Proceedings of the 2004 Winter Simulation Conference
R. G. Ingalls, M. D. Rossetti, J. S. Smith, and B. A. Peters, eds.

APPROXIMATE TIME-PARALLEL CACHE SIMULATION

Tobias Kiesling

Fakultät für Informatik
Universität der Bundeswehr München
85577 Neubiberg, GERMANY

ABSTRACT

In time-parallel simulation, the simulation time axis is decomposed into a number of slices which are assigned to parallel processes for concurrent simulation. Although a promising parallelization technique, it is difficult to be applied. Recently, using approximation with time-parallel simulation has been proposed to extend the class of suitable models and to improve the performance of existing models. In trace-driven cache simulation, sequences of memory requests are processed to determine the performance of variously sized caches. Time-parallel simulation has been applied to trace-driven cache simulation, but only with limited scalability of the parallel algorithm. In order to solve the scaling problem, this work uses approximation with time-parallel cache simulation. Although introducing an uncertainty in the results, the approximate algorithms work in a way that result accuracy increases monotonically with time, allowing a direct control of the quality of results. Experiments with a prototypical implementation indicate the viability of this approach.

1 INTRODUCTION

In classical parallel discrete-event simulation (Fujimoto 1990), the set of state variables of a model is divided into subsets and the corresponding simulations are executed concurrently. Time-parallel simulation takes a different approach by dividing the simulation time into intervals and executing the simulations of these intervals in parallel (Chandy and Sherman 1989). This is a straightforward method of parallelization that is not restricted by the decomposability of the state space in the model. Unfortunately, the *state-match problem* is inherent to time-parallel simulation: the simulation of every time interval has to start execution with an unknown initial state, which will only be known after the simulation execution of the preceding time interval, performed concurrently. Several solutions to the state match problem have been proposed, which are correct in the sense that they guarantee consistent state changes at time interval boundaries: regeneration points (Lin and Lazowska 1991), fix-up computations (Heidelberger and Stone 1990), and recurrence relations (Greenberg, Lubachevsky, and Mitrani 1991).

Alternatively, the concept of *approximate state matching* (Kiesling and Pohl 2004) is a solution of the state match problem that does not preserve correct state changes. This can lead to a significant increase of simulation efficiency, but also introduces an error or uncertainty in the simulation results. For many simulation applications, a small error that does not cross a predefined threshold might be tolerable. Therefore, a mechanism for error control is proposed in (Kiesling and Pohl 2004) and tested successfully for the simple model of an M/M/1 queue.

Simulation of computer caches, with the most important replacement policy of *least recently used* (LRU), has been topic of research for several decades, leading to the development of efficient LRU simulation algorithms, where hit or miss rates are determined for given memory address traces. Different schemes for the parallelization of these algorithms exist for single instruction multiple data (SIMD) (Nicol, Greenberg, and Lubachevsky 1994) and multiple instruction multiple data (MIMD) machines (Heidelberger and Stone 1990). On MIMD machines, the method of temporal parallelization is applied, by splitting a memory address trace into several subtraces that are used as input for parallel cache simulations. Empirical studies (Nicol and Carr 1995) show, that the SIMD algorithms exhibit the best results for small cache sizes, but that the MIMD algorithms are better in the overall case.

In the time-parallel approach, the cache contents at the subtrace boundaries are not known a priori, wherefore they have to be guessed initially and corrected by the use of fix-up computations after a first simulation phase. Depending on the specificities of the input trace and the size of the cache, these fix-up computations can lead to an increase of the parallel simulation runtime to that of the corresponding sequential simulation, or even worse. Experiments with this approach (Nicol and Carr 1995) indicate that the speedup for a small number of processors is excellent, but degenerates with a higher number. Approximation can be applied here

to decrease the cost of the fix-up-computation phase or to avoid it altogether.

The next section discusses several aspects of cache simulation and presents the basic time-parallel MIMD cache simulation algorithms. Section 3 shows how to apply approximate state matching to parallel cache simulation. Results of experiments with the introduced approach are presented in Section 4. Finally, Section 5 concludes the work.

2 CACHE SIMULATION

Simulation of computer caches is an important technique that is widely used by hardware designers to decide on appropriate cache implementations. The behavior of a cache is largely influenced by its *replacement policy*, which determines the page to be removed if the cache is full and a new page has to be loaded. The most important policy for computer caches is *least-recently-used* replacement (LRU), due to its simplicity and yet good results. Therefore, this paper focuses on the simulation of LRU caching.

2.1 Least-Recently-Used Caching

Basic simulation of caching with the LRU policy is straightforward. A data structure for the cache is initialized for a given cache size and an input trace is processed, updating the LRU cache appropriately and recording the number of hits.

This has the disadvantage, that for every cache size a new simulation execution is necessary. Therefore, an approach was introduced to calculate the hit rates for any number of cache sizes in a single pass over the input trace (Mattson, Gecsei, Slutz, and Traiger 1970): simulation is performed as usual, with the exception that the cache size is supposed to be unbounded (i.e. no replacement occurs). For every request, the position of the corresponding page in the LRU stack is recorded as the *stack distance* of the request, which is the minimal size of a cache, such that the request can be served from it. A *distance table* is used to record stack distances. After processing the input trace, the entries of the table can be cumulated to give the *success function* of the simulation, which is defined as $S(c) = \sum_{i=1}^{c} D_i$, where c is the cache capacity for which to calculate <the number of hits and D_i is the number of occurrences of stack distance i in the simulation of the input trace.

