PARALLEL SOFTWARE CACHES

Arno Formella **Jörg Keller**

04/1996

Fachbereich 14 Informatik
Universität des Saarlandes
Postfach 151150
66041 Saarbrücken
Germany

SFB 124, D4

Parallel Software Caches

Arno Formella Jörg Keller*

Universität des Saarlandes, FB 14 Informatik
66041 Saarbrücken, Germany
Email: {formella,jkeller}@cs.uni-sb.de, Phone: ++49 681 302 5538

Abstract

We investigate the construction and application of parallel software caches in shared memory multiprocessors. In contrast to maintaining a private cache for each thread, a parallel cache allows the re-use of results of lengthy computations by other threads. This is especially important in irregular applications where the re-use of intermediate results by scheduling is not possible. Example applications are the computation of intersections between a scanline and a polygon in computational geometry, and the computation of intersections between rays and objects in ray tracing.
A parallel software cache is based on a readers/writers lock, i.e., as long as no thread alters the cache data structure, multiple threads may read simultaneously. If a thread wants to alter the cache because of a cache miss, it waits until all other threads have left the data structure, then it can update the contents of the cache. Other threads can access the cache only after the writer has finished its work. To increase utilization, the cache has a number of slots that can be locked separately. We investigate the tradeoff between slot size, search time in the cache, and the time to re-compute a cache entry. Another major difference between sequential and parallel software caches is the replacement strategy. We adapt classic replacement strategies such as LRU and random replacement for parallel caches.
As execution platform, we use the SB-PRAM, but the concepts might be portable to machines such as NYU Ultracomputer, Tera MTA, and Stanford DASH.

1 Introduction

In time consuming computations, intermediate results are often needed more than once. A convenient method to save these results for later use are software caches. When switching to parallel computations, the easiest method is to give each thread its own private cache. However, this is only useful if the computation shows some regularity. Then, the computation can be scheduled in such a way that a thread that wants to re-use an intermediate result knows which thread computed this result, and that this thread in fact did compute the result already. Due to the arising difficulties in many applications, re-using intermediate results is confined to the generating thread, i.e., the cached information is not shared. Another disadvantage of private caches—especially in massively parallel computers where

*Supported by a DFG habilitation scholarship.

memory resources are often limited—is the fact that for p threads p times as much memory is occupied as in the sequential case.

Many challenging applications lack the required amount of regularity. Coincidently, these are the kind of applications that usually cannot be parallelized well on distributed memory multiprocessors (DMM). For these applications, performance often is better if they are run on shared memory machines (SMM). On an SMM the applications may then benefit from a shared parallel software cache. By this term we mean a software cache in the shared memory, where all threads place their intermediate results and all threads try to re-use intermediate results, no matter which thread computed the results.

One advantage of a parallel cache is that intermediate results are calculated less frequently. This reduces the amount of computation and thus the run time as long as the additional communication time does not compensate the benefit. The memory requirements are reduced because several processors share one data structure. Parallel caches might be suitable for DMMs as well. At least for NUMA-type architectures or small scale distributed memory systems an improvement for the run time seems possible.

There is not much literature available which deals with the possibility to reduce the amount of computation in a parallel application while increasing the amount of communication. For most parallel architectures, communication between processors should be avoided if high performance is the aim [3, 8]. But with the upcoming of shared memory architectures both as massively parallel multiprocessors or as small scale bus oriented multiprocessors the concept of a parallel cache promises additional performance. We show that the SB-PRAM [1, 2] is a good platform to investigate the numerous tradeoffs that one encounters while implementing such a parallel data structure. Some of the concepts might be transferable to other architectures such as NYU Ultracomputer [9], Tera MTA [4], and Stanford DASH [13].

We define the notion of a cache formally in section 2. The classical replacement strategies and possible cache organizations are presented in section 3. The modifications for a parallel cache are explained in section 4. The SB-PRAM as execution platform is briefly discussed in section 5. Section 6 introduces the applications FViewpar and Rayo and presents the performance results we obtained with the parallel data structure on these applications. Section 7 concludes.

2 Definition of a Cache

The notion of a cache is primarily known in hardware design. There, the hardware cache is a well known means to speedup memory accesses in a computer system [11]. We adapted the concept of such an "intermediate memory" to the design of efficient shared memory data structures.

Let us introduce first some notations. An *entry* $e = (k, i)$ consists of a *key* k and associated *information* i. A *universe* U is a set of entries. Given key k the *address function* m returns the associated information $i = m(k)$, if $(k, i) \in U$. Let us assume that for any k, function m succeeds in calculating i. Usually, m is a relatively complex function, we denote the time

to compute $m(k)$ by $t_m(k)$. Universe U can be large and is not necessarily given explicitly. We say that a universe is *ordered* if the keys of its entries can be ordered.

A *cache* C is a small finite subset of U together with a *hit function* h and an *update function* u. Given key k the hit function h returns information i associated with k if the entry $e = (k, i)$ is located in C, i. e., $h(k) = i$ iff $(k, i) \in C$. The hit function h is a relatively simple function, we denote the time to compute $h(k)$ by $t_c(k)$. Time $t_c(k)$ should be much smaller than $t_m(k)$. For an entry $e = (k, i)$ the update function u inserts e in the cache C possibly deleting another entry in C. Usually, u implements some replacement strategy. We denote by $t_u(k)$ the time to update C with an entry which has key k.

The cache C can be used to speedup addressing of U. Given key k, first try $h(k)$ which delivers the information i if $(k, i) \in C$. If an entry is found, we call it a cache hit. If no entry is found, we call it a cache miss. In the latter case, use function $m(k)$ to calculate i. Now, the update function u can be invoked to insert the entry $e = (k, i)$ into C, such that a following request with same key k succeeds in calculating $h(k)$.

