Dynamic Computation Migration in DSM Systems

Wilson C. Hsieh, M. Frans Kaashoek, William E. Weihl

Abstract

Dynamic computation migration is the runtime choice between computation and data migration. Dynamic computation migration speeds up access to concurrent data structures with unpredictable read/write patterns. This paper describes the design, implementation, and evaluation of dynamic computation migration in a multithreaded distributed shared-memory system, MCRL.

Two policies are studied, STATIC and REPEAT. Both migrate computation for writes. STATIC migrates data for reads, while REPEAT maintains a limited history of accesses and sometimes migrates computation for reads. On a concurrent, distributed B-tree with 50% lookups and 50% inserts, STATIC improves performance by about 17% on both Alewife and the CM-5. REPEAT generally performs better than STATIC. With 80% lookups and 20% inserts, REPEATimproves performance by 23% on Alewife, and by 46% on the CM-5.

Keywords: computation migration, data migration, replication, coherence

1 Introduction

Dynamic computation migration is the dynamic choice between computation migration and data migration. Computation migration moves computation to data that it accesses, in contrast to data migration, which moves data to computation. This paper discusses the issues involved in dynamically choosing between computation migration and data migration, and describes an implementation of computation migration in a distributed shared-memory system. Performance measurements demonstrate that migrating computations for some accesses is better than always migrating data, and that choosing dynamically whether to migrate data or computation is better yet for dynamic data structures with unpredictable access patterns.

Computation migration [11, 15] is the partial migration of active threads. Under computation migration, a currently executing thread has some of its state migrated to remote data that it accesses. Computation migration is distinct from RPC (which was explored in systems such as Emerald [13]

Technical paper. Contact information: Wilson Hsieh, University of Washington, Box 352350, Seattle, WA 98195, whsieh@cs.washington.edu, http://www.cs.washington.edu/homes/whsieh, fax: 206-543-2969. Frans Kaashoek, 545 Technology Square, Cambridge, MA 02139, kaashoek@lcs.mit.edu, http://www.pdos.lcs.mit.edu/~kaashoek, fax: 617-258-8682. William Weihl, 130 Lytton Avenue, Palo Alto, CA 94301, weihl@pa.dec.com, http://www.research.digital.com/SRC/staff/weihl/bio.html, fax: 415-853-2104. This paper is a condensed version of the first author's dissertation [10].

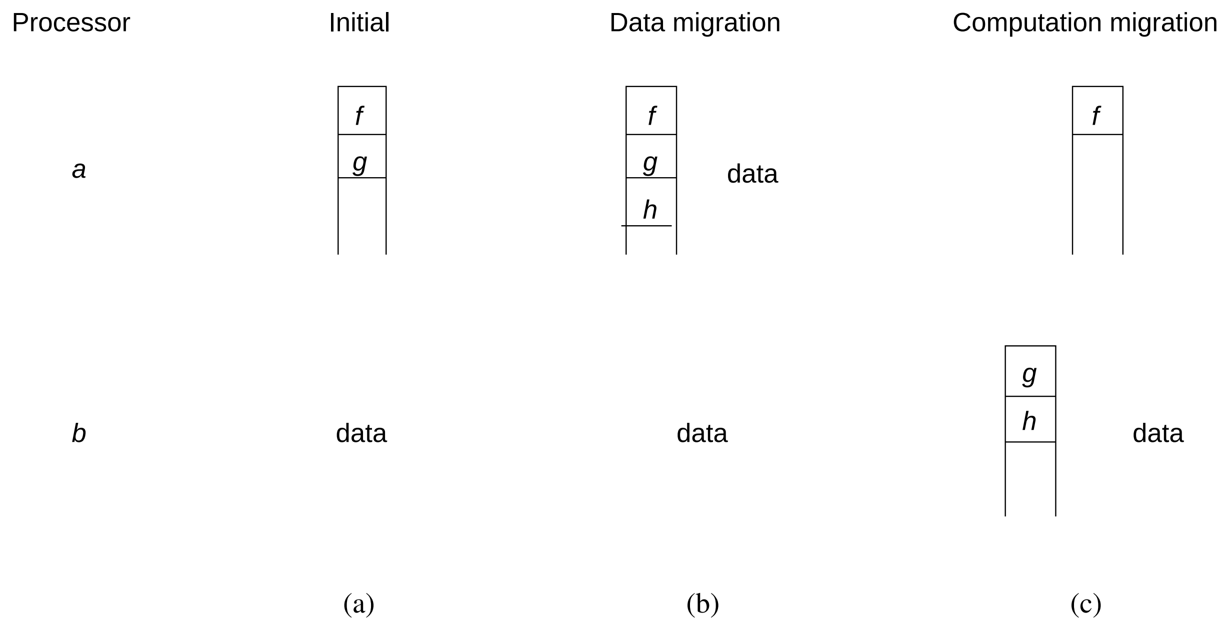

Figure 1. Data migration vs. computation migration. (a) illustrates the initial state of the system. (b) illustrates the state of the system after data migration; (c) illustrates the state after computation migration.

and Amber [7]) in that RPC systems do not migrate pre-existing computation. Computation migration is also distinct from thread migration, as only **part** of a thread is migrated.

Data migration is the migration of data to a thread that accesses it. Data migration is typically used with replication, where multiple copies of data can exist. Replication allows multiple processors to read data in parallel, but requires communication to maintain consistency in the presence of writes. In the remainder of this paper, we shall use "data migration" to mean data migration with replication.

Computation migration performs well when data migration with replication does not; conversely, computation migration performs poorly when data migration with replication performs well. Replication improves the performance of read operations, because it allows them to execute in parallel. Computation migration improves the performance of writes, because it avoids the communication costs of coherence.

