



Hardware monitors for dynamic page migration

Mustafa M. Tikir^{a,*}, Jeffrey K. Hollingsworth^b

^a San Diego Supercomputer Center, 9500 Gilman Drive, 0505 La Jolla, CA 92093, United States

^b Computer Science Department, University of Maryland, College Park, MD 20742, United States

ARTICLE INFO

Article history:

Received 17 September 2006

Received in revised form

2 April 2008

Accepted 31 May 2008

Available online 17 June 2008

Keywords:

Dynamic page migration

Hardware performance monitors

cc-NUMA systems

Multiprocessor systems

OpenMP applications

High performance computing

Address translation counters

Runtime optimization

Full system simulation

ABSTRACT

In this paper, we first introduce a profile-driven online page migration scheme and investigate its impact on the performance of multithreaded applications. We use centralized lightweight, inexpensive plug-in hardware monitors to profile the memory access behavior of an application, and then migrate pages to memory local to the most frequently accessing processor. We also investigate the use of several other potential sources of data gathered from hardware monitors and compare their effectiveness to using data from centralized hardware monitors. In particular, we investigate the effectiveness of using cache miss profiles, Translation Lookaside Buffer (TLB) miss profiles and the content of the on-chip TLBs using the valid bit information. Moreover, we also introduce a modest hardware feature, called Address Translation Counters (ATC), and compare its effectiveness with other sources of hardware profiles.

Using the Dyninst runtime instrumentation combined with hardware monitors, we were able to add page migration capabilities to a Sun Fire 6800 server without having to modify the operating system kernel, or to re-compile application programs. Our dynamic page migration scheme reduced the total number of non-local memory accesses of applications by up to 90% and improved the execution times up to 16%. We also conducted a simulation based study and demonstrated that cache miss profiles gathered from on-chip CPU monitors, which are typically available in current microprocessors, can be effectively used to guide dynamic page migrations in applications.

© 2008 Elsevier Inc. All rights reserved.

1. Introduction

The dominant architecture for current shared-memory multiprocessor systems is cache-coherent non-uniform memory access (cc-NUMA). In cc-NUMA systems, processors have a faster access to the memory units local to them compared to the remote memory units. For example, the remote and local latencies in mid-range Sun Fire 6800 servers is around 300 ns and 225 ns [6], respectively, whereas the remote and local latencies in a 512 processor Altix 3000 are around 605 ns and 145 ns, respectively [21]. Traditionally, cc-NUMA systems use physical first-touch page placement, where memory pages are placed into the memory that is local to the processors that touch the page first. However, first-touch 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 memory intensive applications running on cc-NUMA servers.

In this paper, we first introduce a user-level memory page migration scheme, namely dynamic page migration. In this page

migration scheme, applications are profiled to determine the preferred locations of the memory pages in the memory units using hardware monitors. Then system calls are used to request the kernel to migrate the memory pages to the specific memory units. In this dynamic page migration scheme, both profiling and page migrations are conducted during the same run of the applications. The access frequencies of the memory pages by the processors are gathered continuously at runtime using hardware monitors and the pages are migrated local to the most frequently accessed processors at fixed time intervals.

Although page migration has been extensively studied in prior research, our dynamic page migration approach demonstrates several novel features. First, our goal is not to introduce a new page placement policy. Instead, we demonstrate that the combinations of inexpensive plug-in hardware monitors that sample information about interconnect transactions and a simple page migration policy can be used effectively to improve the performance of real scientific applications. 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 to local memory latency ratio of the Sun Fire system we used is 1.33:1. We believe our page migration scheme will be more effective on systems with large remote to local latency ratios.

The hardware performance monitors we used to gather page access profiles for our actual dynamic page migration scheme are

* Corresponding author.

E-mail addresses: mtikir@spsc.edu (M.M. Tikir), hollings@cs.umd.edu (J.K. Hollingsworth).

¹ This work was done while a student at the University of Maryland, College Park.

centralized external plug-in monitors. These monitors listen to all address transactions on the system interconnect in the cc-NUMA server. However, such monitors are not available in most systems. Moreover, for non-bus based multiprocessors that do not use a common address and data bus, it is difficult to implement such centralized monitors that need to listen to all transactions on the system interconnect. Alternatively, many processors now include on-chip hardware support for performance monitoring, such as MIPS R10000 [22], Compaq Alpha [7], Itanium from Intel [10], Sun UltraSPARC [17].

