Evaluating Cache Coherence in the DSMIO System *

Carla Osthoff[1], Cristiana Seidel[3], Ricardo Bianchini[2], Marta Mattoso[2], and Claudio L. Amorim[2]

[1] Departamento de Computação Eletrônica - LNCC
[2] Programa de Engenharia de Sistemas e Computação - COPPE/UFRJ
[3] Departamento de Engenharia de Sistemas e Computação - UERJ
osthoff@lncc.br, seidel@eng.uerj.br, {ricardo,marta,amorim}@cos.ufrj.br

Abstract—
There has been a significant amount of research on object-based database management systems (ODBMS). In contrast to traditional relational systems, ODBMS use the same data model for the clients and the server. Thus, parallel ODBMS typically adopt a data-shipping architecture, which allows data request processing to be performed at the clients. Although such a strategy can improve performance by moving data closer to clients and by alleviating the load on the server, it raises the issue of cache coherence. Our work is based on the observation that coherence maintenance techniques developed to improve software-based distributed shared-memory systems can be exploited to improve parallel ODBMS. In previous papers we proposed a coherence algorithm, called DSMIO, that uses the Lazy Release Consistency model and the diffing mechanism to limit the number and size of coherence messages and save cache space. These papers evaluated DSMIO in the context of a parallel object-based database system where one machine acted as a server, accessing stable storage and keeping client caches coherent. In this paper we evaluate DSMIO for a system where each machine can act both as a client and a server. We use the state-of-the-art CallBack Locking (CBL) cache coherence algorithm as a basis for comparison. Our results show that DSMIO significantly outperforms CBL for workloads with even a moderately high frequency of database updates.

Keywords— **Client-Server OODBMS, Parallel Database Systems, Cache Coherence, Distributed Shared Memory**

I. Introduction

There has been a significant amount of research on object-based database management systems (ODBMS), e.g. [5, 24], both in their object-oriented and object-relational instantiations. The main goal of the ODBMS is to provide a rich data type system with high performance. This rich data type system is fundamental for applications such as CAD/CAM and CASE, but it is also beneficial for commercial applications. Regardless of the specific application, performance is always an issue when large volumes of data are involved. Advances in the areas of parallel and distributed computing have led to client/server and parallel ODBMS. Given the representation characteristics of ODBMS, these systems have gained popularity in recent years.

In contrast to traditional relational database systems, ODBMS use the same data model for the clients and the server. Thus, parallel ODBMS typically adopt a data-shipping architecture, which allows data request processing to be performed at the clients [4, 18]. With data-shipping, the clients determine which data items are needed to satisfy a given request and obtain those items from the server. However, strict data-shipping implementations are susceptible to network and/or server bottlenecks that can arise when a high volume of data is requested by clients. The key to avoiding these bottlenecks is to cache data at the clients. The combination of data-shipping and client caching can improve performance by moving data closer to clients and by alleviating the load on the server. Unfortunately, client caching raises the issue of cache coherence.

Cache coherence issues arise in many types of parallel and distributed systems, including Distributed Shared-Memory (DSM) systems (e.g. [2, 22, 10, 13, 21]). In fact, there are many similarities between the basic coherence maintenance techniques available in parallel database and DSM systems [7]. Furthermore, the study of database cache coherence is very tightly coupled with the study of concurrency control, since cached data can be concurrently accessed by multiple clients and locks can be cached along with data at clients. Our work is based on the observation that the techniques developed to improve software DSM systems can be exploited to improve parallel database systems that apply client caching. In particular, well-established techniques to reduce false-sharing in page-based software DSM systems, such as relaxed memory consistency and the diffing mechanism, can be very useful for parallel database systems.

With these ideas in mind, in a previous paper [17] we proposed a coherence algorithm, called DSMIO, that uses the Lazy Release Consistency model and the diffing mechanism to limit the number and size of coherence messages and save cache space. That paper presents the main principles behind the algorithm and a set of preliminary performance results. In a more recent paper [15] we presented the current version of the algorithm and a detailed analysis of its performance in the context of a parallel object-based database system where one machine acts as a server, accessing sta-

* Ricardo Bianchini is currently on leave at Rutgers University. This work was supported in part by FINEP and CNPq.

ble storage and keeping client caches coherent. The DSMIO results were compared to those of the state-of-the-art Call-Back Locking (CBL) cache coherence algorithm [6]. CBL has been previously shown to achieve good performance over a wide range of system configurations and workload characteristics [25, 18, 4].

In this paper we extend our previous works by evaluating DSMIO for a parallel database system where each machine can act both as a client and a server, i.e. each machine accesses stable storage and takes active part in keeping data caches coherent. Again, we use CBL as a basis for comparison. Our results show that DSMIO significantly outperforms CBL for workloads with a high frequency of database updates. These results confirm our intuition that techniques borrowed from software-based DSM systems can improve the performance of parallel database systems.

The remainder of this paper is organized as follows. The next section describes the cache coherence problem concentrating on background information about memory consistency models and the description of the CBL algorithm. Section III overviews the DSMIO algorithm by comparing it against CBL. Section IV presents our methodology and application workload. In section V we present the results of our most important experiments. We relate our work to previous research in section VI. Finally, in section VII, we present our conclusions and proposals for future work.

II. The Cache Coherence Problem

Caches are used to amortize the high cost of accesses to lower levels of the memory hierarchy. Caches have been shown effective for sequential, parallel, and distributed systems. The utilization of caches in sequential systems is straightforward. In contrast, caching in a parallel or distributed system is not as simple, since the system may have to keep caches coherent to avoid application malfunction.

In the context of parallel database systems, the cache coherence problem is inherent to the existence of writable client caches. Thus, these systems incorporate cache coherence protocols or algorithms. In addition, these systems must implement a synchronization mechanism to ensure mutual exclusive access to shared data, which is usually implemented by the well-known *lock/unlock* primitives. Both the cache coherence algorithm and the synchronization primitives must be carefully implemented, as they can cause high communication and coherence-induced overheads that can potentially limit performance.

