Adaptive Conflict Unit Size for Distributed Optimistic Synchronization

Kim-Thomas Rehmann, Marc-Florian Müller, and Michael Schöttner

Heinrich-Heine-Universität Düsseldorf
Universitätsstraße 1, D–40223 Düsseldorf, Germany
kimoe001@uni-duesseldorf.de

Abstract. Distributed and parallel applications often require accessing shared data. Distributed transactional memory is an emerging concept for concurrent shared data access. By using optimistic synchronization, transactional memory is simpler to use and less error-prone than explicit lock-based synchronization. However, distributed transactional memories are particularly sensitive to phenomena such as true sharing and false sharing, which are caused by correlated data access patterns on multiple nodes. In this paper, we propose a transparent technique that adaptively manages conflict unit sizes for distributed optimistic synchronization in order to relieve application developers from reasoning about such sharing phenomena. Experiments with micro-benchmarks and an on-line data processing application similar to Twitter (using the MapReduce computing model) show the benefits of the proposed approach.

1 Introduction

In recent years, numerous data sharing techniques emerged, such as in-memory data grids, cloud storage, and network-attached memory, used by distributed and parallel applications. As opposed to earlier distributed sharing techniques such as file sharing and distributed shared memory, these new techniques aim at being versatile and dynamic while at the same time guaranteeing consistency and reliability. Nonetheless, research on scalable, transparent distributed data sharing to complement existing message passing techniques is still under way.

Sharing techniques benefit from the locality principle [1], which allows to improve data access performance based on correlated access patterns. For example, caching is a special form of replication where a processor keeps data that it has used earlier, thereby exploiting temporal locality. In addition, caches operate on cache lines larger than a single machine word, because processors often use adjacent words together, a phenomenon known as spatial locality. In distributed systems, locality of reference is particularly important. Misprediction or lack of locality cause excessive messaging overhead, which is expensive because of higher communication latencies and, depending on the network infrastructure, reduced network bandwidth.

In order to formalize the principle of locality, researchers have defined the notions *true sharing* and *false sharing* [2]. True sharing is a situation where two or more nodes access the same object using read or write operations. If the object is accessed by read-only operations, replication improves access performance. If one or more nodes modify this object, all other nodes must be notified of these modifications. Coherence protocols such as update-on-write or invalidate-on-write require network communication, if the nodes expect to view the system in a consistent state.

The false sharing phenomenon results from nodes being unable to distinguish object accesses. In order to take advantage of spatial locality and to minimize bookkeeping overhead, some systems aggregate objects into consistency units. For example, the memory management unit (MMU) that virtualizes random access memory detects accesses at the granularity of virtual memory pages. If two or more nodes access indistinguishable but different objects and at least one node modifies a single object, all objects appear to be modified. Obviously, if false sharing accesses occur often, they will slow down applications as much as true sharing does.

As opposed to true sharing, it is possible to avoid false sharing without modifying the application by choosing fine-grained consistency units. However, true sharing and false sharing are time-dependent phenomena. When access patterns change, true sharing can turn into false sharing and vice versa.

The contribution of this paper is an adaptive management concept of consistency unit sizes in the context of a distributed transactional memory (DTM) system. DTM extends optimistic synchronization [3], the idea behind transactional memory (TM) [4,5], to distributed systems such as (federated) clusters [6,7,8,9,10,11]. DTM uses transactions to keep replicas consistent, avoiding complicated lock management and deadlocks. Beyond these benefits, speculative transactions bundle operations, allowing bulk network transfers while at the same time providing strong consistency. False sharing in TM leads to false conflicts, causing unnecessary transaction serializations. In contrast to related work, our approach is transparent for the application programmer. Internally we use larger consistency unit sizes whenever possible to allow bulk network transfers (for efficiency reasons). As soon as false sharing situations are detected during runtime, we transparently reduce the granularity of affected consistency units. If false sharing vanishes, we transparently aggregate smaller consistency units.