In this paper, we also investigate the use of several other potential sources of profiles gathered from hardware monitors in dynamic page migration and compare their effectiveness to using profiles from centralized hardware monitors. In particular, we investigate the effectiveness of using cache miss profiles and TLB miss profiles from on-chip CPU monitors, and sampling the content of the processor TLBs. If such sources of information can provide sufficiently accurate information, it would mean software based migration could be performed on systems without the need for dedicated hardware monitors. We also introduce a simple hypothetical modest hardware feature, called Address Translation Counters (ATC), which is specifically designed to gather profiles for dynamic page migration and compare its effectiveness with other sources of profiles. The ATC hardware is a set of additional counters included in the TLBs of a processor and gathers accurate information on access frequencies to the memory pages by the processor.

To evaluate the effectiveness of our dynamic page migration scheme, we implemented our page migration scheme for a Sun Fire 6800 server with Sun Fire Link [13] hardware monitors for the Sun Fireplane system interconnect. To evaluate the effectiveness of using each source of profiles in dynamic page migration, we conducted a simulation based study using a full system simulator, Simics [12]. We present the results of our studies in terms of the number of page migrations triggered, reduction in the number of non-local memory accesses, and improvement in execution times of the applications. We present the results for OpenMP C implementation of the NAS Parallel Benchmark suite [15] for both our actual page migration scheme and our simulation study.

2. Hardware and software components for dynamic page migration scheme

In this section, we describe the hardware and software components used in our actual dynamic page migration scheme. We first describe the architecture of the Sun Fire servers. We next describe the centralized Sun Fire Link hardware monitors. 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) interconnect. The Sun Fireplane interconnect is implemented with up to four levels of interconnect logic depending on the number of processors in the server [6]. In medium and large-sized Sun Fire servers, processors and memory units are grouped together on system boards (locality groups) [17]. Each system board contains four processors and four 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 requesting processor 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 750 MHz CPUs is around 225 ns if the memory is local and 300 ns if the memory is non-local.

The Sun Fire 6800 server is a mid-range cc-NUMA architecture based on 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 six 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 hardware monitors

For our actual dynamic page migration scheme, we use the Sun Fire Link hardware monitors [14] to gather profiling information for page migration. 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 Monitors 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 monitors provide useful information about the addresses and requesting processors, 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 pages in the applications.

2.3. System calls in the 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 [16]. 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.

Solaris places each physical memory page into the memory that is local to the first processor that touches the page. To move pages in our dynamic page migration scheme, 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 a range of virtual memory onto the local memory of the processor that next touches the range.

3. Dynamic page migration methodology

Our dynamic page migration algorithm consists of two different modules. The first module gathers profiling information using the Sun Fire Link monitors. The second module moves memory pages using the profiling information gathered by the first module. In our 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 applications. 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 *exit* 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 the *exit* function detects the application termination and cleans up the hardware monitors.

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. To trigger migration on a page, our scheme uses the move-on-next-touch feature of the *madvise* system call on the page. In addition, to prevent memory pages ping-ponging between memory units, we freeze memory pages that have been migrated recently for a fixed number of page migration iterations (We freeze a page for three consecutive iterations after migration.). Thus, the memory pages are migrated at fixed time intervals and a page may be migrated more than once throughout application execution.

Our migration algorithm does not use a minimum access frequency threshold to trigger the migration of a page. At every migration interval, regardless of the number of accesses to a page, the page is considered as a 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.

