Adaptive DSM-Runtime Behavior via Speculative Data Distribution

Frank Mueller

Humboldt-Universität zu Berlin, Institut für Informatik, 10099 Berlin (Germany)
e-mail: mueller@informatik.hu-berlin.de *phone: (+49) (30) 2093-3011, fax:-3011*

Abstract. Traditional distributed shared memory systems (DSM) suffer from the overhead of communication latencies and data fault handling. This work explores the benefits of reducing these overheads and evaluating the savings. Data fault handling can be reduced by employing a lock-oriented consistency protocol that synchronizes data when acquiring and releasing a lock, which eliminates subsequent faults on data accesses. Communication latencies can be reduced by speculatively replicating or moving data before receiving a request for such an action. Histories of DSM accesses are used to anticipate future data requests.
DSM-Threads implement an adaptive runtime history that speculatively distributes data based on access histories. These histories are kept locally on a node and are distinguished by each thread on a node. When regular access patterns on requests are recognized, the data will be speculatively distributed at the closest prior release of a lock. The benefits of this active data distribution and its potential for false speculation are evaluated for two consistency protocols. The data-oriented entry consistency protocol (Bershad *at. al.*) allows an even earlier data distribution at the closest prior release on the **same** lock. A simple multi-reader single-writer protocol, an improvement of Li/Hudak's work, uses the closest prior release of a speculatively **associated** lock for speculative data distribution and also eliminates data faults. In addition, a framework for speculative release consistency is formulated.

1 Introduction

In recent years, distributed systems have enjoyed increasing popularity. They provide a cost-effective means to exploit the existing processing power of networks of workstations. Shared-memory multi-processors (SMPs), on the other hand, do not scale well beyond a relatively small number of processors (typically no more than 40 CPUs) since the system bus becomes a bottleneck. Distributed systems have the potential to scale to a much larger number of processors if decentralized algorithms are employed that cater to the communication medium and topology. However, distributed systems may cause a number of problems.

First, the communication medium between processing nodes (workstations) often poses a bottleneck, although recent increases in throughput in FDDI, FastEther, ATM and SCI may alleviate the problem in part. Also, communication latencies are not reduced quite as significantly. Latency strongly impacts distributed systems since communication generally occurs as a result of a demand for resources from a remote node, and the node may idle until the request is served. In general, distributed applications must restrict communication to the absolute minimum to be competitive.

Another problem is rooted in the absence of a global state within a distributed system. Conventional techniques for concurrent programming cannot be applied anymore. Instead, distributed algorithms have to be developed based on explicit communication (message passing). But either such approaches use a centralized approach with servers as a potential bottleneck or decentralized approaches require new algorithms or extensive program restructuring. As a result, distributed algorithms are often hard to understand.

An alternative solution is provided by *distributed shared memory (DSM)* [13, 14]. DSM provides the logical view of a portion of an address space, which is shared between physically distributed processing nodes, *i.e.,* a global state. Address references can be distinguished between local memory accesses and DSM accesses, *i.e.,* an architecture with non-uniform memory access (NUMA) is created. In addition, the well-understood paradigm of concurrent programming can be used without any changes since the communication between nodes is transparent to the programmer. Notice that NUMA is experiencing a renaissance on the hardware level as well to cope with scalability problems of SMPs.

DSM systems have been studied for over a decade now. A number of consistency models have been proposed to reduce the amount of required messages at the expense of additional requirements that a programmer has to be aware of. Sequential consistency still provides the conventional programming of SMPs but has its limitations in terms of efficiency for DSM systems. It has been commonly implemented using the dynamic distributed manager paradigm at the granularity of pages [7]. We will refer to this implementation as sequential consistency for DSM. Entry consistency requires explicit association between locks and data that is either provided by a compiler or by the programmer with a granularity of arbitrary sized objects [1]. Release consistency is a weak model that allows multiple writes to a page at the same time, as long as the programmer ensures that the writes occur to distinct parts of the page between nodes. Implementations include *eager* release consistency where data is broadcasted at lock releases and *lazy* release consistency where data is requested upon demand when an attempt to acquire a lock is made [6]. These are the most common consistency models.

This paper explores new directions by adapting consistency models for speculative movement of resources. This approach should address two problems of DSM systems. First, access faults, required to trigger data requests, come at the price of interrupts. The increasing complexity in architecture, such as instruction-level parallelism, has a negative impact on the overhead of page faults since all pipelines need to be flushed. The increased complexity translates into additional overhead that may have been negligible in the past. Thus, avoiding page faults may become more important. Speculative data distribution can eliminate most of these page faults. Second, the number of messages in a DSM system often creates a bottleneck for the overall performance. By forwarding data speculatively ahead of any requests, the latency for data requests can be reduced and messages may be avoided.

