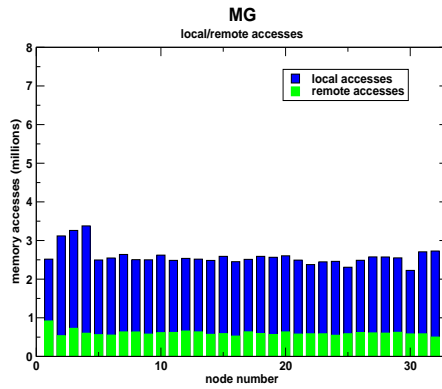
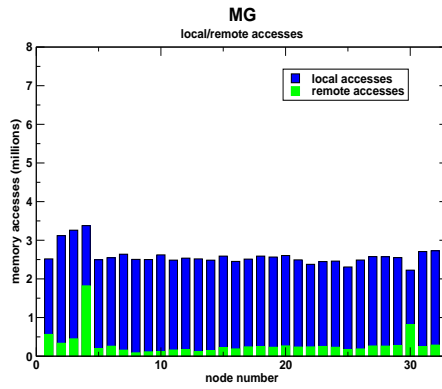
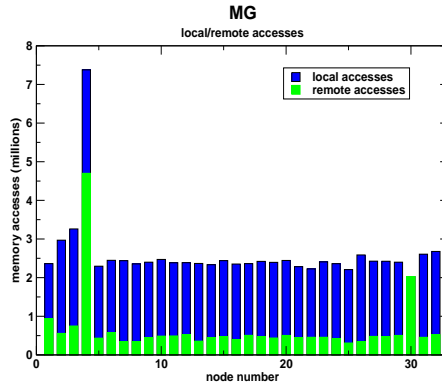


FIG. 11: Access traces of MG before resolving contention (top), after resolving contention by using only competitive page migration (middle) and after resolving contention with the complete contention resolution algorithm.

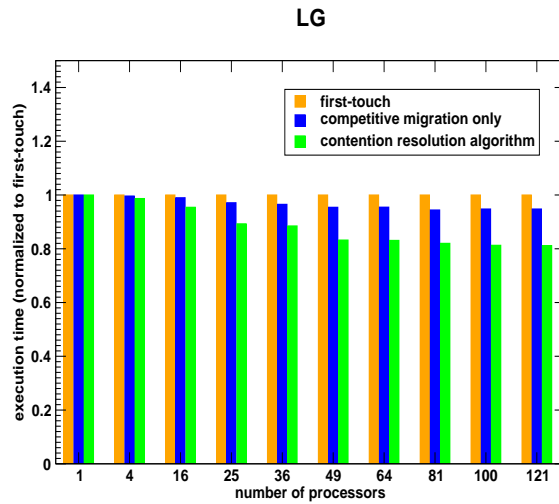


FIG. 12: Execution time of *LG* with competitive page migration and the complete contention resolution algorithm. The execution times are normalized to the execution time of the *FT* version.

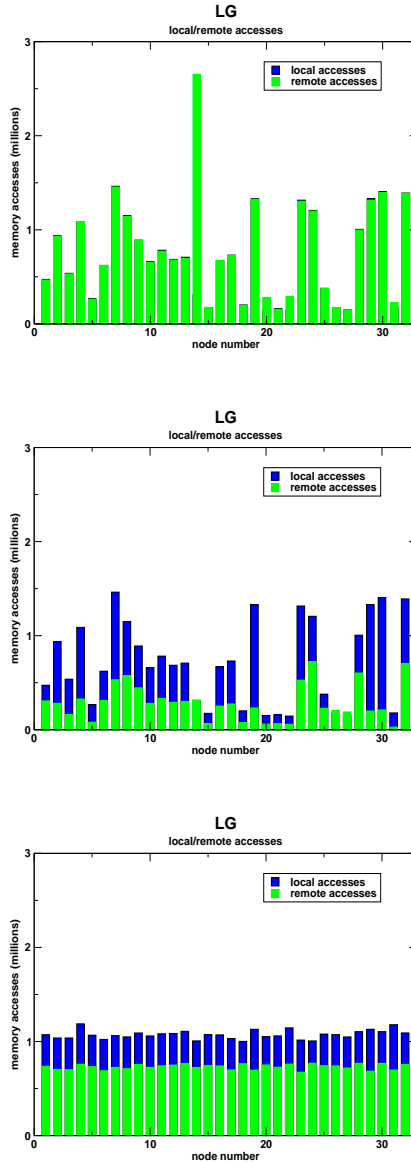


FIG. 13: Access traces of LG before resolving contention (top), after resolving contention by using only competitive page migration (middle) and after resolving contention with the complete contention resolution algorithm.

