Scalability Analysis for Conservative Simulation of Logical Circuits

Jörg Keller
FB Informatik, FernUniversität-GHS Hagen, Postfach 940, 58084 Hagen, Germany,
Tel. +49-2331-987-376, FAX +49-2331-987-308, email `Joerg.Keller@FernUni-Hagen.de`

Thomas Rauber
Institut für Informatik, Universität Halle-Wittenberg, 06099 Halle (Saale), Germany
Tel. +49-345-5524716, FAX +49-345-5527009, email `rauber@informatik.uni-halle.de`

Bernd Rederlechner
Telekom Entwicklungszentrum Südwest, Neugrabenweg 4, 66123 Saarbrücken, Germany
Tel. +49-681-909-2325, FAX +49-681-909-2315, email `Bernd.Rederlechner@ezsw.telekom.de`

principal author for contact: Jörg Keller

Abstract

We investigate conservative parallel discrete event simulations for logical circuits on shared-memory multiprocessors. For a first estimation of the possible speedup, we extend the critical path analysis technique by partitioning strategies. To incorporate overhead due to the management of data structures, we use a simulation on an ideal parallel machine (PRAM). This simulation can be directly executed on the SB-PRAM prototype, yielding both an implementation and a basis for data structure optimizations. One of the major tools to achieve these optimizations is the SB-PRAM's hardware support for parallel prefix operations. Our reimplementation of the PTHOR program on the SB-PRAM yields substantially higher speedups than before.

Keywords: discrete event simulation, conservative parallel simulation, parallel data structures, critical path analysis, shared-memory multiprocessor, PRAM

Most of this work was done while the authors were working at Universität des Saarlandes, Computer Science Dept., Saarbrücken, Germany. This work was supported by the German Science Foundation (DFG) under SFB 124 TP D4, J. Keller was supported by a DFG fellowship. A preliminary version of this paper appeared in Proc. 10th Workshop on Parallel and Distributed Simulation (PADS'96).

Correspondence and requests for reprints should be sent to Jörg Keller, FernUniversität-GHS Hagen, FB Informatik, LG Technische Informatik II, 58084 Hagen, Germany.

1 INTRODUCTION

Large–scale shared-memory multiprocessors are likely to play an important role in parallel computing in the future, because they offer a much simpler programming model than traditional distributed-memory machines. Many of today's shared-memory machines are cache-based machines which show good performance for regular applications with appropriate locality but which fail to get good speedups for irregular applications with a lot of non-local memory accesses. Typical examples of such applications are particle–based simulations like MP3D [24], routing algorithms like LocusRoute [24], and discrete–event simulations like PTHOR [26]. In this article, we consider the execution of discrete-event simulations for logical circuits on shared-memory machines. We try to answer the question which performance we can hope to get on an ideal machine on which the locality of memory accesses can be neglected but for which the overhead for the management of data structures still takes effect. As execution platform, we use the SB-PRAM which has a uniform memory-access time and behaves like a PRAM machine as it is used in theoretical computer science for the analysis of the complexity of algorithms.

We consider the PTHOR algorithm for the parallel simulation of logical circuits, which uses a conservative approach. The PTHOR simulator is based on the sequential THOR simulator and has first been considered for a parallel implementation on the Stanford Dash by Soulé [26]. Soulé investigates the performance of the PTHOR simulator for three platforms: an ideal multiprocessor simulator called Tango [24], an Encore Multimax with 16 processors, and the Stanford Dash with 16 processors.

For a systematic analysis of the attainable speedup, we start with a critical path analysis of PTHOR on the benchmark circuits, which also takes into consideration the partitioning of the LPs among the processors. We extend the partitioning strategies investigated by Lin in [21] from static partitioning strategies to dynamic strategies and stealing strategies. Although this technique yields an upper bound on the speedup for the different benchmark circuits, it does not take into account the overhead for data structures. This can be done by running PTHOR on the SB-PRAM. As the complete SB-PRAM is under construction, we use a simulator that performs a cycle–by–cycle simulation of the actual machine. Thus, the simulator delivers the exact runtime of the real hardware. The accuracy of the simulated runtimes is confirmed by comparisons with measured program runtimes on the available prototype.

Starting with the existing PTHOR implementation from the SPLASH1 benchmark suite [24], we show how the maximum attainable speedup can be increased by several changes in the data structures, including the data structures for the LPs and the memory management. When there are more LPs than processors, the work must be properly partitioned among the processors. We compare a dynamic partitioning scheme using a centralized FIFO queue with a stealing scheme that uses a local queue for each processor. We also show that the use of NULL-messages can result in a large increase of the speedup, depending on the benchmark circuit. The result is an implementation of the PTHOR simulator on the SB-PRAM for which the overhead for the management of data structures is considerably smaller than in the original implementation. Depending on the input circuit, the obtained speedup values even come close to the bound from critical path analysis.

The rest of the paper is organized as follows. Section 2 briefly introduces to parallel discrete event simulation. Section 3 sketches the execution platform used. Section 4 presents the critical path analysis. Section 5 investigates the performance characteristics of the original PTHOR simulator. Section 6 presents the improvements we added and discusses their effects, Sect. 7 summarizes the results.

2 PARALLEL DISCRETE EVENT SIMULATION

A model for discrete event simulation assumes that the system being simulated only changes state at discrete points in time. For the simulation, the system is modeled as a collection of *logical processes* (LPs) that communicate via timestamped messages. For circuit simulations, typical LPs at varying levels of abstraction are transistors, NAND gates, flipflops, multipliers, etc., and their interconnections [5]. The state of the simulated model changes upon the occurrence of *events*, such as the change in output value of an individual gate. An event e may be *scheduled* by a certain number of other events, if these determine the occurrence of e.

The approaches to a parallel execution of discrete-event simulations (PDES) can be distinguished into centralized-time algorithms and distributed-time algorithms. In centralized-time algorithms, a global clock is used and the simulation is executed synchronously. In distributed-time algorithms, each processors has its own clock and the simulation is executed asynchronously. Distributed-time algorithms can be further distinguished into conservative and optimistic approaches. The approaches differ in the way they deal with causality errors caused by the distributed simulation, see [13] for a good overview. The conservative method [10, 12] forces an LP to block until it is safe to simulate an event, i.e., the events are simulated in strict timestamp order. This may lead to deadlocks that have to be recognized and resolved. In the optimistic approaches [3, 16], there is no such restriction, i.e., an LP can execute events in the order in which they arrive. If this leads to a simulation that is not in timestamp order, a roll back to a safe state has to be performed and the effect of messages which should not have been send must be eliminated by appropriate anti-messages.

The limiting factor for a centralized-time algorithm is that the simulation steps proceed in lockstep fashion, waiting for the slowest event to finish [5]. This can greatly slow down the simulation, if there are widely varying event times.

Bailey shows in [4] by a theoretical analysis that the execution time of the conservative asynchronous strategy is a lower bound to the synchronous strategy and that with unit-delay timing, the execution times of the synchronous and asynchronous strategies are equal. The analysis is performed under the assumption that an unlimited number of processors are available and that the inputs to a circuit remain fixed during the simulation. These assumptions are relaxed by Baker in [7] by allowing an arbitrary number of external inputs for each circuit, with each input experiencing different numbers of events at different simulation times. Under these conditions, a relative comparison of the synchronous and conservative asynchronous simulation execution times shows that the conservative asynchronous simulation may execute faster. In particular, the best-case execution times are the same for the synchronous and conservative asynchronous simulation, but the requirements for achieving the minimum time are quite strict. The worst-case execution time of the conservative asynchronous simulation will usually be less than that of the synchronous simulation.

In [26], a parallel, centralized-time logic simulator is discussed. In this practical work, none of both algorithms achieve the best results for all benchmark-circuits.

Supported by the theoretical results above, we decided to research conservative asynchronous simulation and neglect synchronous schemes. The algorithm used in PTHOR offers various possibilities for optimization, with the hope of preserving the benefits of asynchronous simulation.

3 EXECUTION PLATFORM

Most of today's shared-memory machines are cache-based machines, i.e., they still use a physically distributed memory but each processor is equipped with a one-level cache or a two-level cache-hierarchy. The cache coherence is provided by the hardware. The memory access time of these machines is not uniform but depends on the physical location of the data being accessed. For this reason, they are called nonuniform memory access time (NUMA) machines. These machines rely on the locality of most applications and try to hide the memory latency by caching. Examples of NUMA machines are the KSR1/2 [2] from Kendall Square Research, the Stanford Dash [20], and the SPP1000 from Convex [28].

Besides cache-based shared-memory machines, uniform memory access time (UMA) machines have been developed for which the memory access time is independent from the physical location of the data. Examples of such machines are bus-based shared-memory machines like the Multimax [2] from Encore Computer Corp., the C90, J90, and T90 series from Cray Research [2], and the SGI Challenge from Silicon Graphics. The disadvantage of bus-based systems is that they usually can only provide a small number of processors.

The SB-PRAM which is currently under construction at the University of Saarbrücken is an UMA machine that provides a shared address space with a fast memory access time [1]. The latency of the network between the processors and the memory modules is hidden by pipelining of processors, i.e., each physical processor simulates a number of virtual processors. Thus, a write operation to the global memory by a virtual processor takes the same time as an arithmetic operation, independently of the memory location that is addressed. A read operation is also as fast as an arithmetic operation, but the result is available in the next but one instruction. Concurrent accesses to a single memory cell are allowed and combined, making the SB-PRAM behave like the CRCW (CRCW=concurrent read, concurrent write) PRAM model known from theoretical computer science.

