NUMA-Aware Java Heaps for Server Applications

Mustafa M. Tikir Jeffrey K. Hollingsworth
Computer Science Department
University of Maryland
College Park, MD 20742
{tikir,hollings}@cs.umd.edu

Abstract

We introduce a set of techniques to both measure and optimize memory access locality of Java applications running on cc-NUMA servers. These techniques work at the object level and use information gathered from embedded hardware performance monitors. We propose a new NUMA-aware Java heap layout. In addition, we propose using dynamic object migration during garbage collection to move objects local to the processors accessing them most. Our optimization technique reduced the number of non-local memory accesses in Java workloads generated from actual runs of the SPECjbb2000 benchmark by up to 41%, and also resulted in 40% reduction in workload execution time.

1. Introduction

The dominant architecture for the medium and large shared-memory multiprocessor servers is cache-coherent non-uniform memory access (cc-NUMA) machines. In cc-NUMA architectures, 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 300ns and 225ns, respectively where the latencies in high-range Sun Fire 15K servers are around 400ns and 225ns[1].

Prior research[2-5] has shown that dynamic page placement techniques on cc-NUMA systems are most effective for applications with regular memory access patterns, such as scientific applications. In these applications, large static data arrays that span many memory pages are divided into segments and distributed to multiple computation nodes resulting in one or a few computation nodes accessing each data segment most. For example, in prior work[6], we have shown that dynamically placing pages local to the processors accessing them most results in up to 16% performance improvement for a suite of OpenMP applications.

However, unlike scientific applications, Java programs tend to make extensive use of heap-allocated memory and typically have significant pointer chasing[7]. Thus, unlike scientific applications, dynamic page placement techniques may not be as beneficial for Java applications since they allocate many objects, with different access patterns, on the same memory page. Since the page placement mechanism used in the operating system is transparent to the standard allocation routines, the same memory page can be used to allocate many objects that are accessed by different processors. Due to Translation Lookaside Buffer size issues, cc-NUMA servers tend to use super pages of several megabytes, which further increase the likelihood of allocating the objects that have different access patterns on the same memory page. As a result, to better optimize memory access locality in Java applications running on cc-NUMA servers, heap objects should be allocated or moved so that objects that are mostly accessed by a processor reside in memory local to that processor.

In this paper, we propose a set of techniques to both measure and optimize the memory access locality of Java server applications running on cc-NUMA servers. These techniques exploit the capabilities of fine grained hardware performance monitors to provide data to automatic feedback directed locality optimization techniques. We propose the use of several NUMA-aware Java heap layouts for initial object allocation and use of dynamic object migration during garbage collection to move objects local to the processors accessing them most.

We also evaluate the potential of existing well-known locality optimization techniques and present the results of a set of experiments where we applied a dynamic page migration scheme to a Java server application. In our experiments, we used the dynamic page migration scheme we previously proposed[6].

2. Hardware and Software Components

In this section, we describe the hardware and software components we used in this research.

2.1. Sun Fire Servers and Hardware Monitors

In our measurements, we used a Sun Fire 6800 server which is based on the UltraSPARC III processors. It supports up to 24 processors and 24 memory units which are grouped into 6 system boards. Each processor has its own on-chip and external caches. The machine uses a single snooping coherence domain that spans all devices connected to a single Fireplane address bus.

In Sun Fire servers, 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 on a Sun Fire 6800 server is around 225ns if the memory is local and 300ns if it is non-local[1].

The Sun Fire Link Bus Analyzer[8] has an 8-deep FIFO that records a limited sequence of consecutive address transactions on the system interconnect. Each recorded transaction includes the requested physical address, the requestor processor identifier, and the transaction type. The bus analyzer is configured with mask and match registers to select events based on specific transaction parameters.

The information the bus analyzer provides about the addresses in the transactions is at the level of physical addresses. Thus, to accurately evaluate the memory performance of an application, the address transactions have to be associated with virtual addresses used by the application. We used the *meminfo* system call in Solaris 9 to create a mapping between physical and virtual memory pages in the applications.

The Sun Link bus analyzer is a centralized hardware that listens to the system interconnect. However, the techniques we propose in this paper do not require use of such centralized hardware. Alternatively, since most processors now include hardware support for performance monitoring, on-chip hardware monitors of the processors in a multiprocessor server can also be used to gather profiles required by our techniques in a distributed fashion.

2.2. Java HotSpot Server VM (version 1.4.2)

For efficient garbage collection, the Java HotSpot VM exploits the fact that a majority of objects die young[9]. To optimize garbage collection, heap memory is managed in generations, which are memory pools holding objects of different ages, as shown in Figure 1. Each generation has an associated type of garbage collection that can be configured to make different time, space and application pause tradeoffs.

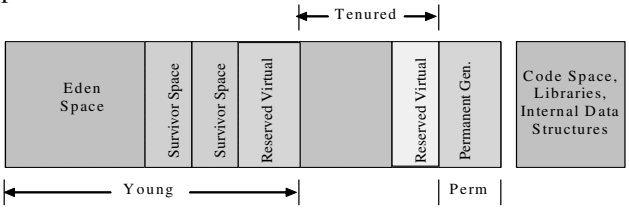

Figure 1 The default memory layout of HotSpot VM.

