Report on WarpKit

Performance Study and Improvement

Zhonge Xiao, Brian Unger
University of Calgary
Canada
email:xiao@cpsc.ucalgary.ca

May 1995

Contents

Abstract

This is a report on the earlier development of WarpKit, a parallel simulation kernel based on shared-memory multi-processor archietectute, as part of the Telesim project. The development is aimed at exploiting shared memory multi-processor paradigm and developing a Parallel Discrete Event Simulation package which is based on shared memory multi-processors and capable of delivering high performance. Three major problems that have great impact on the performance of Time Warp systems are: excessive cost incurred by rollback computation resulting from sole reliance on rollback as a basic synchronization mechanism in a distributed/parallel processing system, large amount of memory space required to run applications, and high system overheads in inter-process communication and global control (e.g. GVT computation and memory management). Shared memory multi-processor architecture provides the potential of delivering much higher performance for Time Warp systems than can be achieved in distributed environment. New approaches could be conceived to address these problems and to realize the potential.

This report covers the results of our effort to improve WarpKit Kernel performance. Incremental State Saving has been implemented on top of existing Kernel which reduces both the time and space spent on state saving, a necessity of Time Warp. Purely asynchronous schemes have been developed and implemented for the global control mechanism. As a result, the system system overhead on global control has been reduced significantly. The new global control mechanism also makes the system overheads insensitivity to to the number of processors as opposed to the distributed situation where system overhead experiences a sharp increase with the number of processors. A global scheduling and load balancing mechanism is expected to restrict the number of rollbacks to a low percentage over net events to be processed by the Kernel. With these new mechanisms in place, one may expect close to linear speedup curve for parallel discrete event simulation on shared memory multi-processors.

1 Overview

WarpKit is a shared memory based TimeWarp simulation executive developed jointly by University of Calgary, Canada, University of Waikato, New Zealand, and Georgia Institute of Technology, US. The primary goal of this ungoing project is to exploit shared memory multi-processors paradigm and develop a TimeWarp Kernel capable of supporting Parallel Discrete Event Simulation (PDES) and delivering high performance. This Kernel has to be not just a experimental model for academic purpose but one that delivers real performance and may serve as the basis for the development of large scale simulation applications. The question is "How" as this has been an area intensive R & D efforts have been going on for many years. Our strategy has been that starting with an existing kernel, study the behavior and performance of WarpKit to identify problems and potentials, research and develop new approaches and methods to solve problems which affect the performance most. The Kernel development work is organized in several phases moving from basic performance study towards tackling with most challenging problems.

The Kernel versions studied in this report are the earlier versions of WarpKit before the formal WarpKit interface was defined and a new state saving mechanism (by Fabian Gomes in PhD. thesis) introduced, hence are refered to as SMTW Kernels. The main futures of the WarpKit, especially the global control mechanisms, have been implemented and experimented in the SMTW kernels.

1.1 Work Phase I

The first phase of work on the kernel which ended with the delivery of SMTW Kernel 2.2 (Oct. 31 version) was instrumentation and problem identification. The original kernel developed at GIT (Fuji89) for the purpose of protocol study is compact and reasonably reliable which provides a very good basis to start with. In order to understand the behavior of SMTW, the effect of underlying shared memory architecture, the application's performance, as well as the kernel's performance, correct and detailed timing data are of fundamental importance. The very first thing we did was restructuring the instrumentation facility and verifying its correctness. Three levels of timing and statistics data collection have been installed reporting with satisfactory accuracy different types of overheads and user times. Also installed is the on-line critical path execution time computation which indicates the maximum parallelism an application permits, what one may expect and how well the kernel and the model perform.

The Incremental State Saving (ISS) mechanism has been incorporated into the kernel, which was also part of the work done in the first phase. This is not directly related to the kernel performance, but certainly is a major factor which has a great impact on speedup. One of the problems with Time Warp is state saving which consumes both memory space and execution time. With large scale application models the overheads of space and time could be so high that it actually prohibits the use of Time Warp mechanism. Modifications have been made in the kernel to support ISS, an implementation of ISS at C language level is built on top of the existing kernel which allows CSS (Copy State Saving) and ISS to coexist, that is CSS and ISS can be used for different parts (logical processes) of an application.

