Parallel Software Caches*

Arno Formella[1] and Jörg Keller[2]

[1] Universität des Saarlandes, FB 14 Informatik, 66041 Saarbrücken, Germany
[2] FernUniversität-GHS, FB Informatik, 58084 Hagen, Germany

Abstract. We investigate the construction and application of parallel software caches in shared memory multiprocessors. To re-use intermediate results in time-consuming parallel applications, all threads store them in, and try to retrieve them from, a common data structure called parallel software cache. This is especially advantageous in irregular applications where re-use by scheduling at compile time is not possible. A parallel software cache is based on a readers/writers lock, i. e., multiple threads may read simultaneously but only one thread can alter the cache after a miss. To increase utilization, the cache has a number of slots that can be updated separately. We analyze the potential performance gains of parallel software caches and present results from two example applications.

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. However, many challenging applications lack the required amount of regularity. Another disadvantage of private software caches in massively parallel computers is the fact that for n threads n times as much memory is needed for software caching as in the sequential case.

Irregular applications, when run on shared memory multiprocessors (SMM), can benefit from a shared parallel software cache. By this term we mean one software cache in the shared memory, where all threads place their intermediate results and all threads try to re-use intermediate results, no matter by which thread they were computed. To allow for concurrent read and ensure exclusive write access of the threads to the cache, a readers/writers lock is used.

Control of multiple accesses of different types to data structures, e.g. by using readers/writers locks, have been investigated in the areas file systems and

* The first author was supported by the German Science Foundation (DFG) under contract SFB 124, TP D4.

databases, see e.g. [6, 11]. While in the first area the focus was on providing functionality such as files being opened by several threads or processes, the focus in the latter area was on developing protocols so that these accesses can be made deadlock free. We will show that in parallel software caches, no deadlock can occur. Our goal is to investigate the potential performance benefits possible from re-using intermediate results, and the tradeoffs that one encounters while implementing such a parallel data structure. We use the SB-PRAM [1] as platform, but the concept should be portable to other shared memory architectures (see Sect. 4).

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

2 Sequential Caches

2.1 Definitions

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 [8]. We adapted the concept of such an "intermediate memory" to the design of efficient shared memory data structures. Software caches can also be regarded as an implementation of the memorization concept in the field of programming languages, where a result for a function is stored rather than recalculated.

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, i. e., $m(k) = i$, if $(k, i) \in U$. Usually, m is a computationally expensive function, let us assume a time $t_m(k)$ to compute $m(k)$. 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, let us assume a time $t_c(k)$ to compute $h(k)$. $t_c(k)$ 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. u usually implements some replacement strategy. Let us assume a time $t_u(k)$ 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, otherwise it is called 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 $\alpha' = (j - s)/j$ is called *hit rate*. For a sufficiently large number of accesses, we can assume an average access time $t_c = 1/j \cdot \sum_{l=1}^{j} t_c(k_l)$. 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 $T_{no} = j \cdot t_m$ and with a cache it is

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

Hence, in case of worst miss rate, i. e., $\alpha = 1$, the run time is increased by a factor $T_c(1)/T_{no} = 1 + (t_c + t_u)/t_m$, and in best case, i. e., $\alpha = 0$, the run time is decreased by a factor $T_c(0)/T_{no} = t_c/t_m$. Thus, the cache improves the run time of j consecutive accesses to U if $T_c(\alpha) < T_{no}$. Clearly, the improvement of the entire program depends on how large the portion of the overall run time is which is spent in accessing U.

2.2 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.

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 an adversary can always find some update patterns which exhibit poor cache 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 the cache consists for LRU and FRQ in 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.

3 Parallel Caches

We assume that our applications are formulated in a task oriented model. The work to be done can be split in a large number of similar tasks which can be executed in parallel. A number of concurrent threads $p_1, p_2, \ldots, p_i$ is used in the parallel program. Each thread computes a task and then picks up a new one until all tasks have been done. Here we mean real parallel threads that are running simultaneously on at least i processors. Hence, we assume that our program can be optimally parallelized, with the possible exception of conflicts in the case of concurrent accesses to a parallel software cache. Also, there will be a sequential part before the spawning of parallel threads. The spawning incurs some overhead.