Garbage collection happens in each generation when the generation fills up. Objects are initially allocated in the young generation. Because of infant mortality, most objects die in the young generation. When the young generation fills up it causes a *minor* collection. Minor collections can be optimized assuming a high infant mortality rate. A young generation full of dead objects is collected very quickly. Some surviving objects are moved to a tenured generation depending on how many minor collections they survived. When the tenured generation needs to be collected, there is a *major* collection, which is often much slower because it involves all live objects.

2.3. The SPECjbb2000 Benchmark

The SPECjbb2000 is a benchmark for evaluating the performance of servers running typical Java business applications. The performance metric used is the throughput in terms of transactions per second.

The SPECjbb2000 represents an order processing application for a wholesale supplier[10] with multiple warehouses. This benchmark loosely follows the TPC-C specification for its schema, input generation, and transaction profile. SPECjbb2000 replaces database tables with Java classes and data records with Java objects. SPECjbb2000 does no disk I/O. It runs in a single JVM.

The SPECjbb2000 emulates a 3-tier system. The middle tier, which includes business logic and object manipulation, dominates the other tiers of the system. Clients are replaced by driver threads with random input to represent the first tier. The third tier is represented by binary trees rather than a separate database and database storage is implemented using in-memory binary trees of objects.

We chose to use SPECjbb2000 for our measurements to be able to isolate the impact of our optimization techniques on the memory performance of the Java server applications. An alternative benchmark is the SPECjAppServer[11] benchmark. However, this benchmark tests performance for a representative J2EE application and each of the components that make up the application environment, including hardware, application server software, JVM software, database software, JDBC drivers, and the system network.

3. Optimizing with Dynamic Page Migration

Prior to evaluating our new object centric optimization techniques, we quantify the impact of existing optimization techniques on the memory access locality of Java applications. Such quantification enables us to compare the effectiveness of specialized techniques with respect to a more general technique and to verify the need for such specialized techniques.

As a general locality optimization technique, we choose dynamic page migration since this technique has been studied extensively and is known to yield performance improvements for many scientific applications running on cc-NUMA servers. For our experiments, we choose the dynamic page migration scheme we had developed for OpenMP applications[6].

To quantify the impact of dynamic page migration on memory access locality of SPECjbb2000, we ran SPECjbb2000 for 6, 12, 18 warehouses with and without

dynamic page migration. For each number of warehouses, we counted the number of non-local memory accesses and measured the percentage reduction in the number of non-local memory accesses due to dynamic page migration compared to the original execution. We also measured the percentage improvement in the throughput for each number of warehouses when dynamic page migration is used.

Table 1 presents the percentage reduction in the total number of non-local memory accesses when dynamic page migration is used. The first column gives the number of page migrations triggered. The third column gives the percentage of non-local memory accesses without page migration and the fourth column shows the percentage of non-local memory accesses with page migration. The fifth column lists the percentage reduction in the total number of non-local memory accesses when page migration is used. The sixth column gives the performance improvement in the throughput.

# of Warehouses	# of Migrations	Non-Local Memory Accesses			% Improvement
		w/o Mig.	with Mig.	Reduction	
6	69,796	72.0 %	52.3 %	27.4 %	-2.8 %
12	145,607	77.0 %	58.1 %	24.5 %	-3.4 %
18	165,794	77.5 %	61.3 %	20.9 %	-3.1 %

Table 1 Performance improvement due to dynamic page migration for SPECjbb2000.

Table 1 shows that running SPECjbb2000 with dynamic page migration, the number of non-local memory accesses are reduced around 25% for all configurations compared to not using dynamic page migration. It also shows that dynamic page migration was not able to improve the throughput for any configuration even though it reduced the number of non-local accesses. Instead, dynamic page migration reduced throughput around 3% since the reduction in non-local accesses did not overcome the overhead introduced by migrating many pages.

More importantly, Table 1 shows that unlike scientific applications where the reduction in the number of non-local memory accesses can be as much as 90%[6], dynamic page migration was not as effective in reducing the number of non-local memory accesses for SPECjbb2000. We suspect this is due to fact that objects that are accessed mostly by different processors are allocated in the same memory page. Instead, to better optimize memory access locality for this type of workload on a cc-NUMA server, we object level migration will be more effective.

4. Inadequacy of Page Level Optimization

Java programs tend to make extensive use of heap-allocated memory and typically have significant pointer chasing. Since typical object sizes are much smaller compared to the commonly used memory page sizes, Java applications are likely to allocate many objects in the same memory page. Moreover, if an application uniformly accesses the objects in a page, a page level memory locality optimization technique may not be as effective in reducing the number of non-local memory accesses to the page.

To investigate whether page level optimization techniques, such as dynamic page migration, are too coarse grained to be effective in reducing the number of non-local memory accesses in Java server applications, it is necessary to measure the memory behavior of these applications at the object granularity.

4.1. Measuring Memory Access Locality

To gather information about the object allocations by a Java application and the internal heap allocations required by the virtual machine, we modified the source code of the HotSpot VM. For each heap allocation, we inserted constructs to record the type of the allocation (i.e. object, array, and code buffer), the address and size of the allocation and the requestor thread. To capture the changes in the addresses during garbage collection, we also modified the source code of garbage collection modules in the HotSpot VM and inserted additional instrumentation code. For each surviving object, the instrumentation code records the new and the old addresses of the object.

We only instrument object allocations that survive one or more garbage collections. During each garbage collection, we map the newly surviving objects back to the corresponding object. Since most of the objects die before one garbage collection, we eliminate overhead due to very short lived objects.