The main problem in discrete event simulations with distributed systems in terms of speedup is the high communication cost of both the hardware and software. The characteristics of system overhead in the distributed simulation is its high sensitivity to the number of nodes and the intensity of inter-node communications. The increase of communication cost easily absorbs the benefit of increased processing power and quickly reaches the top of speedup curve. The desired property of a distributed/parallel simulation kernel which delivers good performance in terms of speedup should be that the overall system overhead does not increase significantly with the increase of the number of processors, i.e. low sensitivity to the number of processors. With such a property, one may expect very good speedup (relative speedup) with applications that permit sufficient parallelism and have reasonably good partitioning so that no excessive rollbacks would occur. This characteristics is what we are trying to achieve with the SMTW kernel as shared memory architecture provides the potential that would be impossible with distributed systems.

Experiments with existing SMTW kernel show that those system overheads like forward execution TimeWarp overhead, referred to as normal TimeWarp overhead and send time do not increase significantly with the number of processors running the simulation. However, The cost of free pool access, GVT computation and fossil collection are very high and worse is that the cost increases drastically with the number of processors. The bottle-neck lies in the time spent on spinning for memory due to the high contention caused by central free pools. These overheads increase very fast with the number of processors, they become dominating cost far before the number of processors getting any large and no further speedup is possible. This behavior is similar to the situation with distributed systems. It suggests that we definitely need new algorithms for shared memory processors, which should avoid use centralized schemes on the regular basis.

1.2 Work Phase II

The second phase of work on the SMTW kernel is to design and experiment with new algorithms which will give the desired characteristics described above. The first thing we did in this phase was experimentation with distributed free lists scheme and this resulted in SMTW kernel 2.3. The improvement is drastic, the serious bottle-neck of spinning for memory buffer no longer exists. Experiments with SS7 model show that rollback overhead has also dropped considerably due to the relieved contention of free pool access. Running parallel pingpong program with long free list (eliminating the effect of GVT computation) gave almost proportional speedup up to 16 processors.

The cost of GVT computation and fossil collection is still high and it increases significantly with the number of processors, mainly because of the synchronized approach (effectively stopping production computation for the entire system). To solve this problem we have worked out a new GVT computation algorithm, called Fast Asynchronous Continuous (FAC) GVT estimation, and a new distributed free lists scheme. The results of tests with FAC algorithm and combined distributed free list scheme are very encouraging. The statistics output from parallel pingpong with short free event buffer list (meaning more frequent GVT computation) and high state change rate (20% on 500 state integers) gave almost ideal relative speedup (15.05 on 16 processors and 17.28 on 18 processors). Tests with SS7-PM model indicate significant decrease in global control overheads. The overall kernel overhead is no longer sensitive to the number of processors and does possess the desired property. These new algorithms have been implemented in SMTW kernel 3.1. There are still rooms for optimization and fine tunning to further improve kernel's performance. This could be done in the future, and now is the high time to move onto the third phase.

1.3 Work Phase III

The statistics outputs from qnet (hyper-cube queueing network) and SS7-PM show that the percentage of rolled back events over total processed events is very high on more than 8 processors. It could reach 60% to 70% on more than 10 processors, this means that committed events account for only 30% to 40% of total events. Rollback is expensive and the cost for rollbacks is high. Better partitioning of application models could improve efficiency as reported for many cases. Our experiments with new partitioning method for SS7-PM model also reported highest speedup of 14 on 30 processors with around 55% efficiency). But we believe that a general method of partitioning for parallel simulation with satisfactory efficiency is almost impossible. TimeWarp is a synchronization mechanism for distributed/parallel processing, but solely depending on TimeWarp to synchronize parallel executions inevitably leads to excessive rollbacks and the high cost incurred. To exploit the possible optimism that TimeWarp may deliver, dynamic load balancing or some other scheduling method, which is efficient but does not have to be perfect, should collaborate with TimeWarp and the latter acts as a backup mechanism for synchronization. The third but not necessarily the last phase of work on the SMTW kernel should be focused on dynamic load balancing or other mechanisms which could do away completely with partitioning at user level. If we are successful in restricting the number of rollbacks, without introducing excessive extra cost, to a low percentage over net events to be processed by the Kernel, one may expect close to linear speedup curve for parallel discrete event simulation on shared memory multi-processors.

2 Statistics and Timing Data report

The SMTW Kernel instrumentation has been designed specifically for the purpose of performance study for both the application model and the Kernel itself. Based on the original version, the instrumentation has been restructured to provide more detailed and classified statistics and timing data. Since the correctness and accuracy of the data are fundamental to the performance study, great effort has been put into verifying and correcting the data collection facility over a long time period.