Besides the usual load and store operations to access memory cells, the SB-PRAM also offers *multiprefix* instructions which enable several processors to perform prefix operations on a memory cell in parallel. As an example, we sketch the execution of a multiprefix addition MPADD. Let $p_1, \ldots, p_n$ be the executing processors where each processor p_i contributes a local value o_i. Let s be a shared memory cell with value o. If $p_1, \ldots, p_n$ execute the MPADD operation synchronously, i.e., each processor p_i executes MPADD s, o_i, then after the operation, processor p_j holds the jth prefix sum $o + \sum_{i=1}^{j-1} o_i$, s contains the sum $o + \sum_{i=1}^{n} o_i$. The multiprefix operations MPMAX, MPOR, and MPAND work similar.

A multiprefix operation is as fast as a read operation, independently of the number of participating processors. It is even possible that different groups of processors perform separate multiprefix operations in parallel. The multiprefix operations can be used for an efficient implementation of synchronization mechanisms (such as barriers without serialization [14]) and for the implementation of various parallel data structures for task management like priority queues or FIFO queues [23]. Because of its memory structure, the SB-PRAM is an ideal machine for the execution of irregular applications. In addition to running an application on the SB-PRAM, the machine can also be used to study the properties of a parallel program under ideal conditions, yielding a prediction of the maximum speedup that can be attained on other machines.

The current prototype provides the user with 128 PRAM processors, the complete prototype will provide 4096 processors. Program runs were executed on a cycle-by-cycle simulator, accuracy was confirmed by comparisons with runs on the actual prototype.

4 CRITICAL PATH ANALYSIS

Not all events occurring while simulating a circuit can be executed in parallel. The result of an event e can only be computed correctly if

1. all events preceding e on the same LP are executed,
2. the results of all events scheduling e are known to e.

4.1 Event Precedence Graphs

Consider the set of the events that occur during the simulation of a fixed experiment on a fixed model. From the above constraints, we can derive a partial order on this set, called "causality". The representation of this order as a directed graph $G = (V, E)$ is called "event precedence graph" (EPG), introduced independently by Berry and Jefferson [8] and Livny [22]. V is the set of events, (e_1, e_2) is an edge iff. e_1 schedules e_2 or e_1 is the last event before e_2 on the same LP. The weight function $\tau : V \to \mathbf{R}_0^+$ assigns to each event the runtime to execute it. This definition can be made independent of the underlying machine by defining $\tau(e)$ as a function on the indegree of e. We call an event e_2 *dependent* on e_1 iff. there exists a path in G from e_1 to e_2.

Only events that are independent from each other can be executed in parallel. Hence, the EPG serves to compute a lower bound on the simulation's runtime. We assume that every LP is simulated on its own processor. Then, because of constraint 1, it can never happen that more than one event e is ready for execution on one processor. This unique event e can be executed as soon as constraint 2 is satisfied. Obviously, events e with indegree 0 can be executed immediately after the simulation starts.

If $START(e)$ and $END(e)$ denote the times when the execution of event e ideally starts and finishes, then

$$\begin{aligned} END(e) &= START(e) + \tau(e)\,, \\ START(e) &= \begin{cases} \max\limits_{(e',e)\in E} END(e') & indeg(e) \geq 1 \\ 0 & \text{otherwise.} \end{cases} \end{aligned}$$

This recurrence equation is well defined because EPGs are acyclic. To compute END, one sorts the vertices topologically and evaluates them in this order. The time

$$T_{crit} = \max_{e\in V} END(e)\,. \tag{1}$$

is the runtime of an ideal simulation on a parallel machine with an arbitrary number of processors. T_{crit} is a lower bound on the parallel runtime of every conservative simulation strategy [17]. It is even a lower bound on optimistic strategies with aggressive cancellation [15].

The path defining the maximum in (1) is called *critical path.* Note that there may be several critical paths in an EPG.

The EPG also serves to compute a lower bound on the sequential runtime by

$$T_{seq} = \sum_{e \in V} \tau(e) \, .$$

So far, the computed runtimes ignore any computational overhead in addition to causality. If we assume that the overhead in a parallel simulation is greater than in a sequential simulation, then the quotient $S_{crit} = T_{seq}/T_{crit}$ defines an upper bound on the possible speedup for a particular experiment.

This overhead assumption is supported by the observation that normally all data structures from the sequential program are needed in the parallel version as well. The parallel program might need additional data structures to support information exchange between LPs.

4.2 Partitioning Strategies

For large circuits, real parallel machines do not have enough processors to assign each LP to a different processor. Hence, the LPs must be partitioned between the available processors.

On distributed memory multicomputers, a commonly used partitioning scheme is *static partitioning.* Every processor is assigned a fixed set of LPs, the sets are disjoint. Examples for static partitioning are cyclic distribution (LP_i is executed on processor $i \bmod p$), blockwise distribution (processor i executes $LP_{in/p+1}$ to $LP_{(i+1)n/p}$), and random distribution (each processor is assigned n/p LPs in a random fashion). If the numbering of LPs in the input data file is arbitrary, then any distribution resembles random partitioning.

There are a number of heuristic approaches to find better static partitionings [9, 18, 19, 27]. However, we did not consider those approaches. They mostly try to optimize communication costs which is not necessary as we use shared–memory machines.

On a shared memory multiprocessor, all processors have access to the data of every LP. Hence, an obvious strategy would be to have a central FIFO queue for LPs that are ready for execution. An idle processor simply picks the first queue element. We call this strategy *dynamic.* The standard method to find out when an LP becomes ready for execution is presented in Subsect. 5.1. The disadvantage of a central FIFO queue is the possible serialization overhead due to concurrent access of multiple processors. This overhead can be eliminated by a serialization–free parallel data structure on the SB-PRAM (see Subsect. 6.5).

Often however, shared memory multiprocessors need some locality in data referencing to exploit their caches and hence to obtain appropriate memory bandwidth. To achieve locality, the PTHOR program of the SPLASH1 benchmark suite [24] uses a so called *stealing strategy*: basically, this is a static strategy with local *task queues* for LPs that are ready for execution. In cases where the load is not balanced, an idle processor can "steal" an LP that is ready for execution but is assigned to another processor. The stealing strategy exploits locality as long as processors are busy and requires remote access only for load balancing when the processor is idle anyway.

In all these strategies, it may happen that a processor must choose between several LPs that are ready for execution. This can happen because either more than one LP assigned to a processor is ready, or because more than p LPs are ready in the central FIFO queue. In PTHOR, the processor chooses the LP that has been ready for execution for the longest time. This is easy to implement. Another popular method is to choose the LP with the smallest timestamp. This method leads to additional overhead because it requires that LPs that are ready to run are kept sorted according to their timestamps.

To get realistic runtime predictions $T_{crit}(p)$ depending on the number of processors p, it is necessary to model the partitioning strategy used in the critical path analysis. Note that these runtimes cannot be shorter than T_{crit}. All delays due to causality apply for both T_{crit} and $T_{crit}(p)$, and partitioning could introduce additional delays. The inclusion of partitioning strategies in critical path analysis was first mentioned by Lin [21], but he only uses a static strategy.

To include one of the above partitioning strategies in critical path analysis, we assume that the number of available processors p is fixed. We maintain a timer $c(i)$ for each processor i, which specifies the computation time performed by i. If this processor executes an event e, the timer is increased by $\tau(e)$. As before, we evaluate the function *END* on the nodes of the EPG in topological order. For an event e executed on processor i, let $c_{old}(i)$ denote the value of $c(i)$ before the execution of e. Then

$$\begin{aligned} END(e) &= START'(e) + \tau(e)\,, \\ START'(e) &= \max\left(c_{old}(i), START(e)\right)\ . \end{aligned}$$

$START(e)$ is defined as above. The execution time consumed by simulating e is taken into account by updating $c(i)$ to

$$c(i) = END(e)\ .$$

The different partitioning strategies lead to different assignments of LPs (and their events) to processors and hence to different results for $T_{crit}(p)$.

Note that the topological sort does not give a unique total order on the vertices, e.g. all vertices with indegree 0 could serve as the first node. Therefore we maintain a priority queue of all events that are ready for execution. The priority is the time when the events became ready. Removing the event with the smallest ready time ensures correct modeling.

4.3 Experiments

We computed the EPGs for three circuits delivered with the PTHOR simulator from the SPLASH1 benchmark suite [24].

- DASH models the cache coherency controller of the DASH multiprocessor [20] and represents 74,000 gate equivalents organized in 24,000 LPs.

- H-FRISC is a small RISC processor generated by a synthesis tool. It represents 7,000 gate equivalents organized in 5,000 LPs.

- Multiplier implements a multiplier of two 16-bit numbers. It also represents 7,000 gate equivalents organized in 5,000 LPs.

We use the input vectors that are delivered with the PTHOR program. We use the unit delay model, i.e., each gate and each register has a delay of 1. We simulate 5000 time units. We computed the speedup bound S_{crit} and bounds

$$S_{crit}(p) = \frac{T_{seq}}{T_{crit}(p)} \ ,$$

where $p = 2^i$, $i = 0, \ldots, 12$, for the three partitioning strategies. For the static and stealing strategies, we use a cyclic distribution. The curves are shown in Fig. 1. $\Longleftarrow$ **Fig.1**

The speedup bounds $S_{crit}(p)$ with partitioning reach the maximum speedup S_{crit} already for small numbers of processors. The dynamic partitioning strategy outperforms the other two in theory. For small processor numbers ($p \leq 16$), the stealing strategy behaves like the static strategy, for larger processor numbers it approaches the dynamic strategy. As the static strategy performs worst, we do not consider it in the sequel.

Second, note that causality restricts the available parallelism severely. The DASH circuit, also the largest one, obtains the worst speedup bound with 7.48. Soulé [26] estimates the inherent parallelism by counting the number of LP-evaluations for each timestep and computing the average over the different timestep-results. With 267 evaluations, the DASH circuit shows a much better result than H-FRISC and Multiplier with the same average of 156 evaluations. The numbers neglect the influences of causality and granularity on the inherent parallelism. This fact as well as the later presented practical results favor our estimation method.