We used the Sun Fire Link monitors to sample the address transactions during the execution of the application and later associate those transactions with the corresponding objects. Even though the information collected by the hardware monitors is sampled and does not include every access, it provides sufficiently accurate profiling information. More importantly, since the monitors are implemented in hardware level, they neither interfere with the memory behavior of the application running nor introduce significant overhead[1].

Our memory access locality measurement algorithm is a two phase algorithm. During the *execution* phase, we run the application on the modified virtual machine to gather information about the heap allocations and to sample the address transactions via hardware performance counters. At the end of execution phase, we generate a trace of heap allocations and memory accesses by the processors. In the *post-processing* phase, we process the generated trace and report measurement results.

[1] We take samples after a fixed number of transactions since our earlier work [7] on dynamic page migration has shown that sampling address transactions at fixed transaction boundaries produces samples that are more representative of the overall transactions compared to other sampling techniques.

We first instrument the executable of the virtual machine at the start of *main* function to create an additional helper thread for sampling the address transactions. We use DyninstAPI[12] to insert the instrumentation code. Moreover, to eliminate perturbation of sampling on the address transactions and memory behavior of the target Java application, we bind the helper thread to execute on a separate processor that does not run any of the threads in the application. The helper thread initializes some instrumentation structures and samples address transactions via the Sun Fire Link monitors for the remainder of the run.

Our algorithm divides the execution of Java applications into distinct intervals. We refer to the time period from the start of a garbage collection until its termination as *garbage collection* interval and the time period between two consecutive garbage collection iterations as *execution interval*.

We do not sample address transactions during garbage collection intervals since current Java virtual machines are engineered to have a small memory footprint that would likely not have a significant impact on the memory behavior of the applications. To associate address transactions with heap allocations during the post-processing phase of our algorithm, we need to store the order information for both address transactions and allocation records. Thus, we use the index of the last sampled address transaction, which is maintained by the helper thread.

The post-processing phase combines the allocation records and address transactions recorded during each execution interval and sorts them according to the order they are requested during the execution. It then tries to associate address transactions with allocation records generated during the same execution interval. If a transaction is not associated with an allocation record in the execution interval being processed, the post processing phase tries to associate the same transaction with an allocation record that is recorded during an earlier execution interval. At termination, the post-processing phase reports memory access locality both for total and non-local accesses.

4.2. Experimental Results

We now present the results of experiments we conducted to measure the memory access locality in a finer granularity for SPECjbb2000 running on HotSpot Server VM. During our experiments, we observed that SPECjbb2000 exhibits similar memory access locality regardless of the number of warehouses. Thus, due to space limitations, we only present the results of SPECjbb2000 for 12 warehouses. In these experiments, we sampled the address transactions every 512 transactions.

Prior to describing the results of experiments, we briefly discuss the execution overhead and perturbation in SPECjbb2000 introduced by our measurements. The results of our experiments show that the throughput of SPECjbb2000 is reduced by 3% due to our source code instrumentation of HotSpot VM. In addition, we observed that 0.08% percent of all address transactions sampled are associated with the additional buffers we used to store allocation records and sampled transactions. Thus, our measurement has neither a significant impact on the execution performance nor a significant perturbation on the memory behavior of SPECjbb2000.

Our measurement technique gathered 10M allocation records during the execution of SPECjbb2000 with 12 warehouses. In addition, it took 33M samples from the address transactions in the system interconnect. The post-processing phase of our technique associated 97.4% of the samples taken with an allocation. That is, 2.6% of all samples taken were not associated with any allocation during the execution. The majority of the unassociated address transactions fall into the code space of the HotSpot VM.

Allocation Type	Number of Allocations	Memory Accesses		Non-Local Accesses	
		Count	Percentage	Count	Percentage
thread local buffer (tlab)	85,351	11,490,677	34.5	9,632,456	83.8
object	9	1	0.0	0	0.0
array	18	159	0.0	3	1.9
large array	1	224	0.0	0	0.0
permanent object	34,907	220,596	0.7	179,899	81.6
permanent array	9,516	15,593	0.0	11,433	73.3
scavenge survivor move	7,376,785	435,170	1.3	354,129	81.4
scavenge old move	602,940	1,849,777	5.6	1,732,677	93.7
compact move	1,932,844	14,628,184	43.9	12,107,329	82.8
active table	1	17,113	0.1	13,259	77.5
code cache	1	3,511,678	10.5	2,821,164	80.3
stack	27	125,564	0.4	102,154	81.4
memory chunks	249	35,644	0.1	18,970	53.2
jni handles	86	65,938	0.2	65,673	99.6

Table 2 Detailed measurement results for memory behavior of SPECjbb2000 with 12 warehouses.

Table 2 presents detailed results of our experiments. In the second column, it gives the number of allocation records gathered from each type of heap allocation. The third and fourth columns give the number of memory accesses associated with the heap allocations for the corresponding allocation type and the percentage of the associated transactions among all transactions. The fifth column gives the number of non-local accesses, and the sixth column presents the percentage of non-local memory accesses for each allocation type. Table 3 presents the results for associated transactions presented in Table 2 for each heap segment in Java heap.

Table 2 shows that the majority of allocation records we recorded were due to garbage collection of surviving objects. It also shows that there are only a few heap allocations for internal data structures used by the virtual machine itself whereas there are moderate numbers of thread-local and permanent allocations.

