Optimal Placement of Java Objects and Generated Code in a Cell-based CC-NUMA Machine

Prakash Raghavendra, Sandya Mannarsamy

Hewlett-Packard, India

srp@hp.com, sandya.s.mannarswamy@hp.com

Abstract

Server-side Java applications are known to run for long time and to create lots of objects during their run. The memory access latencies on a CC-NUMA machine from cell-local memory to non-local memory differ in the order of 10. This means that access times for some data in local memory to a processor would be about 10 times faster than accessing data from a non-local memory (sitting local to other processor). This latency difference may have adverse effect on performance of java applications which create lots of objects and run for long time. The idea presented here is to optimally place these objects in local memories of different cells so that overall access times are reduced significantly. This has seen to improve the performance of the applications in the order of 20-30% for some application like SPECjbbs.

Dynamic optimization systems like JVMs store optimized or translated code in software managed code caches in order to maximize use of native/optimized code. Code caches store dynamic compiler generated code fragments either at method level or at loop level. The code cache is allocated at runtime by the JVM, and as the methods are compiled, the generated code is placed in code caches, and is reused by threads which execute further calls to the method. In this work, we propose a dynamic compiler scheme for splitting the code buffer on a per locality domain, on the basis of the access patterns of these methods by the threads. We use the escape analysis information (as in which method escapes which thread) and the profile information to identify sets of methods to which certain threads have high affinity (in the sense that these methods are mostly executed by only those threads) and use the information to place the generated code for such methods in code buffer fragments on the same locality domain where those threads are scheduled. This is aimed at reducing the remote misses in accessing the code buffers by the threads which have strong affinity for the corresponding code buffer fragments.

1.0 Introduction

Large SMP systems are characterized with multiple locality domains. IBM has MCMs, HP has cells. The memory access latency is different within a locality domain versus between locality domains. There could even be different types of latencies between different levels of locality domains. A Superdome's single cabinet could have slightly lower latency compared to the other cabinet.

The time to access an object/code instruction in local memory of a cell could be in the order of 10 times lesser when compared to time to access an object in non-local memory. This may have adverse effect for Java-based applications, which create large number of objects (like memory intensive applications) and which run for long time (like server-based applications). For example, if a java thread T1, running on processor P1 in cell C1, access an object in memory of cell C2, most of the time, and this region of code happens to be one of the hotspots (that is program spends much time in this part of the code) of the application, then performance would degrade by an order of 5-10. If, however, we had placed this object in C2 in C1, before this hotspot begins to execute, the same could have run much efficiently due to lesser number of non-local access to memory.

The idea presented here would describe a strategy which can be used to statically place these objects (during the start of the program) and eventually re-distribute dynamically depending on the change of access patterns of these objects (from different threads), so that overall non-local accesses for the program is reduced to minimum. This would improve the performance of such server-side java application significantly.

Improving non-local access latencies in CC-NUMA architectures has been studied well by research community. Most adopt dynamic page placement for improving locality. This technique works very well transparent to the applications. However, these seen to work only for C/C++ based applications, which

have regular access patterns, such as scientific applications. However, unlike such applications, java based applications tend to make extensive use of heap allocated memory and typically have significant pointer chasing code. Hence dynamic page placement techniques are known not to work well for these kinds of applications. Other known techniques are applicable for cluster-JVM (cJVMs) which are designed for cluster machines. These techniques are not suitable for CC-NUMA due to the fact that latencies on a cluster system differ more than on CC-NUMA machines.

Optimizing programs for their efficient usage of memory has been well studied in literature. We would like to classify such work as C/C++ based and Java based, since the issues which are applicable to Java are unique. For example, C/C++ usually access the memory in a regular fashion (the locality of reference holds good most of the time). However, with Java, due to heap allocation and garbage collections, the access pattern may not follow any regular pattern and so techniques which work well for C/C++ may not work well for Java as well.

In [4], there is a discussion of various techniques of page mapping techniques, which are quite popular and have been implemented in many operating systems, including HPUX and others. These techniques are known to work well for C/C++ programs. These are transparent to the user as well. [5] discusses a dynamic page placement technique, which decides the placement of pages on a cell and also migrating these as the access patterns change due to different behaviour of the program. The study was specifically done for TPCC, which is not Java based application. [7] discusses a technique, which is based on loads/store prefetching.