Our dynamic migration scheme does not have a particular mechanism for cache coherency but rather relies on the cache coherency mechanism the underlying operation system uses since our approach is designed for cache coherent NUMA systems that already have a cache coherency mechanism implemented. Instead, we advise the underlying OS to move the page to a different location in physical memory, and cache coherency is maintained by the OS by updating the TLB entries and invalidating the cache lines that are indexed using the physical addresses. Moreover, in our scheme, when migration is triggered for a page, we do not have a control whether a victim page will be evicted from the target physical memory if there is no available page for the

migration to succeed. We instead rely on the mechanisms used in the underlying OS.²

In our page migration scheme, the two helper threads are bound to dedicated processors. However, these helper threads are mostly idle other than gathering profiling information and triggering page migrations at fixed time intervals. To isolate the impact of page migration on non-local memory accesses, we chose to bind these threads to dedicated processors. Instead, these threads could run on the processors the application threads run and make use of idle cycles. Alternatively, these threads can be pushed to the OS level by adding two more threads to the OS. More importantly, considering the impact of chip-level multiprocessor architecture on processor costs, additional dedicated processors can be included to the HPC systems for application profiling and page migration.

4. Other sources of hardware profiles for dynamic page migration

In our actual page migration scheme on the Sun Fire server, we use the centralized Sun Fire Link monitors to identify the preferred locations of memory pages for dynamic page migration. However, such monitors are not available in many systems. Moreover, for non-bus based multiprocessors that do not use a common address and data bus, it is difficult to implement such centralized monitors that need to listen to all transactions on the system interconnect. Alternatively, many processors now include on-chip hardware monitors for performance tuning. In this section, we describe other potential sources of profiles that can be used to generate page access frequencies. Later in the experiments section, we present the results of our simulation based study to investigate the effectiveness of these other sources of profiles.

4.1. Profiles from distributed on-chip CPU monitors

Profiles of page access frequencies by processors in an application running on a cc-NUMA system can be gathered by using information about the cache or TLB misses by each processor in the system. If the information about the number of cache or TLB misses on each page by a processor is known, the access frequency of the page by the processor can be approximated. However, for such information to be available, the addresses associated with the cache and TLB misses are needed.

Many processors include hardware support to count events for performance monitoring. Moreover, they often provide mechanisms to trigger an interrupt when a given number of events occur. More recently, an increasing number of processors provide the ability to capture the memory addresses and/or instructions involved in performance critical events. (Note that some monitors may provide approximate information about the instruction(s) involved due to the difficulty of associating information with specific events when many instructions are in flight. However, even approximate information still provides valuable insight for dynamic tuning of applications.) For example, the Itanium 2 processor provides a set of *event address registers* (EARs) that record the instruction and data addresses of data cache misses, the instruction and data addresses of data TLB misses, and the instruction addresses of instruction TLB and cache misses [10]. Thus, by distributed sampling of the addresses associated with the cache or TLB miss events, profiles of page access frequencies by processors

² In the underlying OS we used for this research, page migration fails if there is no available physical page on the target memory. However, we have not seen a case where migration was denied since we track how much memory is used by the application and do not move pages if we run short on memory. In practice, this constraint never caused us to fail to make a desired migration.

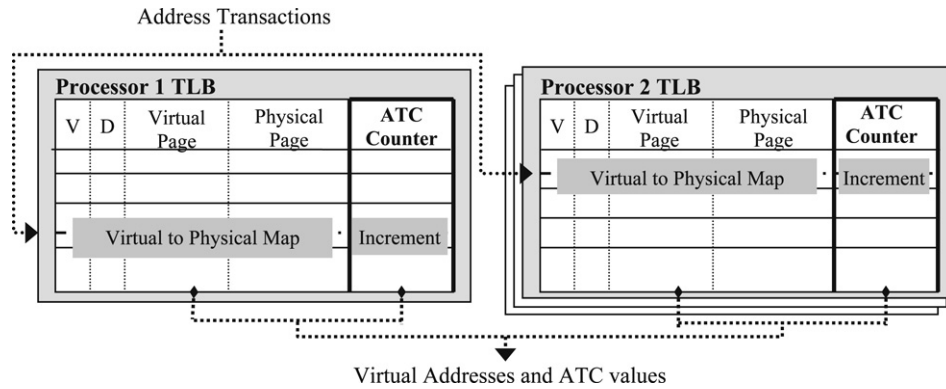


Fig. 1. Information flow in the address translation counters.

can be generated. Moreover, since cache miss events are generally distributed throughout the execution and provide information on fine grain behavior, profiles of page access frequencies gathered from cache miss events may be more representative. Compared to cache misses, the number of TLB miss events is generally lower and these events may not correspond to the pages that are frequently accessed due to the fact that applications tend to keep frequently accessed pages in TLBs. In this paper, we investigate the use of cache and TLB miss information from on-chip CPU hardware.