The scope of this paper is limited to software DSM systems targeted at networks of workstations. The models for speculative data distribution are developed and implemented under *DSM-Threads* [9], a DSM system utilizing an interface similar to POSIX Threads (Pthreads) [19]. The aim of DSM-Threads is to provide an easy way for a programmer to migrate from a concurrent programming model with shared memory (Pthreads) to a distributed model with minimal changes of the application code. A major goal of DSM-Threads is the portability of the system itself followed by a requirement for decentralized runtime components and performance issues. The runtime system of DSM-Threads is implemented as a multi-threaded system over Pthreads on each node and copes without compiler or operating system modifications. Several data consistency models are supported to facilitate ports from Pthreads to DSM-Threads and address the requirements of advanced concepts for multiple writers. Heterogeneous systems are supported by handling different data representations at the communication level. The system assumes a point-to-point message-passing protocol that can be adapted to different communication media. It assumes that messages are slow compared to computation and that the size of messages does not have a major impact on message latencies.

The paper is structured as follows. After a discussion about related work, the basic concept of speculative data distribution are developed. Following this discussion, three specific models of speculation are proposed for common consistency models, namely speculative entry consistency, speculative sequential consistency and speculative release consistency. These models rely on the recognition of access pattern, as considered next. Thereafter, measurements within the DSM-Threads environment are presented and interpreted for speculative locking and speculative data distribution. After an outlook at future work, the work is summarized in the conclusions.

2 Related Work

Software DSM system have been realized for a number of consistency models mentioned before. Nevertheless, the support for speculative data distribution in the context of DSM has only recently been studied. Implementations of sequential consistency, such as by Li and Hudak [7], do not mention speculative approaches. The topic is neither been covered by the original work on entry consistency [1] nor for the associated lock protocols [12]. However, Keleher *et. al.* developed and evaluated a hybrid model as a compromise between eager and lazy release consistency [6]. Data is associated with locks based on heuristics and is speculatively forwarded to a growing number of interested nodes upon lock release. This model differs from our approach in that the latter goes beyond data speculation. We also consider speculative forwarding of lock ownership, varying data access patterns, on-line assessment of the benefits of speculation and suspension and resumption of speculation mechanisms. Recently, Karlsson and Stenström [4] as well as Bianchini *et. al.* [2] proposed prefetching schemes using histories of length two and sequential prefetching but still take page faults on already prefetched pages to ensure that prefetched data will be used. They report speedups ranging between 34% and slight performance degradations depending on the applications but most tested programs show good improvement. These studies differ from our approach since we extend speculation to the synchronization protocol, avoid page faults, incorporate asynchronous message delivery and support sequential and entry consistency. Schuster and Shalev [16] use access histories to steer migration of threads, which shows the range of applications for access histories. While access patterns have also been studied in the context of parallel languages, such as in HPF [15] or filaments [8], such systems rely on compiler support to generate code for data prefetching at user-specified directives. The work described in this paper aims at systems without compiler modifications. Previous work on branch prediction [17, 18] is closely related to the recognition of access patterns proposed in this work.

3 Speculative Data Distribution

Common consistency models for distributed shared memory systems are based on demand-driven protocols. In other words, if data is not accessible locally, it is explicitly requested from a remote node. Speculative data distribution, on the other hand, anticipates future access patterns based on the recorded history of access faults. This is depicted abstractly in Figure 1 where an multiple writes to data x are interleaved between nodes A and B. Instead of serving requests on demand by sending data, the data is forwarded to the node most likely to be accessing it ahead of any requests (see second write to x on node B). If this speculation of access sequences correctly anticipates the application behavior, then considerable performance savings will materialize given that both communication and interrupt overhead can be reduced. Should, however, the speculation not represent the actual program behavior, then the overhead of speculative movements adversely affects performance. Hence, there is a clear trade-off between the number of beneficial and the number of false speculations that can be tolerated.

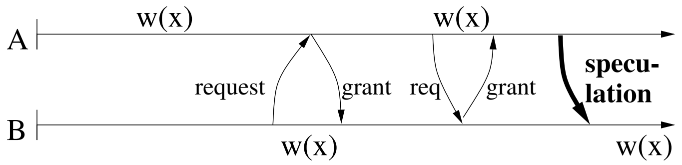

Fig. 1. Abstract Speculation Model for Access Histories

The problem of additional overhead of false speculations can be addressed by various means. One may require a later response from a destination node to confirm that speculative movement was beneficial. In order to keep communication overhead low, this confirmation should be piggybacked with the lock distribution as depicted in Figure 2. When the original node acquires the lock again, it will then be able to evaluate the effect of its previous speculation. Upon realizing that false speculation had occurred, future speculative movements can be inhibited. This confirmation protocol should be very effective since it does not result in any additional messages and the slight increase in size of lock messages is negligible for most communication protocols. An alternative option to confirmations within the runtime system are application-driven control interfaces that govern the use of speculation. This provision allows the programmer to specify access patterns for data areas with respect to program parts. Common access patterns, such as access by row, column, block or irregular, can drive the runtime protocol for speculations. However, this interface requires knowledge by the programmer about algorithms and execution environments that may not always be available.