More importantly, Table 2 shows that accesses to heap allocated objects are mainly due to the Thread Local Allocation Buffers (TLAB) allocated from the eden space of the Java heap and the surviving objects moved into old generation during garbage collection. TLABs are the thread-local storage used by the threads for object allocations in the young generation. Table 2 and 3 show that around 12% of accesses are associated with the internal structures and permanent allocations by the virtual machine and 10% of these accesses are due to the code cache used for interpreter and Java method codes. That is, even though HotSpot VM contributes to the memory behavior of the SPECjbb2000, its contribution is not significant.

Java Heap Region	Memory Accesses		% Non-Local Accesses
	Count	% of All Accesses	
Young Gen.	11,926,231	35.8	83.7
Eden Space	11,389,586	34.2	83.8
Survivor Space	536,645	1.6	82.7
Old Gen.	16,477,990	49.5	84.0
Permanent Gen.	236,189	0.7	81.0
Internal Structures	3,755,937	11.3	80.4

Table 3 Memory activity per Java heap region.

Overall, Table 2 and 3 show that Java server applications are good candidates for memory locality optimizations due to the high percentage of non-local memory accesses. Table 2 and 3 also show that the memory behavior of SPECjbb2000 is mostly defined by the heap allocations and memory accesses it requested and the memory behavior of Java virtual machine is not significant since only a small percentage of memory accesses are due to the internal data structures used by the virtual machine. Thus, locality optimization techniques that focus on optimizing the memory behavior of an application rather than the memory behavior of the underlying virtual machine hold the greatest opportunity.

4.3. Estimating Potential Optimization Benefits

To investigate whether page level optimization techniques are too coarse to optimize the memory locality of Java server applications, it is necessary to investigate potential benefits of possible finer grain optimization techniques. In this section we present an estimation study that roughly predicts the benefits of possible finer grain optimization techniques. The estimation study is based on the heap allocations and accesses gathered during our measurement experiments.

In this study, we consider three object level placement techniques. *Static-optimal* placement has information about all accesses to each heap allocation by processors during the execution and places objects in the memory pages local to the processors that access them most at allocation time. *Prior-knowledge* placement has information about the accesses to each surviving allocation during the next execution interval and moves allocations to the memory pages local to the processors accessing them most in garbage collection intervals. Lastly, *object-migration* placement uses object access frequencies by processors since the start of execution up to the current time. At garbage collection, it migrates heap allocations to memory local to the processors that access them most.

In this estimation study, we measured the potential reduction in the number of non-local memory accesses for each placement technique using heap allocation records and memory accesses we gathered using our measurement tool. Figure 2 presents the percentage of non-local memory accesses in the original execution of SPECjbb2000 as well as using each placement technique.

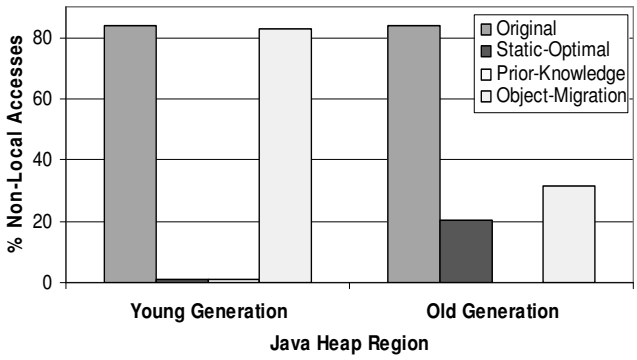

Figure 2 Potential reduction in non-local memory accesses for object level optimization techniques.

Figure 2 shows that heap allocations in the young generation would significantly benefit from both static-optimal and prior-knowledge placement. It also shows that object-migration would not be effective in reducing the number of non-local memory accesses in young generation. Figure 2 also shows that the heap allocations in the

old generation would also benefit from static-optimal and prior-knowledge placements. Unlike heap allocations in the young generation, allocations in the old generation however would benefit from object migrations.

Figure 2 shows that the prior-knowledge placement is more effective in the old generation compared to other placement techniques. It also shows that the static-optimal placement alone yields a significant reduction in non-local accesses in the old generation. This indicates SPECjbb2000 has some dynamically changing memory behavior in the old generation. More importantly, Figure 2 shows that dynamic object migration responds to this changing behavior quite well and yields a significant reduction in the number of non-local memory accesses in the old generation.

In Figure 2, the significant reduction in the number of non-local memory accesses in the young generation for the static-optimal placement indicates that heap allocations in the young generation are mostly accessed by single processors. Thus, we further investigated heap allocations in the young generation. We found that 94% of all accesses to the heap allocations in the young generation are requested by the same processor that requested the allocation. This can be explained with the fact that each thread allocates its TLABs from young generation in which the thread allocates its objects. Moreover, since the mortality rate for the objects in TLABs are high, most of the accesses to those allocations are most likely to be from the same thread. Thus, if thread local buffers were placed local to the processor thread is running on, a substantial memory access locality would be possible.

The thread local allocation buffers were initially created as a way to reduce synchronization overhead for multithreaded applications on UMA multiprocessor systems. Extending them to improve memory access locality on NUMA multiprocessor systems is described in Section 5.

Using the fact that 94% of all observed accesses to the heap allocations in the young generation are requested by the same processors that allocated them, we calculated the potential reduction in the number of non-local memory accesses for a hybrid optimization technique. The hybrid optimization technique places heap allocations local to the processors that requested them in the young generation and uses dynamic object migration in old generation. We have found that such hybrid technique would reduce the number of non-local memory accesses by 73%.