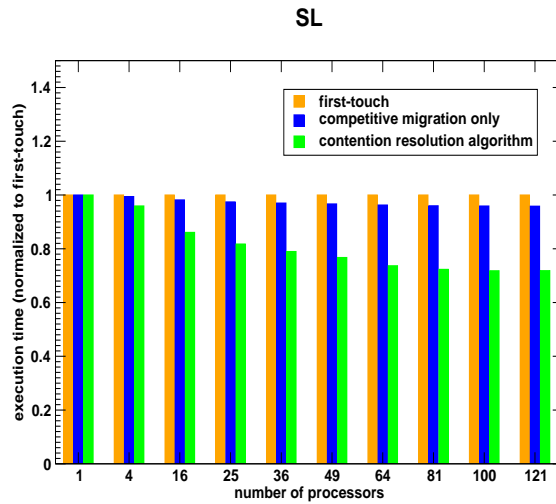


FIG. 14: Execution time of *SL* with competitive page migration and the complete contention resolution algorithm. The execution times are normalized to the execution time of the *FT* version.

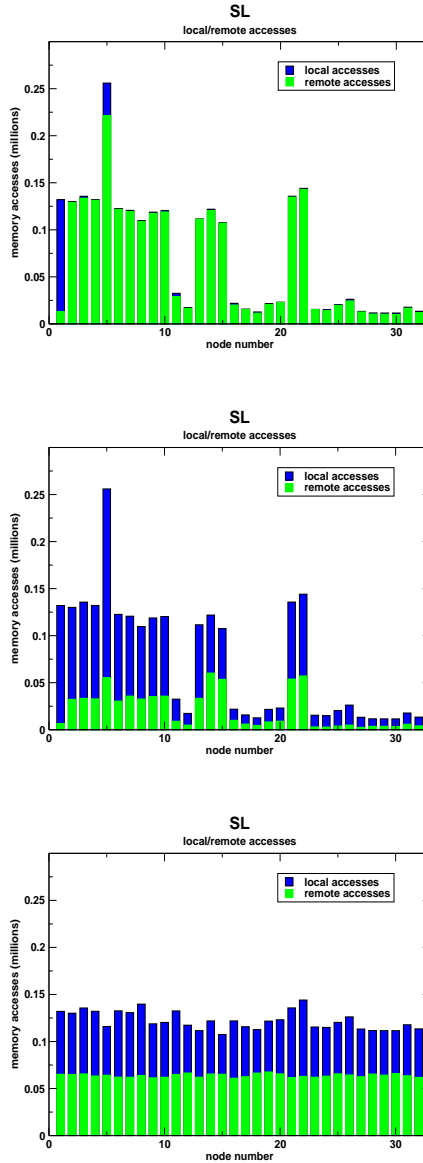


FIG. 15: Access traces of SL before resolving contention (top), after resolving contention by using only competitive page migration (middle) and after resolving contention with the complete contention resolution algorithm.

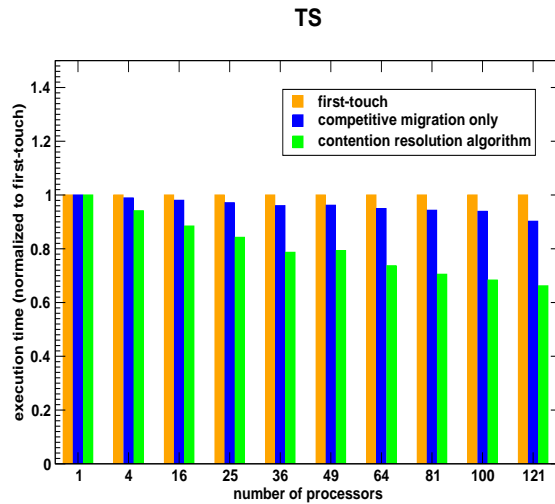


FIG. 16: Execution time of *TS* with competitive page migration and the complete contention resolution algorithm. The execution times are normalized to the execution time of the *FT* version.