Figure 1 illustrates the difference between data migration and computation migration. When a thread executing procedure `g` calls a procedure `h` that accesses remote data, data migration moves the data to the thread. Computation migration moves the activation record for `g` to the data, and executes `h` at the data. Computation migration differs from RPC, where `g` would not be migrated.

For data structures whose read/write ratios are dynamic (unpredictable at compile-time), the decision to use computation migration or data migration should be made dynamically. We call this dynamic choice **dynamic computation migration**. This paper evaluates dynamic computation migration in the context of MCRL, a software distributed shared-memory (DSM) system.

We have made the following contributions:

- We have built a multithreaded distributed shared-memory system called MCRL that supports dynamic computation migration. MCRL extends the CRL system [12]. MCRL runs on a 32-node MIT Alewife machine and on a 128-node CM-5. We will concentrate on our Alewife results, as Alewife's communication costs are very low. Because dynamic computation migration reduces communication costs, the Alewife results represent a lower bound on the performance benefits of using dynamic computation migration.

- We describe two simple policies, STATIC and REPEAT, for choosing between computation migration and data migration, and we evaluate the policies using MCRL. Experiments with a microbenchmark demonstrate that computation migration should always be used for writes, and that a simple policy can make good choices between data migration and computation migration for reads.

- We demonstrate that dynamic computation migration reduces coherence costs for dynamic data structures, where the ratio of read/write accesses is unknown until runtime. For a concurrent, distributed B-tree on Alewife with an operation mix of 80% lookups and 20% inserts, using dynamic computation migration improves performance 23% relative to pure data migration. For the same B-tree on the CM-5, dynamic computation migration improves performance by 46%.

The rest of the paper is structured as follows. Section 2 overviews the implementation of MCRL. Section 3 describes our protocol for dynamic computation migration, and describes two policies for deciding between computation and data migration. Section 4 describes and analyzes the performance results of using dynamic computation migration in MCRL. Section 5 discusses related work, and Section 6 discusses future work. Section 7 summarizes the results and conclusions of this paper.

2 MCRL

We have implemented dynamic computation migration in the MCRL software distributed shared-memory system. MCRL extends CRL [12], a SPMD distributed shared-memory system. CRL provides a global name space for **regions**, which are programmer-defined blocks of memory. CRL replicates regions using a fixed-home, directory-based, sequentially consistent cache-coherence protocol. Locking happens implicitly with the CRL access functions, which acquire regions in either read-mode (shared) or write-mode (exclusive). MCRL extends CRL in two ways: first, it provides support for multithreaded programs; second, it provides support for dynamic computation migration.

MCRL runs on the MIT Alewife machine [1] and on Thinking Machines' CM-5. Alewife is an experimental shared-memory multiprocessor that supports high-performance message passing. Each of the 32 processors is a 20 MHz SPARC-like processor with a 64Kbyte cache and 8Mbytes of physical memory. The machine achieves a peak network bandwidth of 18 Mbytes/second. MCRL uses Alewife's message passing primitives, and does not make use of its shared-memory support. Messages are delivered using interrupts, which are fast because of Alewife's support for fast thread switching.

The CM-5 is a commercial message-passing multiprocessor. Each of the 128 processors is a 32 MHz SPARC processor with a 64Kbyte cache and 32Mbytes of physical memory. The peak achievable network bandwidth is about 11.8 Mbytes/second. We ran with version 7.4.0 Final of the CMOST operating system, and version 3.3 of the CMMD message-passing library. Messages are received by polling, because interrupts are slow on the CM-5. The communication performance of the CM-5 is on a par with today's high-performance distributed systems such as FRPC [17] and U-Net [18], but is lower than Alewife's. Dynamic computation migration improves performance more on the CM-5 than on Alewife, because the communication costs that are removed are greater. We will concentrate on the Alewife results, which represent a lower bound on the benefits of computation migration.

Two approaches have been used to implement computation migration. In the first approach, the compiler generates calls into the runtime system, which manages migration. This approach was used in the Olden system [15]. In the second approach, the compiler manages migration more directly. We took this approach in Prelude [11]. MCRL is designed for the latter approach, but does not provide compiler support. Compiler support should not be difficult to add, although it would require some language restrictions similar to those in Olden.

3 Dynamic Computation Migration

This section describes the mechanics of dynamic computation migration. When a processor accesses a region that is not cached locally, MCRL contacts the region's home node. The home node decides between data or computation migration, since it has the most information about the pattern of accesses to the region. Although a processor can keep track of its own access patterns, it does not have any information about other processors' access patterns. In addition, since the processor must contact the home node anyway, it is advantageous to make use of the home's extra knowledge of global access patterns.

Dynamic computation migration only occurs on remote processors. Operations that occur at a region's home processor do not migrate, even if the only valid copy of the region is currently on a remote processor. Allowing home operations to migrate in such cases could lead to "migration thrashing," where a computation chases a region from processor to processor. For example, if a home operation could migrate, it could migrate to a remote processor while the region is returning to the home; it could then migrate back to the home, and repeat the whole cycle again.

For simplicity, only one outstanding potential migration per region per processor is allowed. That is, if a thread attempts to migrate on a region while another thread on the same processor has sent a migration request on that region, the second thread blocks until a migration acknowledgment or the region is received. Blocking the second thread avoids another form of migration thrashing. For example, in response to the first potential migration the home could send the region back to the remote processor. If the second request did not block, but instead sent another potential migration to the home, the advantage of having sent the region back is lessened.

The MCRL protocol does not add any extra messages for using data migration, but does add an extra message for computation migration. It may be possible to design a protocol that does not require this extra message, but we chose this design point for the sake of simplicity.

In the following two sections we describe two effective policies that a home node can use to choose between data migration and computation migration.