In the DSM world, it is well known that algorithms based on relaxed memory consistency models can reduce these overheads by delaying and/or restricting communication and coherence operations as much as possible. In addition to the use of relaxed consistency models, multiple-writer systems attempt to reduce communication and coherence overheads further by allowing two or more processors to modify their local copies of shared data concurrently and merging modifications at synchronization operations. Relaxed consistency and multiple writers alleviate the effect of false sharing in systems with large coherence units.

A. Relaxed Memory Consistency Models

The memory consistency model provides a formal specification of how the memory system appears to the programmer. A number of relaxed consistency models have been proposed in the literature [1] to allow a process to delay the visibility of its shared data modifications to other processors until certain synchronization accesses occur. Relaxed consistency models allow views of the memory system to become inconsistent until subsequent synchronization events.

The most commonly used relaxed consistency models are the Release Consistency (RC) [8] and Lazy Release Consistency (LRC) models [9]. These two models categorize the synchronization accesses in *acquires* and *releases*. Acquires are used to gain access to shared data (e.g. lock operation) and releases are used to give such access away (e.g. unlock operation).

In simple terms, in RC the modifications made to shared data by a client C_i need to become visible at another client C_j only when a subsequent release executed by C_i becomes visible at C_j. LRC, on the other hand, does not require that modifications be globally visible at the time of a release. Instead, LRC only guarantees that the one processor that acquires a lock will see the modifications that precede the lock acquire. To know which modifications actually precede a lock acquire, LRC divides the program execution into *intervals* in such a way that the intervals can be partially ordered by synchronization accesses. So, on a lock acquire, the acquiring processor asks the last releaser of the lock for the information about the modifications made to shared data during the intervals that precede the acquiring processor's current interval.

LRC has been implemented in several page-based software DSM systems, such as [10, 13, 21]. TreadMarks [10] is one of the most popular page-based software DSM systems and forms the basis of our DSMIO system. In TreadMarks, cache coherence is implemented via invalidations, i.e. the information about modifications to shared data takes the form of invalidations (*write-notices* in TreadMarks terminology) transferred on lock acquire operations. A write-notice informs that a virtual memory page has been modified during a particular interval, but does not contain the new data written to the page. The new data is only requested and later received by the acquiring processor on a subsequent access to the page.

TreadMarks and all other page-based systems can potentially suffer from false sharing. False sharing occurs when multiple processors write to different data structures allocated to the same page. Due to the large size of the virtual memory page, this problem can cause tremendous performance degradation. To counter this problem, TreadMarks and several other systems allow multiple processors to write to the same page concurrently. To avoid conflicts, the modifications made to pages in multiple writer systems are usually stored in the form of page diffs, which encode the modifications leaving the unmodified data out.

B. The CBL Cache Coherence Algorithm

In the CallBack Locking (CBL) cache coherence algorithm, clients access a virtual memory page by acquiring a write or a read lock on the page. When a client C_i tries to acquire a read or a write lock for a page P, it sends a lock request message to the server of the page and blocks until it gets a response from the server. If there is no one else caching page P, the server immediately grants the read or write lock page P to client C_i. If there is another client, C_j, caching page P, the server takes different actions depending on the type of lock C_i is requesting and the type of lock C_j is holding:

(a) If C_i requests a read lock and C_j holds a read lock, the server immediately grants the read lock to C_i, allowing the two clients to hold a read lock for the same page;

(b) If C_i requests a read lock and C_j holds a write lock, the server sends a callback message to C_j, which responds to the server indicating whether it is still updating the page or not. If C_j is updating P, C_i remains blocked until C_j commits. Otherwise, the lock is granted immediately;

(c) If C_i requests a write lock and C_j holds a read lock, the server sends a callback message to C_j. If C_j is not reading page P anymore, it invalidates P from its cache and sends a callback reply to the server. The server then sends a response to C_i granting it an exclusive lock on page P. Thus, when a page is cached at different clients, it is necessary to send callback messages to all other clients that cache the page, and the initial lock requesting client blocks until all of these messages are processed.

(d) If C_i requests a write lock and C_j holds a write lock, the server sends a callback message to C_j. If C_j is still updating the page, C_i has to wait. Otherwise, C_j invalidates P from its cache and sends a callback reply to the server. The server then sends a response to C_i granting it an exclusive lock on page P.

In CBL, before a lock is acquired, the previous writes to the page must be observed at the server and at all the clients that cache the page, so it follows the Release Consistency (RC) model.

The CBL algorithm suffers from two main overhead problems, false sharing and callback delay. As in page-based DSM systems, false sharing is a problem for CBL as a result of its use of the page as the coherence unit. A page is usually much larger than an object, so many objects can fit in a page. False sharing occurs when two clients write different objects allocated to the same page. False sharing is particularly harmful to CBL, since CBL does not allow multiple concurrent writers to the same page, as described above. The callback delay is due to the fact that the server has to wait for a callback reply from each client that has a page copy, before sending a write lock to a client.

III. The DSMIO Cache Coherence Algorithm

The DSMIO algorithm has been described in detail in previous papers [17, 15], so we will focus this section on the differences between DSMIO and CBL.

An important difference between the two systems is the memory consistency model they adopt. CBL is based on the RC model. DSMIO, on the other hand, is based on the LRC model and uses the same infra-structure of intervals and write-notices described in the previous section to keep track of which modifications have to be seen by other processors. In addition, DSMIO allows multiple concurrent writers by using the diff mechanism, while CBL is a single-writer system.

Another difference between the systems lies in the programming model they support. The CBL algorithm requires the programmer to differentiate between read and write locks, while DSMIO only implements mutual exclusion locks. The negative impact of relying exclusively on mutual exclusion locks is limited in DSMIO by its use of LRC. Nevertheless, we intend to add read and write locks to DSMIO in order to address this issue.