4.2. Profiles from valid bit information in TLB entries

Hardware tries to keep virtual–physical page translation entries of the frequently accessed pages in the processor TLBs. Thus, the contents of the valid TLB entries in a processor potentially provide information on the pages that are mostly accessed by the processor. By sampling the content of the TLBs periodically, it is possible to approximate page access frequencies by the processor. Similarly, the information from each processor can be combined and page access frequencies by processors can be generated to guide migrations in a dynamic page migration scheme.

To sample the contents of valid TLB entries of a processor, the underlying operating system needs to provide a software sampling mechanism. In particular, the operating system needs to provide a means to query the list of valid entries and the virtual addresses of the pages for each valid TLB entry. In our research, we assume the underlying operating system provides a system call that returns the list of virtual page addresses in the valid TLB entries for a given processor.

4.3. Address translation counters

To further evaluate the effectiveness of sources of profiles in dynamic page migration, we designed a dedicated hardware monitor that gathers accurate page frequencies and compared the effectiveness of other sources of profiles with the dedicated monitors.

The hypothetical hardware feature we use, *Address Translation Counters* (ATC), is a set of additional counters that is included in the TLBs of the processors. In ATC, a counter is included for each TLB entry in a processor (shown in Fig. 1) and incremented when a virtual to physical address translation is satisfied by the corresponding TLB entry. Moreover, when the contents of a TLB entry is evicted due to a TLB miss or invalidated due to other reasons such as cache coherency operations, the counter associated with the TLB entry is cleared. The ATC is included in each processor and counts the number of accesses to the memory pages by the processor using the virtual to physical address translations requested while the memory pages are actively accessed. Note that

the main reason for introducing these counters is to evaluate the effectiveness of other sources of profiles by comparing them to this hardware feature as ATC counters are more likely to capture page access frequencies more accurately compared to other more indirect sources of profiles. Associating the information hardware monitors provide with specific events with many instructions in flight can be difficult, but that associating TLB misses and ATC events with memory addresses is relatively easy due to the fact that virtual page addresses are already stored in the TLB and that TLB misses must update the TLB.

Information recorded by the ATC hardware can be gathered in several ways. One way is to sample the content of the counters regularly during execution along with the virtual page addresses associated with these counters. Another approach is that the operating system may provide low-overhead software traps such that, when a TLB entry is invalidated due to a TLB miss or cache-coherency operation, the content of the corresponding ATC counter value and the associated virtual page address can be provided to the application (similar to the software TLB miss handler in MIPS processors [9]). Lastly, the underlying operating system could include an additional field for each page table entry where the ATC entry can be saved at context switches. Later, the count information can be gathered via system call by querying the page table content. For our research, we assume the underlying operating system provides means to sample ATC content.

5. Experimental results

In this section we first present the results of experiments we conducted to ensure that we could accurately sample address transactions via hardware monitors in the applications being analyzed. We then present the results of experiments in which we evaluated our actual dynamic page migration scheme on the real hardware and simulation based study where we compared the effectiveness of other sources of profiles to the centralized plug-in hardware monitors we used in our actual migration scheme.

5.1. Interconnect transaction sampling experiments

We sample the interconnect transactions using 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 via hardware monitors is to continuously sample at the maximum speed of the interconnect instrumentation software. We refer to this sampling scheme as *maximum-rate* sampling. Maximum-rate sampling does not capture a complete set of transactions, but it tries to sample as many transactions as possible. Alternatively, transactions

Table 1
Distance values for maximum-rate sampling and interval sampling

| | Max-rate sampling | Interval sampling | | | |
|---------------|-------------------|-------------------|------|------|------|
| | | 4K | 1K | 256 | 64 |
| Processor 0 | 0.51 | 0.03 | 0.03 | 0.03 | 0.09 |
| Processor 1 | 0.61 | 0.04 | 0.04 | 0.04 | 0.09 |
| Processor 2 | 0.47 | 0.01 | 0.02 | 0.02 | 0.23 |
| Processor 3 | 0.58 | 0.00 | 0.01 | 0.01 | 0.02 |
| Processor 4 | 0.65 | 0.02 | 0.02 | 0.02 | 0.12 |
| Processor 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 |

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]. 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 percentage difference between the values of a property 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. This metric indicates how much a set of transactions deviate from another set of transactions in terms of memory behavior. Thus, 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.