Especially the causality has a strong influence on the parallelism. This might result from the form of the LPs. The DASH circuit has LPs with up to 94 inputs. In contrast, the H-FRISC and the Multiplier circuits have LPs with up to 17 and 5 inputs, respectively. The more inputs an LP has, the more it can depend on events occurring on other LPs. The events that schedule an event on an LP with many inputs might finish at vastly different computation times. As a conservative simulation must wait for the last of these events to finish, the delays due to causality can be large. So, it might be wise to split large LPs into smaller units with fewer inputs.

In contrast to this, Soulé [26] proposes to combine LPs to larger units called "globbed elements" to get a larger granularity of the single tasks and so to increase the speedup. As this increases the number of inputs per LP, the benefits due to larger granularity get lost by parallelism degradation. Our results strongly discourage this proposal.

We also investigated the granularity of the LP execution times as a possible source of speedup degradation. On the SB-PRAM the execution time of an LP is proportional to the number of instructions. Figure 2 shows the distribution of the LP execution times. $\Longleftarrow$

Fig.2

First, a single LP needs at most 100 instructions on the SB-PRAM. Thus it does not seem useful to parallelize the execution of LPs. Second, the variance in the execution time is not very large. If we replace the execution time of each LP in an EPG by the average execution time over all LPs of this EPG, then the maximum speedup S_{crit} only increases between 9 % and 17 %. Hence, the difference in execution time cannot explain a large speedup degradation.

5 PTHOR

A widely used algorithm for circuit simulations on parallel machines is the Chandy–Misra–Bryant algorithm (CMB) [10, 12]. This algorithm is a conservative approach. We will first review the PTHOR program [26], which is an implementation of CMB on the Stanford Dash machine and distributed as part of the SPLASH1 benchmark suite [24].

Granularity has a strong influence on centralized-time algorithms. The runtime of each round is bound by the longest task. The asynchronous CMB algorithm is potentially able to simulate events of other simulation timesteps in parallel while a lengthy event runs on one processor. Our granularity measurements show that lengthy tasks exist in the simulation of our benchmark circuits.

Finally, the overhead of synchronization for *each* simulation-time step in synchronous simulation is inevitable. Every element in our benchmark circuits has a non-zero delay and no events are cancelled, so at most one deadlock per simulation-timestep can occur in CMB. With our optimizations discussed later, deadlock resolution runs only slightly slower on the SB-PRAM than a synchronization. So, every timestep without deadlock can help to avoid overhead that must occur in centralized-time simulation.

5.1 Description

PTHOR partitions the LPs of the simulated circuit with the stealing strategy sketched in Subsect. 4.2. It uses a cyclic distribution of LPs to processors. There is a message channel between LP i and LP j if an input of component j in the simulated circuit is connected to an output of component i. If LP i computes a change of the output signal that occurs at *simulated* time t, then this output is put into a message with timestamp t. All LPs connected with LP i get a copy of this message in their appropriate input buffers.

Each processor maintains an *activation list* that contains all of its LPs for which new messages have arrived. If LP i sends a message to another LP j, it generates an entry for LP j in the activation list of the processor to which LP j is assigned.

An event e can only be simulated if all necessary inputs are present in the input buffers. An idle processor j tries to get an LP from its activation list. If its own list is empty, then it tries to steal an

LP from another activation list. If the chosen LP has all necessary inputs, j can simulate one or several events from that LP correctly. In either case, this LP is removed from the activation list. It will be entered again if some new input message arrives.

It can thus happen that all activation lists become empty although some events could be simulated. Such a situation is called *deadlock*. The CMB algorithm tolerates deadlocks, because it is able to detect and to resolve all of them. Deadlock detection can be implemented on a shared memory multiprocessor by maintaining a shared counter which is initially set to zero. A processor whose activation list becomes empty (and does not succeed in stealing) increases the counter. It decrements the counter again if it finds a new event to simulate. A deadlock has occurred if the counter equals the number of available processors.

To resolve the deadlock, one has to find at least one event that can be simulated. To do this, we search for a message m with the minimum timestamp $\tilde{t}$. Chandy and Misra prove that all events that occur at time $\tilde{t}$ (and hence have m as input) can be simulated [12].

5.2 Performance

Figure 3 shows the speedups for the benchmark circuits on three machines, with processor numbers ranging from 2 to 128. Only on the SB-PRAM we obtain a speedup larger than 1. The diagrams show absolute speedups: the sequential runtime is not the runtime of the parallel program with one processor. Instead, it is the runtime of the fastest sequential implementation we were able to develop. For the circuits, the same models and the same implementations were used in the sequential and the parallel case. Only the parts for administrating messages, scheduling LPs and memory management were replaced for the different sequential and parallel measurements. These parts of our sequential simulator had to be optimized: In a sequential simulator, the events must be executed in increasing timestamp order. Thus, in contrast to parallel asynchronous schemes, the sequential queue not only

$\Longleftarrow$ **Fig.3**

Table 1: Slowdown factors

Benchmark	DASH	H-FRISC	Multiplier
SB-PRAM (PTHOR)	10.4	7.4	5.4
Dash (PTHOR)	13.0	9.6	7.5
SB-PRAM (Reimpl.)	3.0	2.1	1.7

schedules the LPs but also has to restore the timestamp order. To perform this task, all messages are held in a priority queue. For the SB-PRAM, we implemented several different data structures like binary heap, fibonacci heaps and calendar queues. We found out that splay–trees [25] give the best runtime results for our application. Besides many small optimizations, an efficient memory management was realized.

Note that the parallel program on one processor is much slower than the sequential program on one processor of the same machine. The quotient between these two runtimes is called *slowdown factor*. Table 1 shows the slowdown factors for the three benchmark circuits on the SB-PRAM and the Dash machine. The latter are taken from [26]. For Dash and Multimax, we used relative speedups from [26] and the above slowdown factors to compute absolute speedups..

The source code of the centralized-time simulator is not delivered with the SPLASH1 benchmarks. So, runtime results for comparison on the SB-PRAM are not available.

The performance of PTHOR suffers from serialization. Serialization occurs during concurrent access to the shared counter for deadlock detection.

The access to the counter is protected by a lock. Figure 4 shows the total number of accesses to the shared counter and the fraction of accesses that were not directly granted. The time to access a lock is one instruction in both the Dash and the SB-PRAM, as both machines provide hardware support for $\Longleftarrow$ **Fig.4** read-modify-write operations.

Serialization is also caused by the computation of the minimum timestamp during deadlock resolution. This computation needs a loop over all processors and barrier synchronizations before and after the loop. The barriers are also implemented by locks. The upper curves of Fig. 5 show the average number of instructions needed to resolve a deadlock in PTHOR on the SB-PRAM. The lower curves show the corresponding numbers for the reimplementation (see next Section). $\Longleftarrow$ **Fig.5**

6 REIMPLEMENTATION

Our reimplementation avoids the serializations mentioned above. We also improved the memory management and the realization of channels between LPs. As mentioned in Sect. 1, the multiprefix operation serves to compute global sums and global minima in a small constant number of instructions. Figure 5 shows the average number of instructions needed for deadlock resolution on the SB-PRAM using multiprefix.

6.1 Memory Management

During the simulation, one has to manage tens of thousands of small list elements for message queues, activation lists etc. PTHOR never recycles elements, it even keeps those elements that are not in use anymore. This is a waste of memory resources and leads to unnecessary shared memory allocations. Furthermore, extracting list elements from the allocated memory leads to serialization because locks are used.

In the reimplementation, each processor maintains a so called *freelist*. After a processor has executed an event, some of the involved list elements might not be needed anymore. Then, the processor adds these to its own freelist. If a processor wants to allocate a list element, it first tries to obtain one from its freelist. If its freelist is empty, then it obtains a list element from an allocated shared memory block.

If a block containing l list elements is allocated, a shared counter c is initialized to l. A so called *R–pointer* is set to the beginning of the memory block. To obtain a list element from that block, a processor decreases the counter c with the help of multiprefix. This allows a concurrent access of multiple processors without serialization. The result r of the prefix operation gives the number of remaining list elements. If $r \leq 0$ the memory block is exhausted. The processor that obtains value 0 then allocates a new memory block, all processors that received values less or equal to zero then repeat the allocation with the new block.

If a processor receives $r \geq 1$, it can cut off a list element from the memory block. To do this, it increases the R–pointer of this block by the size of a list element with the help of multiprefix. The value the processor obtains then determines the position of the list element. Figure 6 shows five processors that try to allocate a list element. Processor 0 finds an element in its freelist, the other four processors $\Longleftarrow$ must allocate from a shared memory block with $c = 2$. After the multiprefix operation, $c = -2$, and **Fig.6** processor i receives value $3 - i$. Thus, processors 1 and 2 get list elements from the current memory block. Processor 3 receives the value 0 and allocates a new block, from which processors 3 and 4 allocate their list elements.

6.2 Channel Queues

The realization of a channel is performed with a FIFO queue where one LP writes a message and all LPs connected to this channel read the message. As it is not clear when all LPs have read a message, PTHOR keeps all messages in these queues. We attach a shared counter to each message in the queue. The counter is initialized to the number of LPs connected to this channel. Each LP reading a message decreases its counter with the help of multiprefix. If the counter has reached zero, the processor accessing the message removes it from the queue and puts it into its freelist. We call this queue organization *single-in multiple-out queue (SIMO).* It needs no locks. Figure 7 shows a SIMO queue where LP 0 writes and LPs 1 to 4 read. The uppermost two messages have not yet been read by any LPand hence have counters with values 4. The next two messages have been read by LP 1 and LPs 1 and 4, respectively, and thus have counters with values 3 and 2. LP 2 has just read the lowermost message and thus decreased the message's counter to zero. The message now is removed from the queue. $\Longleftarrow$ **Fig.7**

6.3 LNE Lists

To resolve deadlocks, one has to inspect all LPs that satisfy the following conditions:

- At the beginning of the deadlock, the LP still has messages in its input buffers,
- the LP has processed at least one message.