For j subsequent accesses to the cache C, i. e., computing $h(k_1), ..., h(k_j)$, the ratio $\alpha = s/j$ where s is the number of misses is called *miss rate*, analogously $\alpha' = 1 - \alpha = (j - s)/j$ is called *hit rate*. For a sufficiently large sequence of accesses, we can assume an average access time $t_c = 1/j \cdot \sum_{k=1}^{j} t_c(k)$ to access the cache. Similarly, we assume an average access time t_m to access the universe, and an average update time t_u after a cache miss. The run time for a sequence of j accesses to U without a cache is $T_0 = j \cdot t_m$ and with a cache it is

$$T_1(\alpha) = j \cdot (t_c + \alpha \cdot (t_m + t_u))$$

Hence, in case of worst miss rate, i. e., $\alpha = 1$, the run time is increased by a factor $T_1(1)/T_0 = 1 + (t_c + t_u)/t_m$, and in best case, i. e., $\alpha = 0$, the run time is decreased by a factor $T_1(0)/T_0 = t_c/t_m$. Thus, the cache improves the run time of j consecutive accesses to U if $T_1(\alpha) < T_0$ or

$$\alpha < \frac{t_m - t_c}{t_m + t_u}$$

Clearly, the improvement of the entire program depends on the portion of the overall run time T which is spent in accessing U.

3 Replacement Strategies and Cache Organization

For the update function u one has to decide how to organize the cache C such that subsequent accesses to the cache perform both fast and with a high hit rate. For a sequential cache the following update strategies are commonly used.

LRU, least recently used: The cache entries are organized in a queue. Every time a hit occurs the appropriate entry is moved to the head of the queue. The last entry in

the queue is replaced in case of an update. Hence, an entry stays at least $|C|$ access cycles in the cache, although it might be used only once. For an unordered universe a linear search must be used by the hit function to examine the cache. Starting at the head of the queue ensures that the entry which was accessed last is found first.

FRQ, least frequently used: Here, the cache entries are equipped with counters. The counter is incremented with every access to the entry. The one with the smallest counter value is replaced in case of an update. Entries often used remain in the cache and the most frequently used are found first if a linear search is employed on a list sorted by frequencies.

CWC, $|C|$–way cache: For a cache of fixed size the cache is simply implemented by a round robin procedure in an array. Thus, after $|C|$ updates an entry is deleted, independently of its usage count. The update of the cache is very fast, because the location in the cache is predetermined.

RND, random replacement cache : The cache entries are organized in an array as well. In case of an update, one entry is chosen randomly and replaced. For the first three strategies one always finds some update patterns which exhibit poor performance. The probability that this happens to a random cache is usually low.

The organization of the cache partly depends on the structure of the universe. If the universe is not ordered, then for LRU and FRQ, the cache preferably consists of a linked list of entries. For a miss, the function h must search through the complete list. For CWC and RND the complete array must be searched, too. However, if the universe is ordered, then we can organize the cache such that the entries appear in sorted order. Given key k, function h must search until either (k, i) or an entry (k', i') with $k' > k$ is found. If the number of entries per cache gets larger, an alternative to speed up the search is to use a tree instead of a list.

4 Parallel Caches

We assume that i concurrent threads $p_1, p_2, \ldots, p_i$ are used in the parallel program. Here we mean real parallel threads that are running simultaneously on at least i processors. The threads need information from the universe U.

First, we want to remark that a parallel cache behaves similar to a normal sequential cache—even in a parallel program—when all accesses to the cache are serialized. The serialization may not be due to some access conflicts. It might just be the case that the different threads are accessing the cache at different instants in time, such that no concurrent access to the data structure is necessary.

In the case that in the parallel algorithm no concurrent access is necessary, the same run time and efficiency remarks as given above for the sequential case hold. For an optimally parallelized program, each individual thread p_i reduces its computation time by the factor

$$s(\alpha_i) = \frac{t_m}{t_c + \alpha_i \cdot (t_m + t_u)}$$

where α_i is the individual miss rate of thread p_i. However, α_i depends on other threads because the cache data structure is updated by all threads. This leads to a tradeoff: the hit rate may be higher, because another thread has pre-calculated a cache entry, or it may be smaller, because another thread has deleted a cache entry.

We assume that the threads access the universe in parallel by executing function m concurrently, and that the access time t_m in the average does neither depend on a specific thread nor the access. Note that this may require replication of the data base on a DMM for some applications.

4.1 Concurrent Accesses

If the program spends a large amount of time in accessing U and if many threads are accessing the cache, it happens more often that concurrent accesses to the cache become necessary. It depends on the architecture of the parallel machine how access conflicts might be solved. In the worst case, all reads and writes to the cache are serialized. On an SMM, however, a more efficient solution is possible.

Concurrent read accesses to the cache are simple to handle. Updating the cache introduces some difficulties: i) one thread wants to delete an entry of the cache which is still or just in the same moment used by another one or ii) two threads might want to change the cache structure at the same time. To overcome the difficulties, a parallel data structure must be created which is protected by a so called readers/writers lock. A thread which wants to perform an update locks a semaphore; when all pending read accesses have finished, the writer gets exclusive access to the cache. During this time other readers and writers are blocked. After the update has been terminated, the writer releases the lock.

A thread p_i first inspects the cache as a reader. After a miss, the thread leaves the readers queue and calculates address function m. This gives other writers the chance to perform their updates. Once a new entry is found p_i enters the writers queue. Because the writers are queued as well, p_i must check again whether the entry has already been inserted in the cache during its calculating and waiting time. The readers/writers lock restricts the speedup to $1/\alpha$, because all misses are serialized. For an architecture that does not allow for concurrent reads, the speedup might be even less.

A machine with atomic parallel prefix and atomic concurrent access avoids serialization while accessing the lock data structure and while updating the waiting queues (see description of the SB-PRAM in section 5).

