Block Migration in Broadcast-based Multiprocessor Architectures

Constantine Katsinis
Electrical and Computer Engineering, Drexel University, Philadelphia, PA 19104
katsinis@ece.drexel.edu

Abstract

This paper presents techniques that improve the performance of parallel programs on distributed shared memory NUMA multiprocessors by implementing dynamic memory block and page migration. Our techniques address the latencies caused by the contention within the network and attempt to enhance data locality by migrating pages to reduce remote references. We analyze the behavior of eight multiprocessor applications which exhibit a wide range of network traffic patterns. Results show that several applications that encounter hot spots and network congestion see a reduction of run time by more than a factor of ten.

1. Introduction

Typically, on cache-coherent DSM multiprocessors, each logical memory page is initially mapped on a physical page at some node in the system which serves as the home node of that page.

Our experiments with real applications show that usually pages are shared by more than two processors and that all processors sharing the page generate approximately the same number of references. Therefore, migrating a page to a new node has little beneficial effect. In fact, experiments have shown [1] that contention at the network interface and the links can be a significant factor affecting performance and must be taken into account when designing page migration protocols.

To reduce the contention, we must detect and eliminate the hot spots as soon as possible after they arise, so that the flow of data request and acknowledge messages in the network is approximately uniform over all nodes. We examine algorithms that continuously attempt to determine if a node is a hot spot. If the node is a hot spot, the algorithm determines if the block being acknowledged by the message under consideration is one of the causes of the hot spot, and may decide to migrate the block.

Our algorithms examine individual memory blocks, or groups of blocks within the same physical page, and may decide to migrate one block or several blocks from the same physical page. They are implemented at the lowest level, between the cache, directory and network controllers, and therefore are completely transparent to the user and require no modification to the application.

We analyze the behavior of the applications using address traces executed by a simulator that uses a broadcast-based architecture as the underlying computer system. One particular implementation of this architecture is the SOME-BUS multiprocessor [2,3] which requires no switches and is capable of supporting multiple continuous broadcasts. Although this broadcast-based architecture can provide better performance than some switch-based architectures, the results obtained here are generally applicable to traditional architectures.

The address space of the application is initially uniformly distributed over all nodes. Directory information for each block includes two counters, indicating the number of local and remote references. Each node has an address translation table which maps logical block numbers to physical block numbers. Initially, the table does not do any remapping, so that logical block b is mapped to physical block b. Block migration is implemented by copying the selected block from the original home node to the new home node and adjusting the translation tables in all nodes. Since previous research uses pages as the granularity for data migration, we also implement page migration. The directory contains two counters per page, indicating the number of local and remote references to that page. These counters are in addition to the two counters per block mentioned above. Our experiments show that the relevant reference counts per block are small numbers, so that the counters associated with each block need only be a few bits wide. Thus, the additional memory requirements per block and per page are very small.

2. Reducing contention

Our assumption is that most shared pages are shared

Proceedings of the 2005 Fourth IEEE International Symposium on Network Computing and Applications (NCA'05)
0-7695-2326-9/05 $20.00 © 2005 IEEE

by more than one processor, the result of repetitive and dynamic single-producer, multiple-consumer combinations. Migrating pages to nodes that access them more frequently offers little benefit and requires such excessive processing that it is not practical to implement in hardware. Instead, we develop very simple algorithms that can be easily implemented in hardware and that use information gathered by hardware monitors with only a minimal overhead and without the involvement of the operating system.

Our protocols are designed to adjust the data distribution at runtime through automatic block migrations. Instead of using memory access histograms, each node monitors its own channel condition to determine if it is becoming a hot spot and, based on this decision, migrates individual blocks to nodes that are not hot spots, to dynamically and transparently modify the data layout. In this way, incorrectly allocated data is moved to other nodes causing an equalization of remote accesses. Blocks being accessed frequently by their current home node are not migrated and hence data locality is preserved.

The performance of the algorithms is examined using a simulator which maintains the precise state of the processor, directory controller, cache controller and channel controller, for every memory reference and every message. The applications executed by the simulator consist of sequences of memory references extracted from actual multiprocessor address trace files. Each thread has its own address sequence and stops running when it has finished processing all of its memory references. The execution of the application is complete when all threads have finished processing their respective memory reference sequences.

The architecture used in this paper has 64 nodes. There is one thread per node. Cache blocks have 64 bytes; DACK messages (with payload of one cache block) have 80 bytes, and request messages with no data have 48 bytes. Smaller messages, such as channel state updates or remap acknowledgments, have 8 or 16 bytes.

A set of four multiprocessor address trace files for programs called *fft*, *speech*, *simple* and *weather* was obtained from the trace database at the TraceBase website. Details about these four applications are provided in [4]. In addition, we use four multiprocessor address trace files from the SPLASH applications *barnes*, *radix and LU using* contiguous (*LUC*) and non-contiguous column allocation (*LUN*).