To speed up deadlock resolution, we maintain a data structure containing only the LPs satisfying the above conditions.

When a processor fetches an LP i from its activation list, it first checks this LP's input buffers and possibly simulates one or several safe events. In either case, this ends by checking the input buffers without finding a safe event. During the test, the above conditions can be checked with little additional overhead. Also, the minimal timestamp of the messages in LP i's input buffers can be computed. This value is called the *LNE time* of LP i (LNE=least next event). It is an upper bound on the time of the next event on this LP.

If we know the LNE times of all LPs which still have messages in their input buffers during a deadlock phase, then we only must compute the minimum $\tilde{t}$ of all LNE times. All events that occur at time $\tilde{t}$ can be simulated (see end of Sect. 5.1). To obtain a faster deadlock resolution, we maintain a list of LNE times:

- After the last test of LP i 's input buffers, the computed LNE time is added to the LNE list. This means that either a reference to LP i containing the LNE time is added to the list, or that the LNE time of LP i is updated if a reference to LP i is already present.

- If the buffers of an LP get empty, its reference is removed from the LNE list.

If we employ a static or stealing partitioning strategy, each processor j maintains a partial LNE list containing references to LPs that are assigned to j. If we employ a stealing strategy, several processors might write into one partial list. Then the partial lists must be protected by locks. However, as stealing happens seldomly, the number of collisions will be low. If we employ a dynamic strategy, each processor maintains the partial LNE list of all LPs that would have been assigned to it in a static partitioning strategy. The lists are also protected by locks.

To find the minimum timestamp $\tilde{t}$, each processor first runs over its own partial LNE list sequentially. Then $\tilde{t}$ is computed by a global minimum over all processors. If the load is balanced, then each processor spends a similar amount of time to compute the local minimum. The global minimum is done with a multiprefix operation in constant time.

Figure 8 shows the partial LNE list of processor 1, when a stealing strategy is employed. Processor 2 has stolen LP 33 from processor 1, has just computed the LNE time of LP 33 to 20, and has inserted a reference to LP 33 at the beginning of the list. Processor 3 has stolen LP 11 from processor 1. The buffers of LP 11 have become empty, therefore processor 3 removes the reference to LP 11 from the list. $\Longleftarrow$ **Fig.8**

6.4 <u>Performance</u>

Figure 9 shows the absolute speedups of PTHOR and the reimplementation on the SB-PRAM. The speedups of the reimplementation are much better than the PTHOR speedups. For the DASH benchmark, the speedup reaches the critical path bound. For H-FRISC and Multiplier there is still a gap $\Longleftarrow$ **Fig.9** between the bound from critical path analysis and the actual speedup. Experiments that try to tighten this gap are discussed in Subsect. 6.5.

The runtime of the reimplementation can be split into four phases:

1. Simulation of logical processes,
2. buffer tests, generation of messages and handling of SIMO, LNE and global activation lists,
3. Waiting,
4. Deadlock resolution.

Figure 10 shows the portions of these phases on the runtime for different numbers of processors. The portions are averaged over all processors. The figure clearly shows that for larger processor numbers, most time is spent in phase 3 (waiting). Hence, the small speedups for larger processor numbers do not result from increased overhead. Optimizations could try to have the processors do something useful during the waiting times. However, it is not obvious how to achieve this without increasing the runtimes of the other phases. $\Longleftarrow$ **Fig.10**

6.5 NULL-Messages and Dynamic Partitioning

First, we incorporate the concept of *NULL-messages*. In PTHOR, a message m is only sent when an LP i changes one of its outputs. In conservative simulation, m can be consumed when no messages with smaller timestamps arrive over this channel. The channel clock shows the timestamp of the last message sent over this channel. Deadlocks occur due to clocks not incremented far enough because of messages not sent. To prevent this, so called NULL-messages containing only a timestamp help to give better guarantees. Chandy and Misra show that deadlocks can be avoided completely if all events send all possible NULL-messages [11].

On distributed memory machines, the flood of NULL-messages can cause more overhead than the deadlock avoidance method. Therefore, one only sends part of the NULL-messages to avoid part of the deadlocks [13]. On shared memory machines, messages need not be sent explicitly. Every event can access each channel data structure in global memory. Therefore, instead of sending a message, one can update every channel clock directly. This removes most of the overhead of message passing (queue organization etc.) and makes NULL-messages a useful tool. To avoid deadlocks completely, every update of a channel clock must be followed by the activation of all LPs connected to this channel.

Figure 11 shows the speedup curves with and without NULL-messages for the Multiplier circuit. The use of NULL-messages almost doubles the speedup. $\Longleftarrow$ **Fig.11**

The situation is different for the H-FRISC circuit. Here, the use of NULL-messages results in an increase of activations by a factor of 6. The speedup drops by a factor of 5 to 6, depending on the number of processors. The reason lies in the different structures of the circuits. While Multiplier is purely combinatorial, H-FRISC contains cycles between registers. In these cycles, often several NULL-messages are sent (and hence activations happen) before an event can be simulated.

Baker and Mahmoody [6] also present an algorithm that optimizes the use of NULL-messages. They report an increase by a factor of three in combinatorial circuits taken from the ISCAS suite. However, the performance of their algorithm on sequential circuits is unknown to us.

Second, we tried to use the dynamic partitioning strategy as an alternative to stealing. To do this, one needs a shared FIFO queue as a global activation list. This list is accessed by all processors and hence need not lead to serialization. With the help of multiprefix, one can implement a FIFO queue that processes inserts or deletions of an arbitrary number of processors in a small constant number of instructions [23].

Figure 12 shows the speedups on H-FRISC for both strategies. The curves for the Multiplier circuit $\Longleftarrow$

Fig.12

look similar. In contrast to theory, the dynamic strategy is not superior to stealing. A reason for this is that more than 90 % of all activations are satisfied from the processors' local activation lists, even for large processor numbers. However, the dynamic strategy leads to a simpler program code. Note that the difference between the two curves is even increasing. This results from a constant runtime overhead while accessing the central FIFO queue.

7 CONCLUSIONS

Our results show that critical path analysis permits good speedup predictions if partitioning strategies are included. For the benchmark circuits, the SB-PRAM comes close to the maximum speedup, allowing more accurate predictions. As a consequence of using a single framework, the tool for critical path analysis also yields an efficient implementation.

For the prediction, we consider absolute speedup values. This is important to evaluate the use of parallel machines in practice as relative speedups are up to 10 times higher than the absolute ones. To make parallel simulators competitive, it might be worth investigating whether the slowdown factors from sequential to parallel can be made smaller.

Experiments with the benchmark circuits reveal that the maximum speedup is strongly dependent on the circuit's structure. Of particular importance are the length of the cycles and the number of inputs per LP. Our results strongly suggest to keep the number of inputs per LP low, if necessary by decomposing one LP into several smaller ones.

We presented several new serialization–free parallel data structures which seem to have a large impact on the programs performance. The efficiency of these data structures is based upon the use of parallel prefix operations.

The Dash machine supports so called fetch&op operations which are parallel increment/decrement.

Hence, SIMO queues and improved deadlock detection could be implemented on the Dash as well. However, the Dash's fetch&op still leads to serialization. Memory management and improved deadlock resolution require parallel prefix sum and maximum with integers, respectively, and thus cannot be used on the Dash.

References

[1] F. Abolhassan, R. Drefenstedt, J. Keller, W.J. Paul, and D. Scheerer, "On the physical design of PRAMs," *Computer Journal*, Vol. 36, No. 8, pp. 756–762, December 1993.

[2] G. Almasi and A. Gottlieb, "*Highly Parallel Computing*," Redwood City, CA:Benjamin/Cummings, 2nd edition, 1994.

[3] H. Avril and C. Tropper, "Clustered time warp and logic simulation," *Proc. 9th Workshop on Parallel and Distributed Simulation*, Lake Placid, NY, June 1995, pp. 112–119.

[4] M. L. Bailey, "A time based model for investigating parallel logic–level simulation," *IEEE Transactions on Computer–Aided Design*, Vol. 11, pp. 814–824, 1992.

[5] M. L. Bailey, J.V. Briner Jr., and R.D. Chamberlain, "Parallel Logic Simulation of VLSI Systems," *ACM Computing Reviews*, Vol. 26, No. 3, pp. 255–294, 1994.

[6] W. I. Baker, J. Herath, A. Jayasumana, and A. Mahmood, "A logic simulation engine based on a modified data flow architecture," *Proc. IEEE International Conf. on Computer Aided Design ICCAD-92*, Santa Clara, CA, November 1992, pp. 377–380.

[7] W. I. Baker and A. Mahmood, "An analysis of parallel synchronous and conservative asynchronous logic simulation schemes," *Proc. 6th IEEE Symposium on Parallel and Distributed Processing*, Dallas, TX, December 1994, pp. 92–99.

[8] O. Berry and D. Jefferson, "Critical path analysis of distributed simulation," *Proc. 1985 SCS Multiconference on Distributed Simulation*, San Diego, CA, January 1985, pp. 57–60.

[9] A. Boukerche and C. Tropper, "A Static Partitioning and Mapping Algorithm for Conservative Parallel Simulations," *Proc. 8th Workshop on Parallel and Distributed Simulation*, Edinburgh, Scotland, UK, July 1994, pp. 164–172.

[10] R. E. Bryant, "Simulation of packet communications architecture computer systems," Technical Report MIT-LCS-TR-188, Massachusetts Institute of Technology, 1977.