The read and write operations themselves are also slightly different in the two systems. On a DSMIO read operation, the client requests the page to the server. The server reads the page from the disk, stores it in the DSMIO shared cache and sends a copy to the client. When the client receives the page from the server, it consults its list of write notices to find out whether it needs diffs from other clients. If so, it requests the diffs and waits for them to be sent back. After receiving all the diffs requested, the client can apply them in turn to its outdated copy of the page.

On a DSM write operation, the client first starts a read operation to bring the page up-to-date. After that, it creates a twin copy of the page and copies a private up-to-date version of the page on top of it. At this point, the DSMIO algorithm generates a diff for the page, comparing the modified version with its twin. The diff is stored in the client's local memory and a write notice is incorporated to its current interval.

IV. Methodology and Workload

A. Platform

Our experimental environment consists of a completely dedicated IBM SP2 with sixteen 66MHz Power2 processors running the AIX operating system. The nodes are connected by a 40 MBytes/s Omega switch. All our systems communicate using the UDP protocol.

B. Workload

The OO7 benchmark [3] has been developed to study the performance of ODBMS, so we use the OO7 benchmark database specification in our performance analysis. The key component of the OO7 benchmark database is a set of *Composite Parts*. Each Composite Part has an associated graph of *Atomic Parts*. The Atomic Parts are the units out of which the Composite Part is constructed.

In our experiments we use the medium benchmark database (100 Mbytes), where each Composite Part contains 200 Atomic Parts. Although Composite and Atomic Parts comprise the bulk of the OO7 database, there is still a higher level in the OO7 hierarchy, *Assembly Parts*. Each Assembly Part is made up of 3 Composite Parts. The medium base has a total of 728 Assembly Parts.

Besides the definition of the database, the OO7 benchmark also specifies the operations made on the database. We are interested in a special class of these operations, called traversals. The OO7 traversal operations navigate procedurally from object to object, updating objects whenever specified. However, the broad range of OO7 traversals is underspecified for our experiments because they do not include the data sharing patterns. So, in our performance analysis we use a single kind of OO7 traversal, **T3**, with and without updates. We refer to these two variations of T3 as the *update* and *read applications*, respectively. We also vary the percentage of data sharing in each of these applications. We do so by diving the database into a set of private regions and a common shared region, and defining three different workloads: *private*, *shared*, and *high contention*. In the private workload, each client reads/writes data from/to its private region for 80% of the time and reads data from the shared region for 20% of the time. In the shared workload, each client reads data from its private region for 80% of the time and reads/writes data from/to the shared region for 20% of the time. In the high contention workload, each client reads data from its private region for 20% of the time and reads/writes from/to the shared region for 80% of the time. There is no data contention in the private workload, while the data contention increases when we move from the shared to the high-contention workload.

The whole database is distributed among the disks of all processing nodes. The data is distributed in round-robin fashion. We implement a balanced distribution in such a way that 75% of the data is accessed locally and the other 25% is accessed remotely.

C. ODBMS

To evaluate the performance of DSMIO and CBL we incorporated them into an object-based database management system called GOA, developed at COPPE/UFRJ [12].

GOA implements a client/server architecture. The client contains the application code, while the server executes the database persistence services and query processing. The GOA client requests objects from the GOA server. The GOA data model is compliant to the ODMG standard, and thus ODL is used to create a database schema and OQL is used to query stored collections of objects.

The original implementation of GOA is single user. Therefore, it lacks concurrency control and transaction management. We eliminated this shortcoming by starting a sequential GOA process on each machine and implementing concurrency control and transaction management within the cache coherence algorithms.

We implemented two parallel GOA systems, CBL and DSMIO-based systems, which only differ in terms of their cache architecture. In the CBL-based system (GOA-CBL), each processor has GOA's original client cache (4 Mbytes) and a server cache (36 Mbytes). In this way the clients access the client cache first and, when there is a cache miss, they request the data to the server. The server either reads the data from local disk (in case it has been assigned the corresponding part of the data set) or forwards the request to another processor (who is the one responsible for storing the corresponding data). The system implements concurrency and transaction management according to the CBL algorithm.

The DSMIO-based parallel GOA system (GOA-DSMIO) alters the original GOA cache architecture in order for both the clients and the server to access a single cache (which is set by default to 40 Mbytes, the same size as the sum of the client and server caches of each processor in GOA-CBL), as in any shared-memory system. When a client cache miss occurs, the client requests the data to the server. The server either reads the data from local disk or forwards the request to another processor, again depending on which machine stores the data locally. The system implements concurrency and transaction management according to the DSMIO algorithm.

V. Experimental Results

In this section we compare the performance of DSMIO's coherence algorithm against that of CBL.