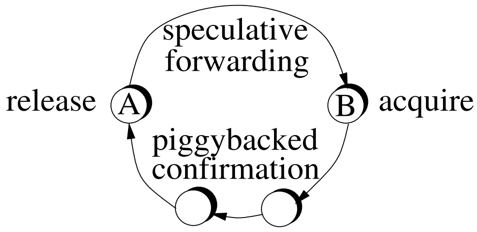

Fig. 2. Confirmation of Speculative Success

The consistency models for distributed shared memory and for distributed mutual exclusion are mostly implemented by synchronous message protocols. These protocols have to be extended to support speculative data distribution. In an asynchronous message-passing model, such as DSM-Threads, the protocols to implement consistency models are slightly more complex [10] and require a few additional considerations than their simpler synchronous counterparts. The main problem of speculative data forwarding is posed by race conditions that may occur in lock protocols and consistency protocols when data is requested while being speculatively forwarded as depicted in Figure 3. A data request may then be forwarded to its requester, a situation that conventional protocols do not anticipate, such that the protocol is broken. This situation should be handled by a silent consumption of data requests on the node that speculatively handed off the data. Such a behavior is safe for protocols that dynamically adjust their knowledge of ownership, such as the distributed dynamic manager protocol [7]. Advanced asynchronous protocols already handle such data races. For example, a token-based protocol for distributed mutual exclusion with priority support prevents these races in the described manner and is directly applicable to non-prioritized protocols as well [11]. The only restriction here is that in-order message handling is required for requests regarding the same data object. Another problem is posed by the requirement to process token/page grant messages when they have not been anticipated, *i.e.,* when they arrive unasked without prior issuing of a request.

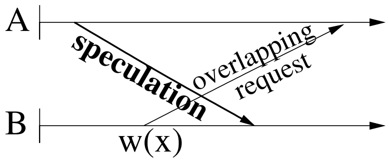

Fig. 3. Overlap between Speculation and Request

The integration of speculation into protocols implementing consistency models also requires a resolution at conflicts between demand-driven requests and speculative information. DSM-Threads always gives demand-driven requests precedence over speculative actions based on the motivation to keep response times of lock requests low and serve requests in a fair manner. If speculative forwarding had the upper hand, the performance may degrade due to false speculation. In the event of multiple pending requests, speculative information could be matched with the requests to serve a later request with a match ahead of earlier ones. However, such behavior may not only be undesirable, distributed data structures may even prevent them, which is the case for distributed lock protocols and distributed data managers.

The speculative distribution of data requires not only the recognition of access patterns but also the association of data with locks. This association can be provided by logging information about the data in the lock object. This simple method may already yield good improvements in performance. A change of the access pattern during program execution may, however, initially result in false speculation. This problem can be addressed by application-driven interfaces but such an approach requires specific knowledge by the programmer, as seen before. An alternative solution is motivated by the observation that a change of access patterns generally occurs between different program parts. If the source of a lock request was related to the program part, speculation could be inhibited when the execution crosses from one part into the next one. This could be achieved by associating the stack pointer and return address (program counter) with a lock upon a lock request. Such information would provide the means to prevent false speculation *after* a new program part is entered but it cannot prevent potentially false speculation at the last release of the lock in the *previous* part. Furthermore, obtaining the stack pointer and return address are machine-dependent operations that restrict portability. Finally, after one false speculation, the confirmation protocol as discussed above should already suffice to reduce the chance of consecutive false speculations.

3.1 Speculative Entry Consistency

Entry consistency associates data explicitly with locks, where the granularity of data is arbitrary (typically objects or abstract data structures). When a lock is transfered between nodes, the data is piggybacked. As a result, accesses to data protected by a lock never result in faults with the associated interrupts and communication. It is assumed data and locks are explicitly associated.

Speculative entry consistency improves upon the lock protocol of the consistency model. Commonly, locks are transfered upon request, *i.e.,* the protocol is demand driven. By logging the history of lock requests, lock movement can be anticipated if regular lock patterns exist. The analogy between lock requests and data accesses illuminates the potential for savings. When a lock request is received on a node that holds the mutual exclusive access (token) of the lock but has already released it, then the requester is recorded as a future speculative destination. The destination node is associated with the same lock object. This has no side-effect on other synchronization objects. Another round of an acquire and release on the same node results in speculative lock forwarding at the point of the release to the recorded destination node as depicted in Figure 4.