For this study, we used the Sun Fire Link counters. 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 to approximate the sampling error of sampling techniques.

For 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 P , denoted C_P . 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 P , denoted S_P . We calculate the ratios for the set of sampled transactions and the set of all transactions as $R_{Sample} = S_P/S_A$ and $R_{All} = C_P/C_A$, respectively. We define the distance as $D = ABS(R_{Sample} - R_{All})/R_{All}$. That is, the distance metric gives an insight as to how far the set of sampled transactions deviate from the set of all transactions.

We conducted a series of experiments for a set of processors while running an OpenMP version of the CG benchmark from NAS Parallel benchmark suite [15]. We ran CG with six threads using the input set of size B. 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 with respect to our distance measure. In Table 1, the second column gives the distance values for maximum-rate sampling, and the third to sixth columns give results for interval sampling with different interval values. 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.

Table 1 shows that, even though the maximum-rate sampling can sample about 18% of all transactions, the distance metric

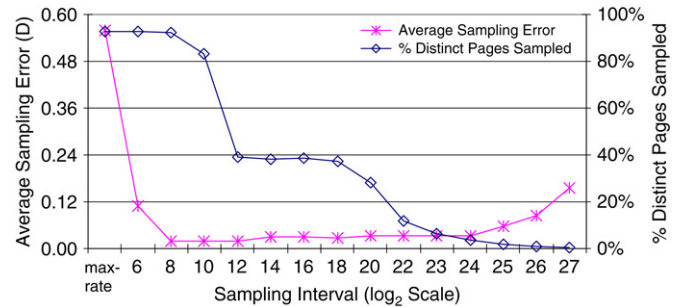


Fig. 2. Average distance and percentage of pages sampled in CG (B).

is significantly higher compared to interval sampling for all processors. 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.

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 are not 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, if the rate of interval sampling exceeds 5% of all transactions, the set of sampled transactions becomes less representative.

To further investigate how representative the samples for larger sampling interval values, we also conducted experiments varying the sampling interval up to every 128M address transactions. In addition, for each experiment, we also recorded the number of distinct pages that are included in the set of sampled transactions. Fig. 2 presents the average sampling error (left y-axis) and the percentage of distinct pages sampled (right y-axis) in the application for the intervals we tested.

Fig. 2 shows that the average sampling error is the highest for maximum-rate sampling and it starts decreasing dramatically as the sampling interval increases. Moreover, the average error stays low and steady for a large range of sampling intervals starting at every-256 transactions sampling to every-8M transactions sampling. The average sampling error starts to increase again after every-16M transaction sampling due to the fact that the number of samples taken is not large enough to accurately characterize all transactions in the application.

Fig. 2 also shows that, for maximum-rate sampling, 93% of all pages in the application are included in the samples taken. Similarly, for smaller intervals, the percentage of distinct pages sampled is around 90% for interval sampling. However, as the sampling interval increases, Fig. 2 shows that the percentage of distinct pages sampled in interval sampling decreases dramatically, resulting in many pages not included in the set of sampled transactions. Fig. 2 shows that even though interval sampling generates more representative samples, the percentage of the pages included in the samples decreases as the sampling interval increases.

5.2. Page migration experiments

To investigate the effectiveness of our actual dynamic page migration scheme on the performance of real applications, we conducted experiments using the OpenMP C implementation of the NAS Parallel Benchmark suite [15]. We chose applications with different sizes ranging from B to C (large data set sizes). We compiled the applications using Sun's native compiler, Sun C 5.5 EA2, with optimizations (`-xopenmp = parallel` and `-O3`) on to support parallelized code.

We conducted all of our experiments on a 24-processor Sun Fire 6800 with 24 GB of main memory. 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 the 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 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 sampled interconnect transactions at every 1024 transactions for experiments with page migration.