The remainder of this paper is structured as follows. Section 2 reviews a static mechanism to avoid false sharing. In Section 3 we present a DTM that adapts its conflict unit sizes to avoid false sharing while supporting spatial locality. In Section 4 we evaluate our dynamic sharing technique. Section 5 discusses related work, and Section 6 concludes with an outlook on further improvements.

2 Static False Sharing Avoidance

Distributed systems that guarantee consistency of shared objects must control accesses to objects. The granularity of access detection influences the performance of distributed and parallel applications. On the one hand, coarse-granular access detection allows bulk network transfers improving performance but may run into false sharing situations drastically degrading performance. On the other hand, fine-granular access detection is not prone to false sharing, but it does not support spatial locality, and it may incur additional run-time overhead.

We classify access detection mechanisms as either being object, attribute or page based. Object-based access detection eliminates false sharing among different objects [12], but it requires either annotations by the programmer or instrumentation by a compiler. For example, the distributed system might rely on applications to notify object accesses by means of operations such as openObject [13]. At a finer granularity, attribute-based access detection completely avoids false sharing. However, allowing applications to modify different attributes of the same objects concurrently is counter-intuitive. In order to detect object accesses transparently with respect to both application and programmer, page-based access detection uses memory protection mechanisms built in hardware.

2.1 Page-Based Access Detection

We have decided to base our implementation of a DTM on page-based access detection for several reasons. First, using pages as consistency units benefits from locality of access and allows bulk network transfers. Second, although the MMU detects object accesses during address translation at hardware level, a user-level library can conveniently control access detection. Therefore, page-based access detection neither depends on the language in which the application was written, nor does it require any special markup for accesses. If a page lacks the requested access privilege, the MMU generates an exception, which the user-level library handles. Furthermore, in a distributed system, high communication latencies mitigate the overhead for local page-based access detection. Third, page-based access detection integrates well with transaction semantics, as we will discuss in Section 3.2.

Although most modern processors support multiple page sizes (e.g. 4 KB and 2 MB respective 4 MB on x86 processors), operating systems usually do not allow applications to select the hardware page size. If objects are smaller than the page size, page-based access detection is prone to false sharing, because accesses to different objects on the same virtual page cannot be distinguished. Page diffing [14] permits locating write accesses at byte granularity. Writable implies readable on x86 processors, such that diffing cannot preclude false sharing, unless it reveals that a page has not been modified at all.

A primitive approach to counteract false sharing would be to allocate objects sparsely in the virtual address space. However, placing each object in a distinct consistency unit trades exact access detection in for internal fragmentation. Although modern machines usually have plenty of physical memory available, internal fragmentation can increase memory consumption by a factor of thousand in extreme cases, for example when wasting a 4 KB page for a 4 Byte object. Moreover, padding irrevocably eliminates the potential benefits of spatial locality. Even worse, the approach cannot adapt to different object usage patterns.

2.2 Multiview/Millipage Address Space Layout

The Multiview/Millipage approach proposed by Itzkovitz and Schuster [15] constructs special virtual-to-physical mappings, allowing for access detection at object granularity, nonetheless avoiding internal fragmentation as with primitive false sharing avoidance. A Millipage region divides a physical page frame into 2^n disjoint Millipages. If the hardware page size is 2^p, one Millipage covers 2^{p-n} bytes. Each Millipage has a distinct mapping in the virtual address space, such that accesses to objects that reside on the same physical page frame are detected independently. A privileged mapping allows to circumvent access detection, thereby enabling atomic updates in multithreaded applications. Figure 1 illustrates Millipage layout with two Millipages per physical page frame.

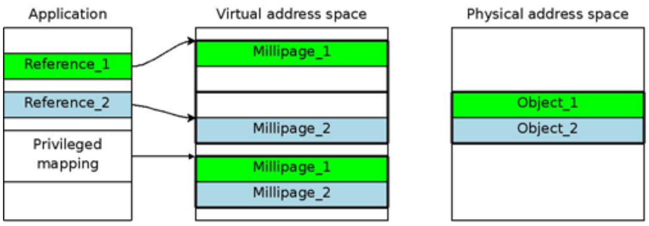

Fig. 1. Millipage mappings ($n = 2$)

In our implementation of Multiview, one region's Millipages reside in consecutive virtual memory pages, such that the region spans a range of 2^{np} bytes in which only a total of 2^p bytes belongs to valid Millipages. This simple convention about memory layout enables cheap access validation: An address's offset in the virtual page must match the page's index in its Millipage region, e.g. the application must reference the first object only through the first virtual page, the second object only through the second virtual page and so on. Failure of access validation indicates a corrupt memory pointer.