5. NUMA-Aware Java Heap Layouts

To optimize memory access locality of Java server applications, we propose the use of two different Java heap configurations. The first one, *NUMA-Eden*, uses a NUMA-aware young generation and the original old generation of the HotSpot VM we used. The second one, *NUMA-Eden+Old*, uses both NUMA-aware young generation and NUMA-aware old generation.

The *NUMA-Eden* configuration focuses on optimizing the locality of the accesses to the objects in the young generation where as the *NUMA-Eden+Old* configuration focuses on optimizing the locality of the accesses to the objects in young and old generations. The *NUMA-Eden+Old* is more likely to be more effective than the *NUMA-Eden* since it targets all memory accesses in the application. However, it requires gathering object access frequencies by processors at runtime.

5.1. NUMA-Aware Young Generation

To optimize the locality of memory accesses to the objects in the young generation, we propose to divide *eden* space in the young generation into segments where each locality group of processors is assigned a segment. We do not change the layout of survivor spaces due to the fact that memory accesses to the survivor spaces throughout the execution of Java sever applications is insignificant compared to memory accesses to *eden* space. In addition, we divide the *eden* space to equal sized segments. Figure 3 shows the layout for the young generation.

To allocate objects in the young generation in the proposed layout, the virtual machine needs to identify the processor that the requestor thread runs on, and place the object in the segment of the corresponding locality group of the processor. If application threads are bound to execute on fixed processors in the cc-NUMA server or affinity scheduling is used in the underlying OS, virtual machines can easily identify the processor an application thread runs through OS provided system calls.

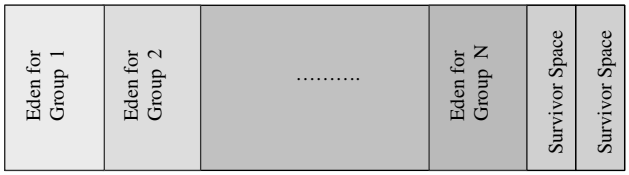

Figure 3 The NUMA-aware young generation layout.

When there is not enough space for object allocations in the young generation, the java virtual machine triggers minor garbage collection. For our NUMA-aware allocator, the java virtual machine may trigger minor collections in several ways; one approach is to trigger minor garbage collection when there is not enough space in a segment for object allocation. However, such an approach may trigger minor collections more often compared to original heap due to fragmentation caused by dividing the region into locality based segments. Alternatively, the virtual machine may fall back to its original behavior and allocate the objects from segments that have enough space, thus eliminating additional minor collections. Since minor collection algorithms are engineered to be fast, we believe additional minor collections will not have a significant impact on the execution performance of Java server applications. Thus, in this paper, when a segment does not have enough space

for object allocations, we trigger minor garbage collection. Moreover, we collect all segments, even if they are not full, in parallel to eliminate future synchronization due to minor collection requests by other segments.

A NUMA-aware young generation may also have additional benefits. If garbage collection threads are bound to execute on processor groups and each collector thread collects the dead objects in the *eden* space associated with the same locality group that the thread is bound to execute, the memory access locality of garbage collection threads can be optimized. Since the garbage collection threads are known to suffer cold cache misses, such optimization can improve the performance of garbage collection threads.

5.2. NUMA-Aware Old Generation

Our experiments described in Section 4 show that almost 50% of all memory accesses are accesses to the objects in the old generation. Moreover, they also show that 84% of accesses to the objects in the old generation are non-local memory accesses. Thus, for fine grain memory locality optimization techniques to be effective for Java server applications, they should also optimize memory access locality for the objects in the old generation. Moreover, to optimize the memory access locality for the objects in the old generation, the fine grain optimization techniques should try to keep the objects local to the processors accessing them most during the lifetimes of objects. We refer to the location of an object as the *preferred location* if the object is placed in a memory page that is local to the processor accessing it most. To be able to identify the processors accessing the objects most, we use the address transaction samples taken form hardware performance counters as described in Section 4.

When an object is promoted to the old generation, it stays in the old generation during the rest of its lifetime. If the object survives another full collection after being promoted to the old generation, its address may change due to the copying collector. More importantly, the object may be accessed by different processors during the distinct intervals of its lifetime. Thus, for a fine grain optimization technique to be effective, it should for each old generation object identify the preferred location of the object and place the object to its preferred location during garbage collection.

To optimize the locality of memory accesses to the objects in the old generation, we propose a dynamic object migration scheme. In this scheme, when an object is promoted to old generation during minor garbage collection, the preferred location of the object is identified and the object is placed in its preferred location. During full garbage collection, the preferred location of each object in the old generation is identified and the object is placed at its preferred location. Moreover, to match the dynamically changing behavior of the application, after a fixed number of minor garbage collections, the preferred locations of the objects in the old generation are re-computed and objects are migrated as needed.

After the preferred location of an object is identified, the virtual machine needs a means to place the object in its preferred location. Thus, similar to NUMA-aware young generation, we propose to divide old generation into segments where each locality group of processors is assigned a segment (Figure 4)[2].

Figure 4 The NUMA-aware old generation layout.

5.3. Experimental Setup

To evaluate the effectiveness of fine grained optimization techniques based on the proposed heap layouts, we evaluated our approach using a hybrid execution simulator. To drive our simulation, we generated a representative workload from an actual run of the Java server application[3].