Three levels of statistics and timing data output are provided. First level is the summary data, and the rest are optional data output for each processor and each logical process respectively. Appendix A gives the exact definition for each data item. Here is the general description of the three levels.

Summary Time Warp Statistics

This is the output data after each successful simulation run. Four sets of statistics and timing data are given under this level.

- The numbers of processed events, rollbacks and cancellations including the numbers of straggler and cancellation induced rollbacks, the number of artificial rollbacks, the numbers of rolled backed events caused by these rollbacks, the number of canceled processed events, the number of aborted events, as well as the total number of events processed whether or not they have been rolled back or canceled.
- summary of the above numbers, i.e. the number of net events (committed events), the total number of rollbacks, the total number of processed events rolled back, the total number of canceled processed events. and their percentages over total processed events.
- The summary timing data, including actual execution time and real time span of the simulation, event rate (net events processed per second). Also given here are the Critical Path user execution time and Sequential user execution time which will be explained in a separate section of this report.

- To be comparable with other discrete event simulation systems, the effective execution time per net event, which is the inversion of net event rate, is reported being broken down into different types of system overhead and user execution time. These are the average time sent or amortized on each net event regarding the multi-processors as a single parallel machine. The real average times per net event should be multiplied by the number of processors running the simulation and are given in brackets so that it is much easier to examine how each type of overhead changes with the number of processors. It is ideal that real per net event times change little with the number of processors.

The following is an example output from a simulation run on eight processors.

```
 *** Summary Time Warp Statistics***

Events processed          517228
Events aborted                 0
Processed Events Cncl'd    38018
Straggler Induced RBs      79840
  Processed Events RB'd   120117
Cncl Evnt Induced RBs      10590
  Processed Events RB'd    15154
Artificial RBs                 0
  Processed Events RB'd        0
Longest rollback              10

Net Events processed      343939 ( 66.50 %)
Number of rollbacks        90430
Proc. Evnts RB'd          135271 ( 26.15 %)
Proc. Evnts Cncl'd         38018 (  7.35 %)
Average RB Distance         1.50

Actual execution time (Sec)        30.579060 (31.729947)
  Initialization time               0.542240
Event Rate (NetEvnts/Sec)           11450.58
```

```
Critcl Path user execution time     3.188215
Sequential user execution time     25.077185 (50.813571)

Effective time per net event      87.33 ( 698.65) (Micro-sec)
  Per event TW overhead           14.88 ( 119.06)
    (copy time)                    0.00 (   0.00)
  other                            8.64 (  69.15)
  wasted TW overhead               7.50 (  59.99)
  idling                           6.28 (  50.25)
  RB overhead                      9.44 (  75.49)
  user execution time              9.11 (  72.91)
  user send time                   9.35 (  74.83)
  wasted user time                 9.30 (  74.44)
  Pools, GVT & termination        12.82 ( 102.53)
```

Monitoring Statistics

This is an optional output of timing data. Detailed per processor average timing data are given in three categories with each further broken down into different types of average time consumption on each processor. The sum of the totals of three are reported under "Total Execution Time" as shown by the following example.

```
 *** Monitoring Statistics***

GVT Computation, Fossil Collection and Cancelback:
  Total spin time                   0.268205 sec
  Total Computation time            0.649925 sec
    Termination time                0.000646 sec
  No. of calls 446
  total                             0.918129 sec

Time Warp Overhead:
  Queue search time                 1.793328 sec
  Idling time                       2.160410 sec
  StragglerRB time                  1.563453 sec
```

```
  SecondaryRB time                  1.682240 sec
  real overhead                     5.904529 sec
    (copy time)                     0.000000 sec
  other                             2.972894 sec
  total                            16.076853 sec

Event processing:
  Computation time                  9.551912 sec
  Spinning for memory               3.489924 sec
  total                            13.041837 sec

Total Execution Time:              30.036820 sec
```

Time Warp Statistics
This is also an optional output of detailed statistics and timing data. Two sets of data are available. The statistics data described in the first set of "Summary Time Warp Statistics" are reported for each logical process. The timing data described in "Monitoring Statistics" are reported separately for each processor instead of per processor average.