The Multiview technique effectively decouples page size and consistency unit size. If each consistency unit holds at most one object, every access uniquely identifies a single object. Therefore, Multiview completely avoids false sharing.

2.3 Handling Huge Objects

The Multiview approach applies in case the consistency unit size is smaller than the page size. In contrast, some objects may be larger than page size, such that these objects must be allocated on consecutive full pages. False sharing cannot occur in these cases, set aside false sharing between attributes of the same object. Nevertheless, it is interesting to note that there are two modes how a sharing service can synchronize access to the pages a large object consists of. Either the service synchronizes each page in stand-alone manner, that is, it allows parts of the object to be modified concurrently, or it handles the object as an aggregation of pages, that is, it automatically ascribes access to all pages simultaneously. If the operating system supports different page sizes in user-space, the sharing service can allocate huge objects on larger pages to improve the performance of access detection. Our DTM supports concurrent modification of object attributes.

3 Dynamic Conflict Size Management

The transaction concept allows concurrent activities to access multiple objects with implicit synchronization. A transaction bundles several read and write operations. Well-known in the context of databases, the so-called ACID properties [16] guarantee atomicity, consistency, isolation and durability of transactions. To ensure atomicity and isolation, transactions execute speculatively. After speculative execution, a subsequent validation phase ensures that a transaction only commits its modifications if speculative execution did not violate the consistency requirement. Durability of database transactions asserts that committed transactions cannot be undone.

Transactional memory (TM) applies the transaction concept to in-memory data [17,5]. By integrating application logic and data access mechanism, TM avoids any potential database overhead and enables specific optimizations. For example, TM relaxes the durability of transactions, such that distributed state need not be written to disk.

The object sharing service (OSS) [18] implements a DTM for XtreemOS [19] but also runs on any x86-based Linux system. The OSS provides shared objects as ranges in virtual memory, such that references to objects are simply memory pointers. For heterogenous setups we plan a pointer swizzling technique like for example implemented in Interweave [20]. Every TM requires application developers to define transaction boundaries by identifying code sections that access shared data concurrently. In the OSS, begin and end of transactions are specified by calling the corresponding library functions.

In order to simplify application development, the OSS controls read and write operations transparently by using page-based access detection. As discussed in the introduction, a consistency unit size of one page (4 KB) is prone to false sharing. In a DTM system, false sharing causes false conflicts between transactions, leading to unnecessary transaction aborts.

The adaptive conflict unit size management we propose in this section is flexible and transparent for the application programmer, relieving him of reasoning about data allocation and memory layouts causing false sharing situations. By providing an adaptive approach, we can support large consistency units and bulk network transfers whenever possible, but we can switch to a fine-grained Multiview consistency unit management in case false sharing shows up.

3.1 Page-Based Access Detection for Transactional Memory

At the beginning of a transaction, the transaction management requests access notification for all objects by revoking read and write permission for the corresponding virtual memory pages. The first read operation to a page causes Linux to report the object's address to the OSS, which in turn inserts the address into the transaction's read set and grants access to the corresponding page. The first write operation proceeds in a similar manner. In addition, the OSS creates a shadow copy containing the page's original content, such that transaction management can restore the page in case the transaction cannot commit and must restart. Repeated read or write accesses to a page within the same transaction are not monitored and can run without any overhead. However, the first write operation on a previously read page causes a shadow copy to be created and moves the address from the read set to the write set.

3.2 Integration of Multiview into Transactional Memory on Linux

Our OSS supports using Millipages of different granularities and full (non-Millipage) pages side-by-side. When allocating an object, the OSS automatically chooses the Millipage granularity coarse enough to hold the object. The Millipage granularity is stored in the virtual page's attributes that are themselves distributed objects with fixed granularity.