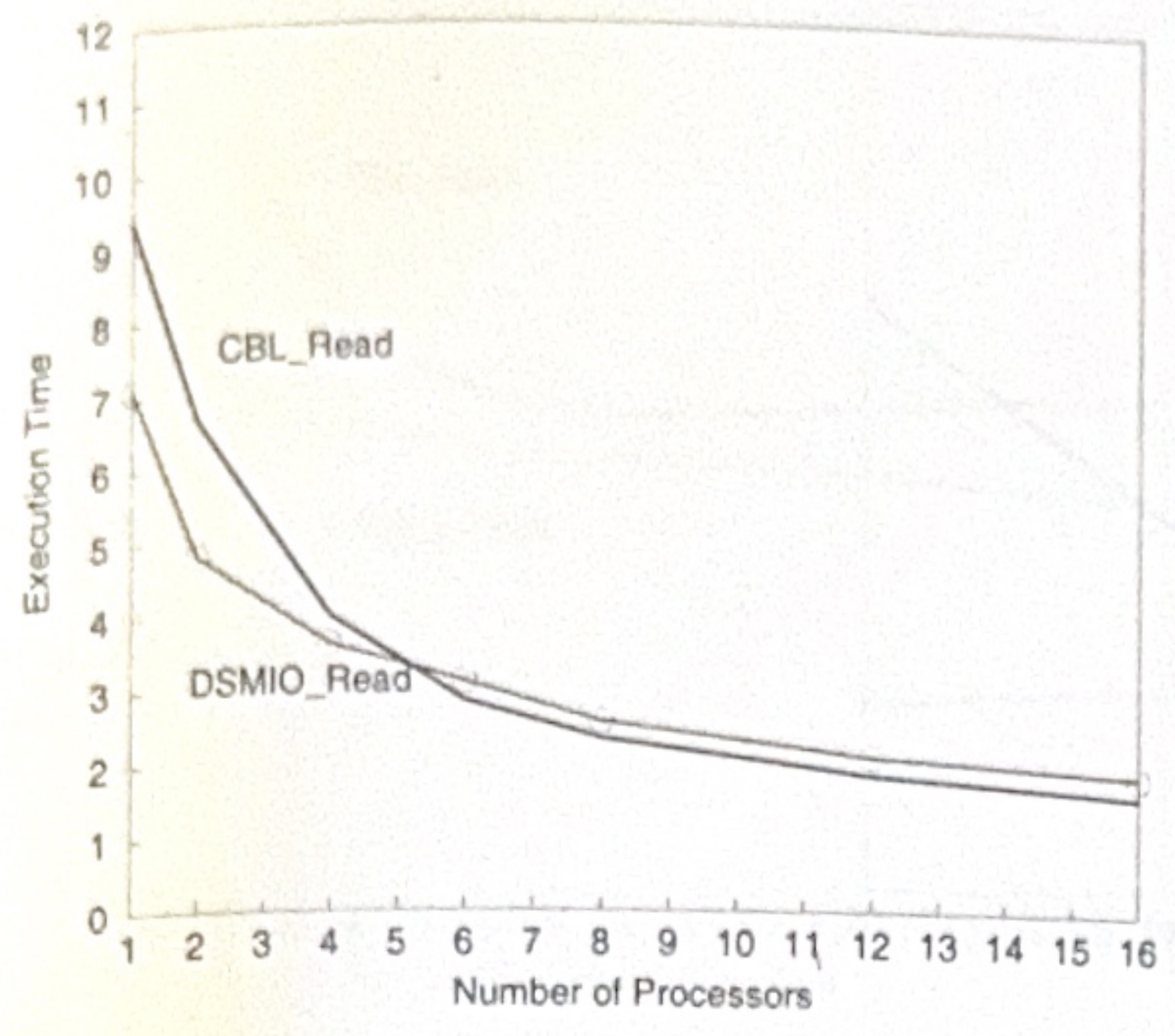

Fig. 1: Private Workload Execution Time (secs)

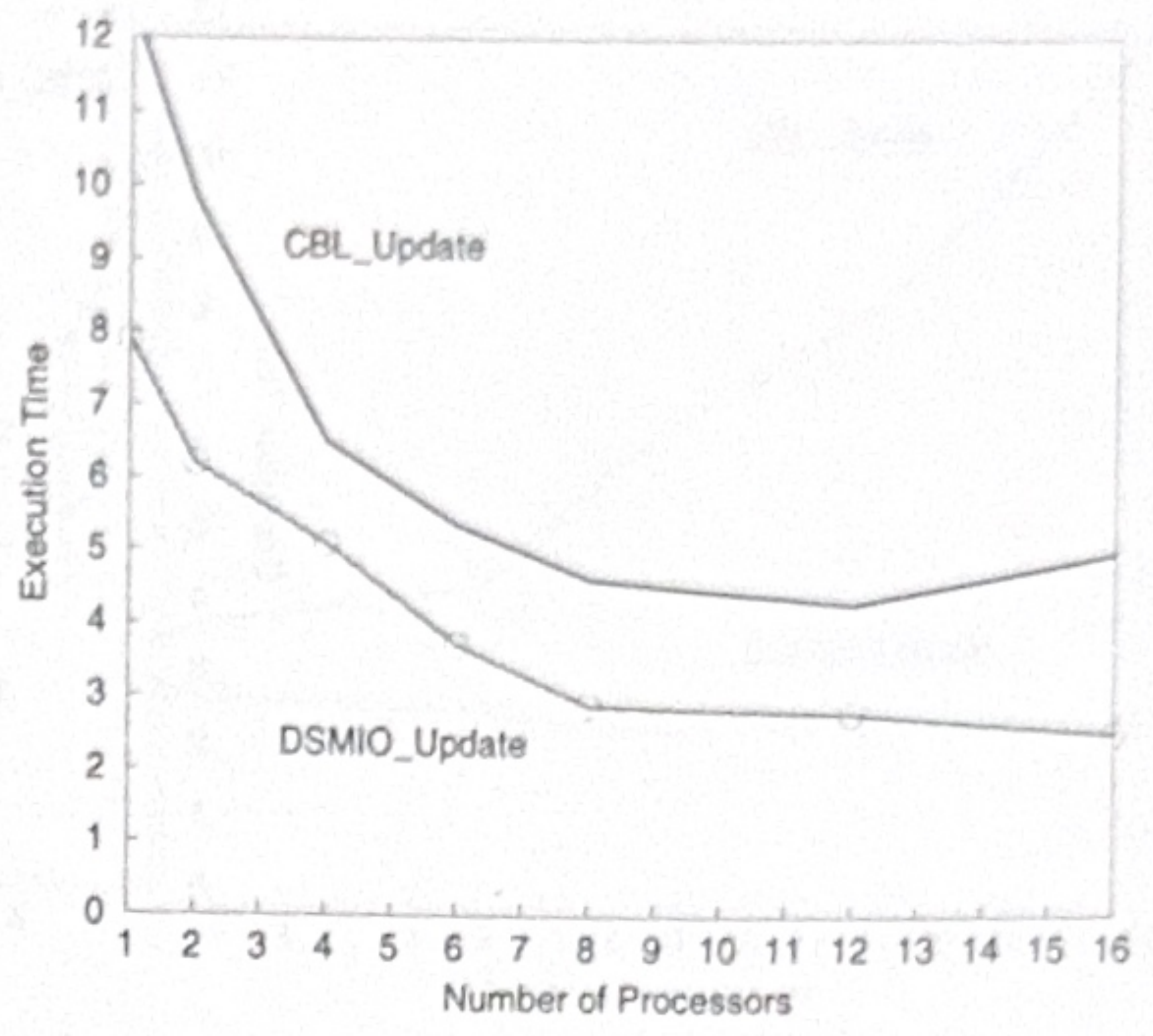

Fig. 2: Private Workload Execution Time (secs)

A. Private Workload

In figure 1 we present the execution times (in seconds) for the private workload, as a function of the number of processors. The different curves in the figure represent the performance of the read application under DSMIO and CBL. The two curves for the read application show that the read performance of DSMIO is better than that of CBL for up to 4 processors. However, from 6 to 16 processors, the CBL read performance becomes slightly better than the DSMIO read performance.