3.1 STATIC Policy

The STATIC policy always migrates computation for remote writes, and always migrates data for remote reads. Assuming that the state of the migrated computation is not much larger than the data, it is a benefit to always migrate writes to data. The semantics of writes is such that they must always be serialized anyway. As a result, it makes sense to execute them all at one processor, which avoids the cost of moving data from one writer to another. The only case where this policy fails to perform reasonably is when exactly one remote processor accesses the region (and the processor occasionally writes the region).

The STATIC policy is not truly a dynamic scheme. Since writes always use computation migration, and reads always use data migration, the decision about which mechanism to use is static. We use the STATIC policy because it is a good baseline dynamic policy.

3.2 REPEAT Policy

The REPEAT policy always migrates computation for remote writes, for the same reasons as the STATIC policy. It dynamically chooses data migration or computation migration for remote reads, depending on the relative frequency of reads and writes. The policy uses computation migration for reads that follow writes, until any processor performs a second read before another write occurs. That is, if a processor executes a second read before another write occurs, data migration will be used.

The REPEAT policy assumes that data migration performs well when replication is useful, which is when multiple reads occur at one processor, without an intervening write. Assuming that the relative frequency of reads and writes remains relatively constant over several operations, the occurrence of two repeated reads means that repeated reads are likely to occur in the near future.

The REPEAT policy performs better than the STATIC policy when there is a large proportion of writes. The STATIC policy migrates data to reads even when there are many writes. As a result, it forces writes to invalidate replicated copies unnecessarily. The REPEAT policy performs slightly worse (a few percent) than the STATIC policy when the operation mix consists predominantly of reads. The percentage of cache hits is lower, since the first read that follows a write will use computation migration under the REPEAT policy.

4 Experimental Evaluation

We evaluate the performance of dynamic computation migration in MCRL, and show how dynamic computation migration can outperform data migration or computation migration alone. The error bars in the graphs represent 95% confidence intervals.

We use the term "pure data migration" to mean the use of the standard CRL interface, which always uses data migration. We use "dynamic data migration" and "dynamic computation migration" to refer to the use of the dynamic protocol in MCRL. In other words, one of the policies is invoked at runtime, and the runtime system decides to use either data migration or computation migration, respectively.

We first describe measurements of MCRL on two microbenchmarks. The first shows how the cost of data migration increases with data size. The second shows how computation migration reduces the cost of coherence. We then describe measurements of MCRL on two data structures: a

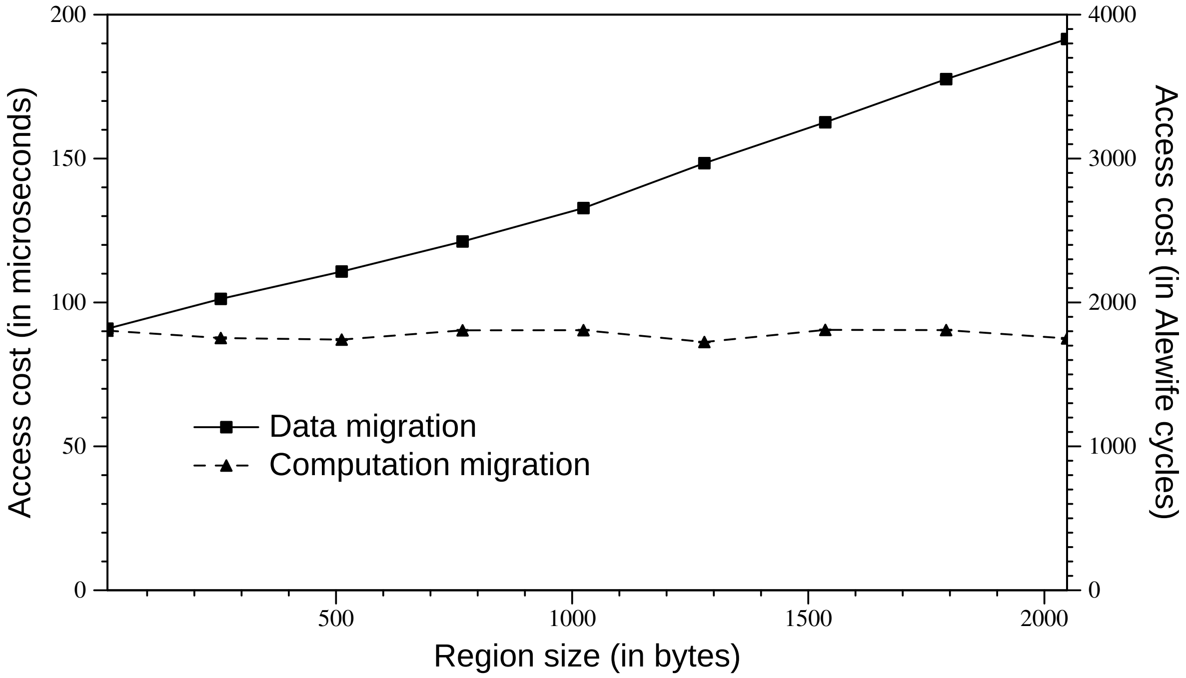

Figure 2. Comparison of data migration costs and computation migration costs for MCRL on Alewife.

counting network and a concurrent distributed B-tree. Both of these multithreaded data structures involve graph traversal, although their read/write characteristics are very different. The B-tree is more dynamic than the data structures used in the SPLASH benchmarks [16]: the pattern of accesses to a particular object cannot be predicted statically.

4.1 Migration Latency

We examine how the size of data affects the relative costs of moving data and computation on Alewife, which has a high-performance communication network. The minimum latency for a round-trip message is approximately 12 microseconds, and the maximum bandwidth is approximately 18 Mbytes/second. The CM-5 results are qualitatively similar, but the effective bandwidth of its network is much lower. The minimum latency for a round-trip message is approximately 12 microseconds, and the maximum bandwidth is approximately 11.8 Mbytes/second. As a result, cost of data migration on the CM-5 increases more quickly as the size of the data increases. It is important to note that the CM-5 performance is only achievable when polling is used.

