Using Hardware Counters to Automatically Improve Memory Performance

Mustafa M. Tikir Jeffrey K. Hollingsworth

Computer Science Department
University of Maryland
College Park, MD 20742
{tikir,hollings}@cs.umd.edu

Abstract
In this paper, we introduce a profile-driven online page migration scheme and investigate its impact on the performance of multithreaded applications. We use lightweight, inexpensive plug-in hardware counters to profile the memory access behavior of an application, and then migrate pages to memory local to the most frequently accessing processor. Using the Dyninst runtime instrumentation combined with hardware counters, we were able to add page migration capabilities to the system without having to modify the operating system kernel, or to re-compile application programs. This approach reduced the total number of non-local memory accesses of applications by up to 90%. Even on a system with small remote to local memory access latency rations, this resulted in up to 16% improvement in execution time.

1. Introduction

Large cache-coherent, shared-memory servers are widely used for high performance computing. Sun Fire servers now support more than 100 processors[6]. Similarly, the IBM pSeries 680[10] scales up to 24 processors, the HP Superdome[9] scales to 64 processors, the Compaq AlphaServer GS-series[7] scales to 32 processors, and the SGI Origin 2000[12] scales up to more than 100 processors[1].

In this paper, we introduce a dynamic page migration scheme that profiles applications to determine the preferred location for each memory page using hardware counters. We then use system calls to request that the kernel move memory pages to their preferred locations.

In our dynamic page migration algorithm, both profiling and page migration are conducted during the runtime of the applications. For each memory page in the application, we continuously monitor hardware performance counters and collect information about which processor most frequently accesses the page. At fixed time intervals during the application's execution, pages are migrated to memory that is local to the processor that most frequently accesses those.

Although page migration has been extensively studied, our dynamic page migration approach demonstrates several novel features. First, the goal in this paper is not to introduce a new page placement policy. Instead, our goal is to demonstrate that the combination of inexpensive plug-in hardware counters that sample information about interconnect transactions and a simple page migration policy can be used effectively to improve the performance of real applications. The plug-in hardware we used in this research is commercially available from Sun Microsystems.

Second, even on multiprocessor systems with small remote to local memory latency ratios, optimizing page placement still provides substantial benefit to some applications. The remote and local latencies in the Sun Fire 6800 servers we used in our research are approximately 300ns and 225ns respectively (i.e. a remote:local memory latency ratio of 1.33:1). This is quite low for a NUMA system, and previous research on optimizing page placements has tended to focus on systems with much larger remote to local memory latency ratios.

Third, using the information provided by hardware counters that gather information based on physical addresses rather than virtual addresses is accurate enough to guide page migration and eliminates the need for getting virtual address information via hardware counters.

The rest of the paper is organized as follows: Section 2 describes the hardware and software components we used; Section 3 describes the steps of our algorithm for dynamic page migration; Section 4 presents the results of our preliminary experiments; Section 5 presents the results for a series of experiments conducted to evaluate our dynamic page migration approach; Section 6 presents the related work. Finally, Section 7 summarizes our results and presents some conclusions.

[1] All names are trademarks of their owners.

0-7695-2153-3/04 $20.00 (c)2004 IEEE

2. Hardware and Software Components

In this section, we describe the hardware and software components used in this research. We first describe the architecture of the Sun Fire servers. We next describe the Sun Fire Link hardware monitors for the Sun Fireplane system interconnect, which we used to measure memory access behavior. Finally, we give a brief explanation about the system calls that we used.

2.1 Sun Fire Servers

The Sun Fireplane interconnect is Sun's fourth generation of Symmetric Multiprocessor Systems(SMP)[2] interconnect. The Sun Fireplane interconnect is implemented with up to four levels of interconnect logic depending on the number of processors in the system[6]. In medium and large-sized Sun Fire servers, processors and memory units are grouped together on system boards[18]. Each system board contains 4 processors and 4 memory units local to the processors.

In Sun Fire servers, the transfer time to move a data block from a memory unit to the requesting device is non-uniform depending on the system boards the memory unit and requestor are on. Processors on a system board have faster access to the memory banks on the same board (local memory) compared to the memory banks on another board (non-local memory). For example, back-to-back latency measured by a pointer-chasing benchmark in a Sun Fire 6800 server with 750MHz CPUs is around 225ns if the memory is local and 300ns if the memory is non-local.

The Sun Fire 6800 server is a mid-range cc-NUMA architecture based on the UltraSPARC III processors and Sun Fireplane interconnect. It supports up to 24 processors and 24 memory units. The processors and memory units in these servers are grouped into 6 system boards. Each processor has its own on-chip and external caches. Mid-range Sun Fire systems use a single snooping coherence domain that spans all the devices connected to a single Fireplane address bus.

2.2 Sun Fire Link Counters and Bus Analyzer

In a cache-coherent shared-memory multiprocessor, the system interconnect is often the performance-limiting component[15]. Moreover, due to complex interactions among the processors and devices that utilize the system interconnect, it is difficult to analyze the performance of the system interconnect. Due to high transaction rates in these systems, gathering a complete set of interconnect transactions is not practical. Instead, these systems often have additional hardware monitors to count and sample the system transactions. Even though the information collected by these hardware monitors is incomplete, it is still an important source of profiling information [15].

In this paper, we use the Sun Fire Link hardware monitors[15] to gather profiling information for page migration (Shown in Figure 1). The Sun Fire Link hardware monitor counts and samples the transactions on the address bus of the Sun Fireplane interconnect. These monitors were developed as part of a system to cluster multiple systems together, thus they listen to the address bus of the system interconnect.