CBL performs worse than DSMIO for read operations and small numbers of processors because CBL maintains two separate caches per node, a client cache and a server cache, so that every read miss on the client cache requires a page copy from the server cache. DSMIO, in contrast, has only one cache per node and thus does not require such page copies. However, the DSMIO read performance does not improve as much as CBL's read performance as we increase the number of processors. The reason for this is that, in our current implementation of DSMIO, the server denies remote requests while it is modifying the shared cache. Thus, when there is a higher demand for server activities, the waiting time increases for remote requests. We are currently working on ways to improve this aspect of DSMIO.

In figure 2 we present the execution times (in seconds) for the private workload of the update application, as a function of the number of processors. We can observe from the figure that DSMIO performs significantly better than CBL for the update application. This advantage is mainly due to the different effects of false sharing in the two systems. As CBL does not allow multiple concurrent writers to the same page, two clients cannot simultaneously write to different objects that belong to the same page. In CBL, only one client can modify the page at a time. DSMIO, on the other hand, uses the diffing mechanism and allows multiple concurrent writers to the page, alleviating the negative impact of false sharing on the system's performance. Furthermore, DSMIO implements the LRC model, which generates fewer coherence messages than the RC model implemented by CBL. In fact, as the number of processors increases, the number of coherence messages sent by DSMIO increases by a factor of 4 (from 800 to 3000 messages), while the number of coherence messages sent by CBL increases by a factor of 17 (from 1800 messages on 2 processors to 30000 messages on 16 processors). The average callback delay of CBL increases only slightly from 22 microseconds on 2 processors to 23 microseconds on 16 processors.

B. Shared Workload

In figure 3 we present the execution times (in seconds) for the shared workload, as a function of the number of processors. The different curves in the figure represent the performance of update application under DSMIO and CBL. Note that the execution time behavior of this workload is similar to that of the private workload. In terms of the performance of the read application, the two systems exhibit the same performance trends and compare in exactly the same way as before, see figure 1.

In terms of the performance for the update application, the two systems again behave in the same way as for the private workload curves. However, the performance difference between CBL and DSMIO is explained by other reasons. Although, the shared update workload also exhibits false sharing, true sharing dominates. Therefore, the performance difference between the two systems comes mainly from the coherence overhead involved in their different con-