Figure 2 compares the cost to touch a region using pure data migration and dynamically choosing computation migration. The cost to choose computation migration is not measured in these experiments. For computation migration, the cost is for a null computation: no computation is migrated. Thus, the measurements are equivalent to measuring the cost of null RPC using the dynamic protocol. These measurements only measure the relative overhead of colocating data and computation.

The costs for computation migration and data migration are virtually identical for tiny regions. As the size of the region increases, computation migration outperforms data migration. The cost for computation migration stays constant, since it is independent of the region size; the cost for data migration increases linearly in the region size. At 2048 bytes, computation migration is 2.1 times faster than data migration.

4.2 Coherence Costs

In this section we analyze the effect of contention on the choice between computation and data migration. The MCRL protocol for dynamic computation migration is used to choose between moving computation and data. The results illustrate how increasing the rate of writes increases the cost of coherence, and how our policies can be used to reduce these costs.

We measure the average cost of touching a single 256-byte region under pure data migration and dynamic computation migration (under both the STATIC and REPEAT policies). Eight processors access the region in a tight loop; the results are similar with more processors. The region size is sufficiently small that the cost of moving data is not a dominant cost; with larger data sizes, the benefits of using computation migration are greater.

Figure 3 demonstrates that our two policies for choosing between computation migration and data migration can reduce the costs of coherence. Pure data migration works well when there mostly reads. The STATIC policy performs well when there are mostly reads or mostly writes, and generally outperforms data migration. The REPEAT policy performs better than the STATIC policy when the numbers of reads and writes are approximately equal, but performs slightly worse than the STATIC policy when there are mostly reads or mostly writes.

Under the STATIC policy at 0% reads, every operation is migrated. The latency is that for always using computation migration. At 100% reads, every operation has data sent back. The latency is that for always using data migration. The curve for the policy bulges upward when the percentage of reads is near 50%. As the percentage of reads reaches 50%, data must be moved between the readers and the home due to invalidations. Once reads begin to predominate, the performance advantage of replication becomes more of a factor. That is, when the percentage of reads makes caching effective, the access cost begins to decrease.

The REPEAT policy performs similarly to the STATIC policy at the endpoints: 0% and 100% reads. (The different executables for the STATIC and REPEAT policies have slightly different cache behavior, which is why there is a difference at 0% reads.) Its curve does not have the upward bulge that the curve for the STATIC policy has, because the REPEAT policy chooses to use computation migration for reads when there is a high percentage of writes. The cost of maintaining coherence is eliminated, so the average cost to access the data remains constant. The REPEAT policy chooses to use data migration for reads when there is a large percentage of reads. This decision is effective, because it allows for replication.

4.3 Counting Network

A counting network is a distributed data structure that supports "shared counting", as well as producer/consumer buffering and barrier synchronization [2, 9]. The simplest data structure for shared counting is a counter protected by a lock. Under such a scheme, a thread acquires the lock

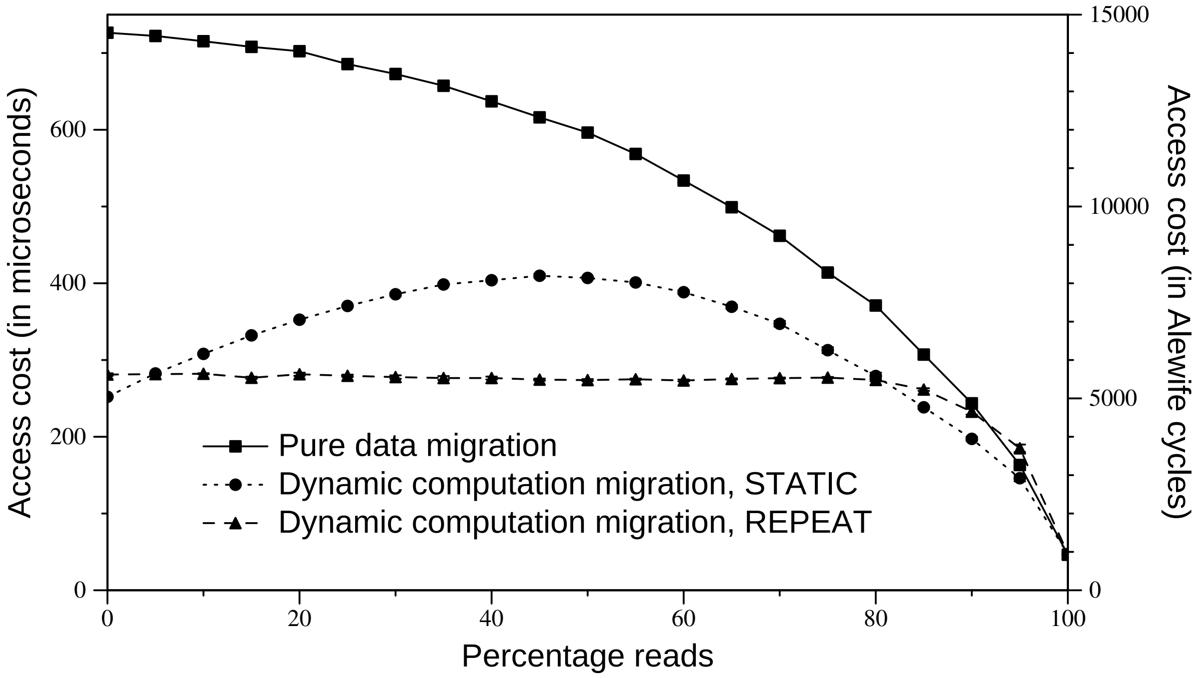

Figure 3. Alewife comparison of pure data migration and dynamic computation migration using the STATIC and REPEAT policies. Cost for one processor to access a single 256-byte region, with eight processors accessing the region.

and updates the counter when it needs a loop index; such a solution scales poorly under contention. A counting network is a distributed data structure that trades latency for higher throughput.