A workload generated for an actual run must be representative of the memory behavior of the actual run. In Section 4.1, we described how we gather a representative trace of heap allocations and accesses to those allocations to measure the memory access locality of Java server applications. To generate a representative workload for an actual run, we use the sampled traces generated by our measurement tool and extrapolate to full workload. A workload is a sequence of object allocation and access requests by processors in the same order they were requested during the actual run of the application.

For each trace entry for a surviving object, we first identify the trace entry that is the source of the object. If the source of the surviving object is a TLAB allocation, we insert a separate allocation request into the workload following the request for the TLAB. We manipulate TLABs and objects allocated in them separately for easy tracking of live objects for garbage collection.

For each trace entry that accesses an allocation, we first identify the trace entry for the heap allocation that is being accessed. If the heap allocation is of type other than TLAB, we identify the corresponding allocation request in the workload. Moreover, if the heap allocation corresponds to a surviving object that has survived multiple garbage collections, we map the object back to the source where the object is allocated. After identifying the allocation request, we insert an access request for it into the workload.

[2] The VM needs to adaptively size the heap segment for each group.

[3] We chose to implement heap management algorithms as separate programs due to the code size and complexity of the JVM implementation as well as to allow controlled experiments not possible in a live system.

For each request inserted in to the workload, the information required for the execution of the request is also extracted from the trace entry it corresponds to. For all requests, we also extract the locality group that will execute the action from the trace entry the action is inserted for.

We implemented a separate program, Workload Execution Machine (WEM), to consume the generated workload trace and issue the memory allocations and accesses to the machine. The WEM takes both a workload and a heap configuration as input and runs the workload using the heap configuration given. At termination, WEM reports the total time spent to run the workload and the total time spent for garbage collection in addition to the number of local and non-local accesses to the heap objects.

5.4. Experimental Results

We conducted experiments using the Workload Execution Machine by running the workload generated from an actual run of the SPECjbb2000 for 12 warehouses on the HotSpot Server VM.

To investigate the impact of higher memory pressures on the effectiveness of the proposed heap configurations, we scaled the sampled set of objects accesses by 16 and 32 times[4]. We chose two different scaling rates to investigate the impact of memory pressure increase on the effectiveness of the proposed heap configurations.

In each workload we generated, accesses to 42% of the object allocations did not exist in the samples taken. We believe these objects were short-lived, thus accesses to them were under sampled. Since we initialize all objects we allocate, similar to the HotSpot VM, we guarantee these objects are touched once.

We ran each workload we generated under three heap configurations, *Original*, *NUMA-Eden* and *NUMA-Eden+Old*. For NUMA-Eden+Old configuration, we triggered dynamic object migration after every 3 minor collections. We chose to trigger object migration after every 3 minor collections for a slower rate of object migrations since our earlier work on page migration has shown that slower rate of migrations are more beneficial[6].

The number of garbage collections triggered for the Original, NUMA-Eden, and NUMA-Eden+Old is 10, 13 and 13 respectively, of which 2 are full collections. The full garbage collections are forced garbage collections requested by the SPECjbb2000 rather than full collections triggered due to lack of heap space. The NUMA-aware heap configurations trigger more minor garbage collections compared to the original heap configuration since in these configurations, a minor collection is executed when a segment for a locality group in the young generation is full, even if the others still have available space.

[4] Replicating the full workload of the original program requires scaling the sampled set by a factor of 70. We chose lower rates due to large memory and disk space requirements of scaled workload.

5.4.1. Reduction in Non-Local Memory Accesses

We conducted a series of experiments where we ran the generated workloads. In these experiments we changed the underlying heap configuration and measured the percentage of the non-local memory accesses to the heap objects. Figure 5 presents the percentage of the non-local accesses to the objects allocated in young and old generations for each heap configuration compared to all accesses to the objects in each generation. It also presents the percentage of the non-local accesses to all objects compared to all accesses. In addition, Table 4 gives the percent reduction in the number of non-local objects accesses for the NUMA-aware heap configurations with respect to the original heap configuration.

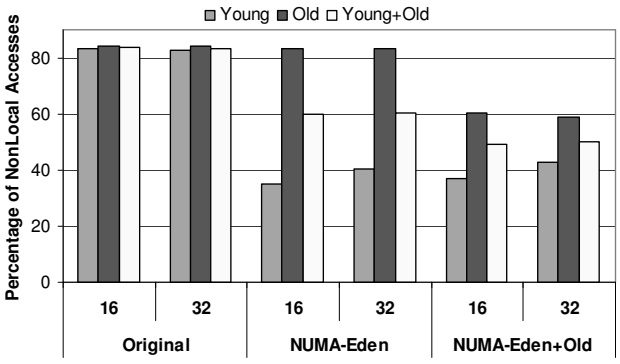

Figure 5 Non-local accesses by heap configuration.

Scale Factor	Heap Configuration	Young Gen.	Old Gen.	All Gen.
16	**NUMA-Eden**	57.6 %	0.3 %	28.1 %
	NUMA-Eden+Old	55.3 %	27.5 %	41.0 %
32	**NUMA-Eden**	50.9 %	1.2 %	27.3 %
	NUMA-Eden+Old	48.0 %	30.2 %	39.5 %

Table 4 Reduction in non-local memory accesses for each heap configuration.

Figure 5 shows that the percentage of non-local object accesses for the original heap configuration is over 80% for all workloads. Moreover, it also shows that our NUMA-Eden heap configuration was able to reduce the number of non-local object accesses by around 28% compared to the original heap configuration whereas the NUMA-Eden+Old configuration reduced the number of non-local object accesses by 39-41% for the workloads. Figure 5 also shows that unlike the NUMA-Eden configuration, using a NUMA-Eden+Old configuration reduces the number of non-local object accesses in the old generation. This is due to the fact that NUMA-Eden uses the original old generation where as NUMA-Eden+Old uses NUMA-aware old generation with object migration.