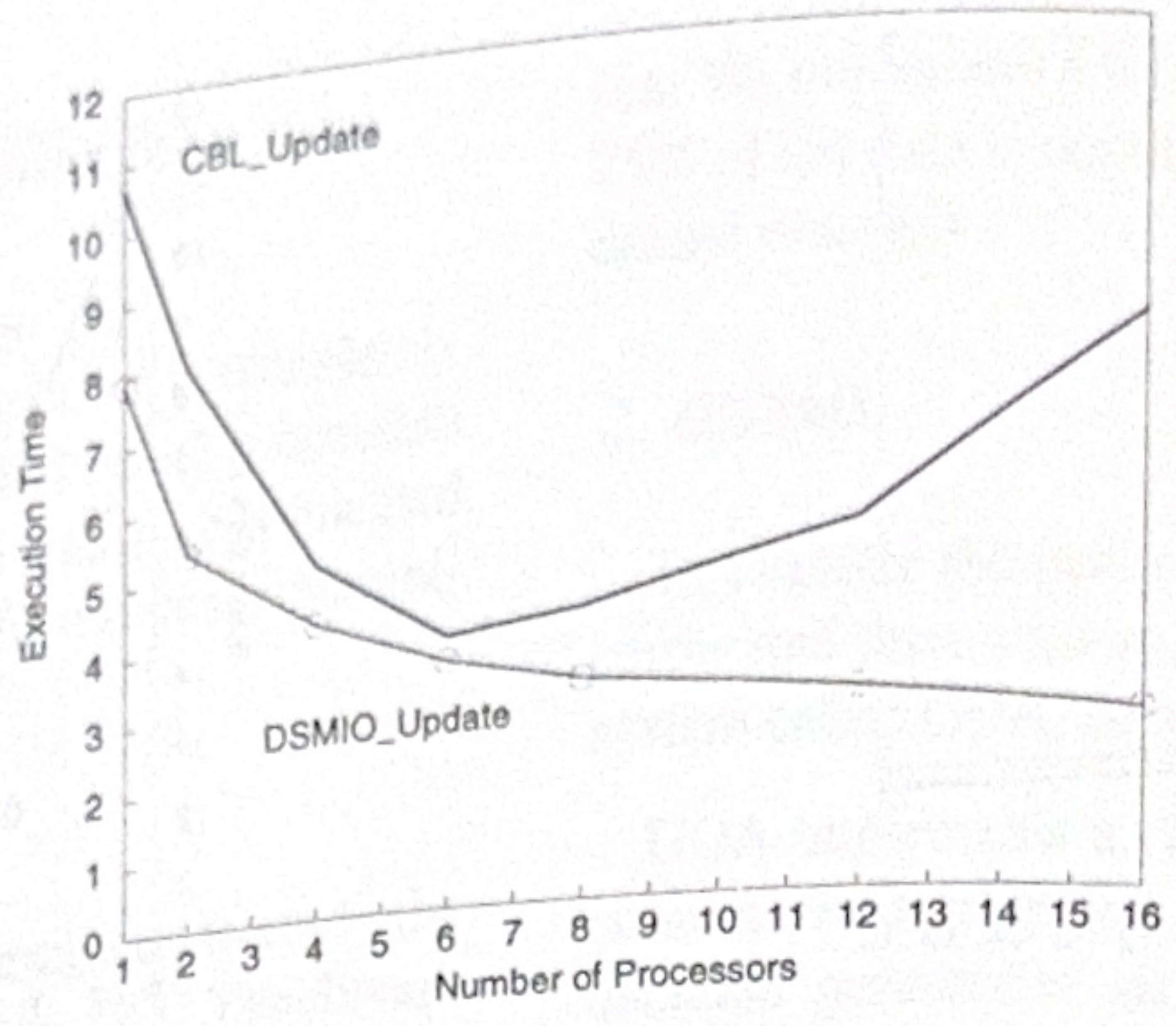

Fig. 3: Shared Workload Execution Time (secs)

sistency models. This overhead manifests itself in terms of coherence messages and the amount of time a processor has to wait for data to be made coherent. As DSMIO uses a more relaxed consistency model than CBL, it sends fewer messages than its counterpart. More specifically, DSMIO sends from 600 messages on 2 processors to 1600 messages on 16 processors, while CBL sends from 700 to 30000 coherence messages. CBL also involves a higher waiting time per page access, as a result of its high callback overhead (the server has to wait for a callback reply from each client that has a copy of the page).

C. High-Contention Workload

In figure 4 we present the execution times (in seconds) for the high-contention workload, as a function of the number of processors. The different curves in the figure represent the performance of read application under DSMIO and CBL. In contrast with previous experiments, the curves for the read application shows that the DSMIO read performance is better than that of CBL for all numbers of processors we study. The reason for this result is that the large number of remote requests cause more cache misses in CBL than in DSMIO. In CBL, the reply to a remote request made by a client is stored in the client cache. As the client cache is small (4 Mbytes) in our experiments, it incurs a significant number of cache misses. In DSMIO, on the other hand, the replies are stored in the shared cache and, thus, generate a smaller number of cache misses. (Recall that the DSMIO cache size is the same as the sum of the server and client caches of each processor in CBL. The performance tradeoff here is that, even though the integrated cache of DSMIO generates fewer remote requests, a DSMIO node has to deny remote requests while it is modifying the shared cache.)

In figure 5 we present the execution times (in seconds) for the high-contention workload of the update application as a function of the number of processors. In terms of the performance for the update application, DSMIO consistently and significantly outperforms CBL for this workload. As explained for the shared workload, the performance difference between the two systems comes from the coherence overhead (the number of messages exchanged and the time a processor has to wait for data to be made coherent). For the high-contention update workload, this overhead is even higher, as there is a much larger number of remote requests. More specifically, CBL sends from 18000 coherence messages on 2 processors to 37000 messages on 16 processors, while DSMIO sends from 2.400 to 12000 messages, respectively. The average callback delay of CBL increases from 23 microseconds on 2 processors to 35 microseconds on 16 processors.

Even though CBL suffers from a large number of coherence messages and a high overhead for making data coherent, it includes read locks, while DSMIO currently relies exclusively on mutual exclusion locks. In order to estimate the negative impact of this limitation, we implemented a variation of the High Contention workload, where data are never written (so mutual exclusion is not necessary), which represents the worst case scenario for DSMIO. On 16 processors, we find that each mutual exclusion lock operation adds an additional 490 microseconds of overhead to our system. We will address this overhead in the next version of DSMIO by differentiating between read and write locks.

VI. Related Work

While there has been a considerable amount of general work on software DSM systems in the recent past and some