[11] K. M. Chandy and J. Misra, "Deadlock absence proofs for networks of communicating processes," *Information Processing Letters*, Vol. 94, pp. 185–189, November 1979.

[12] K. M. Chandy and J. Misra, "Asynchronous distributed simulation via a sequence of parallel computations," *Communications of the ACM*, Vol. 24, No. 11, pp. 198–206, April 1981.

[13] R. M. Fujimoto, "Parallel discrete event simulation," *Communications of the ACM*, Vol. 33, No. 10, pp. 30–53, October 1990.

[14] T. Grün, T. Rauber, and J. Röhrig, "The programming environment of the SB-PRAM," *Proc. 7th IASTED/ISMM Int.l Conf. on Parallel and Distributed Computing and Systems*, Washington D.C., October 1995.

[15] M. A. Gunter, "Understanding supercritical speedup," *Proc. of 1993 Winter Simulation Conference*, Los Angeles, CA, December 1993, pp. 81–87.

[16] D. R. Jefferson, "Virtual time," *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 3, pp. 404–425, 1985.

[17] D. Jefferson and P. Reiher, "Supercritical speedup," *Proc. of 24th Annual Simulation Symposium*, New Orleans, LA, April 1991, pp. 159–168.

[18] K. L. Kapp, T. C. Hartrum, and T. S. Wailes, "An improved cost function for static partitioning of parallel circuit simulations using a conservative synchronization protocol," *Proc. 9th Workshop on Parallel and Distributed Simulation*, Lake Placid, NY, June 1995, pp. 78–85.

[19] P. Konas and P.-C. Yew, "Partitioning for synchronous parallel simulation," *Proc. 9th Workshop on Parallel and Distributed Simulation*, Lake Placid, NY, June 1995, pp. 181–184.

[20] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam, "The Stanford DASH multiprocessor," *IEEE Computer*, Vol. 25, No. 3, pp. 63–79, March 1992.

[21] Y.-B. Lin, "Parallelism analyzers for parallel discrete event simulation," *ACM Transactions on Modeling and Computer Simulation*, Vol. 2, No. 3, pp. 239–264, July 1992.

[22] M. Livny, "A study of parallelism in distributed simulation," *Proc. of 1985 SCS Multiconference on Distributed Simulation*, San Diego, CA, January 1985, pp. 94–98.

[23] J. Röhrig, "Implementation of the P4 Runtime Library on the SB-PRAM (in German)," Master's Thesis, Universität des Saarlandes, 1996.

[24] J. P. Singh, W.-D. Weber, and A. Gupta, "SPLASH: Stanford Parallel Applications for Shared-Memory," *Computer Architecture News*, Vol. 20, No. 1, pp. 5–44, 1992.

[25] D. D. Sleator and R. E. Tarjan, "Self-adjusting binary search trees," *Journal of the ACM*, Vol. 32, No. 3, pp. 652–686, 1985.

[26] L. Soulé, "*Parallel Logic Simulation: An Evaluation of Centralized-Time and Distributed-Time Algorithms*," PhD thesis, Stanford University, June 1992.

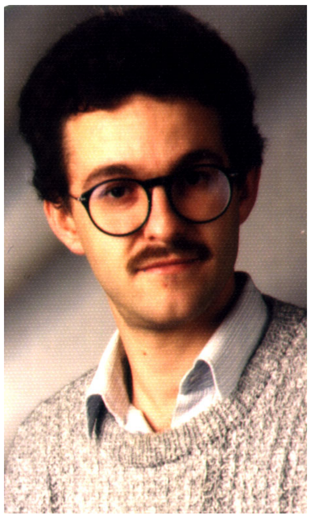
[27] C. Sporrer and H. Bauer, "Corolla partitioning for distributed logic simulation of VLSI-circuits" *Proc. 7th Workshop on Parallel and Distributed Simulation*, San Diego, CA, May 1993, pp. 85–92.

[28] T. Sterling, D. Savarese, P. Merkey, and K. Olson, "An Enpirical Study of the Convex SPP-1000 Hierarchical Shared-Memory System," *Proc. International Conf. on Parallel Architectures and Compilation Techniques PACT '95*, Limassol, Cyprus, June 1995.

Biographies

Jörg Keller received his Masters degree and PhD in computer science from University Saarbrücken in 1989 and 1992, respectively. Since 1996, he is professor for computer engineering at the FernUniversität (Open University) Hagen, Germany. His research interests are parallel architectures, hardware design, and parallel algorithms and data structures.

Thomas Rauber received his Masters degree and PhD in computer science from University Saarbrücken in 1986 and 1990, respectively. Since 1996, he is professor for practical computer science at the Martin-Luther-University of Halle/Wittenberg, Germany. His research interests are parallelizing compilers, performance prediction and modeling of parallel machines, discrete event simulation, and irregular applications on shared-memory machines.



Bernd Rederlechner received a Masters degree in computer science from University Saarbrücken in 1996. The presented article summarizes the results from his master's thesis. He currently works for the german Telekom AG (Entwicklungszentrum Süd-West, Saarbrücken, Germany) as a software developer. His research interests now focus on object-oriented software design, network management, GUI-design and parallel multigrid algorithms.