Figure 5 shows that even though NUMA-Eden and NUMA-Eden+Old use the same layout for the young generation, there is a slight difference in the percentage of

non-local memory accesses in the young generation for these configurations. This is due to the differences in actual page placements in the survivor spaces where we do not use a NUMA-aware layout.

Figure 5 and Table 4 show that NUMA-aware heap configurations are effective in reducing the total number of non-local objects accesses. They also show that using both NUMA-aware young and old generation is more effective in reducing the number of non-local object accesses for each workload compared to using only NUMA-aware young generation. Table 4 also shows that using both NUMA-aware young and old generations reduced the number of non-local object accesses in the workloads by about 40%.

5.4.2. Execution Times

For each experiment, we also measured the total time spent to execute each workload. Figure 6 presents the normalized workload execution time for each heap configuration with respect to the workload execution time for original heap configuration. Figure 6 also presents the normalized time spent for garbage collection. The bottom segment of each bar is just the execution time spent to run each workload whereas top segment of each bar is for the garbage collection time.

Figure 6 shows that the garbage collection times for NUMA-Eden and original heap configurations are comparable even though NUMA-Eden triggers more minor garbage collections. This is due to the fact that minor garbage collection is fairly cheap since it both is executed by multiple GC threads in parallel and does not copy many objects due to the high mortality rate of young objects.

Figure 6 also shows that for each workload, both NUMA-Eden and NUMA-Eden+Old configurations outperform the original heap configuration in terms of workload execution time. While the NUMA-Eden reduces the workload execution times by up to 27%, the NUMA-Eden+Old reduces the workload execution times by up to 40%. Moreover, it also shows that using both NUMA-aware young and old generation is even more effective compared to using only NUMA-aware young generation.

More importantly, Figure 6 shows that as the workload size increases, the effectiveness of the NUMA-aware heap configurations increases. It shows that NUMA-aware heap configurations were able to reduce the workload execution time for the workload that is generated by scaling the original workload 16 times by around 20% compared to original heap configuration, whereas the reduction in the workload execution time by up to 40% for the workload generated by a higher scaling rate of 32. Thus, Figure 6 shows that NUMA-aware heap configurations are effective in the presence of higher memory pressure.

Overall, our experiments show that NUMA-aware heap configurations are effective in reducing the number of non–local memory accesses and workload execution times for Java server workloads. Our approach was able to reduce the number of non-local memory accesses in SPECjbb2000 workloads by up to 41%, and also resulted in 40% reduction in workload execution time.

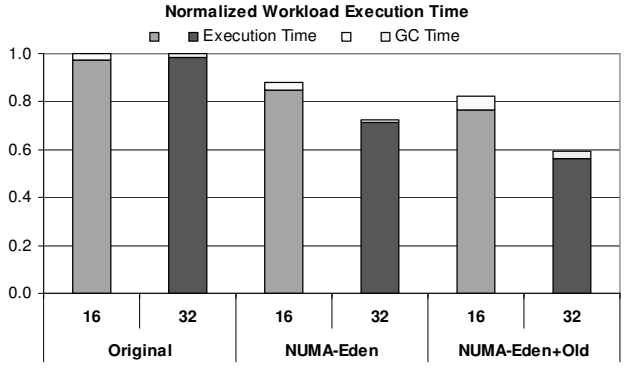

Figure 6 Normalized workload execution time for each scaling factor and heap configuration.

6. Related Work

Most of the prior research on optimizing the locality on cc-NUMA architectures has been in the area of page migration and replication. Unlike our work in this paper, prior research has focused on optimizing the memory access locality of scientific applications using coarse grain optimization techniques. Chandra et al.[13] investigated the effects of different OS scheduling and page migration policies for cc-NUMA systems using Stanford DASH multiprocessors. Verghese[4] studied the operating system support for page migration and replication in cc-NUMA multiprocessors. Noordergraaf and Pas[3] also evaluated page migration and replication using a simple HPC application on the he Sun WildFire servers. More recently, Bull and Johnson[14] studied the interactions between data distribution, migration and replication for the OpenMP applications. More similar to our work is Wilson and Aglietti[5] who used dynamic page placements to improve the locality for TPC-C on cc-NUMA servers.

Karlsson et al.[15] presented memory system behavior of the application servers in ECperf and SPECjbb2000 benchmarks running on commercial cc-NUMA multiprocessor servers and found that a large fraction of the working data sets is shared among processors. Marden et al.[16] studied the memory system behavior of the server applications in the Java versions of SPECweb99 benchmark and reported that the cache miss rate becomes worse for the Java implementation when the size of the cache is increased. Unlike these papers, our work focuses on per-thread memory behavior of server applications, and optimizes the locality of memory accesses in these applications.

Shuf et al.[7] presented a Java allocation-time object placement technique that co-locates heap objects based on the notion prolific types. Calder et al.[17] presented a compiler-directed technique for cache conscious data

placement for pointer-intensive applications. Chilimbi et al.[18-20] described several techniques for improving cache locality of pointer-intensive applications.