The Sun Fire Link Counters consist of two 32-bit counter registers, a programmable control register that activates the counters, two registers to filter transactions based on transaction type, and two sets of mask and match registers to filter transactions based on other parameters, such as physical address range and the device identifier. In addition to counter registers, the Sun Fire Link Bus Analyzer has an 8-deep FIFO that records a limited sequence of consecutive interconnect address transactions. Each recorded transaction includes the requested physical address, the requestor device id, and the transaction type. The bus analyzer is configured with mask and match registers to select specific address ranges, processors or transaction types.

Even though the Sun Fire Link counters and bus analyzer provide useful information about the addresses and requesting processors in the transactions, the information is at the level of physical addresses. To accurately evaluate the memory performance of an application, the address transactions have to be associated with virtual addresses used by the application. This requires us to reverse map physical addresses back to virtual addresses. We used the *meminfo* system call in Solaris 9 to create a mapping between physical and virtual memory pages in the applications.

2.3 Solaris 9 Operating System

To ensure the reusability of local caches in the processors, each application thread should be scheduled on the same processor, if possible, throughout its execution[17]. To ensure the reusability of local caches and to accurately count page access frequencies by processors independent of thread scheduling, we explicitly bind application threads to the processors in the system. We bind application threads to the processors in a round robin fashion using the *processor_bind* system call in Solaris.

[2] We use the term SMP to refer to any machine that allows each processor to access all memory in the system regardless whether the memory access time is uniform or not.

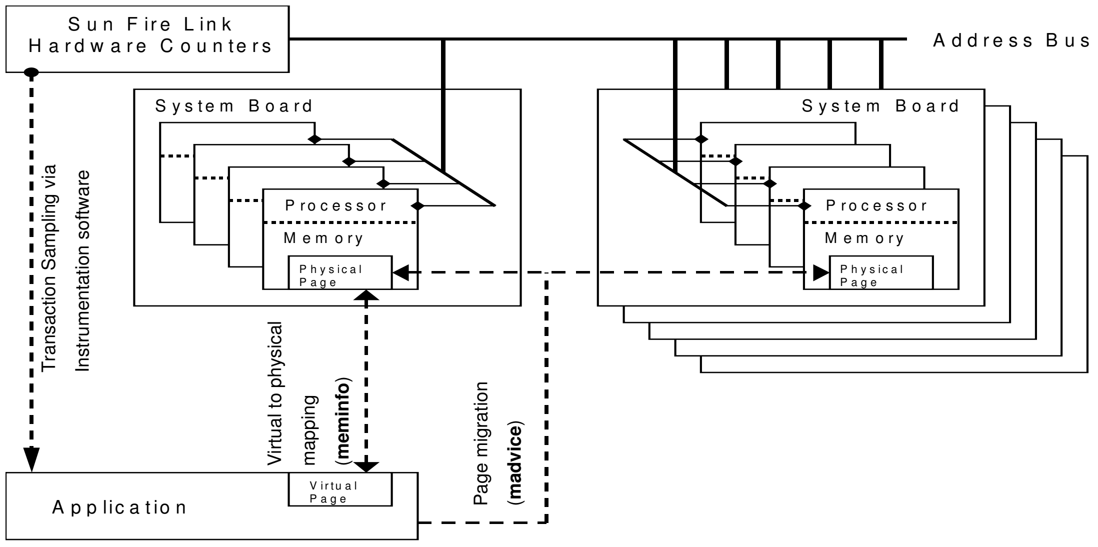

Figure 1 Framework for Our Dynamic Page Migration Scheme

Solaris places each physical memory page into the memory that is local to the first processor that touches the page. However, first-touch page placement may result in non-local placement of a page relative to the processor that accesses it the most, which may have a significant impact on memory performance of the application.

To migrate pages, we use the *move-on-next-touch* feature of the *madvise* system call in Solaris 9. Using the *move-on-next-touch* feature, we request the operating system to move (migrate) a range of virtual memory onto the local memory of the processor that next touches it.

3. Methodology

Our dynamic page migration algorithm consists of two different modules. The first module gathers profiling information using the Sun Fire Link counters and bus analyzer[3]. The second module migrates memory pages using the profiling information gathered by the first module. In our page migration approach, we insert instrumentation code into the application to gather profiling information, to migrate the memory pages, to bind application threads to processors and to detect the application termination.

We used Dyninst[2] to insert instrumentation code into the application being analyzed. Dyninst is a library that permits the insertion of code into a running program. The Dyninst library provides a machine independent interface to permit the creation of tools and applications that use runtime code patching.

For our dynamic page migration algorithm, instrumentation code is inserted at the entry of the *main* function, exit point(s) of *thr_create* function, and the entry of *exithandle* function. The instrumentation code that is inserted at *main* loads a shared library that creates additional helper threads for gathering profiling information and migrating memory pages. The instrumentation code inserted at the exit point(s) of *thr_create* calls the *processor_bind* system call to explicitly bind the newly created application threads to available processors in a round robin fashion. The helper threads are bound to dedicated processors and the remaining processors are used to bind the other threads in the application. The instrumentation code inserted at the entry to *exithandle* detects the application termination and cleans up the hardware counters and software libraries.

Our dynamic page migration algorithm is a two-phase algorithm. It creates two helper threads, one for profiling and another for page migration. The profiling thread samples the interconnect transactions and updates the access frequencies of the memory pages for each system board. The migration thread stops the execution of all other application threads at fixed time intervals and triggers page migration based on the profiling information gathered. In addition, to prevent memory pages ping ponging between memory boards, we freeze memory pages that have been migrated recently for a fixed number of page migration iterations

[3] An HPC application often runs as the only process on the target machine for the highest speedup. Similarly, we assume that at any time there is only one application using the plug-in Sun Fire Link counters. However, if multiple users run HPC applications at the same time, more plug-in performance counters are required. Since these counters are plugged into the I/O boards, as many users as available I/O boards can be provided with similar counters.