4.2 Improvements

To overcome the speedup restrictions that the exclusive writer imposes, one can use several caches $C_0, \ldots, C_{j-1}$, if there is a reasonable mapping from the set of keys to $\{0, \ldots, j-1\}$. An equivalent notation is that the cache consists of j slots, each capable of holding the same number of entries, and each being locked independently. While this realizes the same functionality, it hides the structure from the user, with the exception of the mapping

function. The distribution of the accesses to the different slots will have a significant impact on the performance.

A difference between sequential and parallel software caches is the question of how to provide the result. In a sequential software cache it is sufficient to return a pointer to the cached entry. As long as no entry of the cache is deleted, the pointer is valid. In a sequential software cache, one will use the cached information, continue and access the cache some time later on. Hence, the above condition is sufficient.

In a parallel cache, the cached entry a that one thread requested might be deleted immediately afterwards because another thread added an entry b to the cache and the replacement strategy chose to delete entry a to make room for entry b. Here, we have two possibilities. Either we prevent the replacement strategy from doing so by locking requested entries until they are not needed anymore. Or, we copy such entries and return the copy instead of a pointer to the original entry.

In our applications, all data retrieved from the cache are used in a similar way, i.e. they are needed for about the same amount of time. Hence, we can determine at compile time whether to copy or to lock and simply initialize the cache data structure appropriately. Which one of the two methods leads to higher performance depends on the application, i. e., how long a cache entry might be locked and how much overhead a copying would produce.

For an explicitly given universe, neither locking nor copying is necessary, because the cache contains only pointers to entries. In case of a hit, a pointer to the entry in the universe is returned. The update function safely can replace the pointer in the cache although another threads still makes use of the entry. Additionally, the second check before the cache is updated can be reduced to a simple pointer comparison.

4.3 Replacement Strategies

Another major difference between sequential and parallel software caches is the replacement strategy. The interactions between threads make it more difficult to decide which entry to remove from a slot. We adapt the classic replacement strategies as described in section 3 for parallel caches.

In the sequential version of LRU the entry found as a hit was moved to the beginning of the list. This does not work in the parallel version, because during a read no change of the structure of the cache is possible. The reader would need writers permissions and this would serialize all accesses. Note that for architectures that do not allow for concurrent reads the movement of the entry to the head of the list can be implemented, because requests are serialized anyway.

We implemented the parallel version of LRU as following. Every reader updates the time stamp of the entry that was found as a hit. For unordered universes, a writer sorts all entries in the cache according to their time stamps prior to updating. The least recently used entry is deleted. In order to improve the run time of a write access, the sorting can be skipped, but this might increase the subsequent search times for other threads.

Replacement strategy FRQ is implemented similarly. Instead of the time stamp the reading thread updates an access counter of the entry that was found as a hit. A writer deletes the entry e with lowest access frequency $f(e)$. The frequency is defined as $f(e) = a/n$, where a is the number of accesses to entry e and n is the total number of accesses to the cache since insertion of e. For unordered universes, the entries might be rearranged prior to updating. There is a similar tradeoff between the sorting time and search time as for LRU.

Note that we defined the entries' access frequencies not by absolute numbers (as in Section 3) but relative to the total number of access since their insertion. This removes a preference towards "older" entries.

To go even further, the question arises whether is it fruitful to consider the whole lifespan of an entry in the cache. For example, if an entry is in the cache for a large number of accesses and additionally it has a relatively high actual frequency, then the entry will remain in the cache for a significant amount of time, since its frequency is reduced very slowly. A possible solution to this problem is to treat only the last x accesses to the cache to compute the actual frequency. Previous accesses can be just ignored or one might use some weight function which considers accumulatively blocks of x accesses while determining the frequency.

The replacement strategies CWC and RND can be implemented similarly to the sequential version.

5 Execution Platform

The results presented in section 6.2 have been obtained on the SB-PRAM, a shared memory multiprocessor simulating a priority concurrent read concurrent write PRAM [1]. It consists of p physical processors connected via a butterfly network to p memory modules. A physical processor simulates several virtual processors [12], thus the latency of the network is hidden and a uniform access time is achieved. Each virtual processor has its own register file and the context between two virtual processors is switched after every instruction in a pipelined manner. Accesses to memory are distributed with a universal hash function so memory congestion is avoided. The network is able to combine accesses already on the way from the processors to the memory location. This mechanism avoids hot spots and is extended to employ parallel prefix operations which allow to implement very efficient parallel data structures without serialization.

A first prototype with four physical and 128 virtual processors is running. Although most of the results have been obtained through simulations of the SB-PRAM on workstations, we have verified the actual run times on the real machine [10]. Each virtual processor executes one thread. The predicted run times have been matched exactly with the run times obtained by simulation.

The concepts underlying the SB-PRAM have not been developed independently from others. The concept of virtual processors in hardware was already used in the Denelcor HEP [15], and is used again in the Tera MTA [4]. The concepts of hashing and combining were already used in the NYU Ultracomputer [9] and IBM RP3 [14]. NYU Ultracomputer, IBM

RP3, Tera MTA, and Stanford DASH [13] (a multiprocessor with cache-coherent virtual shared memory) also have hardware support for parallel prefix operations, although DASH is restricted to increment/decrement. The aim of the SB-PRAM is to bring all these concepts together in a single machine.

6 Experiments

6.1 Applications

Application FViewpar [7] realizes a fish-eye lens on a layouted graph, the focus is given by a polygon. Graph nodes inside and outside the polygon are treated differently. To determine whether a node is inside the polygon, we intersect the polygon with a horizontal scanline through the node. The node is inside if the number of intersection points to the left of the node is odd. The distance of the node from the nearest intersection point is used to compute the displacement for that node. If a scanline does not intersect the polygon, the nearest horizontal scanline intersecting the polygon is used. If the scanline hits the polygon only in one intersection point, this point is counted twice. The parallelization is performed with a parallel queue over all nodes of the graph.