The enhanced locking model may be subject to false speculation, just as speculative data distribution, but such a situation may easily be detected by recording if the lock was acquired at the destination node before handing over the token to another node. Upon successful speculation, this protocol reduces the latency of lock operations (acquires) and the amount of messages sent since an acquire to a locally available lock does not result in remote requests anymore. Modifications to existing protocols in order to

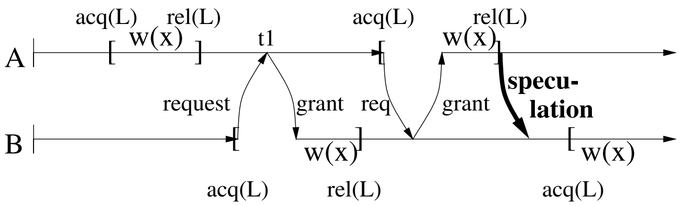

Fig. 4. Example for Speculative Entry Consistency

support speculation have to compensate for requests and speculative motion that occur in parallel as well as reception of a token without prior request. Barriers can be handled in a similar way by considering them as a sequence of a release followed by an acquire [6].

3.2 Speculative Sequential Consistency

Sequential consistency does not provide the luxury of *explicit* associations between data and locks. The model mandates a single-writer-multiple-reader protocol that operates at the granularity of pages. It also requires that the programmer enforces a partial order on memory accesses by using synchronization, reflecting the model of a physically shared-memory environment with multiple threads of control. The lock protocols can be extended to speculatively hand off mutual exclusive access as seen for speculative entry consistency. However, the lack of data association with locks implies that data accesses may still cause page faults in the presence of speculative locking.

Data association can be induced *implicitly* within the sequential consistency model by a reactive response to page faults. Figure 5 depicts the pseudo-code for activating speculations. Upon a page fault in node A, the page is associated with the previous

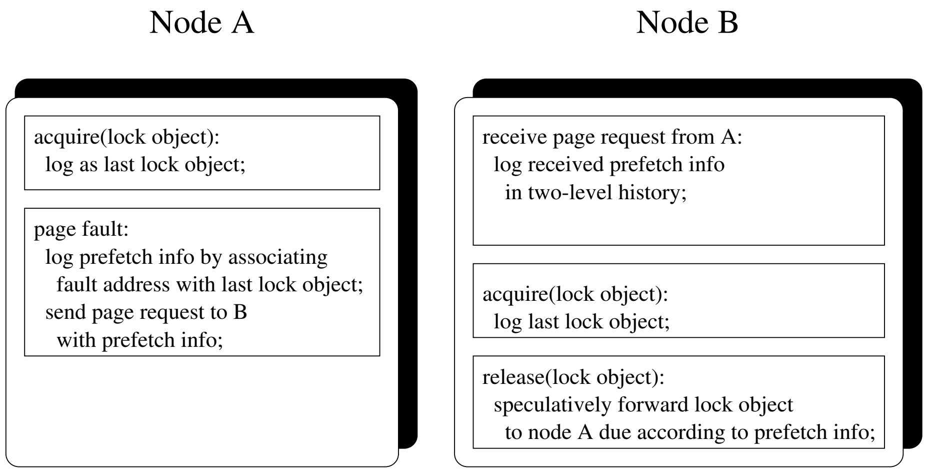

Fig. 5. Protocol for Speculative Sequential Consistency

lock. This information is sent to the owner of the page, node B. Upon reception, node B records this page-lock association in its history and responds with the requested page. At some last point in time, node B locks the same object again and accesses the page in question. Upon releasing the lock, the lock as well as the page are speculatively forwarded to node A if the prefetch information indicates such an action.

In general, a request is locally associated with the lock object of the most recently completed acquire operation but this information is recorded when a remote node receives a page request from the node that had a page fault. Upon the next release operation on the remote node, both the lock token and the data of the page will be speculatively forwarded, even without any remote requests as depicted in Figure 6. Thus, speculation exists at two levels. First, locks and pages are speculatively matched. Second, the association of future destination nodes with locks/pages supplements the speculations. This model has the advantage that the speculative destinations of locks and pages can be compared. In case of a match, the chance of false speculation is reduced. In case of a mismatch, the data may not be associated with the lock, although the local lock history could provide a previously released lock with a matching destination, which may be a likely source for association.