For Java, [6] gives a technique, where the authors have proposed to partition the java heap of a process to be placed on various nodes/cell of a CC-NUMA machine, whereby making the heap aware of NUMA behaviour.

In the Java performance lab, we did several experiments to see whether the current known techniques work well for Java workloads. We ran Specjbb on a 64p (128c) machine, which has 16 cells each having 4 processors. We saw that the performance of the benchmark showed considerable improvement if we turn on the cell interleaved memory instead of cell local memory. Note also that the HPUX operating system has the page placement technique in the kernel. When we switched the interleaved memory the pages of heap were distributed evenly across all nodes in a round robin fashion and all cells accessed them on all the cells and no cell had a huge contention for memory accesses. However with cell local memory, we saw huge queues for updates for some busy part of the heap, which brought down the performance.

Further, all the prior techniques do not work well for long running applications, since they do not re-distribute these objects as access patterns change during the course of the application run. We propose to change these placements as the application behavior change during its course. The other salient feature of our approach is the fact that we use the information that are anyway collected by JVM and we make the decisions of placement or re-distribution of these objects during Garbage collections (GCs). We are also not aware of any work, which place the generated code by the virtual machine, in the code space, which would be accessed optimally across nodes.

The paper is organized as follows: The next section gives the scheme of placement of java objects optimally, section 3.0 explains the scheme to place the code onto the target machine. We give an example placement in section 4.0 and results in 5.0. Finally, we give some ideas on future work and conclude in section 6.0.

2.0 Optimal Placement of objects of a Java based application

The technique follows a two-phased approach. In the first phase, called, "Initial Placement Phase", we place the objects as they get created. In the second phase, we, on a need-basis, re-distribute these objects, depending on the current access patterns of the objects in the application. We will explain each phase in the following sections.

The initial placement phase, does the placement of objects into memory, by doing the static analysis. There are various schemes possible, but, we adopt simple scheme, called "*creator-holds*". This means that the thread which creates an object places the object in local memory of the processor on which it runs. For example, when a thread T1, running on processor P1, in cell C1, creates an object O1, O1 is placed in cell local memory of C1. The idea is that, since we do not know who will be accessing this object most of the time in future, now, we assume

that T1 would be accessing this object most. However, if the access pattern changes, for example, if the thread T2 in cell C2 accesses this object most, then it makes sense to move this object O1 to C2. This is what is done during the second phase, which is "*Redistribution*" phase.

The *Redistribution* phase is dynamic in nature, in the sense that, the decision is based on the access pattern of objects as the application runs. Garbage collector (GC) is an important runtime component of the JVM, which runs every time when JVM is not able to allocate more objects. GC builds a graph of object references to each other and removes those objects which are dis-connected to the main graph (which has root nodes of the program). We re-use this information with a slight modification. When the GC builds such a graph, we also add a counter which registers the number of times the reference was made between two objects O1 and O2. This would give us "hotness" indicator for that reference.

Once such a graph is built, the second step is to search for partitions in this graph such that connectivity between these partitions are minimum. The graph partition is a NP-hard problem and there are many heuristic to solve this problem efficiently. We use the technique developed by [2]*A Multilevel Algorithm for Partitioning Graphs* by *Bruce Hendrickson and Robert Leland in Supercomputing '95*. This technique would do the partition of the graph into k-partitions such that total weight of links between the partitions is minimum. These k-partitions correspond to k cells that are present in the machine. Once we place these objects according to the partitions we obtain, we would be minimal non-local access placement at that time.

The last step is to check for benefit of re-distribution at the current stage. Though the above algorithm gives us a new partition, we need to check whether it would really benefit to do the re-distribution now. This is done by a heuristic. A simple heuristic that we use is to check whether the number of movements that are required is less than 20% of the number of objects. If yes, then we do not think its worth the effort to re-distribute. We see that, with this heuristic, we do not re-distribute during most of the GCs, which is good, since these redistributions would be a big overhead, if done repeatedly.