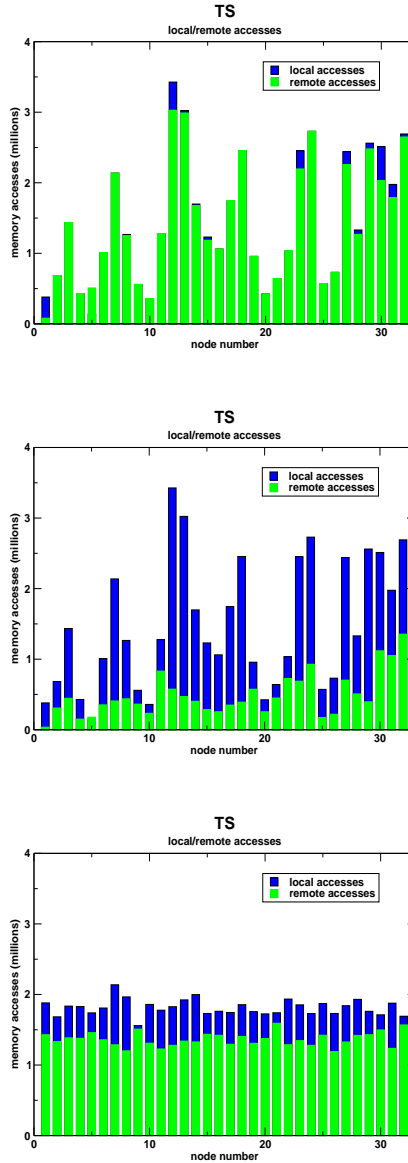


FIG. 17: Access traces of TS before resolving contention (top), after resolving contention by using only competitive page migration (middle) and after resolving contention with the complete contention resolution algorithm.

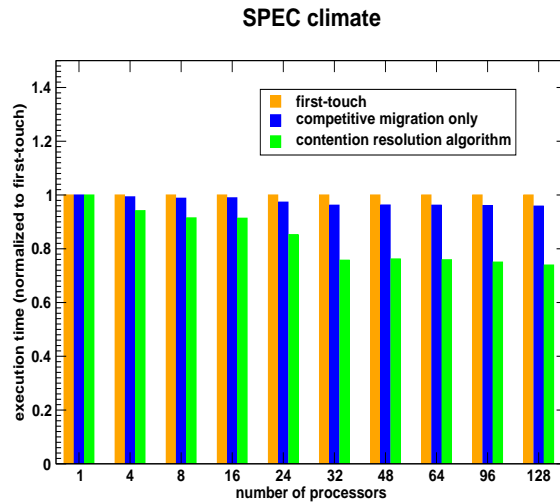


FIG. 18: Execution time of SPECclimate with competitive page migration and the complete contention resolution algorithm. The execution times are normalized to the execution time of the FT version.

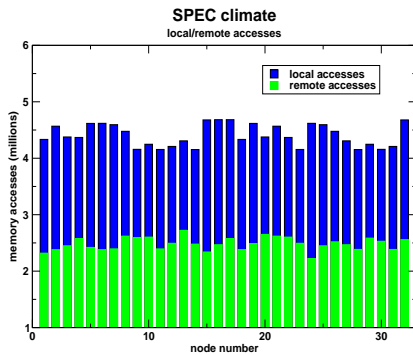
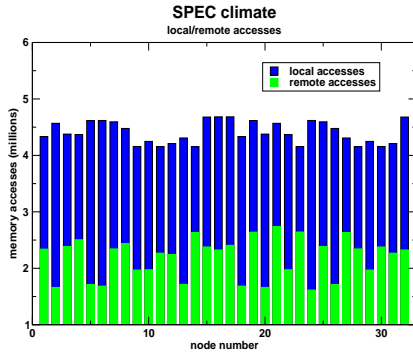
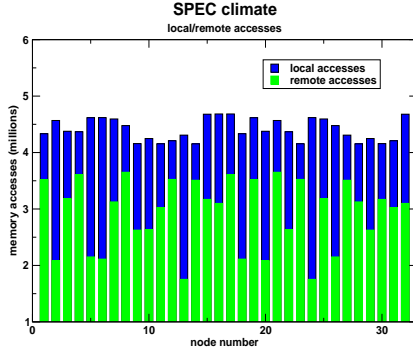


FIG. 19: Access traces of SPECclimate before resolving contention (top), after resolving contention by using only competitive page migration (middle) and after resolving contention with the complete contention resolution algorithm.