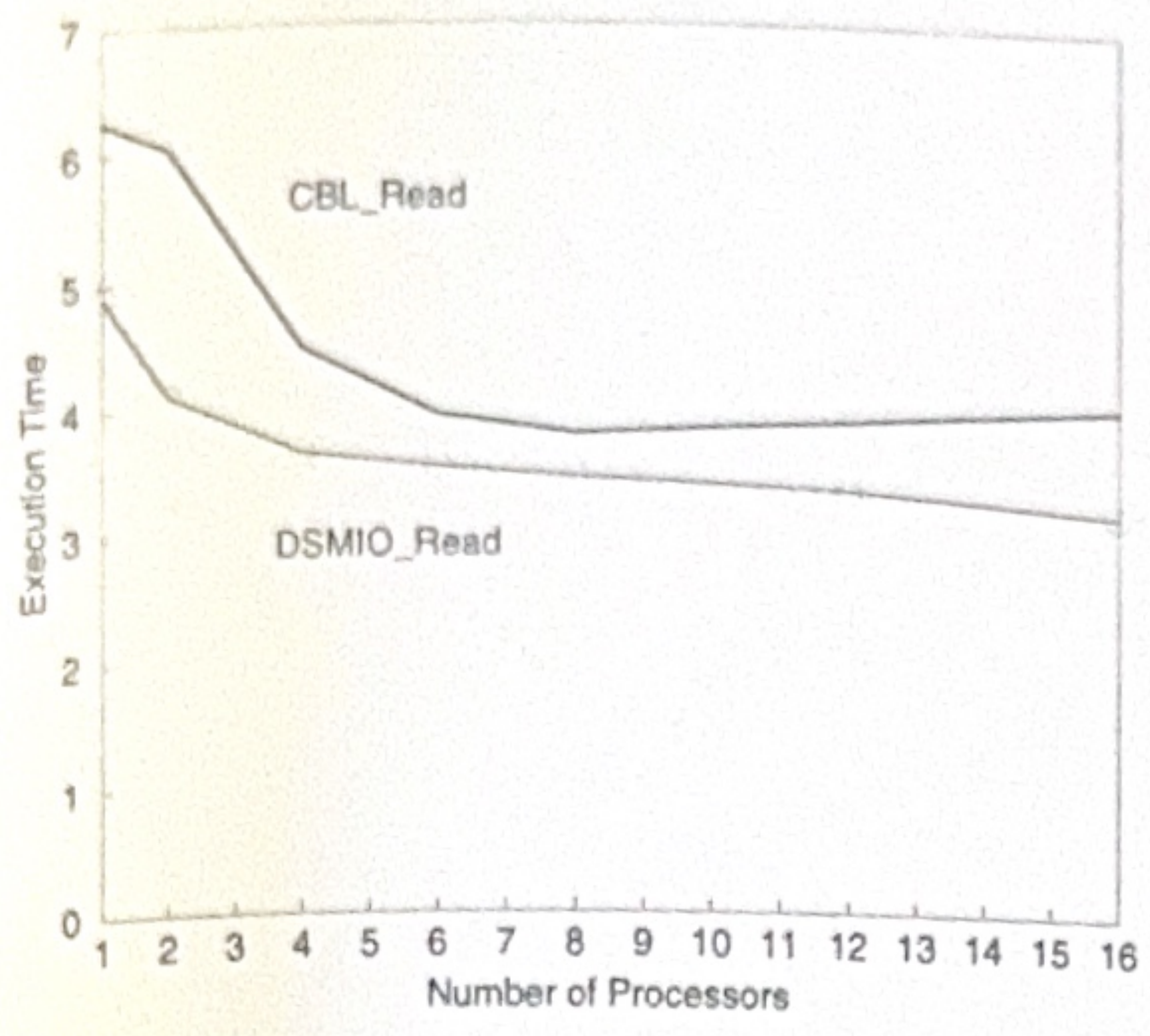

Fig. 4: High Contention Workload Execution Time (secs)

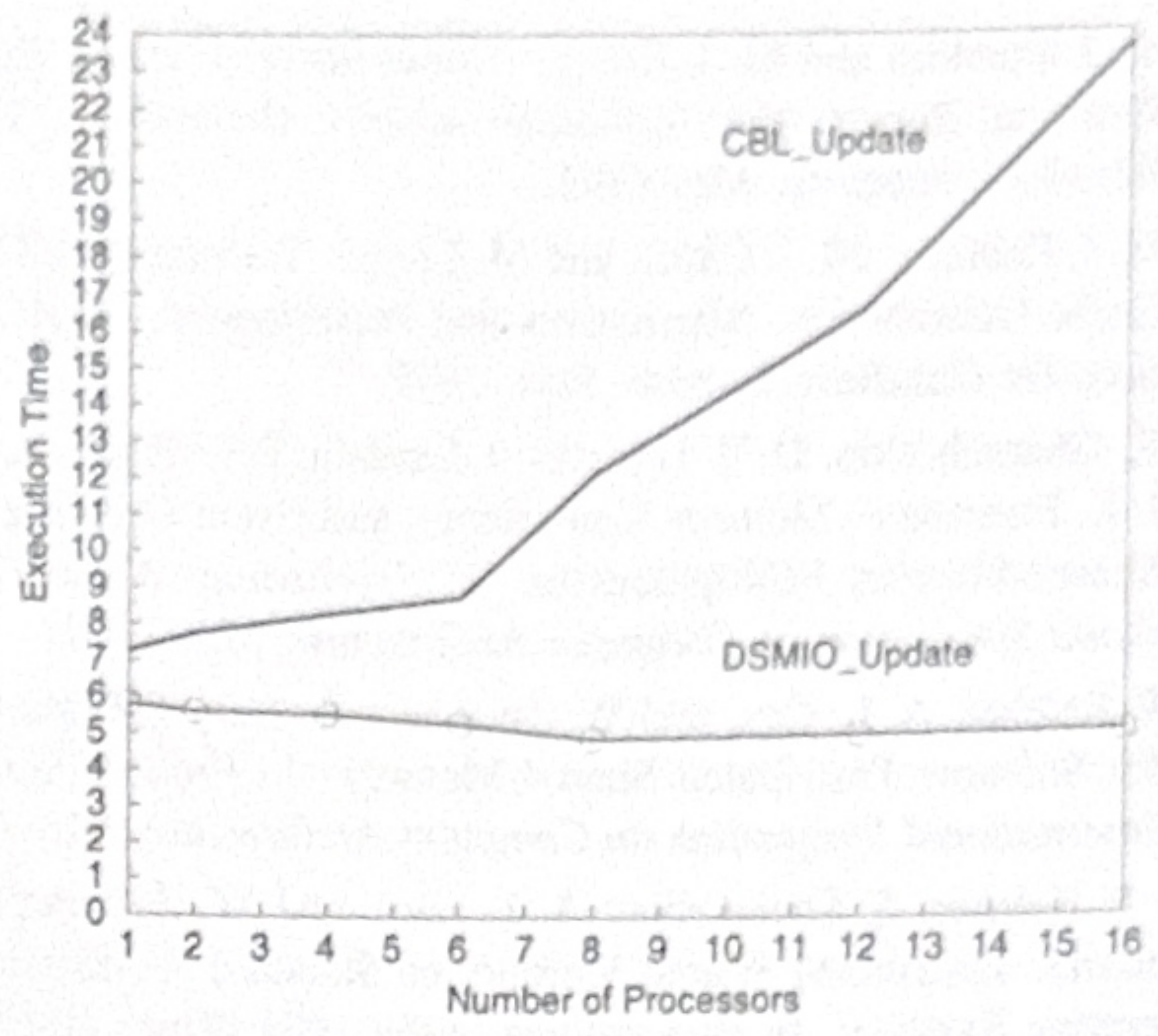

Fig. 5: High Contention Workload Execution Time (secs)

work on database memory performance multiprocessor systems, not much work has been done on running database systems or other applications that make intensive use of operating system functionality on DSM.

The closely related work is a thesis from Rice University [19], in this work TreadMarks is used as a platform for supporting the database Postgres[11]. The work is concerned to research the additional support pieces to handle problems that arose because of the application characteristics, they have no concern on altering application design, therefore the system show slowdown performance.