We performed experiments on an eight-by-eight (eight input, eight output) bitonic counting network. Such a counting network is roughly a six-stage pipeline. Each stage consists of four balancers in parallel. A balancer is a two-by-two switch that alternately routes requests between its two outputs. The counting network is laid out on twenty-four processors, with one balancer per processor. Placing the balancers on separate processors maximizes the throughput of the counting network. Each row of the counting network is spread across one quarter of the machine. This layout balances the amount of bandwidth required across any particular communication link.

This experiment simulates the use of a counting network to distribute the iterations of a parallel loop with an empty body. We create one thread per processor, and each thread accesses the counting network in a tight loop. Only the STATIC policy was run; the behavior of the REPEAT policy would be virtually identical, because all operations on balancers are writes. Therefore, all accesses use computation migration under both policies.

Figure 4 illustrates the results of using our dynamic computation migration protocol on a counting network on Alewife. We measure the throughput of the counting network as the number of processors varies from 1 to 32.

Data migration outperforms computation migration when only one processor accesses the counting network. Each balancer that the processor touches will be migrated to that processor. It takes seven iterations for the processor to touch all seventeen balancers that it accesses in the

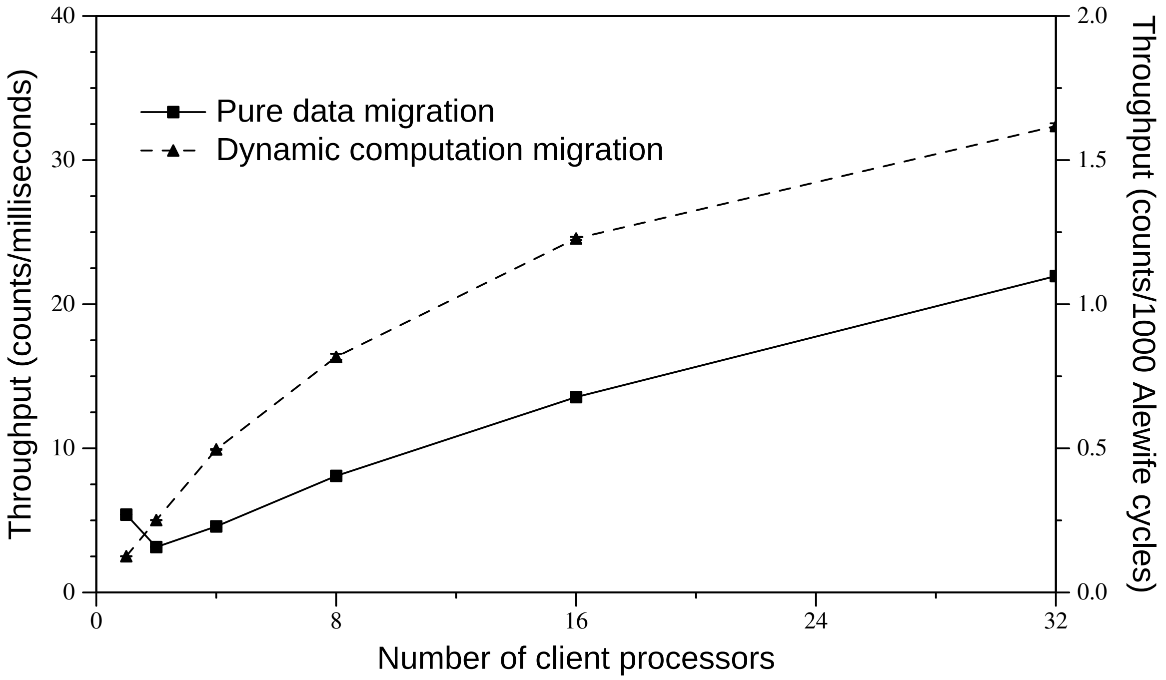

Figure 4. Throughput of an eight-by-eight counting network on Alewife. Performance of pure data migration and dynamic computation migration as a function of the number of client processors.

network. Once it has accessed all of those balancers, all subsequent accesses will be local. As a result, data migration with replication is faster than computation migration. A counting network would not be used with only one client, though, since a centralized solution would give better performance.

The throughput for computation migration scales almost perfectly for two and four processors, because the counting network is four balancers wide. With four or fewer processors, the processors do not interfere much with each other in the counting network. The requests into the counting network move across the counting network without having to wait. With more than four processors, the requests begin to interfere more in the counting network. With more than twenty-four processors, the counting network is mostly full.

Computation migration significantly outperforms data migration when a larger number of processors accesses the counting network. Computation migration does not incur any coherence overhead, whereas data migration must incur coherence overhead on every access. Despite the small size of the counting network nodes (eight words), the cost of maintaining coherence is still more expensive than moving computation. Our experiments confirm our earlier results [11], where we demonstrated that write accesses should migrate computation.

4.4 B-tree

A B-tree [3] is a data structure used to represent a **dictionary**, which is a dynamic set that supports the operations **insert**, **delete**, and **lookup**. The basic structure and implementation of a B-tree is similar to that of a balanced binary tree. Unlike a binary tree, the maximum number of children for each B-tree node is not constrained to two; it can be much larger. The B-tree implementation we use is a simplified version of one of the concurrent, distributed algorithms described by Wang [14, 19]. The form of these B-trees is a variant of the "classic" B-tree: keys are stored only in the leaves, and all nodes and leaves contain pointers to the node to their right. The values of the keys range from 0 to 1,000,000.

The **anchor** of a B-tree is primarily read-only; it is used to locate the **root** of the B-tree. The anchor is only written when the B-tree gains a level. In our experiments the B-tree does not gain levels, which is typical of large B-trees. **Lookup** and **insert** operations traverse the B-tree downward and rightward from the root until they find the appropriate leaf. If an insert causes a leaf to overflow, the leaf is split in two; a background thread is started to insert a pointer to the new leaf node into its parent. This splitting process is continued upwards as necessary. If the root winds up being split, a new root is created and the anchor is expanded.