Example 2.1 *The determination of stack distances for a simple input trace is illustrated in Figure 1. It shows the processing of the trace of page requests where in every time step the stack distance of a request is determined from the current stack and the stack is changed afterwards (either by moving the requested page to the top or by pushing it, if it did not yet appear). Note that in Figure 1, the LRU stack is shown as it appears after processing the corresponding request. The stack distance is either the position of the requested page in the LRU stack, or ∞ if the page is not yet present. Table 1 shows the distance table for the example input trace at the end of processing, which is used to determine numbers of hits for specific cache sizes. E.g., four hits are scored for a cache size of three, which is the sum of the entries of distance three and lower.*

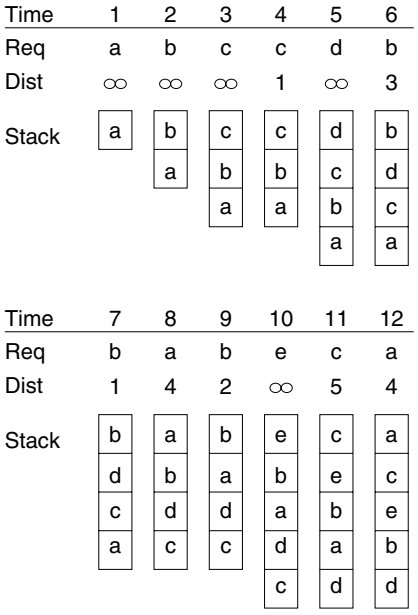

Figure 1: LRU Stack Processing

Example 2.1 leads to two important observations: (i) the determination of stack distances is a generalization of simple LRU cache simulation, as the number of hits for a given input trace is directly related to the stack distances of requests in the trace, and (ii) the stack distance of a request to a specific page is the number of unique requests between the current request and the previous occurrence of a request to the same page plus one. In Example 2.1, the stack distance of the request to page b at time 6 is 3, which is the number of unique requests (c and d) between time 6 and time 2 (the last occurrence of a request to page b) plus one.

Various ways of implementing the LRU stack have been proposed: linked lists (Mattson, Gecsei, Slutz, and Traiger 1970), hash tables (Bennett and Kruskal 1975), and search trees (Olken 1981). An overview of the implementations together with discussions about their strengths and weaknesses can be found in (Thompson 1987).

Table 1: Stack Distances after Processing

Dist	1	2	3	4	5	∞
Count	2	1	1	2	1	5

2.2 Simple Parallel Cache Simulation

Let $T = (t_1, \ldots, t_n)$ be the input trace which is a sequence of n requests. For parallel simulation, T is divided into m non-overlapping subtraces $T_1, \ldots, T_m$. The subtraces do not necessarily have the same length, although this might be preferable for a balanced load distribution among processors.

In the basic time-parallel simulation approach, every subtrace is assigned to a separate processor for simulation, which is performed in several phases: in the *initial simulation* phase, the input subtraces are processed once with an empty initial cache, yielding an incorrect value of the overall number of hits; in the *fix-up computation* phases, resimulation of parts of the subtraces is performed until the correct value for the number of hits is determined. For every pass over a subtrace, the cache contents that have been determined by the simulation of the directly preceding subtrace in the previous pass are used as initial cache. The length of this phase can range from a partial pass over the subtrace to $m - 1$ passes in the worst case.

Note, that it is sufficient to include those requests in the resimulation subtrace that could not be served from the cache and only up to the time where the cache is filled for the first time. Although this leads to an incorrect order of pages in the LRU stack during resimulation, the number of hits are calculated correctly.

Example 2.2 *Let $T = (a, b, c, c, d, b, b, a, b, e, c, a)$ be a sequence of page requests, which is used as input of a cache simulation for a cache size of 3. T is split into two subtraces $T_1 = (a, b, c, c, d, b)$ and $T_2 = (b, a, b, e, c, a)$ for parallel simulation on two processors. As T_1 is the first subtrace, resimulation is not necessary. Two hits are recorded and the cache contents after processing are b, d, c in order of most recently to least recently used. At the same time, simulation of T_2 is performed with an (incorrect) empty initial cache. Therefore, all misses occurring before the cache has been filled for the first time have to be reconsidered for correct simulation results. As the cache of size 3 is filled after the fourth request in T_2, the subtrace b, a, e has to be resimulated with cache contents b, d, c.*

In the original simulation approach, resimulation is performed by the same processor that created the resimulation subtrace with the final cache resulting from the simulation of the previous subtrace. Therefore, resimulation can occur only after a simulation phase has been completed for both subtraces. This is implemented by performing simulation in strict phases, using barrier synchronization of all processes between phases. However, synchronization among processors can be relaxed when processing is changed slightly.

Consider Example 2.2, where the responsibility for processing the resimulation subtrace of T_2 can be pushed to the processor that is responsible for the simulation of T_1. No transfer of cache contents is necessary here. The processor just continues simulation as if the resimulation subtrace is part of its initial subtrace T_1.