The Multiview allocation scheme and the privileged mapping require multiple mappings of the same memory segment. Therefore, the OSS constructs memory mappings using System V shared memory segments, which can be attached repeatedly to a single address space.

We have identified several synergies between memory transactions and the Multiview approach. First, Multiview restrains object size for read and write accesses, such that false aborts are eliminated. Second, Multiview speeds up shadow copy operations. When creating a shadow copy for a Millipage, the OSS needs to backup only a fraction of a full page, at most one physical page frame for an entire Millipage region. Similarly, Multiview restrains the range to compare for diff generation. Third, the privileged mapping allows transactions to run multithreaded in the same process. Otherwise, during non-atomic updates, all of a process's threads would have to be halted.

3.3 Monitoring of Object Accesses

The Multiview technique completely avoids false conflicts if objects are always allocated on distinct pages, but degrades performance for access patterns that do not cause conflicts but could benefit from spatial locality. Moreover, transaction conflicts are dynamic phenomena, which depend on object access patterns. Consequently, we have implemented an access detection technique that dynamically adapts to the degree of false conflicts.

Our technique monitors object accesses to determine whether Millipages should be handled seperately or conjointly. In the context of TM, we designate consistency units as conflict units. A Millipage region that serves as coarse conflict unit is called object access group (OAG).

To avoid exponential state-keeping and limit memory overhead, the monitoring mechanism considers only objects located in the same Millipage region. These objects have been allocated by the same node during some time interval, such that a semantical relationship among these objects is likely. Furthermore, a single system call can set the access protection for a contiguous region of virtual memory, such that using OAGs does not increase the number of costly switches between user and kernel mode.

The dynamic adaptation mechanism bases its decisions only on local information in order to avoid network communication. Each node receives write sets from other committing nodes. Nodes need not transmit read sets, because remote read operations are not relevant to identify false conflicts, given that transactions in the OSS commit using a first-wins strategy.

During the validation phase, transaction management determines whether a transaction conflicts with already committed transactions. In addition, for non-aggregated Millipage regions, our transaction management calculates whether OAGs would have caused hypothetic false conflicts.

3.4 Adapting Sharing Granularity

When aggregating objects into OAGs, it may happen that some objects in the group are not accessed during a transaction. Thus, the transaction's read or write set might contain false positives. For objects in the write set, generating a diff between the actual object and its shadow copy reveals whether the object has been modified. Given that writable implies readable on our target architecture x86, transaction management must not ignore unmodified objects, but it can relocate them from the write set to the read set. For objects in the read set, it is impossible to detect whether they have actually been accessed in the transaction. As a consequence, false positives in read sets increase the probability of false transaction aborts but do not cause inconsistencies.

The dynamic adaptation mechanism handles both the aggregation of objects to OAGs and the division of OAGs to objects with individual access detection. A sharing situation with spatial locality among objects in a Millipage region is characterized by few hypothetic conflicts. If hypothetic conflicts are rare and a read set contains several objects from the same region, the adaptation mechanism combines the Millipages into an OAG. To avoid oscillation, OAGs are formed no sooner than several transactions after splitting the region. We determined empirically that a reasonable stabilization interval is equal to the number of Millipages in the region. An OAG that causes a conflict during validation is subject to false conflicts or even true conflicts. Thus, the adaptation mechanism splits the OAG immediately.

3.5 Hints for the Application Developer

Monitoring of object accesses also assists the developer in identifying those objects that frequently cause true aborts. Conflict rates are aggregated among all participating nodes and published in the built-in name service, including information about which node and which function created the object. The developer can extract true sharing hotspots from the name service either periodically or manually, for example before terminating the application.

4 Performance Evaluation

To evaluate the performance of our adaptive sharing technique, we have run micro-benchmarks under different sharing and allocation strategies. An on-line data processing application demonstrates that transactional memory benefits from adaptive sharing for realistic workloads. We ran our experiments on dual-core nodes equipped with AMD Opteron 244 processors running at 1.8 GHz under Linux 2.6.26. The nodes were connected via Gigabit Ethernet over Broadcom NetXtreme NICs.