The experiments described in this section measure the time for thirty-two processors to access a B-tree. In the experiments an initial B-tree is built with 200,000 keys. Each node or leaf can contain up to 500 children or keys, respectively; nodes and leaves are approximately 4000 and 2000 bytes in size, respectively. With these parameters, the B-tree is three levels deep. The nodes and leaves of the tree are created randomly across thirty-two processors.

After the B-tree is created, all of the B-tree nodes are flushed from every processor, so that only the home nodes have valid data. Thirty-two threads are created, one on each processor, each of which repeatedly initiates either lookup or insert requests into the B-tree. After every processor completes a series of cold accesses, we repeat the series and measure warm access times.

Any background threads that are created (to split nodes, as described above) are started on the home processor of the node that must be written. This optimization, which reduces coherence costs, is performed in all of our experiments; it can be viewed as performing immediate computation migration on the background thread. Dynamic computation migration only occurs at the leaves of the B-tree, since any splits that propagate upwards are executed in background threads.

Figure 5 illustrates the results of using our policies for a B-tree on Alewife, and shows that dynamic computation migration can improve performance for a B-tree. The coherence overhead for B-tree nodes is high, because they are large. Paradoxically, the STATIC policy generally performs better than the REPEAT policy, even when the percentage of lookups is low (and the percentage of writes is high). The reason for this apparent anomaly is that a traversal of the B-tree must visit at least two interior nodes and one leaf node. Visits to the non-leaf nodes from inserts and lookups are always reads, so the rate of writes is very low (writes occur when an insert splits a leaf node). As a result, it is more efficient to always use data migration for the interior nodes, which the STATIC policy does. The REPEAT policy performs worse because it uses computation migration for some reads of interior nodes.

Although the STATIC policy performs better than the REPEAT policy when the percentage of lookups is less than 60%, the REPEAT policy performs better for lookup percentages between 60% and 90%. For example, at 80% lookups, the REPEAT policy performs 23% better than pure data migration, whereas the STATIC policy performs only 5% better than pure data migration. The reason is that when the lookup rate increases to 80%, the rate of writes on the second level of the tree drops, so that the effect described in the previous paragraph becomes less significant. Instead, the performance gain for using the REPEAT policy on the leaf nodes becomes more significant, which results in better performance.

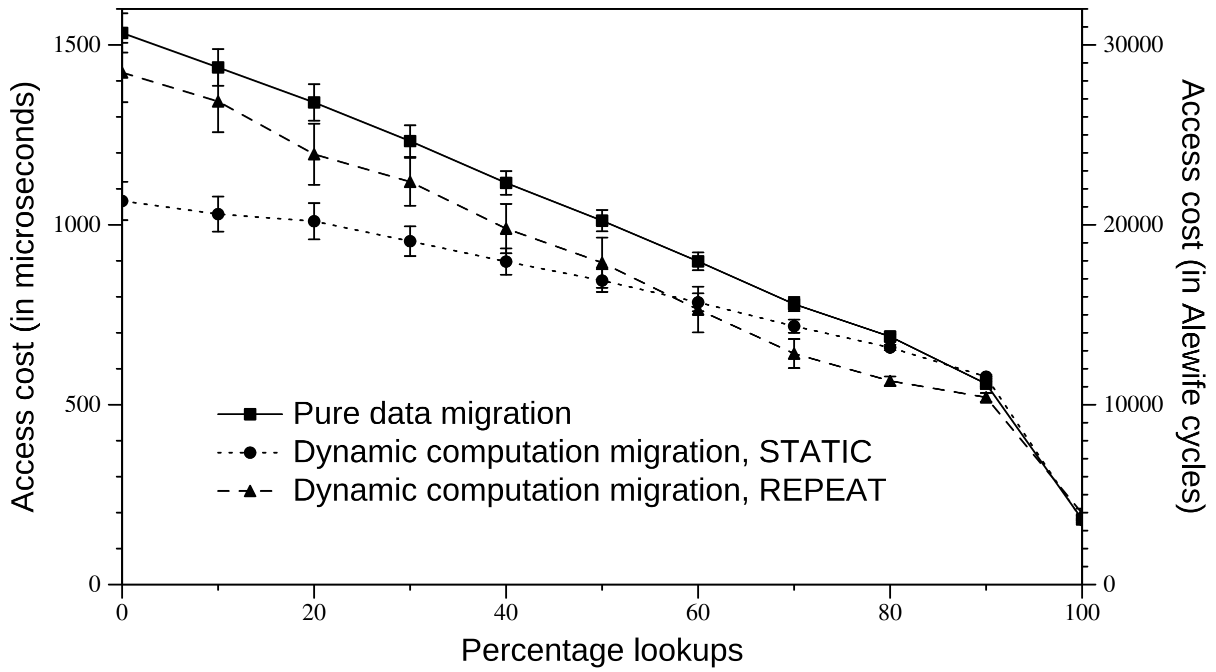

Figure 5. Cost of accessing a B-tree on Alewife. Comparison of pure data migration and dynamic computation migration using the STATIC and REPEAT policies. Thirty-two processors access a three-level B-tree.