The migration protocol operates as follows. At some point in time, the current home node A decides to migrate the block being processed. After the DACK message is enqueued, the directory selects the new home node B and enqueues a remap request message to node B. This message contains the current logical and physical numbers of the block as well as the current contents of the block. The state of the block in node A becomes *old-transient*. During the time that the block is in this state, the old directory in node A continues to respond to data requests from other nodes. It stops responding if it receives an ownership request, which it leaves enqueued together with any subsequently received data requests. The new home node B responds to the remap request message by allocating space for the migrated block, setting its state to *new-transient* and broadcasting a REMAP-ACK message to all nodes. This message also contains the logical and new physical numbers of the block. All nodes adjust their local translation tables, and respond with a REMAP-DONE message to the new home node B. The old home node A responds with a REMAP-DONE message which also includes the current copyset of the block. Node A also changes the state of the block to *old-done*, and forwards to node B any requests that node A may have kept in its queue, or any requests that node A may receive in the future. The new home node B starts responding to data requests after broadcasting the REMAP-ACK message. It delays responding to ownership requests (and all data requests from nodes with pending ownership requests) until it has collected all REMAP-DONE messages (including the one from the old home node). This protocol may only delay the response to ownership requests, but preserves order and write atomicity, and therefore still enforces sequential consistency.

3. Migration algorithms

In the following we describe several block migration algorithms and show their effect on application run times. The basic differences between these algorithms are 1) the quantity used to determine the channel status, and 2) whether additional blocks from the same page may be migrated. All algorithms decide if the current node is a hot spot by comparing the channel state to a threshold. Depending on the algorithm, the state is defined as the utilization, or the channel mean waiting time, or the number of messages queued at the channel. Algorithm 1 decides if the current node is a hot spot using the long-term channel utilization which is the ratio of the time that the channel controller is busy transferring messages over the total run time up to that point in time. Algorithm 2 uses the average long-term message waiting time at the channel queue. This time is updated when a message enters channel service and begins transmission. Algorithm 3 also uses the average long-term queue waiting time and searches the same page for additional blocks that may be migrated. Algorithm 4 operates as Algorithm 3, but uses a short-term estimate of the average channel queue waiting time of messages, by

simply clearing the counters that accumulate waiting time and run time when a busy period of the channel ends. Algorithm 5 requires less hardware complexity by relying on the number of messages in the channel queue to decide if a node is a hot spot or not. The direct result of these algorithms is that blocks that were originally allocated on the memory of the same node X, become remapped onto different nodes and consequently the message traffic which was initially directed toward node X, becomes dispersed onto several nodes. A block located in node X is migrated to any node Y that is determined not to be a hot spot, if the total number of remote memory accesses to the block is larger than the number of local memory accesses. As a result, future reference that would have been directed to the old node are directed to the new node, reducing the traffic into the old node and making it less of a hot spot. At the same time, the new node begins to receive more traffic and becomes more of a hot spot. As it periodically reports its condition, it will eventually stop being selected by the other nodes as a recipient of new remapped blocks, resulting in a balanced message traffic.

The directory maintains two counters for each block, indicating the number of local (C_L) and remote (C_R) references to that block, and two counters for each page (of 16 blocks), indicating the number of local (B_L) and remote (B_R) references to that page. Once the algorithm decides that the current node is a hot spot, it decides whether the current block should be migrated. This decision is based on two thresholds, the remote count of accesses to the block T_R and the local count T_L. T_R has two values T_{RH} and T_{RL} ($T_{RL} < T_{RH}$) which provide some adaptive ability to the algorithm. If the algorithm decides that the current channel state is more than the threshold, it examines the remote and local access counts of the current block and decides to migrate the block if all three conditions are satisfied $C_R > T_R$ and $C_L < T_L$ and $C_R > C_L$. Threshold T_R is normally set to T_{RH} but if the current channel state is much larger than the threshold then T_R is set to the lower value T_{RL} to increase the rate at which blocks are migrated.

If the block must be migrated, then the algorithm selects at random a non-hot-spot node and enqueues a remap request message. If the algorithm can migrate additional blocks, it may examine several more blocks of the same page, if the page counters contain values that satisfy the page thresholds. Specifically, counters P_R and P_L indicate the remote and local count of accesses to the page, and P_N indicates the maximum number of additional blocks within the page that will be considered given that the decision has been made to migrate the first block. A decision to examine additional blocks within the relevant page is made if all three conditions are satisfied $B_R > P_R$ and $B_L < P_L$ and $B_R > B_L$ where B_L and B_R are the local and remote access counts of the page. The algorithm stops examining blocks when migrate messages have been sent for P_N blocks. In our experiments migrate messages for all additional blocks are sent to the same node as the migrate message for the initial block.