We assume that the threads might access the universe U in parallel executing function m without restrictions, and that the access time t_m in the average does not depend on a specific thread nor the access.

3.1 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. In the worst case all reads and writes to the cache are serialized. Often however, a more efficient solution is possible, because many SMMs efficiently handle concurrent read accesses, i.e. CC-NUMA architectures.

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 inspects first 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 been inserted already in the cache during its calculating and waiting time. By moving the execution of address function m outside the region protected by the readers/writers lock, we can guarantee that our protocol is deadlock free: while a thread is reader or writer, it only executes code that works on the cache. It cannot execute other functions that might again try to access the lock, and which could lead to a deadlock.

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. To implement concurrent access to the lock data structure and to the reader and writer queues in a constant number of steps (i. e., without additional serialization), parallel prefix can be used. Thus, a machine with atomic parallel prefix and atomic concurrent access only serializes multiple writers (an example is the SB-PRAM, see Sect. 4).

3.2 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. We assume that a thread will use the cached information, continue and access the cache again only at 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. If entries are locked while they are used, we have to think about possible deadlocks. However, as long as the application fulfills the above condition (each thread uses only one cached entry at a time) the protocol is deadlock free.

If all accesses to the cache use entries for about the same amount of time, then one can decide by example runs whether to copy or to lock entries. It depends on the application which one of the two methods lead to higher performance, 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.

3.3 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 from subsection 2.2 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 data structure is possible. The reader would need writer permissions and this would serialize all accesses. In our parallel version of LRU, every reader updates the time stamp of the entry that was found as a hit. Previously to the update, a writer sorts all entries in the cache according to the time stamps. The least recently used 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 rearranges the list and 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 since insertion and n is the total number of accesses to the cache. It appears a similar tradeoff between the sorting time and search time as for LRU.

Here arises the question whether the whole lifespan of a cached entry must be considered. 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 use 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.

3.4 Performance Prediction

We want to predict the performance of the parallel cache by profiling the performance of the software cache in a sequential program. To do this, we assume that the sequential program consists of a sequential part s and a part p that can be parallelized. We assume that all accesses to the universe (and, if present, to the software cache) occur within part p. We assume that the work in that part can be completely parallelized (see task model in Sect. 3). Thus, the time to execute

the sequential program without a cache is $T_{seq}^{no} = T_s + T_p$, where T_s is the time to execute the sequential part and T_p is the time to execute the parallelizable part. The time $T_{no} = j \cdot t_m$ to access the universe is a fraction $1 - \beta$ of T_p, i.e. $T_{no} = (1-\beta) \cdot T_p$. This means that time $\beta \cdot T_p$ in part p is spent without accessing the universe. Then the time to execute the sequential program when a cache is present is $T_{seq}^{with} = T_s + \beta \cdot T_p + T_c(\alpha)$, where α is the miss rate of the cache as described in Sect. 2. By profiling both runs of the sequential program, we can derive T_s/T_{seq}^{no}, T_p/T_{seq}^{no}, β, α, j, t_u/t_m, t_c/t_m. With $T_{no} = j \cdot t_m = (1 - \beta) \cdot T_p$, the value of t_m/T_{seq}^{no} can be computed from the other parameters.

If we run our parallelized program without a cache on a parallel machine with n processors, then the runtime will be $T_{par}^{no}(n) = T_s + T_p/n + o$, where o is the overhead to create a set of parallel threads. It is assumed to be fixed and independent of n. We derive o/T_{seq}^{no} by running the sequential program on one processor, the parallel program on n processors of the parallel machine and solving $T_{par}^{no}(n)/T_{seq}^{no} = T_s/T_{seq}^{no} + T_p/(n \cdot T_{seq}^{no}) + o/T_{seq}^{no}$ (which is the inverse of the speedup) for the last term. Thus, we only need the runtimes of the programs and need not be able to profile on the parallel machine.