If subtraces are fed to processors through input queues, the fact that resimulation is performed is transparent to the processors and synchronization can be minimized: every processor processes page requests from its input queue until the special symbol $\perp$, signifying end of the input trace, is read. Instead of recording page requests to be resimulated in a separate data structure, they are put directly into the input queue of the preceding processor. Synchronization is performed implicitly by the blocking of processors on an empty input queue. A processor knows that it has finished simulation if it encounters the $\perp$ symbol, passing it on to the previous processor's input queue. In this case, the last processor never performs any resimulation steps, stopping execution as soon as its initial subtrace has been completely processed. This can easily be achieved by putting $\perp$ at the end of the input queue of the last processor.

This approach is summarized in Algorithm 1. Parallelism is introduced by use of the construct

for $l \le i \le u$ **pardo** *statement*

adopted from (JáJá 1992), which indicates concurrent execution of *statement* for every $i \in \{l, \ldots, u\}$. Several functions are used for convenience in the algorithm. ENQUEUE and DEQUEUE execute the corresponding operations on a processor's input queue, PUSH places a request at the top of an LRU stack, REPOSITION moves a request from anywhere in a given stack to the top, and REPLACE removes the bottom request in a stack and puts a new request on top.

Input: *Subtraces $T_1, \ldots, T_m$ and cache size s_{max}. The stack S_i for every subtrace T_i is initially empty. The input queue Q_i for every processor is initialized with the input subtrace T_i. Additionally, $\perp$ is put at the end of Q_m. The hit counters h_i are initialized to 0.*
Output: *The sum of the number of hits h_i of every processor.*

```
begin
    for 1 ≤ i ≤ m pardo
        req = DEQUEUE(Q_i)
        while req ≠ ⊥ do
            if req ∈ S_i then
                REPOSITION(S_i, req)
                h_i = h_i + 1
            elsif |S_i| < s_max then
                PUSH(S_i, req)
                if i > 1 then ENQUEUE(Q_{i-1}, req)
            else
                REPLACE(S_i, req)
            req = DEQUEUE(Q_i)
        if i > 1 then ENQUEUE(Q_{i-1}, ⊥)
end
```

Algorithm 1: Simple Time-Parallel Cache Simulation

2.3 Parallel Full LRU Stack Simulation

Although mentioned in (Heidelberger and Stone 1990) that it is possible to calculate stack distances with the basic time-parallel simulation approach, details were only provided at a later time (Nicol and Carr 1995).

With the simple time-parallel cache simulation structured as in Algorithm 1, it can easily be extended to calculate the success function instead of the number of page hits with three basic changes: every page that has been requested at least once is present in the LRU stack used to determine stack distances (pages are never removed); instead of the absolute number of page hits, stack distances have to be recorded; and for all subtraces, except of the first, stack distances cannot be correctly determined if the corresponding request is not in the local LRU stack, wherefore the processing of these requests has to be delegated to the directly preceding processor.

To see why this is correct, observation (ii) from Section 2.1, relating the stack distance to the number of different occurrences of requests between the current request and its previous occurrence, must be considered. First, the observation implies that in the parallel processing approach, all stack distances whose corresponding requests are found in the LRU stack can be determined correctly. This is due to the fact that the previous occurrence of the current request, as well as all of the different requests in between, have been processed. Second, all of the first occurrences of requests to the same page cannot be determined by the current processor (except of the first), wherefore they are sent to the preceding processor in correct order. Intermediate requests with already determined distances can be skipped, as for the calculation of stack distances only the first occurrence of a request is relevant.

Example 2.3 *Consider the sequence of page requests of Example 2.1, which is split into two subsequences* $T_1 = (a, b, c, c, d, b)$ *and* $T_2 = (b, a, b, e, c, a)$ *to be fed to two processors,* P_1 *and* P_2*, for the determination of stack distances. Recall the correct sequence of stack distances for* T*, which is* $(\infty, \infty, \infty, 1, \infty, 3, 1, 4, 2, \infty, 5, 4)$*. The distances for* T_1 *are correctly calculated by* P_1*. In the simulation of* T_2*, the distances for requests 3 and 6 are determined to 2 and 4. The rest of the requests is sent to* P_1 *for resimulation, as* P_2 *cannot determine the correct distances.*

Algorithm 2 shows the parallel processing to determine the stack distances of an input trace. In addition to the functions introduced with Algorithm 1, here the function POSITION is used, which returns the position of a request in the given LRU stack.

Input: *Subtraces* $T_1, \ldots, T_m$*. The stack* S_i *for every subtrace* T_i *is initially empty. The input queue* Q_i *of every processor is initialized with the input subtrace* T_i*. Additionally,* $\perp$ *is put at the end of* Q_m*. The distance tables* D_i *are initialized with 0 for every possible stack distance.*
Output: *The overall distance table* $D_{overall}$*, which is the sum of the distance tables* D_i *for all processors* i*.*

```
begin
    for 1 ≤ i ≤ m pardo
        req = DEQUEUE(Q_i)
        while req ≠ ⊥ do
            if req ∈ S_i then
                dist = POSITION(S_i, req)
                D_i[dist] = D_i[dist] + 1
                REPOSITION(S_i, req)
            else
                PUSH(S_i, req)
                if i > 1 then ENQUEUE(Q_{i-1}, req)
            req = DEQUEUE(Q_i)
        if i > 1 then ENQUEUE(Q_{i-1}, ⊥)
end
```

Algorithm 2: Full LRU Stack Simulation

3 APPROXIMATION

As introduced in the previous section, there are two approaches for the simulation of LRU caches. The simple approach determines the hit rate for a given trace and cache size. The more general full-stack approach calculates the stack distances of the requests in the input trace, which can be used to implement a success function that returns the number of hits for a given cache size. Here, the algorithms presented in Section 2 are changed to calculate intervals for the number of hits rather than exact values. Depending on simulation needs, these intervals can be used to get approximate results in a much shorter execution time. The following two sections focus on two different aspects of approximate calculation: how approximate simulation results can be determined, and which criteria can be used to decide on the appropriate time to finish simulation execution.