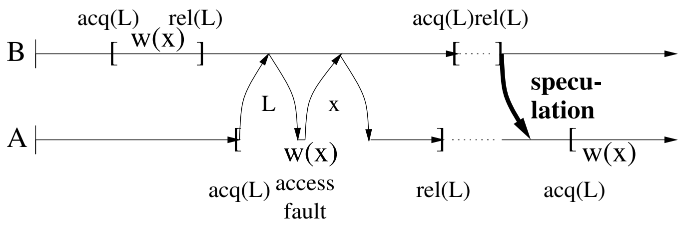

Fig. 6. Example for Speculative Sequential Consistency

Speculative sequential consistency provides potentially high benefits in terms of elimination of access faults and reduction in message traffic. Upon successful speculation, lock requests will not be issued, resulting in lower latencies of locks and fewer messages as seen in the context of speculative entry consistency. In addition, page faults are avoided on the destination node of speculative forwarding and the messages for page requests are saved. The latter savings may be considerable when both a read and a write fault occur for a remote write operation. This claim is based on the observation that there is no portable way to distinguish a read and a write access when a data fault occurs on a non-accessible page. While the faulting instruction may be disassembled to inquire whether the address fault lies within the read or within the written operands, the instruction decoding is highly unportable.

The portable method of distinguishing access modes on faults requires two faults for writes. The first fault results in a request for a read copy of the page. When a second fault occurs on a readable page, the page is supplied in write mode. Thus, the first request requires at least two messages for the read copy plus at least another two messages for the second one for invalidating other read copies, which was not depicted in Figure 6. Speculative sequential consistency can be quite beneficial in such a setting since it can save four page messages, one lock request message, two page faults and reduce the lock latency. False speculation may even be tolerated to some extend if the number of beneficial speculations is sufficient. However, the detection of false speculation with respect to data poses problems since page faults are avoided for successful speculation. Nonetheless, to assess the data/lock association one may perform the lock/data forwarding but, for the first time of speculative movement, leave the page protected to record the next read and write fault. Upon successful speculation, the page access will be enabled at the proper level (read or write) for subsequent speculations. Another option for write accesses will be discussed in the context of speculative release consistency.

3.3 Speculative Release Consistency

Keleher *et. al.* developed a so-called *lazy hybrid* version of release consistency, as mentioned in the related work [6]. The model is thought as an compromise between lazy release consistency and eager release consistency. A heuristic controls the creation of diffs (data differences resulting from write accesses) upon releases of locks and piggybacks them with lock grants. The destinations of such speculative data distribution are members of an *approximate* copy set that continually grows as processes declare interest in data. If a diff is rejected by the heuristic, lazy release consistency is used. A central manager is used to implement this protocol. The main application of this protocol are message-passing systems where the number of messages impacts performance more than size of messages. In comparison with the other variants of release consistency, the model only proves to be marginally better on the average. The reduced number of page faults only played a marginal role in their study.

The speculation realized by the hybrid model neglects opportunities for lock speculation, as explained in the context of speculative entry consistency. It also assumes centralized algorithms for managing pages. Thus, the speculative release consistency proposed here differs in a number of issues. First, speculative locks are utilized. Second, in the absence of a central manager, the approximate copy set has to be created at the root of page invalidation and is subsequently piggybacked with the lock grant message. Third, the approximate copy set can be updated by assessing the effectiveness of prefetching using the confirmation model discussed in the context of speculative sequential consistency. When speculative forwarding occurred without subsequent data access, members are removed from the copy set. This model reduces the overall amount of communication. It also shows that confirmations should be repeated from time to time to reflect a change of access patterns of a different program part in communication patterns. Finally, logging access patterns enables the speculation model discussed in this paper to selectively create diffs and grant locks. This will be discussed in more detail in the following.

3.4 Access Patterns

The collection and recognition of access patterns is a fundamental condition for speculative data distribution. Often, temporal locality exhibits simple patterns where each acquire was followed by an access miss, *e.g.* for sequential consistency and for speculative lock forwarding with entry consistency. Histories of depth one suffice to handle such simple patterns. We also experimented with histories of size two, where prefetching is delayed until an access is repeated. These histories of depth two have been successfully used in the context of branch prediction with certain variations [17]. Due to the similarity between control flow changes and lock requests along control flow paths, these histories are promising for lock protocols as well.

Another common pattern is given by local reuse of resources, for example when every second acquire is followed by an access fault. The latter case also provides opportunities for improvements, in particular for speculative release consistency. Here, not only lock forwarding but also diff creation can be delayed to the point of the second release of the lock. Again, similarities to branch prediction exist, in particular wrt. two-level prediction schemes [21]. However, the problem of DSM access histories requires that addresses of pages or locks are recorded while branch prediction records whether or not a branch was taken. In this sense, DSM access patterns may be closer to predicting the target address of indirect jumps, which is a problem that has also been handled successfully with two-level histories [18]. In the context of DSMs, the first-level history of depth n records the source nodes of the n most recent requests, often in a compressed format within k bits (see Figure 7). Such a *Target History Buffer (THB)* should be kept for each page or lock. (A more simplistic variant for branch prediction uses a single global THB but lacks precision, in particular for more complex access patterns.) The second-level history consists of a *predictor table (PT)* with entries for 2^k patterns indexed by a hashed value calculated from all THB entries. The PT simply contains at the index of the THB the last access recorded for this pattern, which is then used for the next prediction. Thus, the use of two-level histories requires that actual accesses be propagated to the predictor, which can be accomplished by piggybacking this knowledge together with page or lock migration.