The runtime of the parallelized program with a parallel software cache will be $T_{par}^{with}(n) = T_s + \beta \cdot T_p/n + T_c(\alpha)/n + o + w(n)$. The term $w(n)$ represents the additional time due to the serialization of writers. In the best case, $w(n) = 0$. Now, we can compute a bound on the possible speedup. Here, we assume that the miss rate will be the same for the sequential program and each thread of the parallel program, which will be supported by our experiments.

4 Execution Platform

The results presented in Sect. 5.2 have been obtained on the SB-PRAM, a shared memory multiprocessor simulating a priority concurrent read concurrent write PRAM [1]. n physical processors are connected via a butterfly network to n memory modules. A physical processor simulates several virtual processors, 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. Thus, the user sees all virtual processors run in parallel. Accesses to memory are distributed with a universal hash function so memory congestion is avoided. The network is able to combine accesses on the way from the processors to the memory location. This avoids hot spots and is extended to employ parallel prefix operations which allow to implement very efficient parallel data structures without serialization, e.g. a readers/writers lock.

A first prototype with 128 virtual processors is operational [3]. 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. Each virtual processor executes one thread. The predicted run times matched exactly with the run times obtained by simulation. We did no simulations with more

virtual processors, because our workstations did not have enough memory to run such simulations.

Several other multiprocessors provide hardware support for parallel prefix operations: NYU Ultracomputer [7], IBM RP3 [10], Tera MTA [2], and Stanford Dash [9]. The presented concepts should be transferable to these machines. The DASH machine provides a cache-coherent virtual shared memory (CC-NUMA). Here, it would be useful for performance reasons to consider the mapping of the software cache to the hardware caches when designing size and data structures of the cache.

5 Experiments

5.1 Applications

A software cache is part of an application. Hence, we did not test its performance with standard cache benchmarks, but decided to use real applications.

Application FViewpar [5] 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 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 it is not given explicitly. 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. To implement a cache with multiple slots application FViewpar maps a horizontal scanline s given as $y = c$ to slot $g(s) = c \bmod j$, where j is the number of slots.

Application Rayo [4] is a ray tracer. It is parallelized with a parallel queue over all pixels. 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.

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. 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.

5.2 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.

First, we ran the sequential version of FViewpar without cache and found that the sequential part only comprises $T_s/T_{seq}^{no} = 0.004$ of the runtime. Thus with Amdahl's law the speedup can be 250 at most. The parallelizable part consumes the remaining $T_p/T_{seq}^{no} = 0.996$ of the runtime. In part p, a fraction $\beta = 0.121$ is spent without accessing the universe. The function m was executed $j = 6913$ times. When we used a software cache, we found that $t_u/t_m = 0.05$ and that $t_c/t_m = 0.071$. t_c is a bit larger than t_u because of the copying. The miss rate was $\alpha = .355$. By running the parallel program and the sequential program without a cache on the SB-PRAM, we found $o/T_{seq}^{no} = 0.00137$. Now we ran the program with a parallel software cache for $n = 2, 4, 8, \ldots, 128$. We computed $w(n)/T_{seq}^{no}$ from the program runtimes. For increasing n, the value approaches 0.0023 from above.

Then, we simulated application FViewpar for $n = 2^i$ 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 $T_{par}^{x}(n)$ denote the runtime on n processors with replacement strategy x, where no indicates that no cache is used. Figure 1 depicts the speedups $s_x(n)$, where $s_x(n) = T_{seq}^{no}/T_{par}^{x}(n)$, for $x = no, lru, frq, rnd, cwc$. MAX denotes the maximum speedup which is possible by assuming a hit rate of 100% and $w(n) = 0$, while all other parameters are as before.