3.1 Determining Result Bounds

Few effort is required to change Algorithms 1 and 2 to give approximate simulation results during any time of resimulation. However, the calculations presented here only work correctly when execution has been stopped, either permanently (i.e. the simulation is finished), or temporarily for the calculation of the current simulation accuracy.

3.1.1 Simple Cache Simulation

It was already mentioned in (Heidelberger and Stone 1990), that upper and lower bounds on the number of hits are calculated easily at the end of a simulation iteration, where simulation can be stopped if bounds are tight enough. With the parallel simulation algorithm of the previous section, approximate results can be calculated at any time, leading to a higher flexibility for the decision on simulation termination.

In Algorithm 1, upper and lower bounds for the number of hits can be calculated without further mechanisms, if all processors stop execution at least until the calculation of bounds is complete. The minimal number of hits at that time $h_{min} = \sum_{i=1}^{m} h_i$ (where m is the number of processors) is just the sum of the hits recorded by all processors so far. Additionally, every request still in a processor's input queue might be a hit, although this is not known yet. Therefore, the maximal number of hits $h_{max} = h_{min} + \sum_{i=1}^{m} |Q_i|$ is derived from the number of all requests for which the status (hit or miss) could not yet be determined.

Example 3.1 *In Example 2.2, the two subtraces $T_1 = (a, b, c, c, d, b)$ and $T_2 = (b, a, b, e, c, a)$ are simulated by two processors P_1 and P_2. After the requests of both traces have been processed exactly once, a lower bound on the number of cache hits of 4 can be calculated, which are two hits recorded by P_1 for requests 4 and 6 of T_1 and two hits recorded by P_2 for requests 3 and 6 of T_2. As the status of requests 1,2,4, and 5 of T_2 could not be determined yet, they are put into the input queue of P_1 for resimulation, which now has a length of 4, leading to an upper bound on the number of hits of 8.*

3.1.2 Full LRU Stack Simulation

To devise a parallel algorithm for the full stack LRU simulation providing approximate results at any time during simulation execution, the success function is modified to return an interval of the number of hits for a given cache size at any time, instead of the exact value, available only at the end of simulation. Result intervals are specified by their lower and upper bounds which have to be derived from the simulation state. For the identification of lower bounds, no additional mechanism is needed, as they can be directly calculated from the finite stack distances determined during the simulation execution (called *final stack distances* hereafter).

For the determination of upper bounds, all requests which might be hits for a given cache size must be considered, or equivalently, all requests not determined to be sure misses. As a first approximation, in addition to the sure hits, all requests with infinite stack distances (i.e. requests where the correct distances are yet unknown) might be hits for any cache size (except for the first processor, where all stack distances are correctly determined, including infinite ones). However, a more precise determination of upper bounds of result intervals is possible using additional knowledge gained during simulation execution. Let r be a request to a page whose stack distance cannot be determined by a processor p. Then, the correct stack distance d of r must be at least the *preliminary stack distance* $\tilde{d}$, which is the current length of p's LRU stack after pushing r. Recall that a request r with a stack distance of d is a miss for a given cache size s if $d > s$. As just noted, for any preliminary stack distance $\tilde{d}$ of r, $d \geq \tilde{d}$ holds. Thus, if $\tilde{d} > s$, also $d > s$ and r must be a miss for cache size s. If preliminary distances are collected in a distance table in the same way as final distances, the same lookup in the cumulated table can be used to determine the number of possible (but not surely determined) hits for a given cache size, and hence the upper bound on the number of hits. This is due to the fact that only requests with preliminary distances lesser than or equal to the cache size might be hits, all others are known to be misses.

Stack distances are recorded in two different tables, final distances in a *lower-bound distance table* and preliminary distances in an *upper-bound distance table*. The enhanced algorithm processes requests similar to Algorithm 2. If the stack distance of a request can be calculated exactly by the processor, it is recorded in the lower-bound distance table. Otherwise, the preliminary stack distance is determined as the number of elements in the LRU stack after pushing the current request, this distance is recorded in the upper-bound distance table, and the request is sent to the input queue of the preceding processor for resimulation.

Example 3.2 *Consider the subtraces $T_1 = (a, b, c)$, $T_2 = (c, d, b)$, $T_3 = (b, a, b)$, and $T_4 = (e, c, a)$ which were created by dividing the input trace of Example 2.1 into four subtraces for parallel simulation. After all requests have been processed once, only the stack distance for the second b in T_3 could be determined exactly to 2, as well as the distances of ∞ for all requests in T_1. For the rest of the requests, preliminary stack distances can be calculated as explained above. Table 2 shows the final and preliminary distances after this first iteration. The distances in these tables can be cumulated to give the upper and lower bounds of the number of hits that are returned by a call to the approximate success function. E.g., for a cache size of 2, the exact number of hits must be contained in the interval* $[1, 6]$*, for a cache size of 4 in the interval* $[1, 8]$.