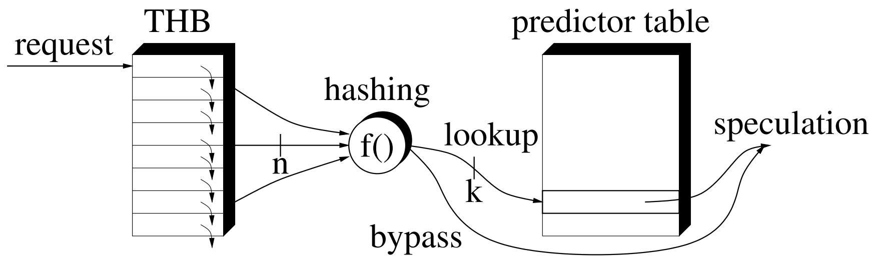

Fig. 7. Two-Level Access Histories

There are several options for using two-level histories within DSMs. One may either use the program counter the base address of a page (or a lock index) for requests. Alternatively, the fault address may be used. The difference between these options becomes clear when considering spatial locality in an SPMD model with synchronization at each iteration. When fault addresses are used, the addresses differ from iteration to iteration resulting in ever new indices for the PT. For column-oriented spatial locality, the base addresses of a page would always result in the same PT index yielding better predictions. Another way to handling this problem would be by considering only the most significant k bits of the fault address (and the least significant k bits for lock indices). For row-oriented spatial locality, on the other hand, the lower bits of the faulting address may be more telling assuming a uniform stride. To deal with both of these cases, the entire address should be saved in the THB and the hashing process can be augmented to recognize strides. For small strides (column indexing), the highest k bits are hashed while large strides may bypass the PT (see Figure 7) in order to prefetch the next page (in direction of the stride).

A problem is posed by irregular access patterns. While complicated patterns may still be recognized by two-level histories, a completely chaotic access behavior cannot be dealt with. First, such a situation has to be recognized. This can be accomplished by adding a two-bit saturating up-down counter to each PT entry (also similar to branch prediction [17]) that is incremented on successful speculation and decremented otherwise. Speculation will simply be inhibited when this counter is less than 2 (see Figure 8). This limits the amount of false speculation (with prefetching) to two instances and requires another two correct speculations (without prefetching) before prefetching is reactivated.

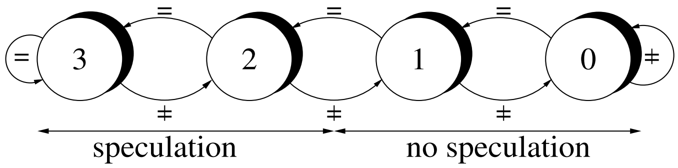

Fig. 8. DFA for a Two-Bit Saturating Up-Down Counter

In an SPMD programming model, a number of distributed processes may issue a request for the same page or lock almost at the same time. Sometimes, node A may be slightly faster than node B, the next time B may be faster. The resulting access patterns may seem irregular at first but the group of processes issuing requests within a certain time frame remains the same. Thus, two-level histories may also help out but it may take slightly longer before different patterns are stored in the corresponding PT-entries. In addition, it would help to delay transitive migrations to allow the current owner to utilize the forwarded information, thereby compensating for jitters of requests. Our system uses a short delay before it responds to further requests for this reason, which is similar to timed consistency but is often used in conjunction with consistency protocols that support data migration.

The two-level history scheme described here records the first access between synchronizations for implementations where subsequent faults are still forced before pages are validated (see rated work). The framework described in this paper avoids faults altogether after successful speculation but will only record the first two faults (for depth two histories or two-bit counters). It may also be useful to experiment with a combination of these options where subsequent faults are only forced on every ith iteration, depending on the stride. This would allow prefetching of the next page at page boundaries and re-confirmation of speculation.