(We currently freeze a page if it migrated during the last 3 consecutive intervals). Thus, the memory pages are migrated at fixed time intervals and a page may be migrated more than once throughout the execution of the application.

Our migration algorithm does not currently use any minimum access frequency threshold for the migration of a page. At every migration interval, regardless of the number of accesses, each page is considered as candidate for migration. Alternatively, we could limit migration to the pages with a minimum number of accesses or cache misses and thus migration overhead would potentially be eliminated for pages with little contribution to the application's memory time.

4. Preliminary Experiments

In this section we will discuss the results of our preliminary experiments to ensure that we could accurately sample interconnect transactions in the applications being analyzed, and that placing pages local to the same board as the processor can have a significant impact on the memory performance of an application.

4.1 Interconnect Transaction Sampling

We sample the interconnect transactions using the Sun Fire Link hardware monitors and approximate the access frequencies for the memory pages. However, for sampling to be effective, the sampling technique has to be representative of all transactions that occurred during the execution of the application being analyzed.

One approach to sample interconnect transactions using the Sun Fire Link bus analyzer is to continuously sample at the maximum speed of interconnect instrumentation software. We refer to this sampling scheme as *maximum-rate* sampling. (Note that maximum-rate sampling does not capture a compete set of transactions, but it tries to sample as many transactions as possible). Alternatively, transactions can be sampled at fixed time intervals or at every N^{th} transaction occurrence, where N is a constant that defines the interval of sampling [3]. In this paper, we refer to sampling at every N^{th} transaction occurrence as *interval* sampling.

We conducted a series of experiments to compare how representative the maximum-rate and interval sampling techniques are of all transactions. To objectively compare the two sampling techniques we designed a distance metric D that given a set of transactions and a set of samples from the set, measures the percent difference between the values of a property P for these sets. The property we used in our experiments is the ratio of transactions requested by a specific processor to the total number of transactions in a given time interval. The closer the value of our distance metric is to 0, the more representative the set of sampled transactions is of the set of all transactions. Since the Sun Fire Link counters can accurately count the number of transactions as well as the number of transactions from a given processor, we counted both of these values and compared them with samples taken via Sun Fire Link bus analyzer.

During each experiment, we configured one of the two counters in the Sun Fire Link hardware monitors to count the number of transactions requested by a selected processor N, denoted C_N. The other counter is configured to count all transactions, C_A. Using the Sun Fire Link bus analyzer we also sampled interconnect transactions and recorded the number of transactions sampled, denoted S_A. In the set of sampled transactions, we count the number of transactions that are requested by processor N, denoted S_N. We calculate the ratios for the set of sampled transactions and the set of all transactions as $P_{Sample} = S_N/S_A$ and $P_{All} = C_N/C_A$, respectively. We define the distance as $D=ABS(P_{Sample} - P_{All})/P_{All}$. That is, the distance metric gives an insight as to how far the set of sampled transactions deviate from the set of all transactions with respect to the property P.

We conducted a series of experiments for a set of processors while running OpenMP version of the CG[4] benchmark from NAS Parallel benchmark suite[16]. We repeated the experiments with different sampling intervals in which samples taken at every 64, 256, 1024 and 4096 transactions.

Table 1 presents the results of the experiments conducted to compare how representative the sampled transactions are of all transactions. In Table 1, the second column gives the distance values for maximum-rate sampling. The third to sixth columns give results for interval sampling with different interval values. In Table 1, the rows that are labeled with processor identifiers give the distance between the set of all transactions and the set of sampled transactions with respect to that processor. The second from the last row averages the distance values of all experiments.

Table 1 shows that maximum-rate sampling can sample about 18% of all transactions. Table 1 also shows that for maximum-rate sampling, for each processor, the distance metric is significantly higher compared to

[4] We have also conducted same set of experiments for the other applications in the NAS Parallel benchmark suite and observed similar results as application CG. We chose to present the results for only CG due to the space limitations.

interval sampling. Moreover, for maximum-rate sampling, the average distance over all processors is 0.56, which shows that the set of sampled transactions is quite different from the set of all transactions (Recall a value of 0 for distance *D* is perfect sampling correlation).

	Maximum-rate Sampling	Interval Sampling			
		4K	1K	256	64
Proc 0	0.51	0.03	0.03	0.03	0.09
Proc 1	0.61	0.04	0.04	0.04	0.09
Proc 2	0.47	0.01	0.02	0.02	0.23
Proc 3	0.58	0.00	0.01	0.01	0.02
Proc 4	0.65	0.02	0.02	0.02	0.12
Proc 5	0.57	0.03	0.02	0.03	0.15
Average Dist.	0.56	0.02	0.02	0.02	0.11
% Sampled	17.56	0.19	0.78	3.07	9.75

Table 1 Comparison of distance values for maximum-rate sampling and interval sampling

During maximum-rate sampling, the maximum number of transactions the instrumentation software can record bounds the number of samples that can be taken for a processor. Thus, if a processor requests transactions faster than the maximum rate the instrumentation software can read, many transactions for the processor will not be recorded. Similarly, if a processor requests transactions slower than the rate of instrumentation software, almost all of its transactions will be recorded as samples. Thus, maximum-rate sampling results in a skewed distribution of sampled transactions with respect to the level of memory system activity on processors and the sample set does not accurately represent all transactions.