Universe U is the set of all possible horizontal scanlines intersecting the polygon, thus all (s, l) where s is a scanline and l is the list of intersection points between the scanline s and the polygon. Universe U is not given explicitly. Thus, with the notation introduced above, a key k is a scanline s, information i is a list of intersection points, and the address function m is the procedure which intersects a scanline with the polygon. The hit function h checks whether the intersection points of the scanline s with the polygon have been calculated already. Time $t_m(k)$ reflects the time for the calculation of all intersection points of the scanline with the focus polygon. Time $t_c(k)$ is just the search time in the cache for the existence of the key.

The cache makes use of two facts: first, the intersection points of a scanline are used for all nodes of the graph that are located on a horizontal line in the original image, and thus exploits regularity in the input graph. Second, the first and last scanline intersecting the polygon are used for all nodes that are located above, respectively below, the polygon.

In order to implement a cache with multiple slots application FViewpar uses a simple mapping function g. A horizontal scanline s given as $y = c$, where c is constant, is mapped to slot $g(s) = c \bmod j$. Here, j is some small prime number, so the accesses to the slots are distributed sufficiently uniform.

Application Rayo [6] is a ray tracer. The cache is used to exploit image coherency. In the case presented here, we reduce the number of shadow testing rays. Those rays are normally cast from an intersection point towards the light sources, so that possibly blocking objects are detected. An intersection point is only illuminated if no object is found in direction towards the light source. We use a separate cache for each light source which is a standard means to speedup ray tracing. If two light sources are located closely together, one might unify their caches.

Universe U is the set of all pairs (v, o) where v is a shadow volume generated by object o and the light source. Due to memory limitations U is not given explicitly. A key k is a shadow volume, information i is the blocking object o, and the address function m is simply the ray tracing procedure for ray r finding a possible shadow casting object. The hit function h examines for a new ray r, whether its origin is located in a shadow volume of an object in the cache associated with the light source. The cache makes use of the coherency typically found in scenes: if two intersection points are sufficiently close to each other then the same object casts a shadow on both points. Here, $t_m(k)$ is the time needed by the tracing procedure that finds the closest intersections point of a ray in the scene. $t_c(k)$ is the time to check whether the ray does not leave at least one shadow volume in the cache.

An alternative approach does not compute the shadow volume explicitly, because it might not have a simple geometrical shape. One verifies for a certain object in the cache whether the object really casts a shadow on the origin of the ray. Hence, an entry (k, i) can be replaced simply by the information (i), coding a previously shadow casting object. A cache hit returns a pointer to the object that casts the shadow. Now, the universe is explicitly given, because the objects are always available. Note, that all objects in the cache must be checked for an intersection with the ray, because no key is available to reduce the search time. Time $t_c(k)$ reduces to simple intersection tests of the ray with objects contained in the cache.

For application Rayo, the mapping function g takes advantage of the tree structure while spawning reflected and transmitted rays. For each node in the tree a slot is created. Thus, the slots allow to exploit the coherency between ray trees for adjacent pixels.

On the shared multiprocessor each processor has access to the entire scene description and the underlying space subdivision. Because we parallelized one of the fastest ray tracers [5] and because the problem shows a high degree of parallelism, linear absolute speedup can be achieved. The load distribution is performed almost optimally with the help of a parallel queue which contains all first level rays.

6.2 Results

We tested several aspects of the concept of software caches: its scalability, the influence of the replacement strategies, whether copying or locking of the information is more effective, and the tradeoffs due to size of the cache and its organization.

We simulated application FViewpar for $p = 2^i$ processors, $i = 0, \ldots, 7$, with and without cache. For the cache, we used a fixed size of 16 slots, each capable of holding 4 entries. Accessed entries were copied from the cache to the memory space of the particular thread. Let $T_x(p)$ denote the runtime on p processors with replacement strategy x, where no indicates that no cache is used. Figure 1 depicts the speedups $s_x(p)$, where $s_x(p) = T_{no}(1)/T_x(p)$, for $x = no, lru, frq, rnd, cwc$.

For $p = 1$, all replacement strategies give a runtime improvement by a factor of about 1.8. As p increases, the curves fall into two categories. RND and CWC strategies provide less improvement, until they make the application slower than without cache for $p = 128$. LRU