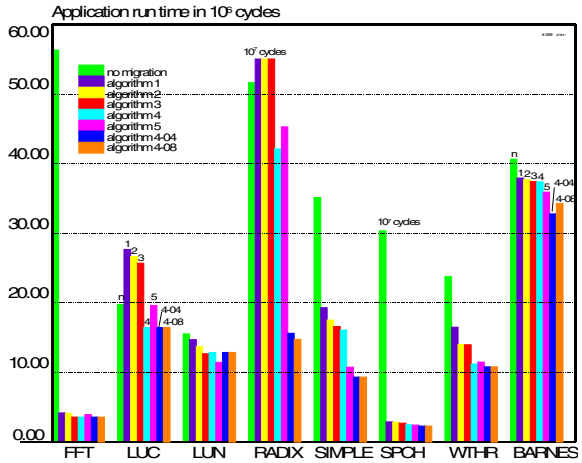

Figure 1: Application run times.

Although Algorithm 1 succeeds in reducing the application run times, it does not eliminate the hot spots. The reason is that it relies on a long-term utilization measurement that does not quickly reflect the increased latencies experienced at the hot spots. Still, Algorithm 1 is useful because of its simplicity and as a basis for further comparisons. Algorithm 2 is more responsive to the occurrence of large latencies at the hot spots and results in significantly lower run times compared to Algorithm 1. Figure 1 shows the run times of 8 applications for the case when no block migration is used and for Algorithms 1, 2, 3, 4 and 5. Algorithm 1 uses utilization threshold $T_U = 0.50$ and Algorithms 2, 3 and 4 use waiting time threshold $T_W = 160$, and $T_{RL} = 4$, $T_{RH} = 6$, $T_L = 9$, with $P_N = 1$. Algorithm 5 uses queued messages threshold $T_N = 10$. Using channel wait times and expanding the migration decision to rely on page information has a direct effect on equalizing the channel utilizations. Applications *fft* and *speech* exhibit one dominant hot spot, which is easily eliminated by all algorithms with about the same effectiveness. *Radix* also has two dominant hot spots, but most accesses involve a very small number of blocks, so that migrating a block is not as effective because it would only move the hot spot from one node to another. *Simple* and *weather* are the more representative applications that exhibit several hot spots that are effectively eliminated by block migration. Applications *LUC* and *LUN* also show some benefit from block migration, however they place much less traffic on the network and messages do not encounter large

latencies. As a result, the migration algorithms are called infrequently. The figure shows that use of Algorithm 3 increases the run time in applications *LUC* and *radix*. Because it relies on the average long-term message channel queue waiting time, it goes through periods of time when several blocks are migrated, increasing the network traffic and message latencies, without reducing the hot-spot effects. This is due to the fact that in *radix* the hot spot is concentrated on a few blocks, and in *LUC* the latencies are relatively small even in the initial run with no migration. Algorithms 4 and 5 are more responsive and never increase the run time. In most cases, the performance of Algorithm 5 is better or approximately equal to the performance of Algorithm 4.

4. Conclusion

In DSM multiprocessor architectures, application run time can be reduced, sometimes quite significantly, by the use of simple algorithms that automatically migrate memory blocks. Their primary purpose is to reduce the latency caused by network contention due to near-simultaneous accesses of memory blocks by multiple processors. In addition, they take into account the local accesses of the home node, thereby preserving application locality. Our results show that very simple measurements, easily implemented in hardware, such as the number of messages waiting transmission at the node channel queue can be used by migration algorithms to successfully improve performance. As Figure 1 shows, run time is reduced. Applications that exhibit one dominant hot spot, such as *fft* and *speech*, encounter significant improvement, since all processors, except the one at the hot spot, see reduced request-response latencies and are idle for a smaller fraction of time. In the *fft* application, run time is reduced by a factor of 14 and processor utilization more than doubles. In the *speech* application, run time is reduced by a factor of 12.5 and average processor utilization increases by a factor of 1.523. In applications that exhibit several independent hot spots, such as *simple* and *weather*, run time is reduced to half the original time, while average processor utilization increases slightly. Applications that place less traffic on the network and whose messages do not encounter large latencies, such as *LUC* and *LUN* see a smaller benefit from block migration. Finally, in applications with few hot spots where only a small number of memory blocks are involved, migration algorithms have a smaller effect, since migrating a block only causes the channel congestion to reappear at the new home node.

5. References

[1]. Donglai, Dai, Panda, DK, "How much does network contention affect distributed shared memory performance", Intern. Conference on Parallel Processing, 1997, pp. 454-461.

[2]. Katsinis, C. and D. Hecht, "Fault-tolerant Distributed- shared-memory on a broadcast-based architecture", Tr. on Parallel and Distributed Systems, v. 15, n. 12, Dec. 2004, pp. 1082-1092.

[3]. Katsinis, C., "Performance Analysis of the Simultaneous Optical Multiprocessor Exchange Bus", Parallel Computing Journal, Vol. 27, No. 8, July 2001, pp. 1079-1115.

[4]. http://tracebase.nmsu.edu/tracebase.html