The work by Scales and Gharachorloo [22] does address many of the issues we dealt. They implement software DSM Shasta [23], a fine-grained memory consistency, at the cache block-level, that request to rewrites hardware multiprocessors binaries. Their goal was also to run a parallel database over a cluster with their DSM. They used the commercial Oracle database to run queries from TPC-D.

An important problem in distributed object and object-relational DBMSs is client cache consistency. Lots of recent work have been proposed on this topic [4, 18, 25]. Their main goal is to decrease false sharing problems throught changing CBL algorithm cache coherence granularity design. As these are simulated work we do not know the real impact on CBL algorithm perfomance, we have plans to add these algorithm extensions on DSMIO in order to study the performance impact.

VII. Conclusion and Future Works

In this paper we evaluated the DSMIO system in the context of a parallel object-based database system where each machine can act both as a client and a server. We used the state-of-the-art CallBack Locking (CBL) cache coherence algorithm as a basis for comparison. Our results show that DSMIO significantly outperforms CBL for workloads with even a moderately high frequency of database updates.

Based on these results, we conclude that DSMIO, and in particular Lazy Release Consistency and multiple writer protocols, indeed improve the performance of parallel database systems.

In the near future, we will address the few limitations of DSMIO that were mentioned throughout the text and will start more explicitly tackling fault tolerance issues. In this context, we will be touching on some of the same issues in Recoverable DSM systems [14]. We will also evaluate DSMIO for larger databases and with more nodes in order to evaluate the system's performance in the face of cache overflow and operating system I/O buffer saturation. Finally, we will compare the DSMIO results to those of CBL-based systems that use a finer coherence unit granularity.

References

[1] S. V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer, December 1996.*

[2] A. Agarwal, R. Bianchini, D. Chaiken, K. 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 Annual International Symposium on Computer Architecture*, pages 2–13, June 1995.

[3] M. J. Carey, D. J. DeWitt, and J. F. Naughton. The 007 Benchmark. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, pp.12-21, 1993.*

[4] M. J. Carey, M. J. Franklin, and M. Zaharioudakis. Fine-Grained Sharing in a Page Server OODBMS. In *Proceedings ACM SIGMOD Conference, 1990.*

[5] R. Cattell The Object Database Standard: ODMG-3.0 In *Morgan Kaufmann Publisher, 2000*

[6] M. J. Franklin and M. J. Carey. Client-Server Caching Revisited. In *Technical Report 1089, Computer Science Department, University of Wisconsin-Madison, May 1992.*

[7] M. J. Franklin, M. J. Carey, and M. Livny. Transactional Client-Server Cache Consistency: Alternatives and Performance. In *ACM Transactions On DataBase Systems, Sept. 1997.*

[8] K. Gharachorloo, D. E. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. L. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessor. In *Proceedings of the 17th International Symposium on Computer Architecture, May 1990.*

[9] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proceedings of the 19th International Symposium on Computer Architecture, May 1992.*

[10] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Treadmarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the 1994 Winter USENIX Conference, January 1994.*

[11] A. Martin PostgresSQL Home Page: http://www.postgressql.org/index.html

[12] M. L. Q. Mattoso. Aspectos de Paralelismo na Gerência de Dados e Objetos no GEOTABA. In *Tese de D.Sc, COPPE/UFRJ, RJ Brasil, 1993.*

[13] L. R. Monnerat and R. Bianchini. Efficiently Adapting to Sharing Patterns in Software DSMs. In *Proceedings of the 4th IEEE Symp. on High-Performance Computer Architecture (HPCA-4), February 1998.*

[14] C. Morin and I. Puaut. A Survey of Recoverable Distributed Shared Virtual Memory Systems. In *IEEE Transactions on Parallel and Distributed Systems, sept.1997.*

[15] C. Osthoff, M. Mattoso, C. Seidel, R. Bianchini and C. L. Amorim. Avaliação do Algoritmo de Coerência de *Cache* de Disco DSMIO

[16] C. Osthoff. Proposta e Avaliação de Mecanismos de *Software* de Memória Compartilhada Distribuída para E/S Paralela. In *Tese de D.Sc, COPPE/UFRJ, RJ Brasil, 2000.*

[17] C. Osthoff, R. Bianchini, M. Mattoso, C. Seidel, and C. L. Amorim. Explorando Conceitos e Mecanismos de Memória Compartilhada Distribuída em E/S Paralela. In *Proceedings of XV Brazilian Symposium on Computer Architecture, October 1999.*

[18] M. Ozsu, K. Voruganti, and R. Unrau. An Asynchronous Avoidance-Based Cache Consistency Algorithm for Client Caching DBMSs. In *Proceedings of the 24th Very Large DataBases Conference, 1998.*

[19] T. Parker, and A. Cox I/O-Oriented Applications on a Software Distributed-Shared Memory System. Master of Science Thesis - Computer Science Dept- Rice university, 1999.

[20] A. Purrakayastha, C. Ellis, D. Kotz, N. Nieuwejaar, and M. Best. Characterizing Parallel File-Access Patterns on a Large-Scale Multiprocessor. In *Proceedings of the International Parallel Processing Symposium, April 1995.*

[21] C. B. Seidel, R. Bianchini, and C. L. Amorim. The Affinity Entry Consistency Protocol. In *Proceedings of the 1997 International Conference on Parallel Processing (ICPP'97), August 1997.*

[22] D. Scales and K. Gharachorloo. Towards transparent and efficient software distributed shared memory In *Proceedings of the 16th ACM Symposium on Operating System Principles,Oct. 1997.*

[23] D.Scales, K. Gharachorloo and C. Thekkath. Shasta: A low overhead,software-only approach for supporting fine-grain shared memory. In *Technical Report 96/2, Western resarch Laboratory, Compaq Corporation,1996.*

[24] O2 TECHNOLOGY In *The O2 System Admnistration Guide,Version 4.5, 1995.*

[25] K. Voruganti, M. Ozsu, and R. Unrau. An Adaptative Hybrid Server Architecture for Client-Caching Object DBMS. In *Proceedings of the 25th Very Large DataBases Conference, 1999.*