We ran all applications with 12 threads on six system boards of the Sun Fire 6800 server where each board executed two threads rather than running the applications with 24 threads where each processor is assigned a thread. This is due to the fact that we noticed a form of intra-board locality in the Sun Fire servers [18] that can mislead the benefits of page migration in isolation. We observed the choice of the processor from the group of processors on the same system board can also have an impact on the execution times of applications. We implemented a simple benchmark and measured the execution time of this benchmark when different processors in the same system board are used to execute the application. In each execution, to eliminate the effect of memory page placements, all memory pages in the benchmark are explicitly placed locally. We observed that our simple benchmark took up to 11% more time to execute when it is bound to the second processor of the system board compared to when it is bound to the first processor even though the array pages are placed local to the processors [18]. The intra-board variations can be partially explained by resource sharing contention among processors, bookkeeping operations running on processors by OS and 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. To eliminate any possible contention due to resource sharing among processors, we scheduled only two threads on each system board rather than four threads such that we would isolate the benefits of page migration alone for our experiments and the gain due to the page migrations is not overcome by the intra-board variations.

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 s.

For the experiments with page migration, the migration interval is given as a parameter to our dynamic page migration scheme.

To investigate the impact of migration intervals and choose the migration interval for the experiments, we conducted a sensitivity analysis in which we ran each application under different migration intervals ranging from 1 s to 50 s. Our experiments showed that the migration interval used does not have a major impact on the performance of the applications except MG. For MG, the migration interval has a significant impact due to the fact that MG is a short running program, and when migration is triggered at a slower rate, MG does not benefit from page migrations. Thus, for our page migration experiments, we chose to trigger page migration at every 5 s. We chose 5 s as the migration interval such that we would trigger a sufficient number of migrations in all applications to benefit from dynamic page migration but still keep a slower rate of migrations in the other applications for a lower overhead.

5.2.1. Reduction in non-local memory accesses due to page migrations

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.

Table 2 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 2 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 reduction for applications is 58.3%.) Table 2 also shows that, for MG, a significant number of non-local memory accesses were eliminated when memory pages were migrated. This is due to the fact that the first-touch policy in the underlying operating system placed pages poorly in a single memory unit and our migration policy was able to migrate pages to several memory units according to their access pattern.

Unlike MG, for LU our dynamic page migration approach was not able to reduce the number of non-local memory accesses significantly. For LU, the first-touch policy placed memory pages better. Moreover, system boards uniformly access the majority 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 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 may also be used to better decide whether a page should be migrated [19].

5.2.2. 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

Table 2
Reduction in non-local memory accesses due to page migration

| | # of address transactions (millions) | Percentage of non-local 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 3
Percentage change in the number of write-back transactions

| | # 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 |

migration approach does not have a significant impact on the cache behavior of applications. To quantify the cache usage of the applications, we counted the number of conflict and capacity misses (i.e. non-compulsory misses) during the execution of the applications with and without dynamic page migration using the Sun Fire Link monitors. The Sun Fire Link monitors measure non-compulsory misses 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 3 presents the number of WB transactions with and without our page migration approach.

Table 3 shows that our dynamic page migration approach does not have a significant effect on the cache behavior of applications. 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 and thus the absolute number of cache misses is more than a factor of 20 lower than any other application we measured. Moreover, since the working set of EP fits in local caches, the increase in cache misses in EP is mainly due to the invalidation of cache lines caused by migration of memory pages.

5.2.3. Execution times with page migration

While reducing the number of non-local memory accesses in an application is important, what matters is the impact of this reduction on the application's runtime. Thus, we measured 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 for each experiment.

First, we ran each application using our dynamic page migration approach and measured the total execution time including the overhead due to the creation of the helper threads and triggering 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 4 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 pages are migrated 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 4 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 4 shows that binding application threads to the processors is almost always beneficial even though the performance gain is not significant.

Table 4 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 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 partially reduce the page migration overhead.

Fig. 3 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, Fig. 3 also presents the migration overhead percentage with respect to the original execution time of the application. The migration overhead includes time spent for the suspension of all threads and moving pages to their target memory location. Fig. 3 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 dynamic page migration.

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 2).