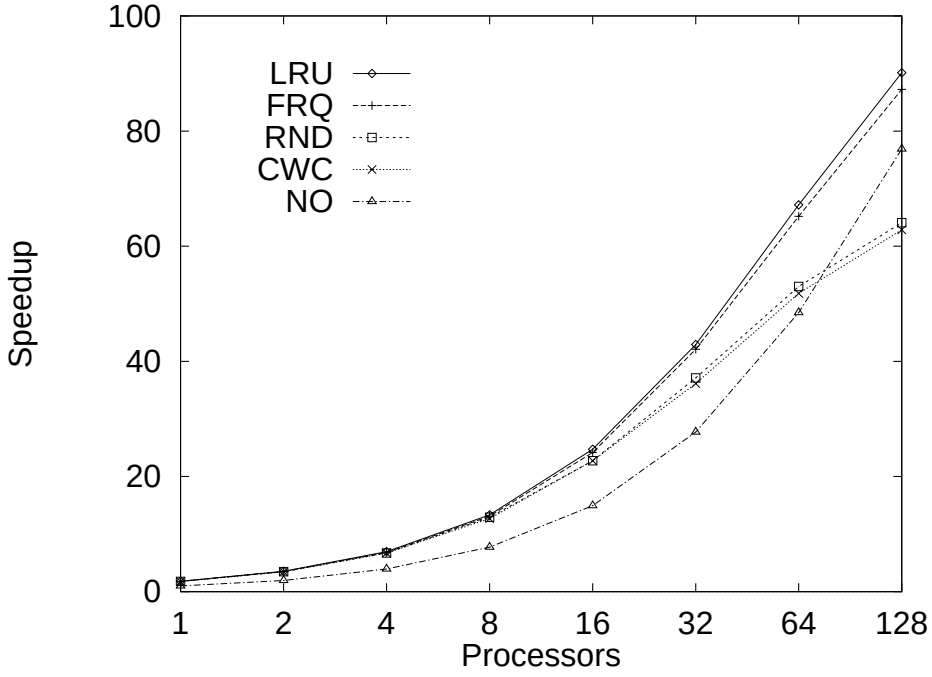

Figure 1: Scalability of the software cache

and FRQ remain better than without a cache for all p, with LRU being slightly faster than FRQ. Their curves slowly approach $s_{no}(p)$, but this might be partly caused by saturation, as the input graph used has only 3600 nodes to be moved, so with $p = 128$, each processor has to move just 28 nodes.

The miss ratios were close to 38 percent for all replacement strategies, hence the miss ratios cannot cause the different behavior. It seems to be caused either by different search times in the cache, or misses that occur concurrently for one strategy might lead to additional serialization. Further investigations are necessary to understand this behaviour.

As LRU turns out to be the best of the replacement strategies, we used it to compare locking and copying of cached entries. Figure 2 depicts the two speedup curves, processor numbers and cache sizes were chosen as before. The size of the cached entries is 9 words. Locking is 15 to 25 percent faster than copying, so it is a definite advantage in this application.

Last, we compared different cache sizes and organizations. Again, we used LRU as replacement strategy, and we fixed $p = 32$. Let $\widetilde{T_j}(k)$ denote the runtime with a cache of size k and j slots, so each slot is capable of holding k/j entries. Figure 3 depicts the speedup curves $s_j(k) = T_{no}(1)/\widetilde{T_j}(k)$ for $k = 2^i$, $i = 0, \ldots, 10$, and for $k = \infty$, i. e., a cache of unrestricted size. Note that for a cache with j slots, $k \geq j$.

For a fixed cache size k, $s_j(k)$ grows with j, if we do not consider the case $k = j$, where each cache slot can contain only one entry. This means, that for a cache of size k, one should choose $j = k/2$ slots, each capable of holding two entries. The only exception is $k = 16$. Here $j = 4$ is better than $j = 8$.

For fixed j, the performance improves up to a certain value of k, in our case $k = 4j$ or $k = 8j$. For larger cache sizes, the performance decreases again. Here, the searches through