For $n = 1$, all replacement strategies give a runtime improvement by a factor of about 1.8. As n increases, the curves fall into two categories. RND and CWC strategies provide less improvement, until they make the application slower than without cache for $n = 128$. LRU and FRQ remain better than without a cache, with LRU being slightly faster than FRQ. Their curves slowly approach $s_{no}(n)$, but this might be partly caused by saturation, as the input graph used has only 3600 nodes to be moved, so with $n = 128$, each processor has to move just 28 nodes. Also, as $s_{no}(n)$ approaches the maximum possible speedup of 250 and

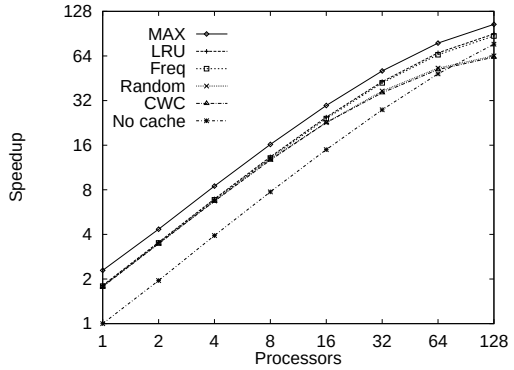

Fig. 1. Scalability of the software cache

hence $s_{max}(n)$, not much can be gained anymore from using a cache. Thus, at some larger number of processors the overhead in using a cache will be larger than the gain. This is supported by computing $s_{no}(256)$ and $s_{LRU}(256)$ with the formulas from Sect. 3.4. The program without a cache is predicted to be slightly faster than with a cache[3]. The miss rate for LRU was $\alpha = 0.355$ for $n \leq 32$ and sank to 0.350 for $n = 128$.

As LRU turns out to be the best of the replacement strategies, we used it to compare locking and copying of cached entries. 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.

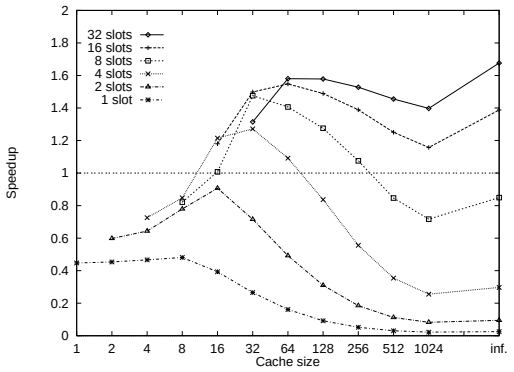

Fig. 2. Comparison of Cache Sizes

[3] This behavior has also been observed in a simulation with 256 processors, done by J. Träff, Max-Planck-Institute for Computer Science, Saarbrücken, Germany.

Last, we compared different cache sizes and organizations. Again, we used LRU as replacement strategy, and we fixed $n = 32$. 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 2 depicts the speedup curves $s_j(k) = T^{no}_{par}(32)/\widetilde{T_j}(k)$ 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 longer lists need more time than caching of more entries can save. If we give the cache an arbitrary size $k = \infty$, then the performance is increased again. The reason seems to be that from some k on, each entry is only computed once and never replaced. Note that the miss rate remains constantly close to 35.5 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.

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 sequel, 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 $n = 2^i$ processors, $i = 0, \ldots, 7$. Figure 3 shows some relative speedups, where we varied the size and the number of slots. Let us denote with $T_x(n)$ the run of the inner loop running on n processors. 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. s_4, the best one in Fig. 3, 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.

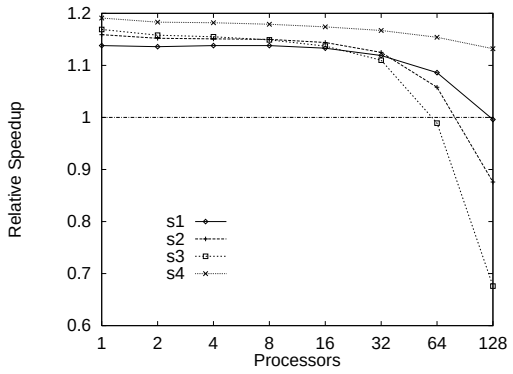

Fig. 3. Relative Speedups for Different Cache Sizes and Numbers of Slots

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, 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. Nevertheless, the run times in Fig. 3 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 n times the memory size occupied by the cache structure, such that for large numbers of processors memory limitations may become problematic. Figure 4 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

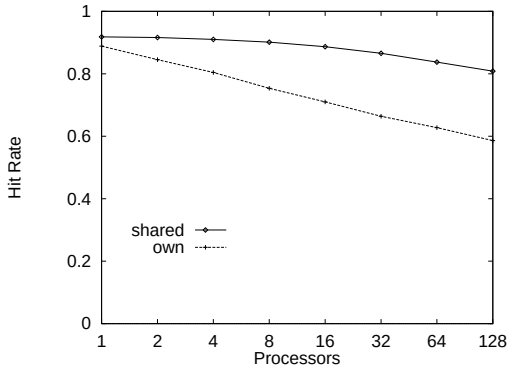

Fig. 4. Hit Rates for Individual and Parallel Cache

in run time, as it is illustrated in Fig. 5. 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 slightly improves the run time.