Table 1 also shows that for interval sampling, the distance values depend on the sampling rate. The distance values are low and similar to each other except for the experiments where transactions are sampled at every 64 transactions. In particular, if the samples are taken at every 256 transactions or more, the set of sampled transactions is fairly representative of all transactions. Table 1 also suggests that for interval sampling, if the rate that samples are taken exceeds 5% of all transactions, the set of sampled transactions becomes less representative.

Since transaction sampling competes for bus bandwidth with the application being measured, it is also necessary to quantify the bus load due to the sampling technique used. To quantify the bus load of each sampling technique, we conducted an experiment where we counted the number of address transactions due to accessing the hardware monitor. From this, we calculated the additional bandwidth consumed.

Our experiments showed that both maximum-rate and interval sampling produce the same bus load of around 0.5MB/sec (0.005% of the maximum data bandwidth). This is due to the fact that the dominant part of the bus load is produced by sampling the counter contents to determine whether it is time to take a sample rather than getting the sample. If the counters had an interrupt on overflow feature (common in current CPU counters), we could eliminate much of this bus load.

4.2 Impact of Local Page Placement

Before testing the effectiveness of our page migration scheme on multithreaded applications, we wanted to assess the impact of page placement on the memory performance of a single threaded application. We designed a simple application that sequentially traverses over the elements of an array repeatedly. Before each array element is accessed, the cache line containing the element is invalidated and the access is satisfied by the memory in which the array pages are placed. Note that this application is designed to exercise memory heavily and real applications would not have as many cache misses.

We conducted experiments running the single threaded application under local and non-local page placement, and we measured the total time spent to access array elements. Moreover, to eliminate factors such as pre-fetching or speculative loads, we also implemented a variant of this benchmark that uses a random number generator to decide on the next element to be accessed.

Table 2 presents the memory access times for our test programs. In Table 2, the first column lists the applications, where each row is labeled by the pattern in which the array elements are traversed. The second and third columns give the memory access times for local and non-local placements of the array pages, respectively. The fourth column lists the slowdown ratios when array pages are placed non-locally compared to being placed locally on the processor running the application.

Array Access Pattern	Array Page Placement		Slowdown
	Local	Non-Local	
Sequential	546.6	685.8	1.25
Strided	659.7	810.1	1.23
Random	512.5	606.0	1.18

Table 2 Array access times for local and non-local page placement

For each program, Table 2 shows a significant slowdown in array access times when array pages are placed non-local to the processor running the application compared to placing array pages locally. The slowdown ratios for array access times range from 1.18 to 1.25. More importantly, Table 2 shows that the slowdown due to non-local page placement is directly proportional to the back-to-back latencies measured by the pointer-chasing latency benchmark listed in Section 1.

We noticed a form of intra-board locality in the Sun Fire servers. That is, although the array pages are local, the choice of the processor from the group of processors on the same system board also has an impact on the array access times. Table 3 presents the array access times for each application when different processors in the same system board are used to execute the application. In each execution, array pages are placed identically. In Table 3, the second column presents array access times when the test programs run on the first processor in a system board where the third column presents array access times when they run on the second processor.

Array Access Pattern	Processor on Local System Board		Slowdown
	CPU 0	CPU 1	
Sequential	546.6	604.3	1.11
Strided	659.7	715.4	1.08
Random	512.5	546.0	1.07

Table 3 Intra-board variation in array access times

Table 3 shows that the programs spent 7-11% more time traversing the array elements when they are bound to the second processor of the system board compared to when they are bound to the first processor even though the array pages are placed local to the processors. We believe intra-boards variations in array access times are to due to whether the array pages are placed on the memory banks controlled by the processor running the application or on the memory banks controlled by another processor in the same system board. We also believe increasing the number of memory banks controlled by each processor will reduce the intra-board variations.

5. Page Migration Experiments

To investigate the effectiveness of our dynamic page migration approach on the performance of real applications, we conducted experiments using the OpenMP C implementation of the NAS Parallel Benchmark suite[16]. We chose applications with different sizes ranging from B to C such that each application would have a similar memory footprint. We compiled the applications using Sun's native compiler, Sun C 5.5 EA2, with optimizations(-O3) on to support parallelized code.

We conducted all of our experiments on a 24-processor Sun Fire 6800 with 24GB of main memory. The system clock frequency is 150MHz. The processors are 750MHz UltraSPARC III. The memory in each system board is 8-way interleaved where each processor controls two banks of memory. The Sun Fire Link hardware is plugged into an I/O drawer in this system. The Sun Fire Link instrumentation has full visibility into all transactions on Fireplane interconnect.

To quantify the benefits of our dynamic page migration approach, we conducted a series of experiments with and without page migration. For all applications, we measured both the original execution times and the execution times when memory pages are migrated using our dynamic page migration approach. For each application, we also measured the percentage reduction in the number of non-local memory accesses when memory pages are dynamically migrated compared to its original execution.

We ran all applications with 12 threads on 6 system boards of the Sun Fire 6800 server. To eliminate any possible contention due to resource sharing among processors, we scheduled two threads on each system board. We sampled interconnect transactions at every 1024 transactions.

For the experiments with page migration, we triggered page migration at every 5 seconds. To choose the migration interval, we conducted a sensitivity analysis in which we considered different migration interval values ranging from 1 second to 50 seconds. The sensitivity analysis showed that the migration interval does not have a major impact on the performance of the applications when page migration is used.

As explained in Section 3, we insert instrumentation code into the application using the Dyninst library. For each application, the instrumentation overhead is a one-time overhead since the Dyninst library has a capability of saving instrumented executables for later reuse. Moreover, the instrumentation overhead for our page migration approach is independent from the execution times of the applications we analyzed. We measured the instrumentation overhead for all applications for our dynamic page migration approach and it is typically around 2 seconds.