Berger et al.[21] introduced the Hoard memory allocator for multithreaded applications to avoid false sharing by using per-processor heaps. Steensgaard[22] and Domani et al.[23] investigated the benefits of using thread-local heaps to improve garbage collection performance in Java applications. Unlike these papers, our techniques work on objects shared among threads and use dynamic analysis of object access frequencies during program execution.

7. Conclusions

In this paper, we introduced new NUMA-aware Java heap layouts and dynamic object migration to optimize the memory access locality of Java server applications running on cc-NUMA servers and investigated the impact of these layouts on the memory performance of SPECjbb2000. We evaluated the effectiveness of our techniques using workloads we generated from actual runs of SPECjbb2000.

Our proposed Java NUMA-aware heap layouts always reduced the total number of non-local object accesses in SPECjbb2000 compared to the original Java heap layout used by the HotSpot VM by up to 41%. Moreover, our proposed NUMA-aware heap layouts reduced the execution time of Java workloads generated from actual runs of SPECjbb2000 by up to 40% compared to original layout in the virtual machine.

We have shown that using both the NUMA-aware young and old generations combined with dynamic object migration is more effective in optimizing the memory performance of SPECjbb2000 compared to using only the NUMA-aware young generation. Lastly, we have shown that as the memory pressure increases in the Java server applications, our proposed NUMA-aware heap configurations are more effective in improving the memory performance of Java server applications. We believe the use of NUMA-aware heap layouts will be even more effective in improving the performance of Java server applications running on larger cc-NUMA servers where difference in access times between local and non-local memory accesses is larger.

Acknowledgements

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] A. Charlesworth, The Sun Fireplane System Interconnect, ACM IEEE SC'01, Denver, CO, 2001.

[2] R. P. LaRowe, C. S. Ellis, L. S. Kaplan, The Robustness of NUMA Memory Management, SOSP, Pacific Grove, CA, 1991.

[3] L. Noordergraaf, R. van der Pas, Performance Experiences on Sun's WildFire Prototype, ACM IEEE SC'99, Portland, OR.

[4] B. Verghese, S. Devine, A. Gupta, M. Rosenblum, Operating System Support for Improving Data Locality on CC-NUMA Compute Servers, ASPLOS, Cambridge, MA, 1996.

[5] K. M. Wilson, B. B. Aglietti, Dynamic Page Placement to Improve Locality in CC-NUMA Multiprocessors for TPC-C, ACM IEEE SC'01, Denver, CO, 2001.

[6] M. M. Tikir, J. K. Hollingsworth, Using Hardware Counters to Automatically Improve Memory Performance, ACM IEEE SC'04, Pittsburgh, PA, 2004.

[7] Y. Shuf, M. J. Serrano, M. Gupta, J. P. Singh, Characterizing the Memory Behavior of Java Workloads: A Structured View and Opportunities for Optimizations, ACM SIGMETRICS, Cambridge, MA, 2001.

[8] L. Noordergraaf, R. Zak, SMP System Interconnect Instrumentation for Performance Analysis, ACM IEEE SC'02, Baltimore, MD, 2002.

[9] Sun Microsystems, Tuning Garbage Collection with the 1.4.2 JVM, 2003, http://java.sun.com/docs/hotspot/gc1.4.2/.

[10] Standard Performance Evaluation Council, SPECjbb2000 Benchmark, 2000, http://www.spec.org/osg/jbb2000.

[11] Standard Performance Evaluation Council, SPECjAppServer Development Page, 2000, http://www.spec.org/osg/jAppServer.

[12] B. R. Buck, J. K. Hollingsworth, An API for Runtime Code Patching, International Journal of High Performance Computing Applications, 14(4), 2000, p. 317-329.

[13] R. Chandra, S. Devine, B. Verghese, A. Gupta, M. Rosenblum, Scheduling and Page Migration for Multiprocessor Compute Servers, ACM ASPLOS, San Jose, CA, 1994.

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

[15] M. Karlsson, K. E. Moore, E. Hagersten, D. A. Wood, Memory System Behavior of Java-Based Middleware, HPCA, Anaheim, CA, 2003.

[16] M. Marden, S.-L. Lu, K. Lai, M. Lipasti, Comparison of Memory System Behavior in Java and Non-Java Commercial Workloads, Workshop on Computer Architecture Evaluation using Commercial Workloads, Cambridge, MA, 2002.

[17] B. Calder, C. Krintz, S. John, T. Austin, Cache-Conscious Data Placement, ACM ASPLOS, San Jose, CA, 1998.

[18] T. M. Chilimbi, B. Davidson, J. R. Larus, Cache-Conscious Structure Definition, ACM PLDI, Atlanta, GA, 1999.

[19] T. M. Chilimbi, M. D. Hill, J. R. Larus, Cache-Conscious Structure Layout, ACM PLDI, Atlanta, GA, 1999.

[20] T. M. Chilimbi, J. R. Larus, Using Generational Garbage Collection to Implement Cache-conscious Data Placement, ISMM, Vancouver, Canada, 1998.

[21] E. D. Berger, K. S. McKinley, R. D. Blumofe, P. R. Wilson, Hoard: A Scalable Memory Allocator for Multithreaded Applications, ACM ASPLOS, Cambridge, MA, 2000.

[22] B. Steensgaard, Thread Specific Heaps for MultiThreaded Programs, ISMM, Minneapolis, MN, 2000.

[23] T. Domani, G. Goldshtein, E. K. Kolodner, E. Lewis, E. Petrank, D. Sheinwald, Thread-local Heaps for Java, ISMM, Berlin, Germany, 2002.