Further processing decreases the size of the returned intervals (increasing the accuracy of results). Table 3 shows final and preliminary distances after all requests to resimulate have been processed twice. Here, for a cache size of

Table 2: Distances in Example 3.2 after First Iteration

Final distances

Dist	1	2	3	4
Count	0	1	0	0

Prelim. distances

Dist	1	2	3	4
Count	3	3	2	0

Table 3: Distances in Example 3.2 after Second Iteration

Final distances				
Dist	1	2	3	4
Count	2	1	1	1

Prelim. distances				
Dist	1	2	3	4
Count	0	0	1	2

2, the number of hits is known to be exactly 3, whereas for a cache size of 4, the number of hits must be contained in the interval [5, 8].

Without further provisions, the preliminary stack distances for a request might be recorded multiple times by different processors. Therefore, before updating its upper-bound distance table, a processor must make sure that the record of the corresponding previously determined preliminary distance is removed. As the determined stack distances are reflected by the values of simple counters, this can be achieved by decrementing the value for the old preliminary distance in the table, ensuring consistent global values for the preliminary distances. For requests not processed before, no change of the upper-bound distance table is required. To implement this approach, requests that are kept in an input queue for processing must be annotated with their preliminary stack distance. Yet unprocessed requests are not annotated (or annotated with 0).

The overall approximate full LRU stack simulation approach is shown in Algorithm 3. The annotation of requests is realized here by using request/distance pairs as queue entries. Therefore, the ENQUEUE function takes a pair as second argument and the DEQUEUE function returns a pair. As termination of the algorithm is discussed in the next section, the termination of the while loop is not explicitly specified here. It depends on the return value of the opaque TERMINATE_ALGORITHM function, which might also need some parameters not shown for reasons of conciseness.

Input: *Subtraces $T_1, \ldots, T_m$. The stack S_i for every subtrace T_i is initially empty. The input queue Q_i of every processor is initialized with the input subtrace T_i, where every request is annotated with 0. The upper-bound distance tables U_i and the lower-bound distance tables L_i are initialized with 0 for every possible stack distance.*
Output: *The overall upper-bound distance table $U_{overall}$ and lower-bound distance table $L_{overall}$, which are the sum of the tables U_i and L_i, resp., for all processors i.*

```
begin
    for 1 ≤ i ≤ m pardo
        (req, odist) = DEQUEUE(Q_i)
        while ¬ TERMINATE_ALGORITHM() do
            if odist > 0 then
                U_i[odist] = U_i[odist] − 1
            if req ∈ S_i then
                dist = POSITION(S_i, req)
                L_i[dist] = L_i[dist] + 1
                REPOSITION(S_i, req)
            else
                PUSH(S_i, req)
                if i > 1 do
                    pdist = |S_i|
                    U_i[pdist] = U_i[pdist] + 1
                    ENQUEUE(Q_{i−1}, (req, pdist))
            (req, odist) = DEQUEUE(Q_i)
end
```

Algorithm 3: Approximate Full LRU Stack Simulation

3.2 Termination Conditions

The global calculations of result intervals presented in Section 3.1 are correct if processors do not continue simulation during the time of calculation. Otherwise, hits might be lost or recorded multiple times. In cases where a global execution strategy independent of the current accuracy of result intervals is chosen, this does not pose any problems. E.g., if runtime requirements exist, the simulation can be executed for a fixed amount of time and the accuracy of results can be determined afterwards.

However, if termination of the simulation algorithm has to depend on the current result accuracy, or if termination control should be implemented locally in the processors, further mechanisms are needed. A straightforward approach to determine simulation runtime by result accuracy is to halt execution of the processors periodically in order to check the current accuracy. The whole simulation process can be finished if accuracy is satisfying, or resumed otherwise.

Although the implementation of termination control locally in the processors is preferable, it is much harder to achieve. This is due to the implicit synchronization of processors, which only have limited knowledge about the status of the simulation in adjacent processors. Therefore, for a local termination control, additional messages have to be passed between processors. These might range from flags that give the status of the preceding processor (e.g. running or finished) to informations about its local result accuracy. For a very detailed termination control, sophisticated mechanisms are needed, which might incorporate a high overhead that is not justified by the finer granularity of the termination control.

4 EXPERIMENTS

To test the viability of approximate cache simulation, Algorithms 2 and 3 were implemented in C using the message passing interface MPI (Message-Passing Interface Forum 1997) to communicate between processes. For the determination of speedups, a sequential simulator was implemented.

Only the more interesting full stack simulation approaches were considered, using the search-tree based implementation of the LRU stack described in (Olken 1981).

Extended experiments with the prototypes were conducted on an SGI Origin 3000 with 64 500 MHz MIPS R14000 processors, a total of 32 GB of main memory, and the IRIX 6.5 operating system. The sources were compiled with the MIPSpro 7.2 Compiler with switches `-O3 -n32` enabled, using the MPI 1.2 implementation in the SGI Message Passing Toolkit (MPT).