5.1 Reduction in Non-Local Memory Accesses

To quantify the benefits of our dynamic page migration approach, we counted the total number of non-local memory accesses for all applications with and without using dynamic page migration. We used the Sun Fire Link hardware monitors to measure the total number of non-local memory accesses in the applications.

Due to limitations in the number of counters in the Sun Fire Link hardware monitor, we were not able to count the per board number of non-local memory accesses in an application during a single run. Instead, we ran each application once for each system board and counted the number of non-local memory accesses requested by the group of processors in that system board. We later calculated the total number of non-local memory accesses for an application as the sum of the non-local memory accesses for all system boards.

Table 4 presents the percentage reduction in the total number of non-local memory accesses when dynamic page migration is used compared to when memory pages are not migrated. In the second column, we give the total number of address transactions requested by each application during its execution. The third column gives the percentage of non-local memory accesses without our page migration approach and the fourth column shows the percentage of non-local memory accesses when memory pages in the application are migrated using our dynamic page migration approach. The fifth column lists the percentage reduction in the total number of non-local memory accesses when dynamic page migration is used.

Table 4 shows that for all applications, our dynamic page migration approach was able to reduce the number of non-local memory accesses by 19.7-89.6% (The average is 58.3%).

Table 4 also shows that for MG, a significant number of non-local memory accesses were eliminated when memory pages were migrated. However, for LU our dynamic page migration approach was not able to reduce the number of non-local memory accesses significantly. In LU, all system boards uniformly access most of the memory pages that our dynamic approach was able to migrate. That is, while migrating those pages to a system board reduces the number of non-local memory accesses requested by the processors in that system board, the number of non-local memory accesses by the processors in all other system boards increases. Our dynamic page migration approach currently uses a simple decision mechanism that identifies the preferred location of a memory page as the system board that accesses it most. It does not take the access frequencies by other system boards into consideration. The access frequencies by other system boards might be useful to better decide whether a page should be migrated.

	Memory Accesses (Million)	**Percentage of Non-local Memory Accesses**		**% Reduction**
		w/o Page Migration	**Page Migration**	
BT (B)	38,507	40.9	25.3	38.0
CG (C)	15,721	80.9	15.3	81.0
EP (C)	42	85.4	28.2	67.0
FT (B)	2,329	64.2	29.6	54.0
LU (C)	48,682	41.2	33.1	19.7
MG (B)	841	80.5	8.3	89.6
SP (C)	116,116	55.0	22.7	58.8

Table 4 Reduction in non-local memory accesses when memory pages are dynamically migrated

5.2 Impact of Page Migration on Cache Usage

The UltraSPARC III processors in the Sun Fire servers use physical addresses to index their external caches. Since page migration changes the physical addresses of the memory pages in an application, it is also necessary to ensure that our page migration approach does not have a significant impact on the external cache usage of the applications. To quantify the external cache usage of the applications, we counted the number of conflict and capacity misses during the execution of the applications with and without dynamic page migration. We counted the number of conflict and capacity misses in the applications using Sun Fire Link counters by measuring the number of write-back (WB) transactions requested. A WB transaction is requested when a dirty cache line is evicted from the external cache due to a capacity or conflict miss.

Table 5 presents the number of WB transactions with and without our page migration approach. Table 5 shows that our dynamic page migration approach does not significantly affect the number of conflict and capacity cache misses. It also shows that our dynamic page migration approach has a higher impact on EP compared to other applications. However, EP does not allocate a significant number of memory pages during its execution and thus the absolute number of cache misses is more than a factor of 20 lower than any other application we measured. Moreover, the total number of address transactions requested by EP is not significant due its effective use of local caches. The increase in cache misses in EP is mainly due to the invalidation of data in processor caches caused by migration of memory pages.

	Number of WB Transactions (Millions)		% Change
	w/o Page Migration	Page Migration	
BT (B)	14,948.8	14,900.1	-0.33
CG (C)	270.6	268.7	-0.67
EP (C)	12.3	12.6	2.38
FT (B)	855.0	851.8	-0.37
LU (C)	18,252.8	18,171.6	-0.44
MG (B)	217.4	218.0	0.28
SP (C)	39,223.3	39,139.9	-0.21

Table 5 Percent change in the number of write-back transactions

5.3 Execution Times

While reducing the number of non-local memory accesses in an application is important, what matters is the impact of this reduction on application's runtime. In this section, we look at the impact of our page migration approach on the execution times of the applications. For each application, we conducted three different experiments and measured the total execution time of the application in each experiment.

First, we ran each application using our dynamic page migration approach and measured the total execution time including overhead due to the creation of the helper threads and triggering memory page migrations. Even though the migration thread runs in parallel with other threads of the application, it suspends all application threads to trigger the actual page migrations and later resumes their executions. During the second set of experiments, we measured the original execution times of the applications with no intervention. Lastly, we conducted a third set of experiments to investigate the impact of binding application threads to fixed processors, and therefore the impact of dynamic page migration in isolation. During these experiments, we ran each application with page migration disabled but bound the threads to the processors in the system.

For each application and experiment, we repeated the experiment seven times and recorded the minimum of the execution times among all runs. We used the minimum execution time since we noticed higher variation in the original execution times for some applications. We suspect the higher variation in the original execution times of those applications is due to differences in the initial page placements and thread scheduling by the operating system.

Table 6 presents the execution times of the applications we analyzed. The second column lists the original execution times of the applications. In the third column, we present the execution times when the application threads are bound to the processors throughout the executions. The fourth column lists the execution times of the applications when we migrate memory pages using our dynamic page migration approach. The fifth column presents the number of page migrations triggered. Lastly, the sixth column presents the overhead due to page migrations.