3 Benchmark Programs

Three benchmark programs are used in the Kernel performance study. Each has quite different chracteristics so that the performance of the Kernel can be observed in both normal and stressed situations.

3.1 Parallel Pingpong

Parallel Pingpong is a modified Pingpong program which has been designed specifically to measure the SMTW Kernel performance. The user event processing time, or event granularity, could be set to high or negligible so the basic overheads of the Kernel can be measured more closely. The Parallel

Pingpong program consists of equal number of processes (LPs) as the number of processors (PEs), each LP runing on a separate PE. So the number of LPs (N) varies depending on the number of PEs that execute the program. Initially, each LP sends a message to its next neighbor and another self-directed message with both send-time equal to its ID number (from 0 to $N-1$). For each received message, a LP advances its local clock to the receive time of the message, stamps the receive time of the out going message to its clock time plus N time unit, then sends the message to its next neighbor if the message comes from its previous neighbor, otherwise the message will be self-directed. When running on single processor, there is only one LP and all the messages are self-directed. Artificial computation could be inserted and parameterized, so is the state variables. The program has the following properties:

- At any time there will always be at lease $2N$ events either being processed or pending. Each LP has at least one message for processing, this eliminates idling. Total parallelism is guaranteed by the application and a few rollbacks are possible in normal situations (small number of PEs or enough event granularity).
- Ideal relative speedup, defined as the net event rate divided by the event rate obtained when runing the program on single processor, equals to the number of PEs on which the program is running. The closeness of real relative speedup curve to the ideal one can be a measurement of the Kernel's overhead characteristics and how well the algorithms adopted in the Kernel performance.
- The program could also become very stressing if the event granularity set to very low and the number of PEs getting large. This is due to the self-directed messages and could be used to test the global control mechanism under irregular conditions.
- The amount of committed messages up to a certain simulation time is fixed (twice the number of simulation time units) no matter how PEs are involved.

3.2 Queueing Network

The second benchmark program is a simulation model of closed queueing networks configured in a hypercube topology (Fuji89). A fixed number of jobs (messages) randomly circulate throughout the network, referred to as *message population*. To process an event, the simulator first selects an outgoing link using a uniformly distributed random variable. A server is associated with each outgoing link. The queueing discipline used is First-Come-First-Serve (FCFS). The service time is selected from an exponentially distributed random variable with mean of 1.2. No additional delays or busy wait loops are inserted into the program to artificially increase the user execution time of each event. A four dimensional hypercube was used for experimentation. Among the properties of this model are:

- it contains a reasonable amount of parallelism (about maximum 8 speedup with 4 dimension hypercube).
- it is homogeneous and highly symmetric, so it is free of any inherent bottlenecks and has uniform computation and communication patterns.
- the four dimensional hypercube when combined with the relatively small event granularity and a modest message population, an erroneous computation tends to spread very rapidly, but still well restrained so no one can go too far ahead.

Compared to the SS7-PM model described below, the Queueing Network model behaves well in the sense that no cancelback invocation is needed except in extreme situations.

3.3 SS7 Performance Model

A performance model of a Signaling System No.7 (SS7) was built and used as a more realistic large-scale application to test the Kernel's performance (Gom93). The functions in the layered SS7 protocol can be modeled by the following five types of components:

- Data Link functions of MTP level 1 and 2 are modeled by the *Link*.
- Network function (MTP level 3 and SCCP) modeled by the SS7 *Network Server*.
- Network management function modeled by the *SS7 Network Manager*.
- User part function modeled by *ISUP*.
- Database function modeled by the *SCP Database*.

The trunk traffic generation and trunk network management functions can be modeled by the following three components:

- *End Offices* model end offices and tandem switches.
- *Trunks* model trunk groups connecting end offices and tandem switches.
- *Network Processors* model trunk network management functions.

The performance model (SS7-PM) was also built using these seven types of objects. The SS7-PM was designed so as to have the same run time characteristics of any real SS7 simulation model, particularly in terms of:

- the number of LPs in the real simulation.
- the number of instructions executed on an average to characterize LP and event processing simulation code.
- the number and types of events scheduled at a LP.
- the size of a LP's state and amount of state modified on an average.

The traffic for two service groups, namely the basic bearer service and 800 service were modeled within the trunk network. In the SS7 network the generation of Signaling Units for each call in the service group were modeled.