4.1 Micro-Benchmarks

We have run synthetic workloads with four different allocation schemes. The dlmalloc allocator is a general-purpose allocator, similar to the one used by the GNU standard C library. We use its MSpaces variant, which enables multi-threaded allocations of transactional memory. MSpaces is quite space-efficient but prone to false sharing. The Page allocator places each object in a separate physical page frame. It implements the primitive approach against false sharing and causes internal fragmentation for object sizes that are not a multiple of page size. The Millipage allocator statically places all objects on Millipages and does not aggregate objects. The Adaptive allocator is based on the Millipage allocator and implements adaptive sharing based on OAGs. To express an allocation scheme's reaction on an access pattern, we have measured the number of detected accesses.

For the first test, the setup consists of two nodes accessing two objects that have been allocated consecutively. The examined node reads both objects, the second node writes to one object, causing frequent transaction restarts on the examined node. A simple fairness strategy in OSS ensures that a transaction will commit after restarting once. Figure 2 impressively demonstrates that the MSpaces allocator is susceptible to false sharing, whereas the other allocators enable to distinguish both objects.

In the second test, a single node accesses two objects conjointly in a loop of 2^{16} transactions. The Page and Millipage allocator detect each access separately, as depicted in Figure 3. The MSpaces and Adaptive allocator only detect one access per transaction because of spatial locality between the objects.

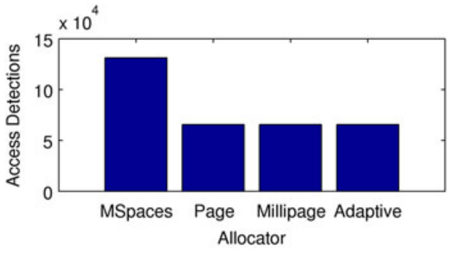

Fig. 2. Access detections induced by accessing distinct objects

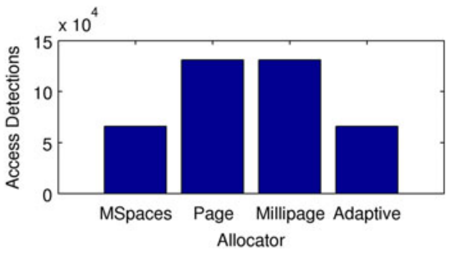

Fig. 3. Access detections induced by accessing objects conjointly

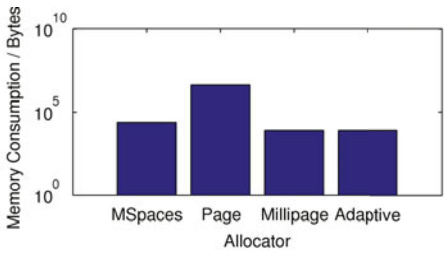

Fig. 4. Memory consumption (logarithmic scale)

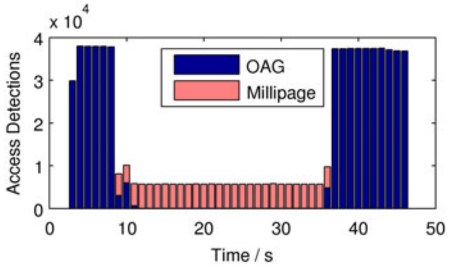

Fig. 5. Dynamic adaptation from OAGs to Millipages and vice versa

For a synthetic workload with 1024 4-Byte objects, the memory consumption of the Page allocator is severe, whereas the other allocators allocate only the requested object size plus some allocation meta-data (see Figure 4).

We have also evaluated how well our technique adapts to varying object access patterns. The setup consists of two nodes, one of which is running transactions in a loop for 2^{20} times, reading from two objects. The other starts up about six seconds later, runs transactions in a loop for 2^{19} times, writing to one of the objects. Initially, the second node does not run transactions at full speed, which causes the first node to switch several times between coarse-granular and fine-granular access detection. Figure 5 subdivides the number of access detections for OAGs and for Millipages.