Speculative Locking			Speculative Data Forwarding		
Speculation	No	Yes	Speculation	No	Yes
Messages	38	22	Messages	76	8
Local Locks	1	18	Page Faults	18	4

Table 1. Performance Evaluation

4 Measurements

Experiments were conducted with the aforementioned distributed shared-memory system called DSM-Threads. The user interface of the system closely resembles the POSIX Threads API [19]. It distributes threads remotely upon requests by spawning a remote process (if none exits) and executing the requested thread (instead of the main program). Each logical node is itself multi-threaded, *i.e.,* local threads can be executed in a physically shared memory. Nodes are identified by a tuple of host and process. The system supports asynchronous communication for message-passing systems, and the low-level message-passing implementations currently support TCP sockets and the SCI common message layer [3]. The lock protocol is an implementation along the lines of Naimi *et. al.* [12] although DSM-Threads supports asynchronous handling of the lock. The system also supports priority-based locks but this option was not used for the experiments to minimize the influence of more complex protocols [11]. Sequential consistency is realized by Li/Hudak's work [7] but uses an asynchronous variant of the protocol [10]. The system also provides entry consistency much alike Bershad *et. al.* [1]. The implementation of lazy release consistency following Keleher's work is still in progress [5]. All algorithms use decentralized distributed methods to prevent scalability problems due to software restrictions.

The DSM-Threads system runs under SunOS 4.1.x and the Linux operating systems at present. It utilizes POSIX threads at runtime and Gnu tools for compilation, assembling, linking and extraction of information from the symbol table [9]. The following experiments were conducted under Linux. A ping-pong test was designed to assess the potentials of speculation. Two logical nodes share data on the same page that is accessed for reading and writing. The accesses occur between an acquire and a release of the same lock object. Each node repeats the sequence of acquire, read accesses, write access and release 10 times. The runtime system was instrumented to report the number local locks (without communication) and the number of messages sent with respect to pages and locks. The amount of page faults could be inferred from these numbers.

The first experiment compares a lock protocol without speculation with a version that speculatively forwards locks, only. The left part of Table 1 depicts the number of messages recorded for locks within the program. Each acquire required a request message and a granting response without speculation, except for the first acquire since the token was locally available. With speculation, the first acquire on each node required two messages as before and resulted in a recording of the access pattern. All other acquires were preceded by speculative forwarding of the token at a release to the other node, resulting only in one message per release. The acquires proceeded without message latency as they could be granted locally. Excluding the start-up overhead, the number of messages can be reduced by 50% with this method.

The second experiment compares data migration without speculation with a version that speculatively forwards data at release operations. The right part of Table 1 depicts the number of messages recorded for data requests within the program. Lock requests are not depicted. Each access required a request message and a granting response without speculation, except for the first read and write accesses since the initial node had write access to the data. The remaining 9 iterations in the initial node and 10 iterations in the other node constituted of a read access (request and grant read) and a write access (invalidate copy set and acknowledge), for a total of $19 * 4 = 76$ messages. Each of the 19 instances resulted in a read fault followed by a write fault. With speculation, the first round of a read and write access pair on each node required four messages (as before) and resulted in recording the access pattern. All other accesses were preceded by speculative piggybacking of the data with the (speculative) token forwarding to the other node at the release point, resulting in no data messages at all. Notice that the overhead of the release message was already accounted for in the lock protocol. The data accesses on the destination node proceeded without message latency as they could be satisfied locally without a page fault. Thus, after a start-up overhead dependent on the number of nodes interested in the data, all data messages and page faults are eliminated. For n interested nodes, the initial overhead would amount to $4n$ messages and $2n$ faults. In the event of confirmation for speculations, an additional $2n$ faults would be required in the second round of access pairs.

The actual savings may depend on the communication medium and processing environment, which is common for DSM systems. If communication latencies are small relative to processing time, the observed savings can be achieved in practice. For longer latencies, chances increase that a lock request is issued before or while speculative forwarding occurs. In that case, the advantages of speculation may not materialize in full but the message performance would not be degraded. This applies mostly to speculative entry consistency since the lock protocol is the primary cause of messages. The book-keeping overhead in terms of local computation for speculation is relatively low.

However slow the communication system, speculation may still result in savings of messages although long latencies cause parallel request issues. In particular for speculative data distribution, the data will be supplied at the correct access level (read or write). In case of write accesses, the overhead of a prior read fault with its two messages can still be avoided. In conjunction with lock association, the full savings of speculative data distribution should still materialize for speculative sequential consistency.

5 Future Work

On-going work includes an implementation of lazy release consistency within DSM-Threads. A comparison with speculative release consistency would be interesting with respect to the competitiveness of speculative sequential consistency. Current work also includes the recognition of more complex access patterns, speculative confirmation during initial speculation and temporal re-confirmation of speculations. We are also considering to log access patterns on the faulting node and piggybacking the information with the lock token to reduce the chance of false associations between data and locks. Furthermore, it may be possible to avoid request messages (or delay them) if a demanding node can be ensured that speculation will provide him with the data right away. Finally, selected programs of the *de-facto* standard benchmark suite “Splash” are being adapted for DSM-Threads for a more comprehensive evaluation [20].