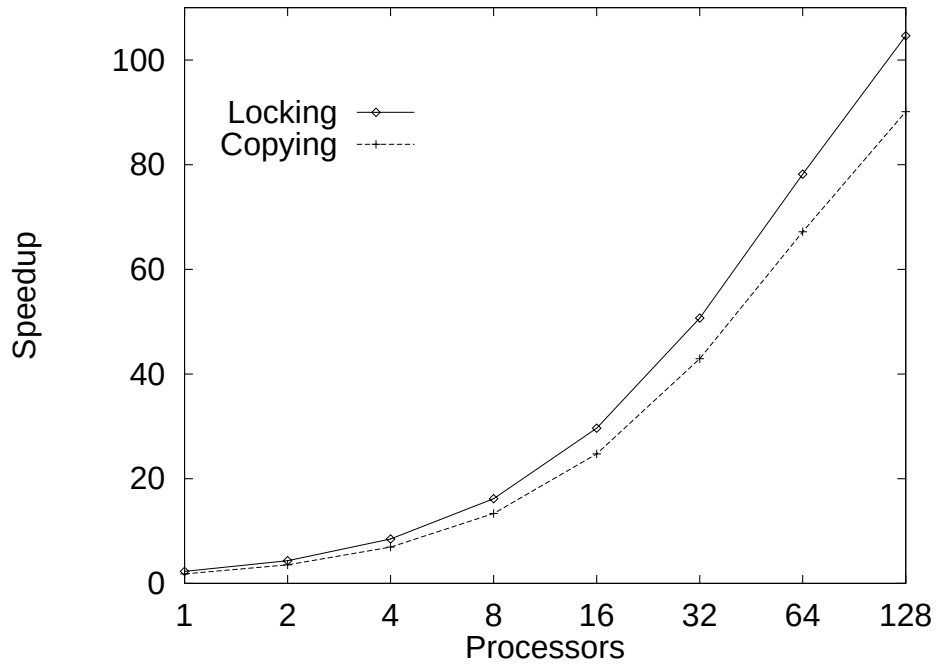

Figure 2: Comparison of Copying and Locking of Items

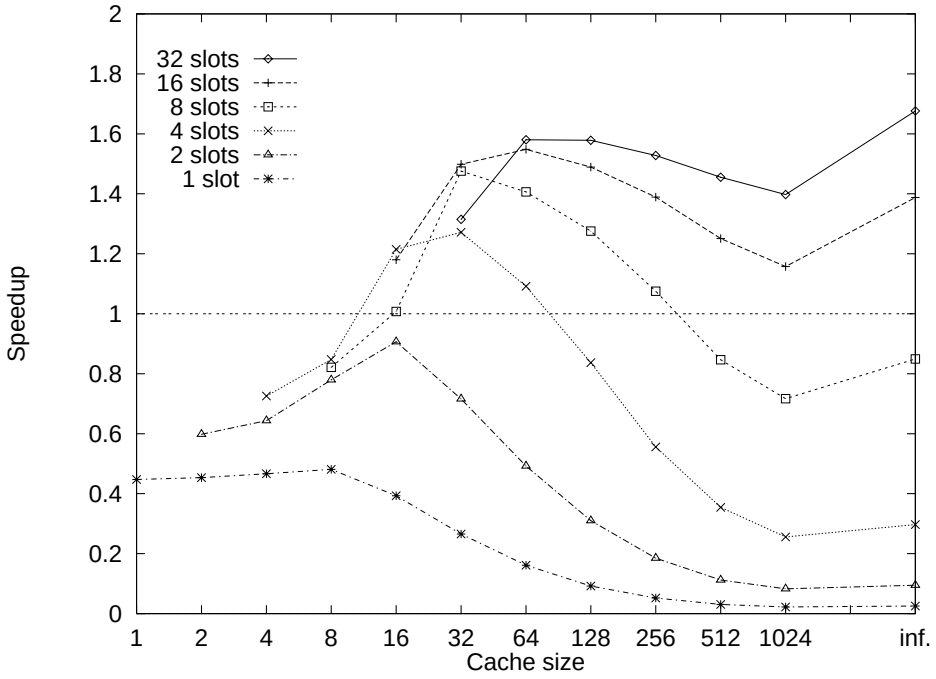

Figure 3: Comparison of Cache Sizes

longer lists need more time than caching of more entries can save. Astonishingly, if we give the cache an arbitrary size $k = \infty$, then the performance is increased again. The reason might be that from some k on, each entry is only computed once and never replaced. Note that the miss ratio remains constantly close to 38 percent for $j = 32$ and $k \geq 64$.

If the cache size is chosen too small, the speedup is less than 1, i. e., the program is slower than without cache for $k \leq 8$. For $k \geq 16$, the gain when doubling the cache size gets smaller with growing k. In this spirit, the choice $k = 64$ and $j = 16$ for the comparison of speedups was not optimal but a good choice.

Ordered versus unordered implementation shows a slightly better performance. Almost independently from the number of processors the ordered organization is about 3 percent faster than the unordered implementation.

For application Rayo we decided to implement only the cache with LRU replacement strategy. The decision is based on the fact, that usually the object which was found last is a good candidate as blocking object for the next intersection point. As we will see in the on-going, the optimal cache size is quite small, so one can infer that at least for the presented scenes the update strategy has not a large impact on performance. The results are presented for a scene of 104 objects and four light sources. Image resolution was set to 128×128, 16384 primary rays and 41787 secondary rays are traced. Four light sources make 205464 shadow rays necessary, 85357 of them hit a blocking object. We measured the hit and miss rates in the cache respective to the actually hitting rays, because if the shadow does not hit any object we cannot expect to find a matching cache entry. The cache can only improve the run time for hitting shadow rays, thus it can improve at most 32 percent of the run time. We focus only on the inner loop of the ray tracer, where more than 95 percent of the run time is spent.

We simulated application Rayo for $p = 2^i$ processors, $i = 0, \ldots, 7$. Figure 4 shows some relative speedups, where we varied the size and the number of slots. Let us denote with $T_x(p)$ the run time of the inner loop running on p processors. Here, x indicates the number of entries in the cache; s_1 is the relative speedup $T_0(p)/T_1(p)$, s_2 is the relative speedup $T_0(p)/T_2(p)$, and s_3 is the relative speedup $T_0(p)/T_4(p)$, respectively. Speedup s_4, the best one in figure 4, is obtained if we use one slot in the cache for every node in the ray tree. The size of the slot was set to only one entry. Increasing the slot size to two entries already led to a small loss of performance.

For small numbers of processors, a larger cache has some advantages, but with increasing number of processors the smaller cache becomes the better one. As the curve for s_4 implies, this is due to the conflicts during updating the cache. The processors are working at different levels in the ray tree and one single cache cannot provide the correct blocking object. As long as few processors are competing, the larger the cache the better the performance is. The search time in the larger cache together with the serialization during update has a negative impact on performance for a large number of processors. However, adapting the cache to the structure of the ray tree exhibits a large speedup s_4. Even for 128 processors a speedup of 13 percent has been achieved. Note that only 32 percent of the run time can be improved, thus, 40 percent of the run time during shadow determination has been saved.

For the run time of one single processor a slightly better update strategy was implemented,