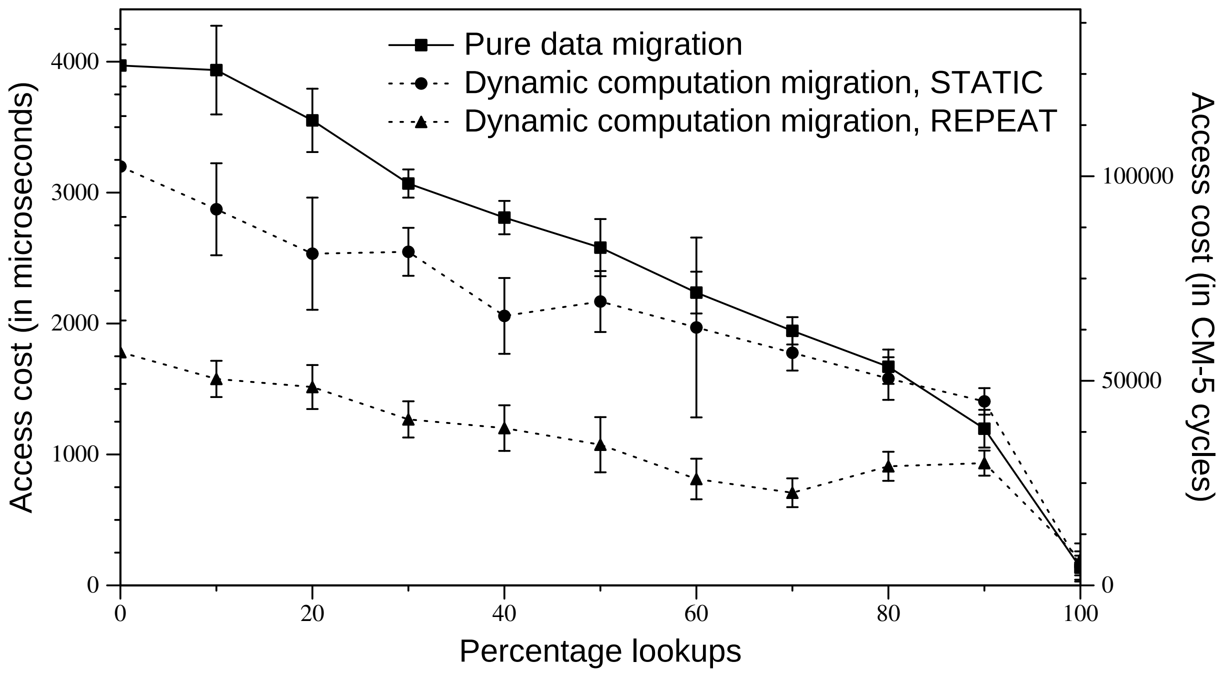

Figure 6. Cost of accessing a B-tree on the CM-5. Comparison of pure data migration using the STATIC and REPEAT policies. Thirty-two processors access a three-level B-tree.

Although the REPEAT policy generally performs worse than the STATIC policy, both policies outperform pure data migration when the percentage of lookups is lower than 90%. The nodes are large enough to make writes expensive under data migration; as a result, computation migration improves performance even when there is a moderate proportion of writes.

Figure 6 illustrates the results of using our policies for a B-tree on the CM-5. The shapes of the curves are similar, although the STATIC and REPEAT curves do not cross. Communication is much more expensive on the CM-5 than on Alewife. As a result, computation migration is more effective, and using computation migration when there are more than 10% writes improves performance.

The communication to computation performance of the CM-5 is close to what we would expect of a workstation cluster. Therefore, we expect that our results would apply to such a parallel system, and that dynamic computation migration would be effective on a workstation cluster. Alewife has relatively high communication performance, relative to its computation performance, and we view this as a lower bound on the performance benefits of dynamic computation migration.

The fact that most B-trees used in databases exhibit very high read rates does not invalidate our results. First, the pattern of accesses to a B-tree may go through different phases: in phases where new data is being added to the B-tree, the write rate may be fairly high. Second, our B-tree does not implement deletes, and the rate of writes in a real B-tree would include deletes. Third, our B-tree nodes are not that large; with larger nodes, the cost of coherence is higher. Finally, we expect that other concurrent, distributed data structures will exhibit read/write ratios less than 90%.

5 Related Work

The work described in this paper builds upon our earlier work, in which we explored the use of static computation migration [11]. The Olden project [4] has also explored static computation migration. Rogers and Carlisle have investigated language and compiler support for statically determining when to migrate computation or data. The programmer is required to give the compiler indirect knowledge of how data is laid out by specifying how likely a data structure traversal is to cross processors. Such a strategy works well for scientific applications, where the layout of data structures, and the pattern of accesses to those data structures, is fairly predictable. A static decision between computation migration and data migration does not work for truly dynamic data structures.

DSM systems typically use only one protocol for maintaining the coherence of data, whereas the access patterns for data vary widely. The Munin project [5] provides various different coherence protocols that can be used for different types of data. As in other DSM systems, Munin restricts its attention to the migration of data, and does not allow for the migration of computation. Munin does provide support for migratory data in the form of RPC.

Several systems, such as Mercury [8] and COOL [6], use cache-conscious scheduling, which is a scheduling policy designed to increase cache reuse. The use of computation migration can increase global cache effectiveness, because by avoiding replication it reduces pressure on the cache. Nonetheless, this effect is second-order in comparison to the performance benefit of eliminating coherence messages.

6 Future Work

The exploration of user annotations to control dynamic migration directly is likely to be fruitful. Such annotations could allow the user to specify hints for dynamic migration in a region-specific manner. For example, in the B-tree it would help to use the level of a node when deciding whether to use computation or data migration: the root node should always be replicated, whereas the lowest level should use dynamic computation migration. Since it is implemented in software at user-level, MCRL is sufficiently flexible to allow for application-level decisions.

This paper has examined dynamic computation migration in a DSM on homogeneous, tightly coupled multiprocessors. The results should remain valid for DSM systems on more loosely coupled parallel systems, such as workstation clusters. An interesting area of investigation will be the use of dynamic computation migration in heterogeneous systems, especially because the same program image would not be loaded on every processor.