6 Conclusion

This work presented methods for speculative distribution of resources as a means to address the main short-comings of DSM systems, namely the amount of communication overhead and the cost of access faults. A number of models with speculative support are developed. Speculative entry consistency forwards lock tokens at the point of a release to another node based on past usage histories. Speculative sequential consistency not only utilizes a speculative lock protocol but allows data association with the most recently released lock. Future lock grants will then piggyback the associated data, thereby circumventing access faults at the destination node. Speculative release consistency extends Keleher’s hybrid model to provide speculative lock forwarding, initial confirmation of the benefits of speculation and temporary re-confirmation of these benefits to detect changes in access patterns. Initial measurements show that the number of messages for the lock protocol can be almost reduced by half under ideal circumstances. The savings for speculative data distribution for sequential consistency are even more dramatic. After a start-up of a constant number of messages, all subsequent messages may be avoided by piggybacking the data with lock messages. Furthermore, all page faults may be avoided as well when the speculation is successful. The main problems of speculation are twofold. First, there exists a potential of false speculation, where resources are sent to a node that does not use them. The paper suggests two-level histories to recognize access patterns and use re-confirmation to avoid unnecessary communication. Second, large communication latencies relative the short processing intervals may allow request messages to be issued even though the requests will be granted by speculation. Speculation does not cause any penalty in this case, in fact, it may still reduce latencies but it cannot reduce the amount of messages if requests are sent in parallel with speculations. Overall, existing consistency protocols only require moderate changes to support speculation so that it should well be worth to incorporate these methods into existing DSM systems.

References

1. B. Bershad, M. Zekauskas, and W. Sawdon. The midway distributed shared memory system. In *COMPCON Conference Proceedings*, pages 528–537, 1993.
2. R. Biachini, R. Pinto, and C. Amorim. Data prefetching for software DSMs. In *Int. Conf. on Supercomputing*, 1998.
3. B. Herland and M. Eberl. A common design of PVM and MPI using the SCI interconnect. final report for task a 2.11 of ESPRIT project 23174, University of Bergen and Technical University of Munich, October 1997.

4. M. Karlsson and P. Stenström. Effectiveness of dynamic prefetching in multiple-writer distributed virtual shared memory systems. *J. Parallel Distrib. Comput.*, 43(7):79–93, July 1997.
5. P. Keleher, A. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *International Symposium on Computer Architecture*, pages 13–21, 1992.
6. Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. An evaluation of software-based release consistent protocols. *J. Parallel Distrib. Comput.*, 29:126–141, September 1995.
7. K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Systems*, 7(4):321–359, November 1989.
8. D. K. Lowenthal, V. W. Freeh, and G. R. Andrews. Using fine-grain threads and run-time decision making in parallel computing. *J. Parallel Distrib. Comput.*, August 1996.
9. F. Mueller. Distributed shared-memory threads: DSM-threads. In *Workshop on Run-Time Systems for Parallel Programming*, pages 31–40, April 1997.
10. F. Mueller. On the design and implementation of DSM-threads. In *Proc. 1997 International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 315–324, June 1997. (invited).
11. F. Mueller. Prioritized token-based mutual exclusion for distributed systems. In *International Parallel Processing Symposium*, pages 791–795, 1998.
12. M. Naimi, M. Trehel, and A. Arnold. A log(N) distributed mutual exclusion algorithm based on path reversal. *J. Parallel Distrib. Comput.*, 34(1):1–13, April 1996.
13. B. Nitzberg and V. Lo. Distributed shared memory: A survey of issues and algorithms. *IEEE Computer*, 24(8):52–60, August 1991.
14. J. Protic, M. Tomasevic, and V. Milutinovic. Distributed shared memory: Concepts and systems. *IEEE Parallel and Distributed Technology*, pages 63–79, Summer 1996.
15. Rice University, Houston, Texas. *High Performance Fortran Language Specification*, 2.0 edition, May 97.
16. A. Schuster and L. Shalev. Using remote access histories for thread scheduling in distributed shared memory systems. In *Int. Symposium on Distributed Computing*, pages 347–362. Springer LNCS 1499, September 1998.
17. J. E. Smith. A study of branch prediction strategies. In *Proceedings of the Eighth International Symposium on Computer Architecture*, pages 135–148, May 1981.
18. J. Stark and Y. Patt. Variable length path branch prediction. In *Architectural Support for Programming Languages and Operating Systems*, pages 170–179, 1998.
19. Technical Committee on Operating Systems and Application Environments of the IEEE. *Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API)*, 1996. ANSI/IEEE Std 1003.1, 1995 Edition, including 1003.1c: Amendment 2: Threads Extension [C Language].
20. S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characteriation and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–37, New York, June 22–24 1995. ACM Press.
21. T.-Y. Yeh and Y. N. Patt. Alternative implementations of two-level adaptive branch prediction. In David Abramson and Jean-Luc Gaudiot, editors, *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 124–135, Gold Coast, Australia, May 1992. ACM Press.