The experiments consisted of the determination of the success functions of various input traces which had been obtained from the NMSU TraceBase facility (<`http://tracebase.nmsu.edu/tracebase.html`>). Table 4 summarizes the properties of the traces, which are collections of memory references from programs in the SPEC92 benchmark suite (Gee, Hill, Pnevmatikatos, and Smith 1993). A smaller trace (001) with a moderate number of unique references is included. The remaining traces are of about the same length, but differ significantly in the number of unique references. From those, the 085 trace is most complex with the highest number of unique references, leading to interesting results discussed later. Figure 2 shows a comparison of the success functions of all traces, where the hit rate in the interval $[0.85, 1.0]$ is plotted against an increasing cache size. Traces 001 and 085 exhibit a moderate locality, with a significant number of misses occuring even for cache sizes larger than 2000 entries. Trace 023 has a much higher overall locality indicated by the success function increasing faster than the functions of the other traces. Trace 097 shows a special behavior, where only a limited number of different hit rates exist.

When processing the traces, no distinction was made between data and instruction references. A 16 byte cache line was supposed, masking off the last four bits of every reference. Execution times of the calculations of stack distances were recorded during experiments of both the sequential and parallel cache simulators. The results presented here are averages of ten repetitions of every experiment.

Speedups for the various traces are shown in Figures 3 to 6. For all of the traces, speedup of the basic cache simulation (mcs) is excellent for small numbers of processors (≤ 16). However, with the number of processors increasing to 32, a degradation of performance is notable with the smaller 001 trace. With 64 processors, the degradation

Table 4: Properties of Input Traces

Trace	No. of refs.	No. of unique refs.
001 cexp	18,782,144	26,198
023 eqntott	100,000,000	89,857
085 gcc	100,000,000	162,997
097 nasa 7	99,731,776	30,109

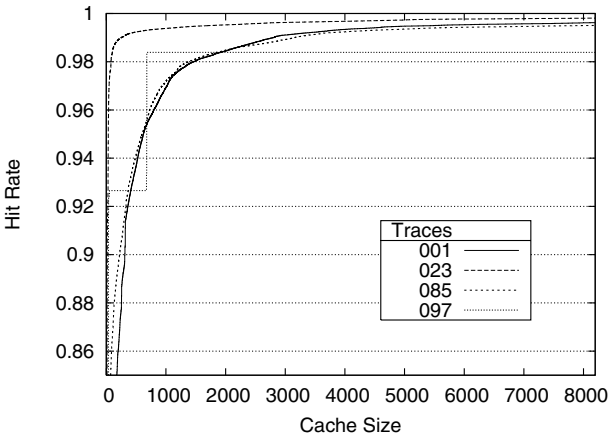

Figure 2: Success Functions of Traces

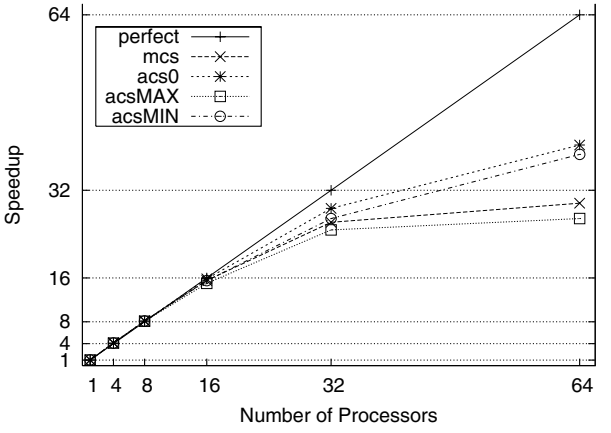

Figure 3: Speedups for Trace 001

reaches a significant level for all of the traces. In Figure 3 of the 001 trace, the speedup for 64 processors even stays on about the same level as that for 32. The speedups of the 085 trace shown in Figure 5 exhibit a more interesting pattern. Most notably, a better-than-perfect speedup is achieved for 16 and 32 processors. This is due to the smaller size of the local LRU stacks in the parallel simulation, resulting in a better cache performance and hence a faster execution. Another observation is the bad performance of mcs for the 085 trace with 4 processors. This results from the size of the resimulation subtraces of a single process exceeding the limit of the MPI message buffer, which leads to a much higher synchronization overhead, as the processes have to wait until the resimulation subtraces have been fetched by the peer before being able to continue simulation.

Additionally, Figures 3 to 6 show the speedups of three different types of experiments with the approximate parallel cache simulator (acs): acs0 does not perform any resimulation, i.e. it stops execution if every request from the input trace has been processed exactly once, acsINF performs resimulations until exact results are known, and acsMIN limits the amount of resimulation in every process