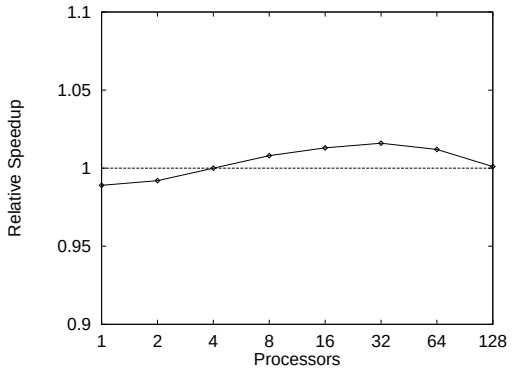

Fig. 5. Relative Speedup for Individual and Parallel Cache

6 Conclusion

We introduced the concept of a parallel cache and implemented the data structure on the SB-PRAM multiprocessor. In our applications, the software cache

leads to performance improvements, but investigations on more diverse workloads are necessary. Providing several slots in the cache which can be updated independently reduces serialization after cache misses. The modified LRU strategy together with locking of requested entries was found to be best in the presented applications. The other parameters have to be chosen carefully, too.

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 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 should be portable to other shared memory architectures.

References

1. Abolhassan, F., Drefenstedt, R., Keller, J., Paul, W. J., Scheerer, D.: On the physical design of PRAMs. Comput. J. **36** (1993) 756–762
2. Alverson, R., Callahan, D., Cummings, D., Koblenz, B., Porterfield, A., Smith, B.: The Tera computer system. In Proc. Int.l Conf. on Supercomputing (1990) 1–6
3. Bach, P., Braun, M., Formella, A., Friedrich, J., Grün, T., Lichtenau, C.: Building the 4 Processor SB-PRAM Prototype. In Proc. Hawaii Int.l Symp. on System Sciences (1997) 14–23
4. Formella, A., Gill, C.: Ray Tracing: A Quantitative Analysis and a New Practical Algorithm. Visual Comput. **11** (1995) 465–476
5. Formella, A., Keller, J.: Generalized Fisheye Views of Graphs. Proc. Graph Drawing '95 (1995) 242–253
6. Fussell, D.S., Kedem, Z., Silberschatz, A.: A Theory of Correct Locking Protocols for Database Systems. Proc. Int.l Conf. on Very Large Database Systems (1981) 112–124
7. Gottlieb, A., Grishman, R., Kruskal, C. P., McAuliffe, K. P., Rudolph, L., Snir, M.: The NYU ultracomputer — designing an MIMD shared memory parallel computer. IEEE Trans. Comput. **C–32** (1983) 175–189.
8. Handy, J.: The Cache Memory Book. Academic Press, San Diego, CA (1993)
9. Lenoski, D., Laudon, J., Gharachorloo, K., Weber, W.-D., Gupta, A., Hennessy, J., Horowitz, M., Lam, M. S. The Stanford DASH multiprocessor. Comput. **25** (1992) 63–79
10. Pfister, G.F., Brantley, W.C., George, D.A., Harvey, S.L., Kleinfelder, W.J., McAuliffe, K.P., Melton, E.A., Norton, V.A., Weiss, J: The IBM research parallel processor prototype (RP3): Introduction and architecture. In Proc. Int.l Conf. on Parallel Processing (1985) 764–771
11. Silberschatz, A., Peterson, J. L., Galvin, P. B.: Operating System Concepts, 3rd Edition. Addison-Wesley, Reading, MA (1991)

This article was processed using the LaTeX macro package with LLNCS style