4.2 MapReduce

MapReduce [21] is a computing model for processing large amounts of data. The model applies to certain problems where mapping the input data to a different domain allows highly parallelized computations. Being easy to understand, MapReduce has reached widespread use. For example, Google uses MapReduce for different search and extraction problems in more than 4000 applications.

The common introductory example for MapReduce is word frequency analysis. For determining the frequency of words in an input text, the map phase emits each individual word with a count of 1. The reduce phase afterwards collects all identical words from map phase's output and sums up their counts, yielding the total count per word.

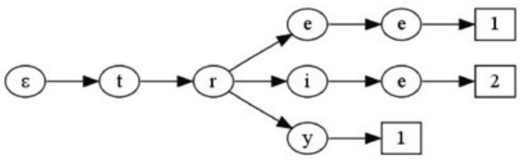

Fig. 6. Example trie storing the frequency of the words *tree, trie (2) and try*

The word frequency problem exemplifies that MapReduce is well-suited for analyzing static data. Dynamic, interactive information sharing, such as Web 2.0 applications that are currently emerging, needs processing facilities for continuous data streams. To achieve good scalability, continuous data should be processed in parallel. As a consequence, the computing model must efficiently support concurrent access to shared data.

We have applied MapReduce to processing of continuous data streams. Our implementation bases on our DTM. Extending the word frequency example, we illustrate the effectivity of adaptive conflict size management with continuous analysis of text data streams, using a scenario resembling the well-known Web 2.0 application Twitter.

The continous word counting example operates on a trie [22] where each word is represented by a path from the tree's root to a node. The node at the end of a word stores the frequency of the word it terminates, possibly other statistical information such as time stamps too. Intermediate nodes represent prefixes of a word, storing at most 26 references to next prefix characters (see Figure 6). In our implementation, each node has a size of 216 Bytes, which equals 26 references to child nodes plus a 64-Bit counter. When allocating nodes for the trie, the Adaptive allocator splits a physical page frame in 16 Millipages, each 256 Bytes large, causing 16% internal fragmentation.

The trie representation of words already counteracts false sharing by enforcing a high fan-out, e.g. compared to a representation of words in a binary tree. Our implementation serves back-to-back allocations from the same Millipage region, if space allows so. Therefore, nodes tend to reside in the same region as their ancestors and descendants, such that grouping adjacent objects makes sense.

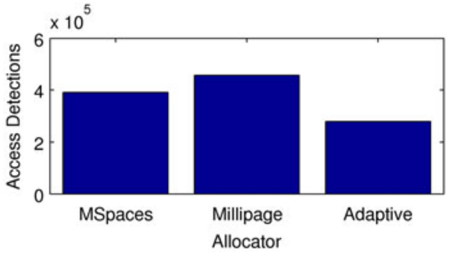

Fig. 7. Access detections for word frequency analysis

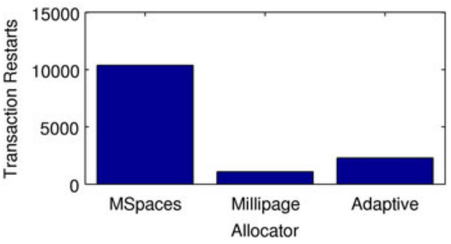

Fig. 8. Transaction aborts for word frequency analysis

In our experiment, we connected two nodes using our DTM. Each node simulated a user who entered some text (the novel *Kim* written by Rudyard Kipling). The text consisted of 107585 words in total, thereof 10636 different words. Again, we measured the number of access detections representing how well an allocator makes use of locality (see Figure 7). Additionally, we determined the number of transaction restarts, which indicates how much access detection suffers from false sharing (see Figure 8). Our adaptive access detection mechanism triggers only 60% access detections compared to the Millipage allocator, and it causes less than 25% transaction restarts compared to the MSpaces allocator.