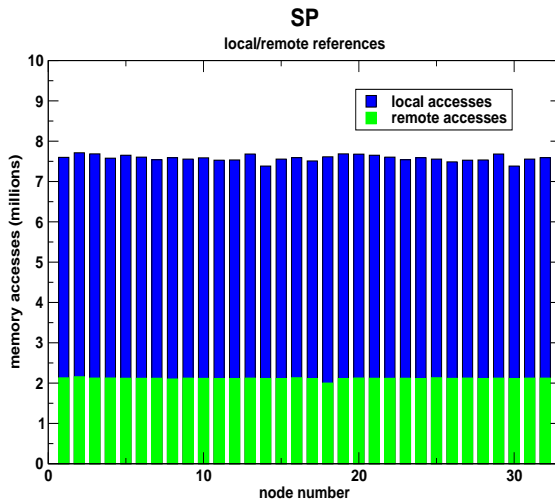
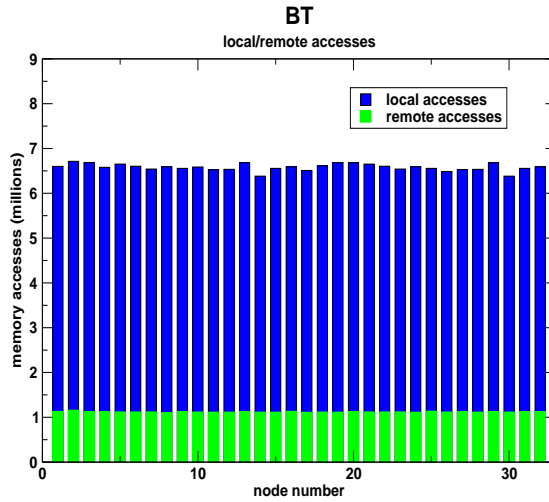


FIG. 20: Memory access patterns of NAS BT and SP, taken from executions on 64 processors (32 nodes) of the Origin2000.

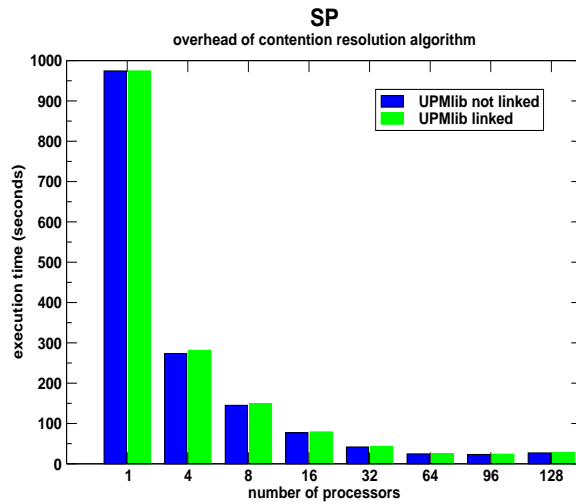
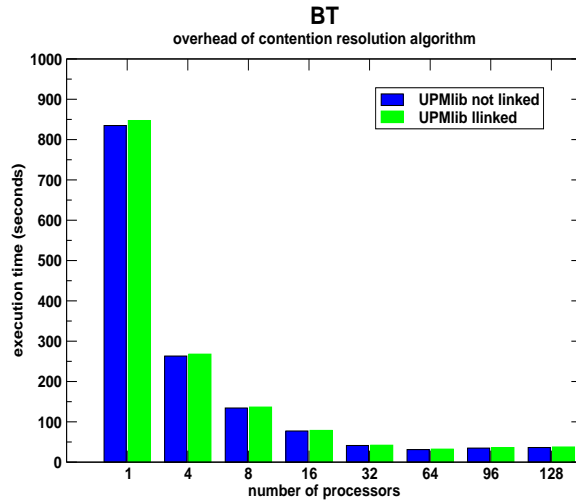


FIG. 21: Execution times NAS BT and SP, when UPMlib is not linked to the codes (left bars) and when UPMlib is linked and activated to execute the page migrations derived from the contention resolution algorithm (right bars).

²We use the term *node* to signify the building block of a cache-coherent DSM multiprocessor. A node may have one or more processors. For the purposes of this study, we consider the accumulated number of remote memory accesses issued by the processors on the same node as a single set of remote memory accesses.

³We are measuring contention due to 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 effects of sharing the bus of the node between two processors. Since this is an architecture-specific feature of the Origin, we do not investigate it further for the purposes of this paper.

⁴Snapshots are retrieved periodically, using a timer interrupt. The sampling points are not correlated with the actual execution status of the program [16].

⁵The Origin2000 provides two automatic page placement algorithms, round-robin and first-touch. First-touch performed better than round-robin with all benchmarks used in this study.