Table 6 shows that for all applications except LU and MG, when the application threads are bound to processors the applications run faster by 0.16-1.76% compared to their original executions. However, LU slows down by 0.6% where MG slows down by 2.2% when their threads are bound to the processors. Table 6 shows that binding application threads to the processors is almost always beneficial even though the performance gain is not significant.

	Execution Times (sec)			Number of Migrations	Overhead (sec)
	Original	Bound Thread	Page Migration		
BT (B)	996	992	966	112,310	11.8
CG C)	625	613	534	47,213	4.4
EP (C)	293	292	292	2,071	0.3
FT (B)	113	112	118	177,602	15.1
LU (C)	1981	1994	1978	132,696	13.1
MG (B)	31	32	26	49,884	2.7
SP (C)	3901	3854	3347	138,943	17.1

Table 6 Execution times of the applications for their original execution, for the execution where application threads are bound to processors, and for our dynamic page migration approach. Migration overhead is also included in the listed times for page migration column.

Table 6 also shows that the overhead due to page migration is mainly proportional to the number of page migrations requested and it ranges up to 12.8% compared to the original execution times of the applications. To guarantee that the migration thread touches the page next before all other threads, all other threads have to be suspended. If the operating system instead provided a system call that would allow applications to indicate the target locations of the memory pages, it would permit migration of pages to their target locations during the next available opportunity, and thus reduce the page migration overhead.

Figure 2 presents the performance improvement when our page migration approach is used compared to both the original execution time and the execution time when the threads of the applications are bound to processors. Under the label of each application on the x-axis, Figure 2 also presents the migration overhead percentage with respect to the original execution time of the application. Figure 2 shows that our dynamic page migration approach was able to improve the execution performance of the applications except FT by up to 15.9% compared to their original executions. However, FT runs slower under our dynamic page migration approach.

Our dynamic page migration approach improved the performance of CG and SP by 14.5% and 14.2%, respectively, compared to their original execution times. CG and SP request many memory accesses and our dynamic page migration approach was able to eliminate many of the non-local memory accesses (see Table 4). In addition, dynamic page migration improved the execution performance of CG and SP by 12.8% and 13.2% respectively, compared to the executions where application threads are bound.

Like CG and SP, our dynamic page migration approach was also able to improve the performance of MG by 15.9% compared to its original execution time. Even though MG does not request many memory accesses, our page migration approach was still able to reduce the number of non-local memory accesses significantly (see Table 4). Compared to the execution of MG when its threads are bound to the processors, dynamically migrating memory pages in MG improved the execution performance by 18.1%.

Figure 2 also shows that our dynamic page migration approach improved the execution performance of BT by 2.9% compared to its original execution. Dynamically migrating memory pages in isolation for BT improves the execution performance by 2.6%. Figure 2 also shows that our page migration approach is not as effective for BT as for CG, MG, and SP, which is partially due to fact that the reduction in the number of non-local memory accesses in BT is not as high. Similarly, our page migration approach improved the performance of LU by 0.8%, which is also mainly due the small amount of reduction in number of non-local memory accesses in LU.

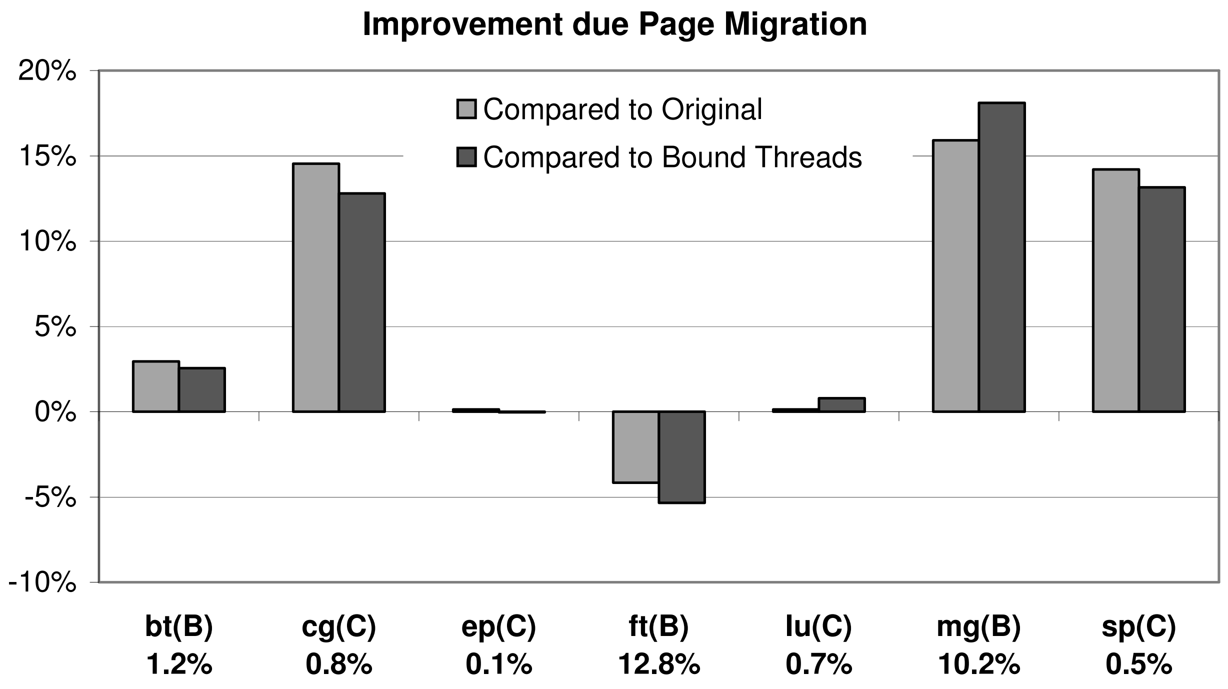

Figure 2 Performance gain for the applications using our dynamic page migration approach compared to original execution time and to the execution time when threads are bound to processors. Percentage value below each application label corresponds to page migration overhead introduced

Figure 2 also shows that our dynamic page migration approach was not as effective in improving the execution performance of EP even though it reduced the number of non-local memory accesses by 67.0%. This due to fact that EP reuses data in the local caches of the processors, and the majority of its memory accesses are requested at the beginning of its execution, before the memory pages are migrated.