5 Related Work

The implications of sharing phenomena and their interdependencies have been discussed mainly in the context of caching hardware and distributed shared memory (DSM). Several consistency models that take account of sharing have been defined, for example scope consistency [23] and view-based consistency [24]. These models provide weaker consistency than TM. The Region-trap library [25] combines pointer swizzling and virtual memory protection to trap accesses to individual objects, requiring region pointer annotation. Amza et al. [26] describe the dynamic aggregation of pages for lazy release consistency [14]. Our work has some similarities with ComposedView [27]. ComposedView provides transparent aggregation of small consistency units for sequential consistency, but to our knowledge the technique has not been applied to TM yet.

The impact of false sharing on TM has been discussed recently, for example in the VELOX project [28]. Burcea et al. [29] propose to vary access tracking granularity. In contrast to our approach, the authors focus on per-object granularity that does not adapt dynamically to access patterns. Bocchino et al. [7] implement a DTM for large-scale clusters. They define eight design dimensions for their TM, one dimension is the static size of conflict detection units.

6 Conclusion

We have presented an approach for the adaptive management of conflict unit sizes for a distributed transactional memory system. The combination of a smart allocation strategy and transparent access monitoring avoids false sharing and thus unnecessary transaction aborts caused by false conflicts. At the same time we support locality whenever possible, allowing bulk network transfers to speed up distributed processing. The proposed solution is transparent for the application programmer and is able to adapt its strategy to changing access patterns.

The evaluation using micro-benchmarks and a MapReduce application demonstrate the benefits of the adaptive conflict unit size management while at the same time introducing only minimal overhead.

Clearly, transactional memory is attracting a lot of people in research and industry, and recently some of these ideas have also shifted to distributed systems. Therefore, we expect more and more transactional applications, also for distributed environments.

We plan to study more flexible object access groups containing objects from different Millipage regions. In this context, Bloom filters [30] are a promising data structure for statistical monitoring of large data sets. Finally, we have started with large-scale experiments on the Aladdin-Grid'5000 platform.

References

1. Denning, P.J., Schwartz, S.C.: Properties of the working-set model. ACM Commun. 15(3), 191–198 (1972)
2. Torrellas, J., Lam, M.S., Hennessy, J.L.: False sharing ans spatial locality in multiprocessor caches. IEEE Trans. Computers 43(6), 651–663 (1994)
3. Kung, H.T., Robinson, J.T.: On optimistic methods for concurrency control. ACM Trans. Database Syst. 6(2), 213–226 (1981)
4. Herlihy, M., Moss, J.E.B.: Transactional memory: architectural support for lock-free data structures. SIGARCH Comput. Archit. News 21(2), 289–300 (1993)
5. Felber, P., Fetzer, C., Guerraoui, R., Harris, T.: Transactions are back—but are they the same? SIGACT News 39(1), 48–58 (2008)
6. Kotselidis, C., Ansari, M., Jarvis, K., Luján, M., Kirkham, C., Watson, I.: DiSTM: A software transactional memory framework for clusters. In: ICPP 2008: Proceedings of the 37th IEEE International Conference on Parallel Processing, September 2008, IEEE Computer Society Press, Los Alamitos (2008)
7. Bocchino, R.L., Adve, V.S., Chamberlain, B.L.: Software transactional memory for large scale clusters. In: PPoPP 2008: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, pp. 247–258. ACM, New York (2008)
8. Herlihy, M., Sun, Y.: Distributed transactional memory for metric-space networks. Distributed Computing 20(3), 195–208 (2007)
9. Manassiev, K., Mihailescu, M., Amza, C.: Exploiting distributed version concurrency in a transactional memory cluster. In: PPoPP 2006: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming, pp. 198–208. ACM, New York (2006)
10. Romano, P., Carvalho, N., Rodrigues, L.: Towards distributed software transactional memory systems. In: LADIS 2008: Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware, pp. 1–4. ACM, New York (2008)
11. Couceiro, M., Romano, P., Carvalho, N., Rodrigues, L.: D2STM: Dependable distributed software transactional memory. In: PRDC 2009: Proc. 15th Pacific Rim International Symposium on Dependable Computing (November 2009)
12. Bal, H.E., Bhoedjang, R., Hofman, R., Jacobs, C., Langendoen, K., Rühl, T., Kaashoek, M.F.: Performance evaluation of the Orca shared-object system. ACM Trans. Comput. Syst. 16(1), 1–40 (1998)
13. Herlihy, M., Luchangco, V., Moir, M., Scherer, I.W.N.: Software transactional memory for dynamic-sized data structures. In: PODC 2003: Proceedings of the twenty-second annual symposium on Principles of distributed computing, pp. 92–101. ACM, New York (2003)
14. Keleher, P., Cox, A.L., Zwaenepoel, W.: Lazy release consistency for software distributed shared memory. In: Proc. of the 19th Annual Int'l Symp. on Computer Architecture (ISCA 1992), pp. 13–21 (1992)