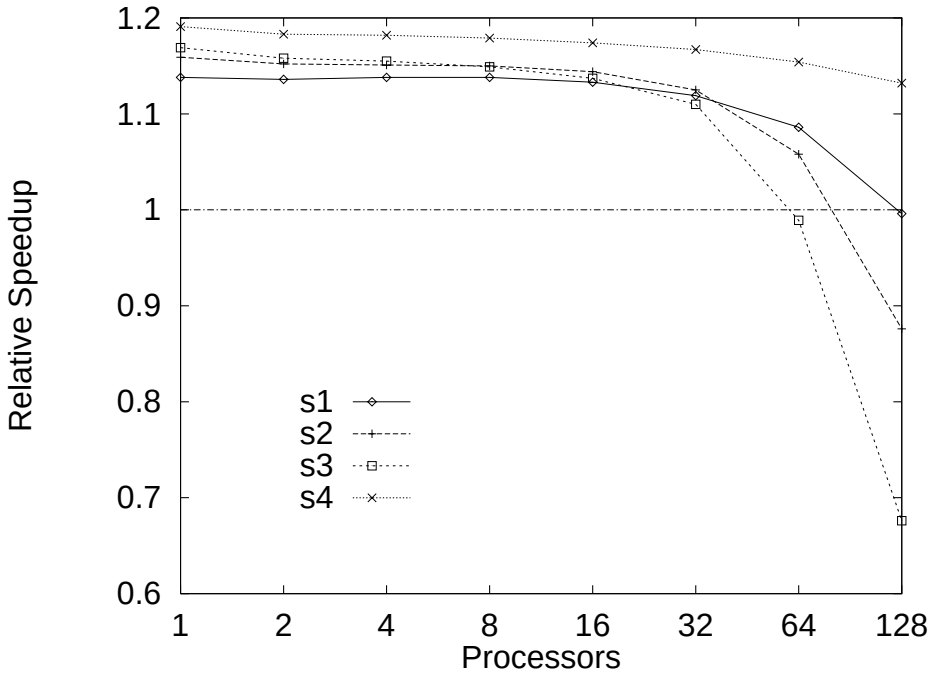

Figure 4: Relative Speedups for Different Cache Sizes and Numbers of Slots

because we can afford an update of the cache during every access. After a cache miss, the least recently used object is removed from the cache if the update function u does not provide a blocking object. This performs better for a single processor because after a shadow boundary has been passed, it is quite unlikely that the previous object which cast the shadow will be useful again. This method cannot be implemented when more than one processor is present because too many serializing writes to the cache would occur. Nevertheless, the run times in figure 4 demonstrate that the parallel cache even with the weaker replacement strategy outperforms the version with no cache.

Instead of sharing one data structure, one might provide each processor with its own cache. This leads to p times the memory size occupied by the cache structure, such that for large numbers of processors memory limitations may become problematic. Figure 5 shows that the hit rate for the parallel cache is significantly larger than the average hit rate for the individual caches. The difference increases with larger numbers of processors. The difference for one processor in the figure is explained by the alternative implementation of the replacement strategy. If the cache is owned by a single processor, we always deleted the least recently used object.

The large difference in the hit rates does not imply necessarily a large gain in run time, as it is illustrated in figure 6. The relative speedup between a version with individual caches and a version with a parallel cache is always close to one, but tends to be larger for 16 and 32 processors. Remembering that the cache improves at most the run time of 32 percent of the overall run time, in this portion of the program almost 5 percent are gained. The effect is due to the cache overhead and the serialization while updating. Nevertheless, the parallel cache saves memory and additionally improves the run time slightly.