Figure Caption List

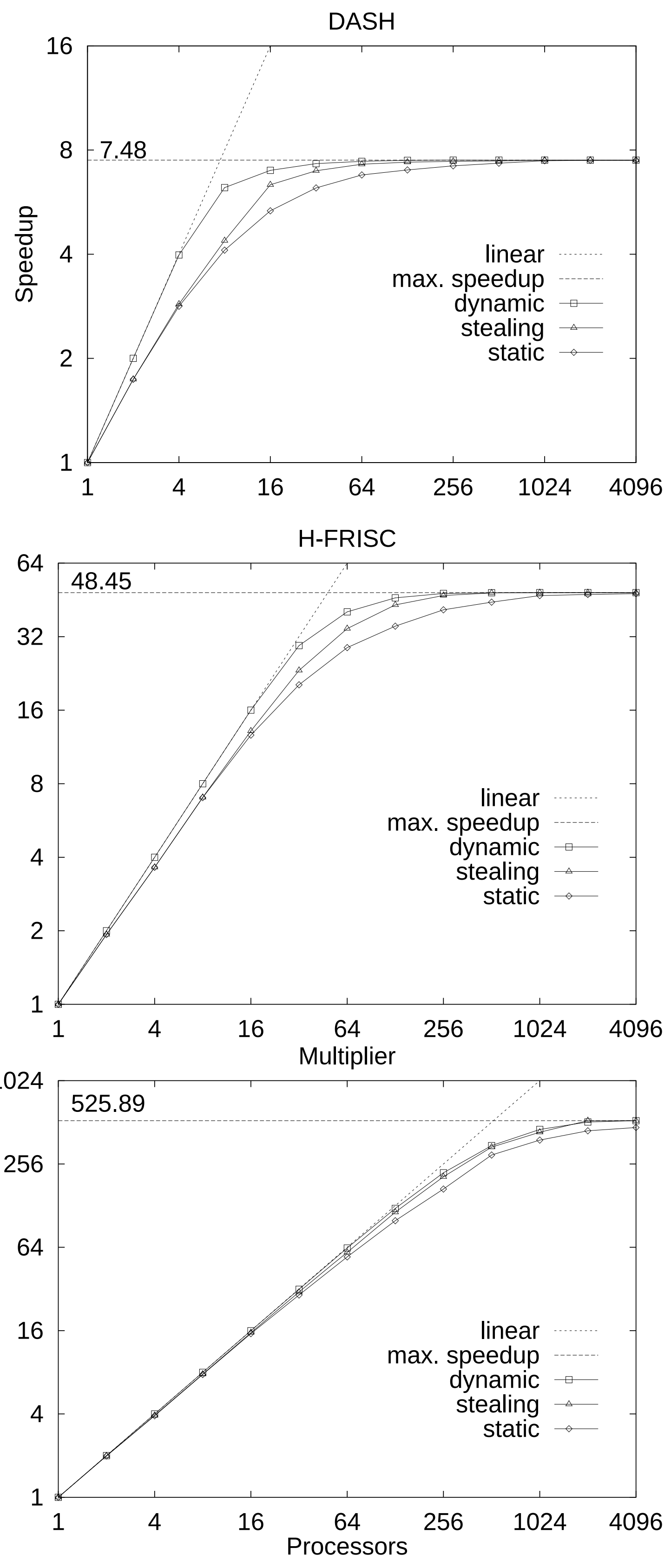


Figure 1: Speedup bounds for different partitioning strategies

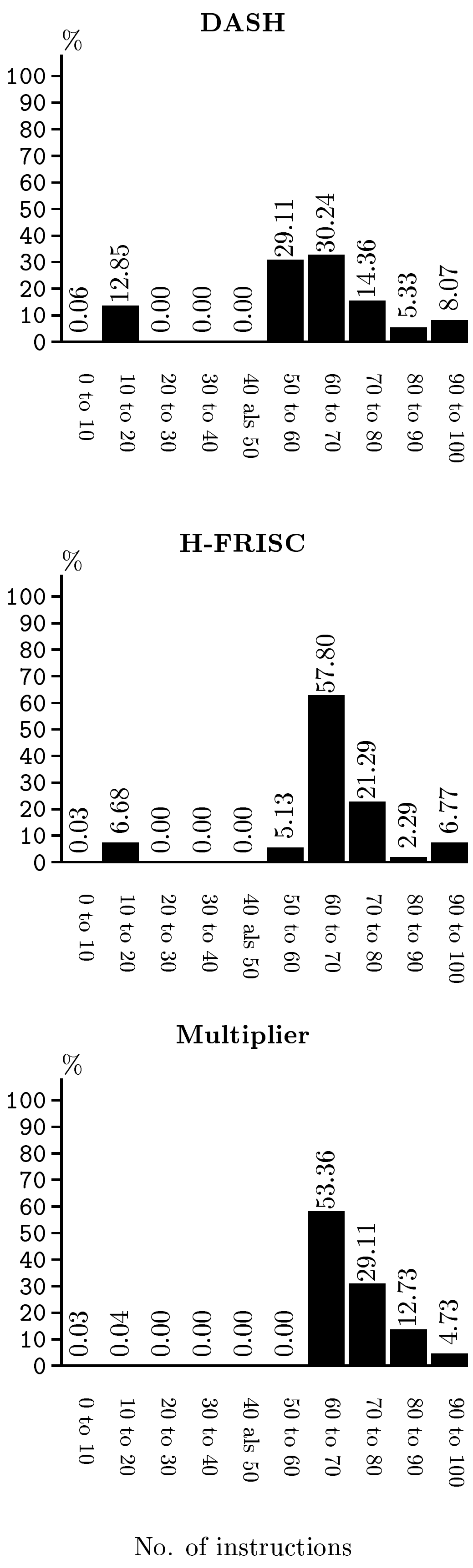


Figure 2: Granularity of LP execution times

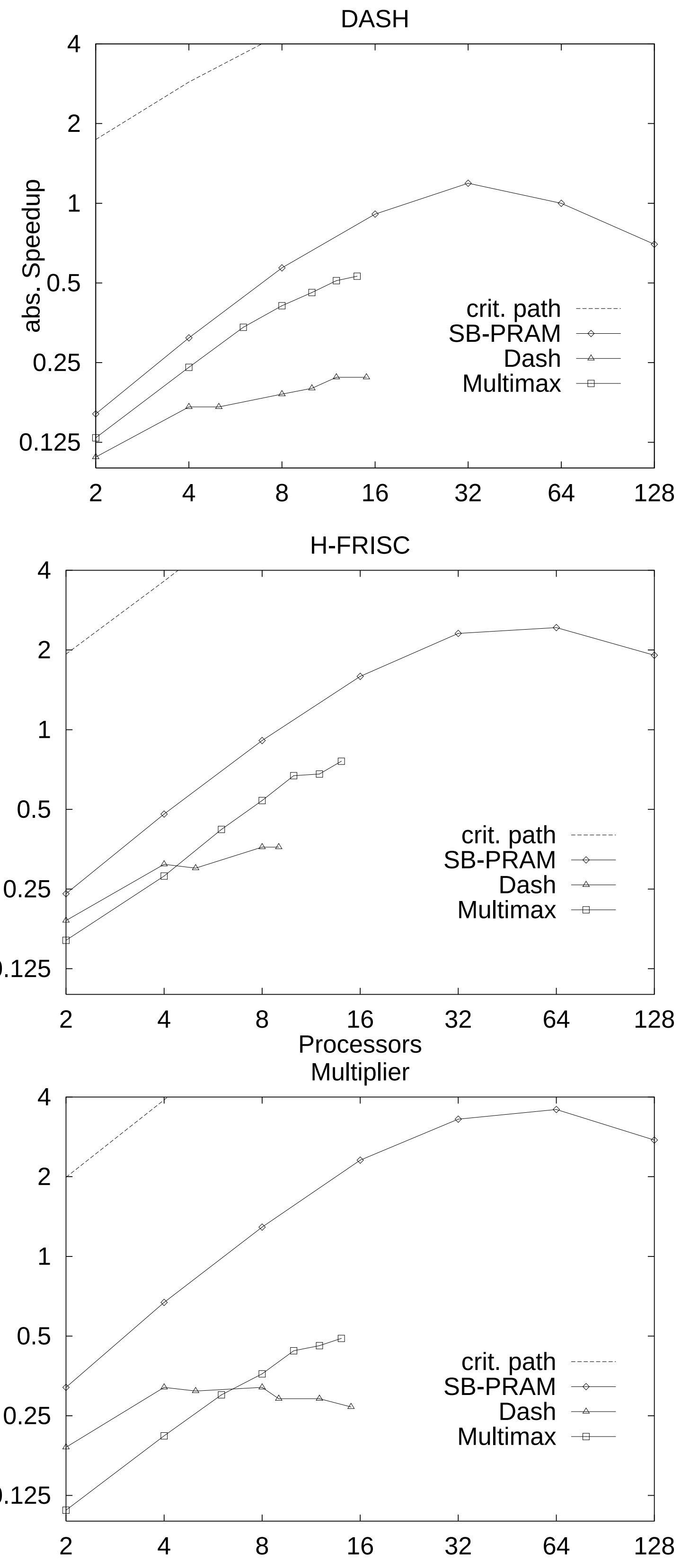


Figure 3: Absolute speedups of PTHOR on the Dash–, Multimax– and SB-PRAM–Multiprocessor.