Figure 2 shows that our dynamic page migration approach was not able to improve the execution performance of FT even though it reduced the number of non-local memory accesses in FT by 54.0% (Table 4). Instead, our page migration approach slowed down the execution of FT by around 4.2% compared to its original execution. However, Figure 2 also shows that the slowdown for FT is mainly due to the overhead introduced by page migration, which is 12.8% of the original execution time for FT. That is, the reduction in the number of non-local memory accesses did not overcome the overhead introduced by migrating many pages that are initially placed poorly. Moreover, the page migration overhead for FT would be reduced significantly if the operating system did not require suspending application threads to trigger the actual migrations by touching pages and instead provided a mechanism to directly request migration.

6. Related Work

Noordergraaf and Zak[15] describe a set of embedded hardware instrumentation mechanisms implemented for monitoring the system interconnect on Sun Fire servers. The instrumentation supports sophisticated programmable filtering of event counters. Their implementation results in a very small hardware footprint making it appropriate for inclusion in commodity hardware. In our page migration scheme, we heavily used these instrumentation mechanisms to sample interconnect transactions.

Many prior page migration policies[1,11] have been in the context of non-cache-coherent NUMA multiprocessor systems. These kernel-level policies were based on page fault mechanisms and studied different page placement and migration policies for NUMA multiprocessors with large remote to local latency ratios. Bolosky et al[1] used memory reference traces to drive simulations of NUMA page placement policies. LaRowe et al[11] modified OS memory management modules to decide whether a page will be migrated. In contrast, our research introduces page migration policies for cache-coherent shared memory multiprocessor systems with small remote to local latency ratios. Moreover, our approach implements the migration policy at the user level and uses access frequencies gathered from the plug-in hardware counters.

Chandra et al[5] investigated the effects of different OS scheduling and page migration policies for cache-coherent shared-memory multiprocessors using the Stanford DASH multiprocessor. Although they mainly focused on OS scheduling policies, they also investigated page migration policies based on TLB misses. Chandra et al. reported that page migration did not improve the response time for the workloads used due to overhead incurred by the operating system. They also performed a trace-driven study to explore the possible benefits of memory page migrations. Compared to their approach, our page migration approach is more effective partially due to elimination of most of the operating system overhead by using a slower migration rate.

Verghese et al.[19] studied operating system support for page migration, and replication in cache-coherent shared-memory multiprocessors. They introduced a decision tree to select the action to be taken on memory pages upon cache misses. The actions taken for a page include replication, migration and freeze, depending on the threshold values used in the decision tree. Using the thresholds that gave the best results, they evaluated the approach using a machine simulator for SGI Origin2000 multiprocessors. The multiprocessor system they used also had a large remote to local memory latency ratio of 4:1. They reported that dynamic page placements did not yield performance gains due to the overhead introduced by the operating system. They also reported that the primary sources of overhead were processor synchronization and TLB flushing. Unlike their approach, our page migration approach eliminates most of the operating system overhead due to using a slower migration rate. Moreover, the Sun Fire servers we used in our research incur a lower overhead due to TLB flushing since TLB misses are serviced by hardware.

Kernel-level dynamic page placement schemes are also extensively studied in the Sun(TM) WildFire systems[4,8,14]. The Sun WildFire system is a prototype cache coherent NUMA architecture, built from small number of large standard SMP nodes and has large remote to local latency ratios. Hagersten and Koster[8] evaluated the impact of coherent page replication and hierarchical affinity scheduling on TPC-C execution. They used excess-remote-cache-miss counts to guide page placement. Noordergraaf and vander Pas[14] also evaluated kernel-level page migration and replication using a simple HPC application in a large Sun Wild-

Fire system. To identify memory pages for migration, they used excess misses that indicate conflict and capacity misses in a local node's cache. They reported that using a replication-only policy yielded much better performance than policies that included migration.

Recently, Bull and Johnson[4] studied the interactions between data distribution, migration and replication for the OpenMP applications. Although they primarily focused on a data distribution extension for OpenMP, they also studied the impact of page migration and replication. Their study also showed that page replication is more beneficial than migration. This is mainly due to higher overhead in page migration from copying a memory page from its local node to a remote node. In comparison, our page migration approach also has a slower migration rate, which partly explains the reduction in page migration overhead.

Recent work has used dynamic page placements to improve the locality for TPC-C in cc-NUMA servers. Wilson and Aglietti[20] used Verghese's dynamic page placement algorithm to tune TPC-C execution on Sybase. They used a one-second trace of TPC-C execution and a simulator for a 4-node multiprocessor system to study the performance, bandwidth and locality of TPC-C. They used hand-tuned threshold values for dynamic page placements in their simulations. Wilson and Aglietti showed that dynamic page placement could be effective if operating system overhead is hidden within the idle CPU cycles.

7. Conclusions

In this paper, we introduced an automatic profile-driven page migration scheme and investigated the impact of our page migration scheme on the memory performance of multithreaded programs. We used commercially available plug-in hardware monitors to profile the applications. We tested our dynamic page migration approach using the OpenMP C implementation of the NAS Parallel Benchmark suite.