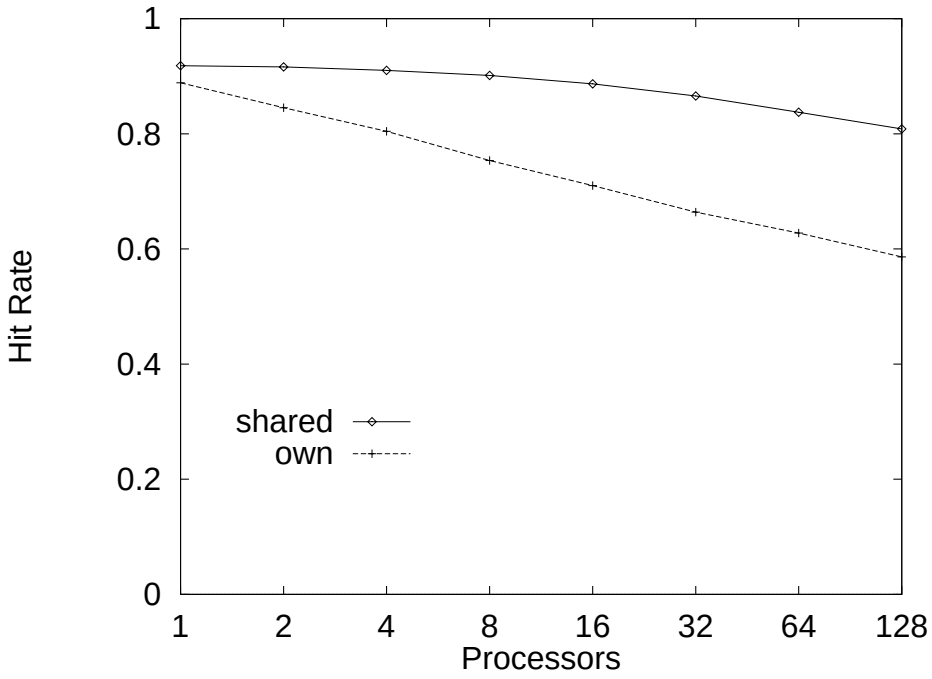

Figure 5: Hit Rates for Individual and Parallel Cache

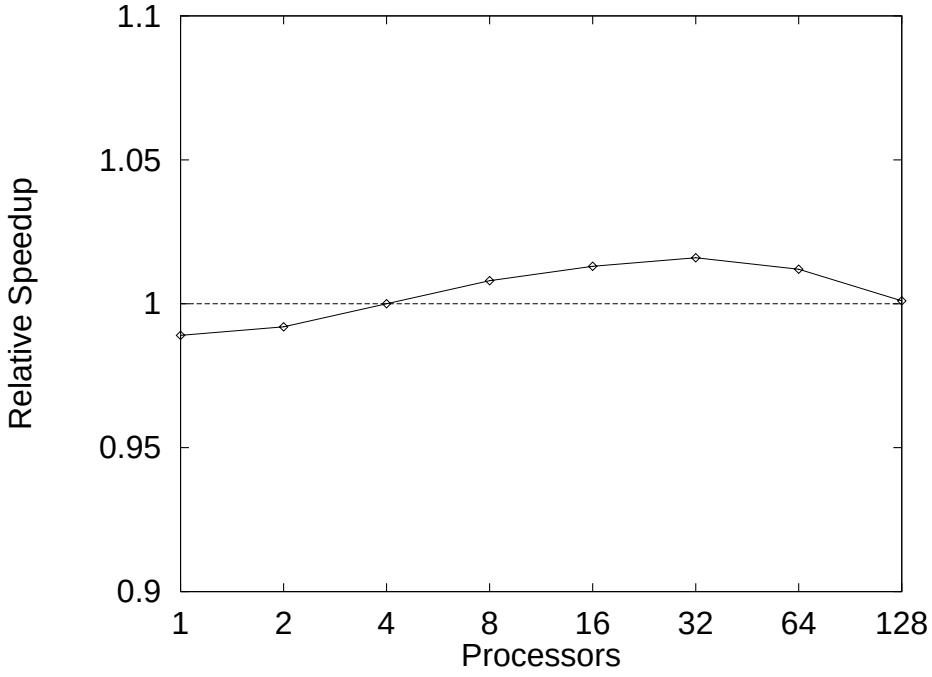

Figure 6: Relative Speedup for Individual and Parallel Cache

7 Conclusion

We introduced formally the concept of a parallel software cache and implemented the data structure on a parallel machine SB-PRAM. We demonstrated the advantages of such a cache. It uses less memory than individual caches for each thread would occupy. Moreover, for certain applications a faster execution time can be expected, because intermediate results are shared.

We adapted several classical replacement strategies, such as least recently used, least frequently used, random replacement, and $|C|$–way cache to the parallel case. A readers/writers lock which is used to protect the cache during update serializes all write accesses. Providing several slots in the cache which can be updated independently reduces the negative impact of the serialization. The modified LRU strategy was found to be best, at least in the presented applications FViewpar and Rayo.

Concurrent read and a fast lock structure are essential to achieve good performance. The parallel multi-prefix operation which is available in the SB-PRAM appeared to be a very powerful operation to implement the parallel data structure efficiently. As other machines such as Tera MTA, Stanford DASH and NYU Ultracomputer support such operations as well, the solution seems portable. The cache entries can be locked by a reader or copied to the readers' memory space. For the SB-PRAM locking performed better than copying. We analyzed several tradeoffs which occur in the design of parallel caches. For the ray tracer, we have seen that the version with a parallel cache outperforms the program where each thread uses its own individual cache. The parallel version both wins in run time—although only slightly—and in memory usage.

The shared memory data structure allows to hide many implementational details of the concept. The same functions can be used in a wide range of applications which at first sight do not exhibit a large amount of regularity. The dynamic behavior of the parallel cache can improve the run time as long as the parameters of the cache are carefully chosen. The concept of a parallel cache as a data structure might be useful for sequential programs consisting of several interacting threads as well. Here there might exist data exchange between the threads which is not predictable statically in advance. The parallel cache which is accessed by all threads might improve the efficiency of the program.

The SB-PRAM as simulation platform allows for a quantitative analysis, because as a UMA-architecture its performance is predictable and explainable. Once crucial parameters have been detected, the promising implementation can be ported to other shared memory architectures (NUMA) or even in certain cases to distributed memory machines.

References

[1] Ferri Abolhassan, Reinhard Drefenstedt, Jörg Keller, Wolfgang J. Paul, and Dieter Scheerer. On the physical design of PRAMs. *Computer Journal*, 36(8):756–762, December 1993.

[2] Ferri Abolhassan, Jörg Keller, and Wolfgang J. Paul. On the cost–effectiveness of PRAMs. In *Proceedings of the 3rd IEEE Symposium on Parallel and Distributed Processing*, pages 2–9. IEEE, December 1991.

[3] George S. Almasi and Allan Gottlieb. *Highly Parallel Computing*. The Benjamin/Cummings Publishing Company, Inc., 2nd edition, 1994.

[4] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera computer system. In *Proceedings of the 1990 International Conference on Supercomputing*, pages 1–6. ACM, 1990.

[5] Arno Formella. Ray Tracing Complex Scenes: Parallel or Sequential? In M.H̃. Hamza, editor, *Proceedings of 7^{th} IASTED/ISMM International Conference on Parallel and Distributed Computing and Systems*, pages 89–92. IASTED–Acta Press, October 1995.

[6] Arno Formella and Christian Gill. Ray Tracing: A Quantitative Analysis and a New Practical Algorithm. *The Visual Computer*, 11(9):465–476, December 1995.

[7] Arno Formella and Jörg Keller. Generalized Fisheye Views of Graphs. In *Proceedings Graph Drawing '95*, Lecture Notes in Computer Science, LNCS 1027, pages 242–253. Springer Verlag, December 1995.

[8] Geoffrey C. Fox, Roy D. Williams, and Paul C. Messina. *Parallel Computing Works*. Morgan Kaufmann, 1994.

[9] Allan Gottlieb, Ralph Grishman, Clyde P. Kruskal, Kevin P. McAuliffe, Larry Rudolph, and Marc Snir. The NYU ultracomputer — designing an MIMD shared memory parallel computer. *IEEE Transactions on Computers*, C–32(2):175–189, February 1983.

[10] Thomas Grün, Thomas Rauber, and Jochen Röhrig. The programming environment of the SB–PRAM. In *Proceedings of the 7th IASTED/ISMM International Conference on Parallel and Distributed Computing and Systems*, pages 504–509. Acta Press, 1995.

[11] Jim Handy. *The Cache Memory Book*. Academic Press, San Diego, CA, 1993.

[12] Jörg Keller, Wolfgang J. Paul, and Dieter Scheerer. Realization of PRAMs: Processor design. In *Proceedings WDAG '94, 8th International Workshop on Distributed Algorithms*, Lecture Notes in Computer Science 857, pages 17–27. Springer, September 1994.

[13] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam. The Stanford DASH multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[14] G.F. Pfister, W.C. Brantley, D.A. George, S.L. Harvey, W.J. Kleinfelder, K.P. McAuliffe, E.A. Melton, V.A. Norton, and J. Weiss. The IBM research parallel processor prototype (RP3): Introduction and architecture. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 764–771. IEEE, 1985.

[15] B.J. Smith. A pipelined shared resource MIMD computer. In *Proceedings of the 1978 International Conference on Parallel Processing*, pages 6–8. IEEE, 1978.