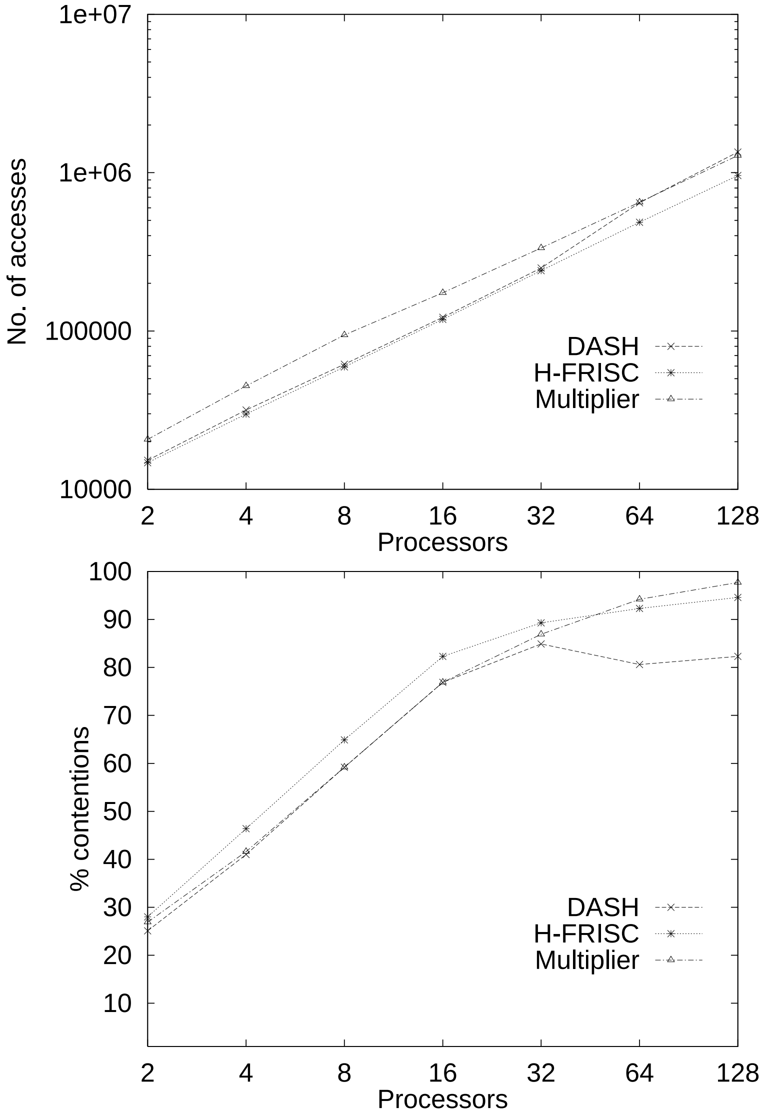


Figure 4: Lock contention on SB-PRAM

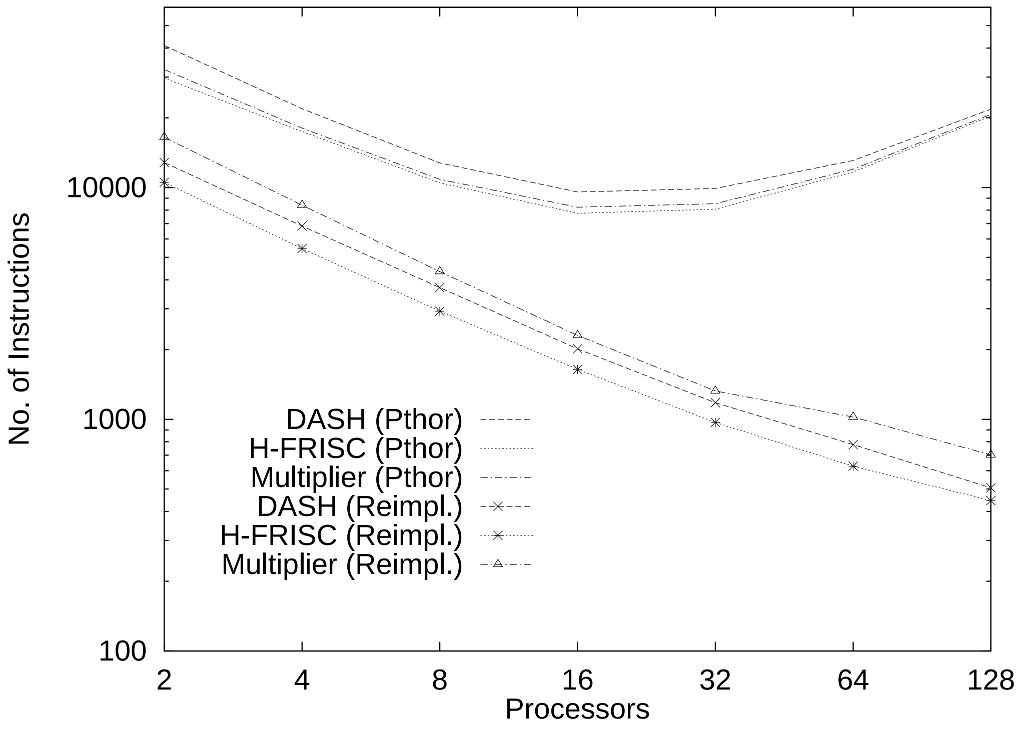


Figure 5: Duration of deadlock resolution on SB-PRAM

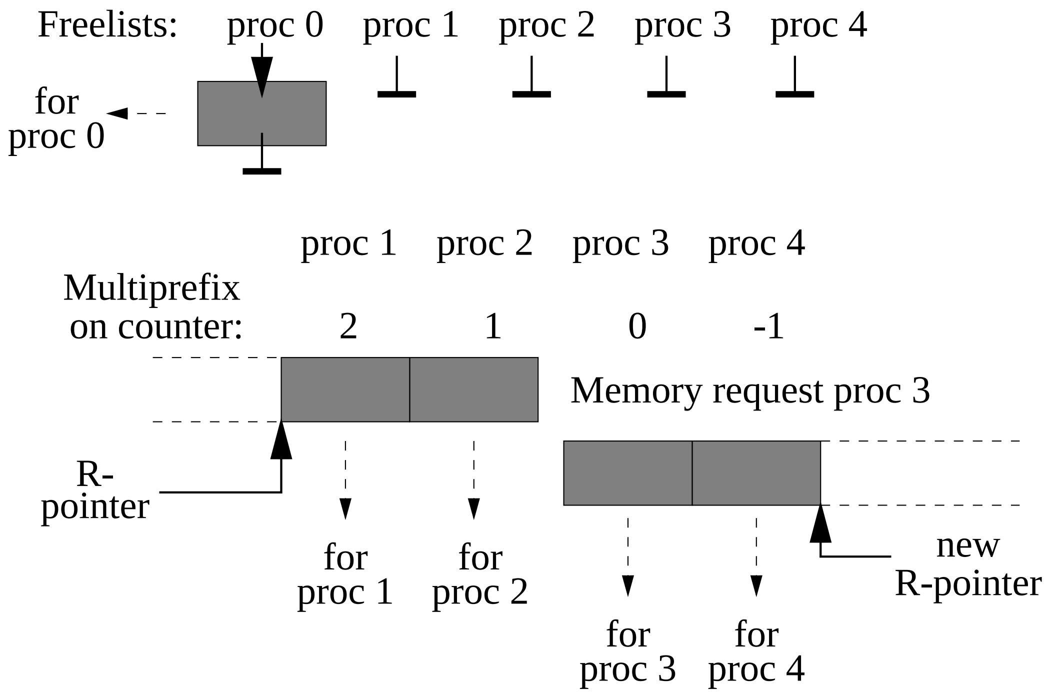


Figure 6: Memory management of list elements

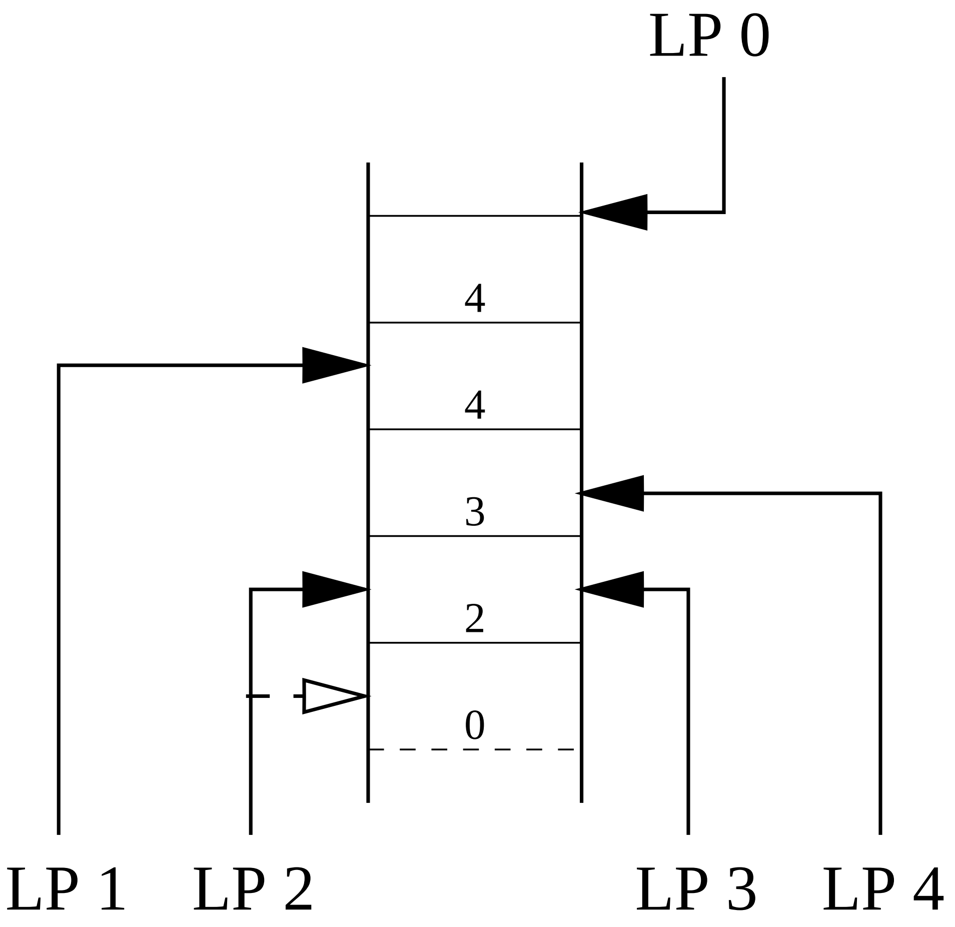


Figure 7: Single-In Multiple-Out queue

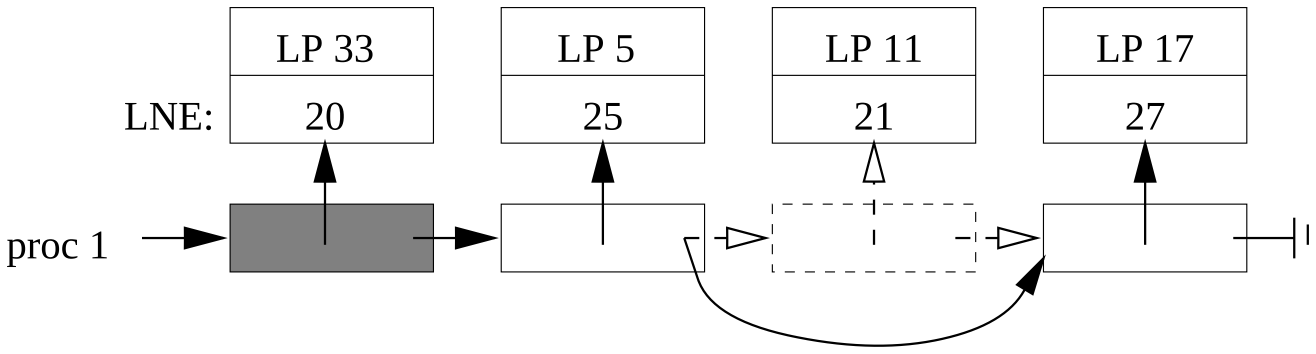


Figure 8: Partial LNE list of processor 1