3.0 Optimal Placement of dynamic code in JVMs

Code is generated in much smaller snippets and its access patterns can be determined by the run time environment. For instance, in a multi-threaded program, different threads may be doing different work and so be working with different Java methods. In these cases, we can generate code with closer locality to methods generated for a thread. Even if we do nothing else and the total code needed by a thread falls in its own set of virtual pages, the OS can locate those pages into physical memory closest to the processor on which the corresponding threads are running.

But a further optimization would be to break the code buffer into multiple pieces, each corresponding to a thread, and maybe a common area. Then the code is generated into each such piece and placed inside individual (Cell Local Memory) CLMs. In Java virtual machines, the compiler generated code is maintained in a separate software managed code cache. The management of code cache in order to reduce memory latencies by cache misses is an important factor in improving the overall performance of the JVM.

In general, “escape analysis”[3] information is gathered by the compiler in order to recognize which objects can be allocated on the stack instead of on the heap. Escape analysis gives for each thread, what are the set of methods that do not escape the thread. This means those methods which are executed only by this thread. We use this information to divide the set of hot methods into different partitions such that each partition consists of a set of methods and a set of threads such that each method falls into the non-escape set of the thread. We use the online profile information to find the methods which suffer from remote memory misses in their code cache accesses. This information is fed back to the JVM such that the hotspot compiler can allocate the code cache for each set of methods such that the methods are mapped on to the same locality domain where the threads can be scheduled. This information is also outputted to the programmer as an advice so that if the programmer desires, he can use the pset calls to tie the threads to the corresponding locality domain.

5.0 Results

We implemented the above scheme (for optimal object placement) for two applications on a Tru64 system running FastVM JVM. The first application was from a customer (part of the application) which had lots of reads/writes to many objects that it creates. Second is the SPEC java benchmark, called Specjbb. Since currently there is no support from Tru64 OS to place the objects in a given cell-local partition (local to a CPU), we simulated the condition by counting the accesses to a local cell memory and to a non-local cell memory during interpretation of the JVM. It was found to improve application performance by 20% in case of SPECjbb and around 12% for the customer application. We plan to provide more detailed results during the final poster presentation.

6.0 Conclusions and Future work

In this work, we presented schemes to place both objects and generated code in JVM in an optimal fasion on a CC-NUMA machine. These machines scale well, however, they incur significant costs in accessing non-local accesses. We presented the results from these to show that such methods when implemented for a large NUMA machines (like Superdomes) can result in significant performance improvement (especially for server-side Java applications).

We are currently working on optimally placing the code, C-heap structures also in the current implementation. The C-heap structures are structures malloced by JVM code itself. Since JVM also runs significant part of the run-time of any Java applications, the accesses made by JVM itself also account for about 10-15% of times of the overall accesses made by the Java process. So, placing these optimally also has an effect on the performance on a CC-NUMA machines. We are using the similar schemes (of access patterns of these structures to make decisions on the placement on cells) to place these as well.

References

[1] Kim Hazelwood and James E. Smith. "**Exploring Code Cache Eviction Granularities in Dynamic Optimization Bruce Hendrickson and Robert LelSystems**," *Second Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO-04)*. Palo Alto, March 2004, pp. 89-99.)

[2] Bruce Hendrickson and Robert Lel, "**A Multilevel Algorithm for Partitioning Graphs",** Bruce Hendrickson and Robert Leland in Supercomputing '9*5*.

[3] Bruno Blanchet. "**Escape Analysis: Correctness Proof, Implementation and Experimental Results**" In 25th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'98*)*, pages 25-37, San Diego, California, January 1998.

[4] Jian Huang, et al, "**Page-Mapping Techniques for CC-NUMA Multiprocessors** », Intl. Conference on Algorithms and Architectures for Parallel Processing, 1997.

[5] Kenneth Wilson and Bob Aglietti, "**Dynamic Page Placement to improve locality in CC-NUMA Multiprocessors for TPCC**", SuperComputing Conference, 2001.

[6] Mustafa Tikir, Jeffery Hollingsworth, "**NUMA-Aware Java heap for Server Applications**", 19th IEEE Intl. Parallel and Distributed Processing Symposium (IPDPS), 2005.

[7] Stefanos Kaxiras, James Goodman, "**Improving CC-NUMA performance using Instruction-based Prediction**", 5th Intl Symposium on High Performance Computer Architecture, 1999.