Our dynamic page migration approach always reduced the total number of non-local memory accesses in the applications we analyzed compared to their original executions, by up to 90%. Our page migration approach was also able to improve the execution time of the applications up to 16% compared to their original execution time.

We conducted our experiments on a Sun Fire 6800 server which has only small differences between local and non-local memory access times (225ns vs. 300ns). We believe our page migration approach will be even more effective in improving the performance of the applications running on larger cc-NUMA servers such as the Sun Fire 15K. In these larger cc-NUMA servers, the data transfer times differ significantly among local and non-local memory accesses (225ns vs. 400ns). For our page migration approach to work on these larger cc-NUMA servers, however, separate plug-in hardware counters for each coherency domain would be required.

More importantly, the effectiveness of our page migration approach shows the importance of inexpensive hardware counters in automatic performance tuning of the applications. In this paper, our page migration approach depends on accurate interconnect transaction samples gathered from hardware counters. We believe this type of hardware counters and tools like our page migration approach will be of increasing utility as memory systems become more complex.

We believe the effectiveness of our page migration approach also shows the advantage of putting the page migration policy at the user level while only relying on the operating system kernel to provide the actual migration mechanism.

We also believe that for page migration mechanism to be more beneficial, underlying operating system should provide means to trigger page migration without stopping the application. That is, if the user could simply request migration of a page and the underlying operating system could migrate the page during available idle cycles, most of the migration overhead would be hidden.

Acknowledgements

We thank Lisa Noordergraaf for her valuable comments on this paper and help in understanding the internals of the Sun Fire servers and Sun Fire Link hardware monitors. We also thank Jeffrey Odom who helped configure the system used in this paper; and Bryan Buck and Chadd Williams for their comments. This work was supported in part by NSF awards EIA-0080206, and DOE Grants DE-FG02-93ER25176, DE-FG02-01ER25510 and DE-CFC02-01ER254489.

References

[1] Bolosky, W.J., Scott, M.L., Fitzgerald, R.P., Fowler, R.J., Cox, A.L., *NUMA* Policies *and Their Relation to Memory Architecture*, International Conference on Architectural Support for Programming Languages and Operating Systems, 1991, San Jose, CA.

[2] Buck, B.B., Hollingsworth, J.K., *An API for Runtime Code Patching*, The Journal of High Performance Computing Applications, 2000, **14**.

[3] Buck, B.R., Hollingsworth, J.K., *Using Hardware Performance Monitors to Isolate Memory Bottlenecks*, SC2000, 2000, Dallas, TX.

[4] Bull, J.M., Johnson, C., *Data Distribution, Migration and Replication on a cc-NUMA Architecture*, The Fourth European Workshop on OpenMP, 2002, Rome, Italy.

[5] Chandra, R., Devine, S., Verghese, B., Gupta, A., Rosenblum, M., *Scheduling and Page Migration for Multiprocessor Compute Servers*, 6th International Conference on Architectural Support for Programming Languages and Operating Systems, 1994, San Jose, CA.

[6] Charlesworth, A., *The Sun Fireplane System Interconnect*, SC2001, Denver, CO.

[7] Gharachorloo, K., Sharma, M., Steely, S., Doren, S.V., *Architecture and Design of AlphaServer GS320*, International Conference on Architectural Support for Programming Languages and Operating Systems, 2000, Cambridge, MA.

[8] Hagersten, E., Koster, M., *WildFire: A Scalable Path for SMPs*, Fifth IEEE Symposium on High-Performance Computer Architecture, 1999.

[9] Hewlett Packard, *HP Superdome White Paper*, , http://www.hp.com/products1/servers/scalableservers/superdome/infolibrary/, 2000.

[10]IBM, *The IBM pSeries 680 Technology and Architecture*, 2000, http://www.ibm.com/servers/eserver/pseries/hardware/ whitepapers/p680_technology.html.

[11]LaRowe, R.P., Ellis, C.S., Kaplan, L.S., *The Robustness of NUMA Memory Management*, Symposium on Operating Systems Principles, 1991.

[12]Laudon, J., Lenoski, D., *The SGI Origin: A ccNUMA Highly Scalable Server*, International Symposium on Computer Architecture, 1997.

[13]Laudon, J., Lenoski, D., *System Overview of the SGI Origin 200/2OOO Product Line*, IEEE Computer Society International Conference, 1997, San Jose, CA.

[14]Noordergraaf, L., Pas, R.v.d., *Performance Experiences on Sun's WildFire Prototype*, SC1999, Portland, OR.

[15]Noordergraaf, L., Zak, R., *SMP System Interconnect Instrumentation for Performance Analysis*, SC2002, Baltimore, MD.

[16]Omni OpenMP Compiler Project, *NAS Parallel Benchmarks OpenMP C Versions*, http://phase.hpcc.jp/Omni/benchmarks/NPB.

[17]Squillante, M.S., Lazowska, E.D., *Using Processor-cache Affinity in Shared Memory Multiprocessor scheduling*, IEEE Transactions on Parallel and Distributed Systems, 1993, **4**(2).

[18]Sun Microsystems, *UltraSPARC III Cu User's Manual (version 1.0)*, 2002,

[19]Verghese, B., Devine, S., Gupta, A., Rosenblum, M., *Operating System Support for Improving Data Locality on CC-NUMA Compute Servers*, International Conference on Architectural Support for Programming Languages and Operating Systems, 1996, Cambridge, MA.

[20]Wilson, K.M., Aglietti, B.B., *Dynamic Page Placement to Improve Locality in CC-NUMA Multiprocessors for TPC-C*, SC2001, Denver, CO.