7 Conclusions

We have described dynamic computation migration, which is the dynamic choice between using computation migration and data migration. We have also described MCRL, a multithreaded DSM system that we built to evaluate dynamic computation migration. We demonstrated that dynamic computation migration can improve performance on dynamic data structures:

- Using computation migration for writes performs well, even with the overhead of making a dynamic choice. In general, synchronization objects tend to be written frequently, which makes them good candidates for computation migration.

- The STATIC policy performs well for a B-tree. Using the STATIC policy improves performance by 17% on Alewife, relative to pure data migration, when the operation mix is 50% lookups and 50% lookups. The STATIC policy improves performance by 16% on the CM-5 for the same operation mix.

- The REPEAT policy generally performs better than the STATIC policy on a B-tree. On Alewife, with an operation mix of 80% lookups and 20% inserts, the REPEAT policy improves performance by 23% relative to pure data migration. On the CM-5, it improves performance by 46%.

Dynamic computation migration is useful for reducing coherence costs in applications that do not have statically known (or predictable) data access patterns. Most scientific applications, such as those in the well-known SPLASH benchmark suite [16], have communication patterns that are very predictable: they have relatively static data and thread layout. Future parallel applications are unlikely to have such characteristics.

Although network performance continues to improve, sending messages will remain significantly more expensive than local computation. Therefore, computation migration will remain important, because computation migration reduces communication costs. Our Alewife results are optimistic with respect to communication costs, since the processor architecture is slow, compared to the networking architecture. Our CM-5 results should be more representative of the behavior that future multiprocessors will exhibit.

Acknowledgments

We thank Kirk Johnson for building CRL, and the anonymous reviewers for their helpful comments.

References

[1] A. Agarwal, R. Bianchini, D. Chaiken, K.L. Johnson, D. Kranz, J. Kubiatowicz, B.H. Lim, K. Mackenzie, and D. Yeung. **The MIT Alewife Machine: Architecture and Performance**. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 2–13, Santa Margherita Ligure, Italy, June 22–24, 1995.

[2] J. Aspnes, M. Herlihy, and N. Shavit. **Counting Networks**. *Journal of the ACM*, 41(5):1020–1048, September 1994.

[3] R. Bayer and E.M. McCreight. **Organization and Maintenance of Large Ordered Indexes**. *Acta Informatica*, 1(3):173–189, 1972.

[4] M.C. Carlisle and A. Rogers. **Software Caching and Computation Migration in Olden**. In *Proceedings of the 5th Symposium on Principles and Practice of Parallel Programming*, pages 29–38, Santa Barbara, CA, July 1995.

[5] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. **Techniques for Reducing Consistency-Related Communication in Distributed Shared-Memory Systems**. *ACM Transactions on Computer Systems*, 13(3):205–243, August 1995.

[6] R. Chandra, A. Gupta, and J.L. Hennessy. **COOL: An Object-Based Language for Parallel Programming**. *IEEE Computer*, 27(8):13–26, August 1994.

[7] J.S. Chase, F.G. Amador, E.D. Lazowska, H.M. Levy, and R.J. Littlefield. **The Amber System: Parallel Programming on a Network of Multiprocessors**. In *Proceedings of the 12th Symposium on Operating Systems Principles*, pages 147–158, December 1989.

[8] R.J. Fowler and L.I. Kontothanassis. **Improving Processor and Cache Locality in Fine-Grain Parallel Computations using Object-Affinity Scheduling and Continuation Passing (Revised)**. Technical Report 411, University of Rochester Computer Science Department, June 1992.

[9] M. Herlihy, B.H. Lim, and N. Shavit. **Scalable Concurrent Counting**. *ACM Transactions on Computer Systems*, 13(4):343–364, November 1995.

[10] W.C. Hsieh. **Dynamic Computation Migration in Distributed Shared Memory Systems**. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, September 1995. Available as MIT/LCS/TR-665.

[11] W.C. Hsieh, P. Wang, and W.E. Weihl. **Computation Migration: Enhancing Locality for Distributed-Memory Parallel Systems**. In *Proceedings of the 4th Symposium on Principles and Practice of Parallel Programming*, pages 239–248, San Diego, CA, May 1993.

[12] K.L. Johnson, M.F. Kaashoek, and D.A. Wallach. **CRL: High-Performance All-Software Distributed Shared Memory**. In *Proceedings of the 15th Symposium on Operating Systems Principles*, pages 213–228, Copper Mountain, CO, December 3–6, 1995. http://www.pdos.lcs.mit.edu/crl.

[13] E. Jul, H. Levy, N. Hutchison, and A. Black. **Fine-Grained Mobility in the Emerald System**. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[14] P.L. Lehman and S.B. Yao. **Efficient Locking for Concurrent Operations on B-Trees**. *ACM Transactions on Database Systems*, 6(4):650–670, December 1981.

[15] A. Rogers, M. Carlisle, J. Reppy, and L. Hendren. **Supporting Dynamic Data Structures on Distributed-Memory Machines**. *ACM Transactions on Programming Languages and Systems*, 17(2):233–263, March 1995.

[16] J.P. Singh, W. Weber, and A. Gupta. **SPLASH: Stanford Parallel Applications for Shared Memory**. *Computer Architecture News*, 20(1):5–44, March 1992.

[17] C.A. Thekkath and H.M. Levy. **Limits to Low-Latency Communication on High-Speed Networks**. *ACM Transactions on Computer Systems*, 11(2):179–203, May 1993.

[18] T. von Eicken, A. Basu, V. Buch, and W. Vogels. **U-Net: A User-Level Network Interface for Parallel and Distributed Computing**. In *Proceedings of the 15th Symposium on Operating Systems Principles*, pages 40–53, Copper Mountain, CO, December 3–6, 1995.

[19] P. Wang. **An In-Depth Analysis of Concurrent B-Tree Algorithms**. Master's thesis, Massachusetts Institute of Technology, January 1991. Available as MIT/LCS/TR-496.