15. Itzkovitz, A., Schuster, A.: MultiView and Millipage – fine-grain sharing in page-based DSMs. In: OSDI 1999: Proceedings of the third symposium on Operating systems design and implementation, pp. 215–228. USENIX Association, Berkeley (1999)
16. Haerder, T., Reuter, A.: Principles of transaction-oriented database recovery. ACM Comput. Surv. 15(4), 287–317 (1983)
17. Dias, R.J., Lourenço, J.M.: Unifying memory and database transactions. In: Sips, H., Epema, D., Lin, H.-X. (eds.) Euro-Par 2009 Parallel Processing. LNCS, vol. 5704, pp. 349–360. Springer, Heidelberg (2009)
18. Müller, M.F., Möller, K.T., Sonnenfroh, M., Schöttner, M.: Transactional data sharing in grids. In: PDCS 2008: Proceedings of the International Conference on Parallel and Distributed Computing and Systems (2008)
19. Christine, M.: XtreemOS: A Grid operating system making your computer ready for participating in virtual organizations. In: ISORC 2007: Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing, Washington, DC, USA, pp. 393–402. IEEE Computer Society, Los Alamitos (2007)
20. Chen, D., Dwarkadas, S., Parthasarathy, S., Pinheiro, E., Scott, M.L.: Interweave: A middleware system for distributed shared state. In: Languages, Compilers, and Run-Time Systems for Scalable Computers, pp. 207–220 (2000)
21. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. ACM Commun. 51(1), 107–113 (2008)
22. Knuth, D.E.: The art of computer programming. In: sorting and searching, 2nd edn., vol. 3. Addison Wesley Longman Publishing Co., Inc., Redwood City (1998)
23. Iftode, L., Singh, J.P., Li, K.: Scope consistency: a bridge between release consistency and entry consistency. In: SPAA 1996: Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures, pp. 277–287. ACM, New York (1996)
24. Huang, Z., Sun, C., Purvis, M., Cranefield, S.: View-based consistency and false sharing effect in distributed shared memory. SIGOPS Oper. Syst. Rev. 35(2), 51–60 (2001)
25. Brecht, T., Sandhu, H.: The region trap library: handling traps on application-defined regions of memory. In: ATEC 1999: Proceedings of the annual conference on USENIX Annual Technical Conference, p. 7. USENIX Association, Berkeley (1999)
26. Amza, C., Cox, A., Rajamani, K., Zwaenepoel, W.: Tradeoffs between false sharing and aggregation in software distributed shared memory. In: PPOPP 1997: Proceedings of the sixth ACM SIGPLAN symposium on Principles and practice of parallel programming, pp. 90–99. ACM, New York (1997)
27. Niv, N., Schuster, A.: Transparent adaptation of sharing granularity in MultiView-based DSM systems. Softw. Pract. Exper. 31(15), 1439–1459 (2001)
28. Harmanci, D., Felber, P., Gramoli, V., Fetzer, C.: TMUNIT: Testing transactional memories. In: TRANSACT 2009: 4th Workshop on Transactional Computing, Feburary (2009)
29. Burcea, M., Steffan, J.G., Amza, C.: The potential for variable-granularity access tracking for optimistic parallelism. In: MSPC 2008: Proceedings of the 2008 ACM SIGPLAN workshop on Memory systems performance and correctness, pp. 11–15. ACM, New York (2008)
30. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. ACM Commun. 13(7), 422–426 (1970)