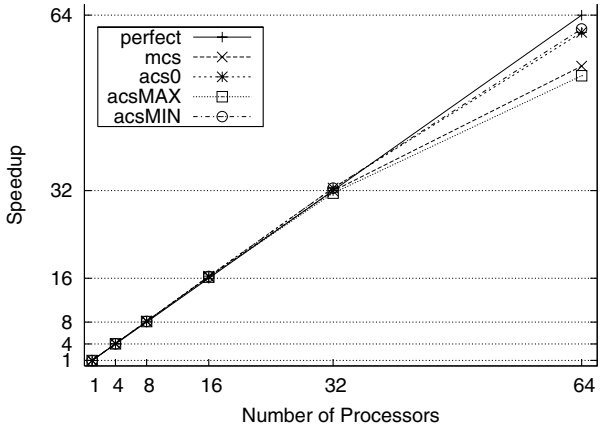

Figure 4: Speedups for Trace 023

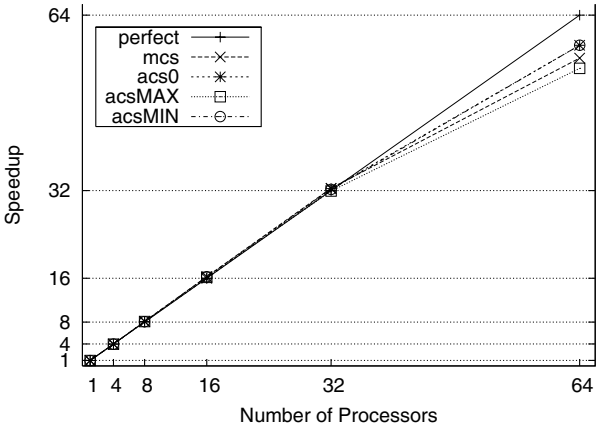

Figure 6: Speedups for Trace 097

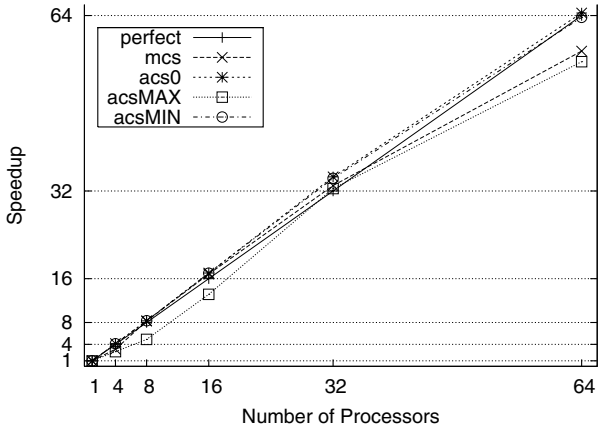

Figure 5: Speedups for Trace 085

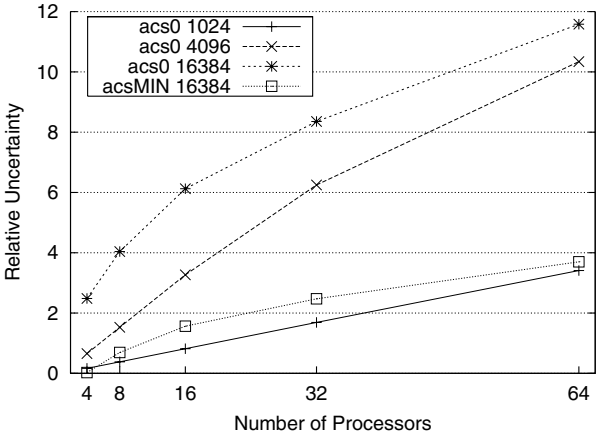

Figure 7: Relative Uncertainty for Trace 001 (in ‰)

to the minimal number of resimulation steps necessary in any one of the parallel processes without approximation. Both of acs0 and acsMIN notably increase scalability, as the speedups drop only marginally for 64 processors (except for the 097 trace, where in comparison to mcs, the speedup is not increased significantly by the approximate simulators). As can be noticed, the speedups of acsINF only slightly drop below that of mcs, which indicates a low overhead for approximate processing (with the exception of the 085 trace, where the increased size of resimulation messages leads to an even worse depletion of message buffers with consequences described above).

The amount of uncertainty in the results can be measured by the size of result intervals for the different cache sizes, giving the maximum range of different possible results. This can be divided by the number of request in the input trace to get the maximum possible deviation of the approximate hit rate for a given cache size from the correct hit rate. Figures 7 and 8 present these *relative uncertainties* in ‰ of the hit rate for the 001 and 085 traces. Uncertainties are shown for three different cache sizes for the acs0 experiments and one cache size for the acsMIN experiments. For the smallest cache size of 1024, uncertainties seem to grow linearly with the number of processors, whereas for larger caches, the growth is clearly sub linear. Note also that uncertainties are generally low, lying in the range of a few per thousand of the hit rate. The acsMIN experiments, which did perform almost as well as acs0, exhibit very small uncertainties, even for higher cache sizes.

5 CONCLUSIONS

Temporal parallelization of simulation models is a promising alternative to the more traditional parallelization approaches. The difficulty in applying temporal parallelization to different application domains lies in an efficient solution of the state-match problem. The usage of approximation with time-parallel simulation has been proposed earlier in order to extend its applicability to new classes of models, as well as a means to improve the efficiency of existing time-parallel simulation models. The focus of this work is the application of approximation for the parallel simulation of LRU caching.