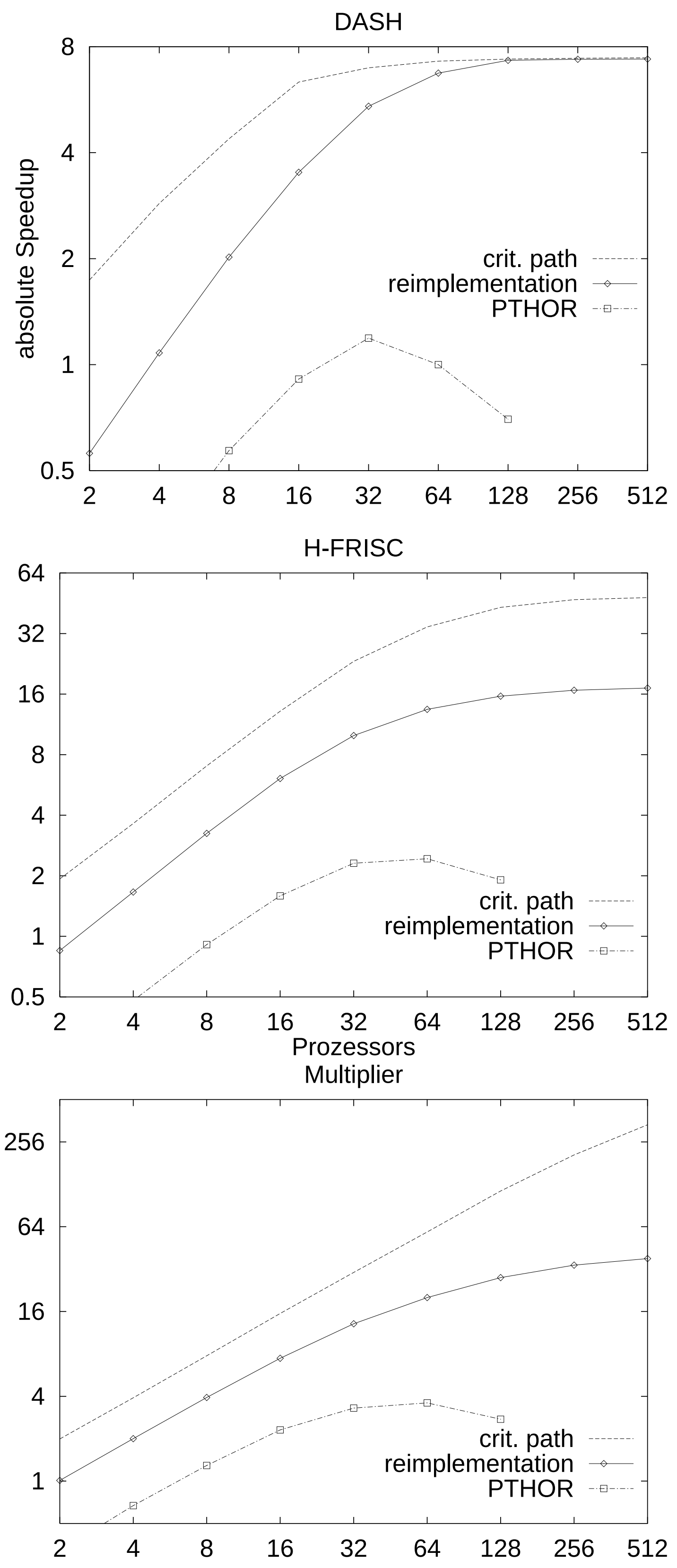


Figure 9: Absolute speedups before and after reimplementation on the SB-PRAM

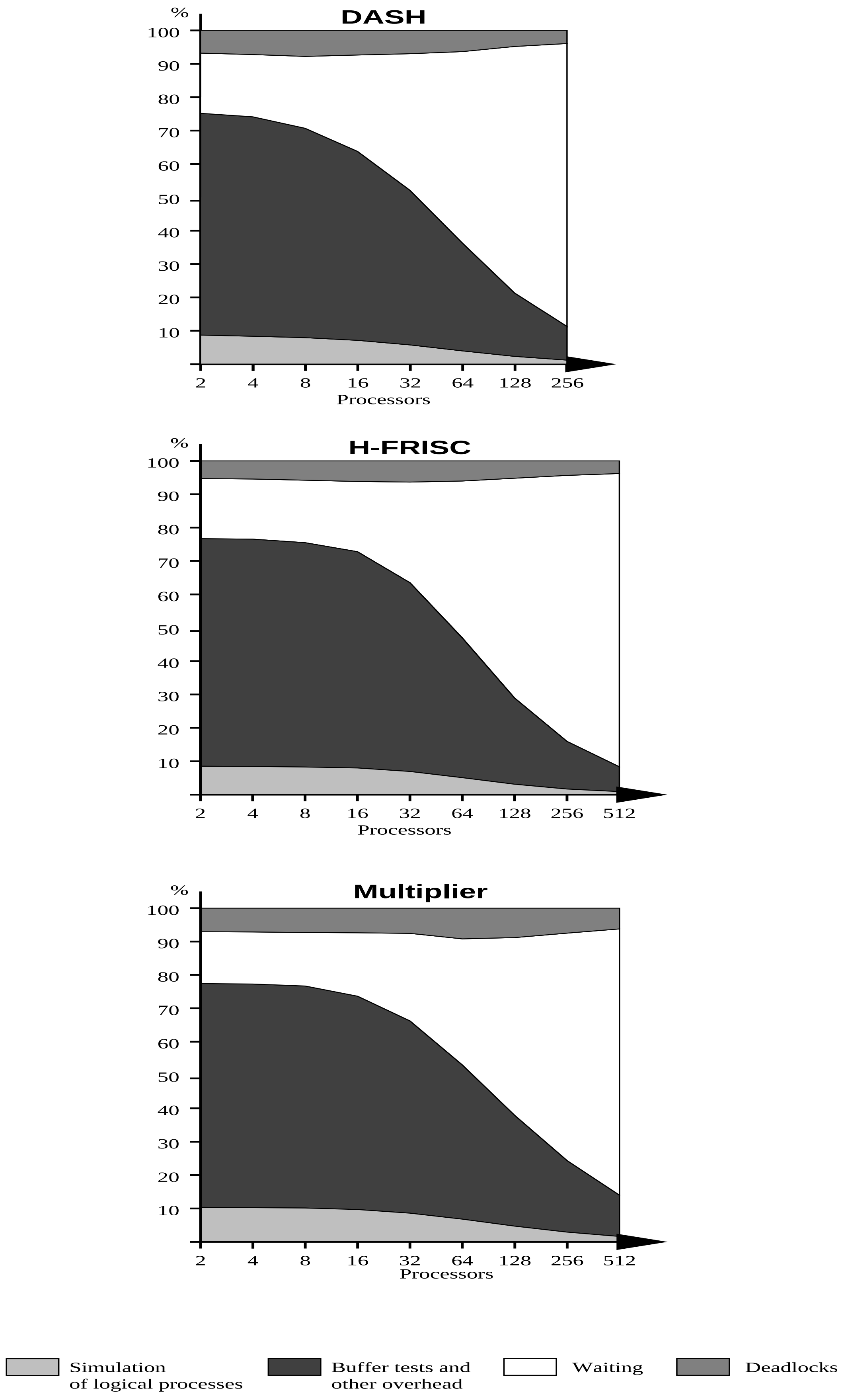


Figure 10: Portions of phases on execution time

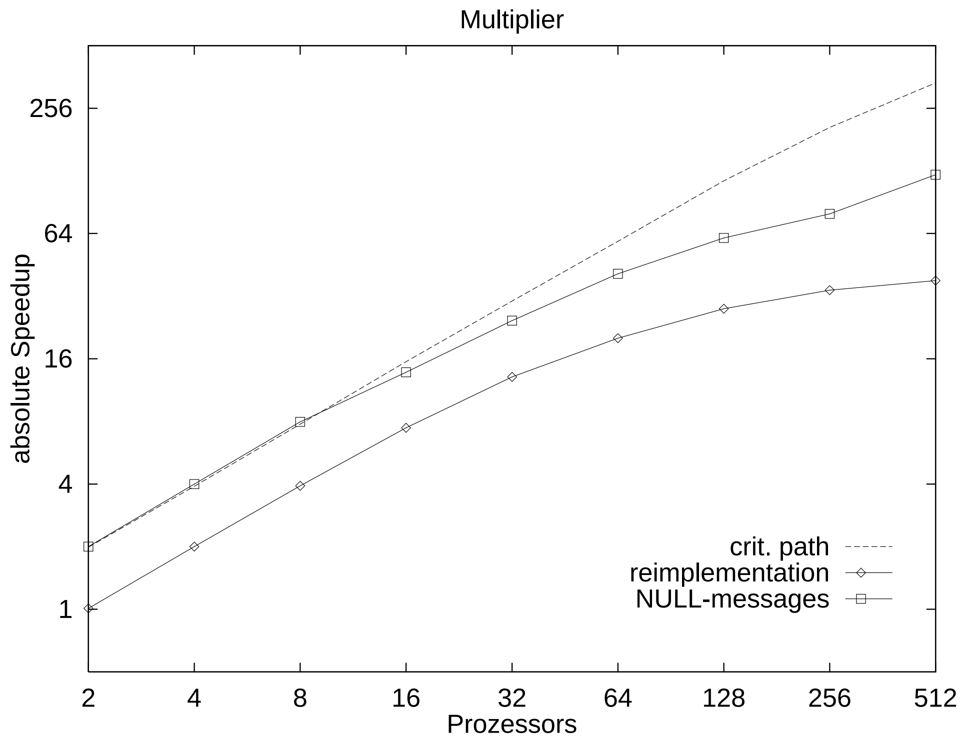


Figure 11: Use of NULL-messages

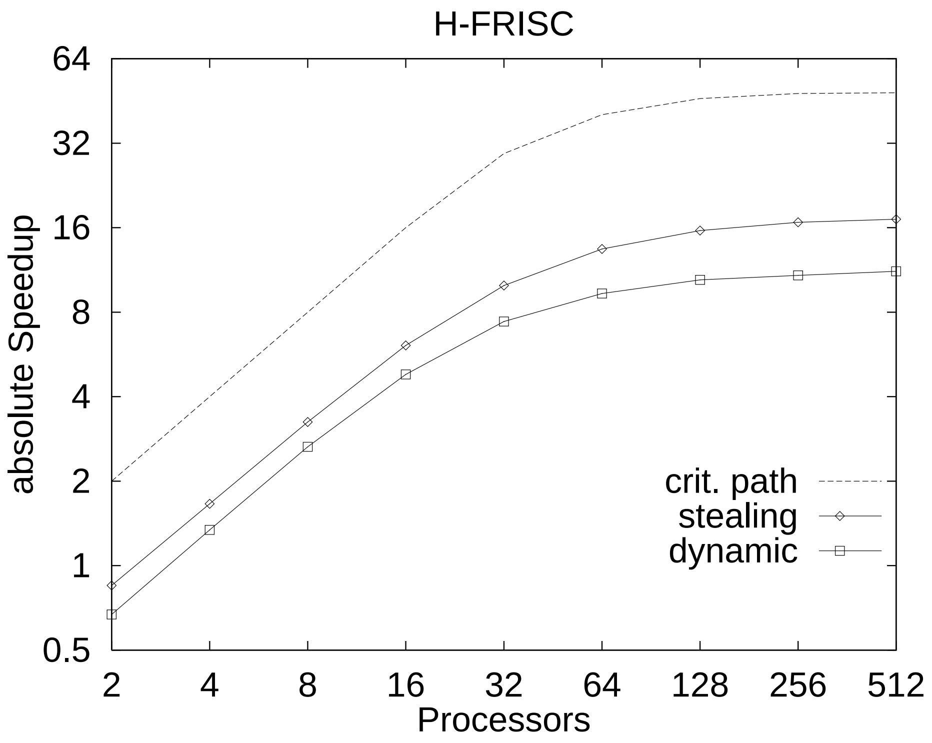


Figure 12: Absolute speedups for dynamic and stealing partitioning