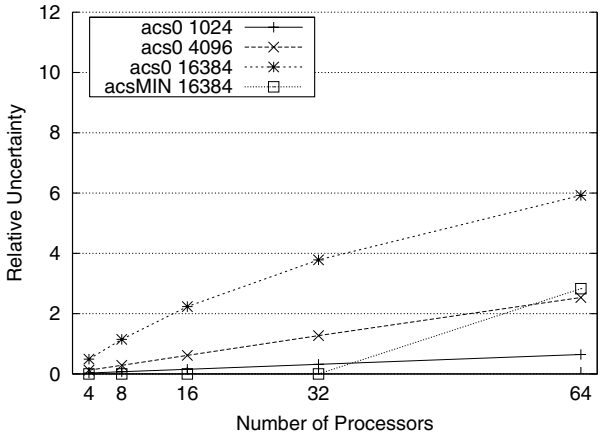

Figure 8: Relative Uncertainty for Trace 085 (in ‰)

First, existing time-parallel simulation algorithms, both for simple LRU cache simulation and full LRU stack simulation, have been presented. These algorithms have then been used as the basis for approximate simulation algorithms, where at any time during the simulation, approximate results can be calculated in the form of intervals giving the range of possible values of results. Strategies for termination of the simulation at an appropriate time have been discussed.

The algorithms presented in this work (the basic as well as the approximate ones) have been prototypically implemented and experiments have been conducted. These indicate a significant increase in the speedup of the simulation with a reasonable accuracy of simulation results.

Previous work on the parallel simulation of LRU caching has either restricted itself to the simple cache simulation approach, or exhibited a serious decrease of the speedup achieved for higher numbers of processors. This work uses approximation to increase the overall speedup, allowing the parallel simulation to scale to very high numbers of processors. Two properties of the approximation algorithms allow a direct control of the introduced uncertainty: accuracy of simulation results increases monotonically with time and it can be calculated without much overhead at any time of the simulation execution. In theory, linear speedup can always be achieved by allowing an arbitrary uncertainty. In practice, this is hard to achieve, due to the synchronization required for input and collection of results. However, experiments indicate, that even with a very small uncertainty, significant increases in speedup are possible.

ACKNOWLEDGMENTS

The author would like to thank Manfred Neumann of the RZ UniBwM for providing access to the SGI Origin 3000 used for the conduction of experiments.

REFERENCES

Bennett, B. T., and V. J. Kruskal. 1975. LRU stack processing. *IBM Journal of Research and Development* 19 (4): 353–357.

Chandy, K., and R. Sherman. 1989. Space-Time and Simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation*, 53–57.

Fujimoto, R. M. 1990. Parallel Discrete Event Simulation. *Communications of the ACM* 33 (10): 30–53.

Gee, J. D., M. D. Hill, D. N. Pnevmatikatos, and A. J. Smith. 1993. Cache Performance of the SPEC92 Benchmark Suite. *IEEE Micro* 13 (4): 17–27.

Greenberg, A. G., B. D. Lubachevsky, and I. Mitrani. 1991. Algorithms for Unboundedly Parallel Simulations. *ACM Transactions on Computer Systems* 9 (3): 201–221.

Heidelberger, P., and H. S. Stone. 1990. Parallel Trace-Driven Cache Simulation by Time Partitioning. In *Proceedings of the 1990 Winter Simulation Conference*, ed. O. Balci, 734–737. Piscataway, New Jersey: IEEE Press.

JáJá, J. 1992. *An introduction to parallel algorithms*. New York: Addison-Wesley.

Kiesling, T., and S. Pohl. 2004. Time-Parallel Simulation with Approximative State Matching. In *Proceedings of the 18th Workshop on Parallel and Distributed Simulation*, 195–202.

Lin, Y., and E. Lazowska. 1991. A Time-Division Algorithm for Parallel Simulation. *ACM Transactions on Modeling and Computer Simulation* 1 (1): 73–83.

Mattson, R. L., J. Gecsei, D. R. Slutz, and I. L. Traiger. 1970. Evaluation Techniques for Storage Hierarchies. *IBM Systems Journal* 9 (2): 78–117.

Message-Passing Interface Forum 1997. *MPI-2.0: Extensions to the message-passing interface*. MPI Forum.

Nicol, D. M., and E. Carr. 1995. Empirical Study of Parallel Trace-Driven LRU Cache Simulators. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, 166–169.

Nicol, D. M., A. G. Greenberg, and B. D. Lubachevsky. 1994. Massively Parallel Algorithms for Trace-Driven Cache Simulations. *IEEE Transactions on Parallel and Distributed Systems* 5 (8): 849–859.

Olken, F. 1981. Efficient Methods for Calculating the Success Function of Fixed-Space Replacement Policies. Technical Report LBL-12370, Lawrence Berkeley Laboratory.

Thompson, J. G. 1987. Efficient Analysis of Caching Systems. PhD thesis, University of California, Berkeley. Also published as: UCB, CSD TR-87/374.

AUTHOR BIOGRAPHY

TOBIAS KIESLING is a Research Assistant at the University of the Federal Armed Forces (UniBwM) in Munich, Germany. He holds a german Vordiplom-Degree (B.S. equivalent) in Statistics (1997) as well as a german Diplom-Degree (M.S. equivalent) in Computer Science (2002), both from the Ludwig-Maximilians-University in Munich, Germany. He is currently working on his Dr. rer. nat. (Ph.D. equivalent) in the field of parallel and distributed simulation. His research interests include time-parallel simulation, the usage of approximation techniques in parallel and distributed simulation, and distributed hybrid simulation models. His e-mail address is <kiesling@informatik.unibw-muenchen.de>, and his homepage is <http://fakinf.informatik.unibw-muenchen.de/~tkiesling/>.