This performance model represents an application of real challenge in that the requirement for huge memory space (run time program memory size well exceeds 1 Gbytes), large number of LPs (4036 LPs), difficulty in obtaining good partitioning. Very rapid spread of erroneous computations and excessive rollbacks are observed in the experiments. Improper partitioning easily leads to extremely irregular workload and unusual simulation behavior.

4 Performance Metrics and Charts

4.1 Performance Metrics

Speedup is the most important performance metrics that the users really care about. The main goal of this ongoing project on the SMTW kernel part is to study and to improve kernel's performance by experimenting with different algorithms and mechanisms. Here we define speedup as relative to the execution time on a single processor with the same SMTW Kernel version. That is the execution time of a simulation model on a single processor divided by the execution time of the same model on multi-processors, it is referred to as *real relative speedup*. For Time Warp simulations, the time spent on events that eventually will be rolled back is a wasted time and prolongs the total execution time resulting in lower speedup. To estimate the possible speedup without rollbacks, which would be a good measure of the performance of both the kernel and the application model, we define *expected relative speedup* as *real relative speedup* divided by the fraction of committed events over total events. The accuracy of this estimation varies and therefore it is not meant as a precise measurement.

The overheads incurred by the Kernel affect the speedup significantly. In fact, the main problem in discrete event simulations with distributed systems in terms of speedup is the high communication cost of both the hardware and software. The increase of communication cost easily absorbs the benefit of increased processing power when more nodes are added and quickly reaches the top of speedup curve. The characteristics of system overhead in the distributed simulation is its high sensitivity to the number of nodes and the intensity of inter-node communications. The desired property of a distributed/parallel simulator which delivers good performance in terms of speedup should be that the overall system overhead does not increase significantly with the increase of the number of processors, i.e. low sensitivity to the number of processors. Ideally, suppose that the system overhead spent on each event (t_{sys}) is constant, it does not change with the number of processors (n) on which the simulation is running, also assume that the event processing time on the application part is also constant (t_{appl}), this is probably true if locality is well observed on a shared memory machine, and enough parallelism is provided by the application (no processor idling), then a full speedup of n can be achieved when running simulation on n processors:

$$T_n = \frac{(t_{sys} + t_{appl}) * N_{event}}{n}$$

$$T_1 = (t_{sys} + t_{appl}) * N_{event}$$

$$S_n = \frac{T_1}{T_n} = n$$

where T_n is the execution time of a simulation of N_{event} events
on n processors,
T_1 is the execution time of the simulation on a single processor,
S_n is the ideal *expected speedup.*

The performance of an algorithm or a mechanism used in the simulator can also be described as the amount of overhead introduced by it and the characteristics of the changes of overhead with the number of processors. *low sensitivity* of overhead to the number of processors is a desired property of algorithms in PDES.

4.2 About the Performance Figures

Three types of charts are drawn based on the statistics output from the benchmark tests. The *Actual Per Event Times* chart depicts by stacked areas the composition of average per event overheads and the event processing time spent in user code on varied number of processors. At the bottom is the global control overhead caused by GVT computation, Fossilcollection and memory management, also including the time spent in acquiring a free event buffer or a state block, it is amortized to each committed event. Above that are normal TimeWarp overhead and message send time, these are the average intrinsic overhead incurred by the Kernel when processing a event. On top is the idling time distributed to each committed event, below idling time is the average user execution time for a event. All the rollback related overheads are also distributed to each committed event. The *Other Overhead* refers to the time that can not be recorded in any of the above overheads and it is incurred by both forward execution and rollback. The *Global Control Overheads* chart breaks down the global control overhead into two parts. The overhead of *Spinning for Memory* turned out to a dominant fact in a centralized free list scheme. The *Relative and Expected Speedup* chart displays the relative real speedup, the expected speedup. The ideal speedup is also displayed if it is conceivable.

5 SMTW Kernel 2.2 and Performance

5.1 SMTW Kernel 2.2

In shared memory systems, increased inter-processor communication does not necessarily increase message passing overhead significantly as opposed to the distributed systems, unless excessive contention exists. It is possible that the TimeWarp overheads be reduced close to sequential overheads, which implies much greater potential for speedup by parallel execution.

The original SMTW Kernel developed at GIT (Fuji89) has been successful in attempting to minimize TimeWarp overheads arising from various sources.

- Although the Kernel does not alleviate state saving overheads, it eliminates the need to keep extra copies of outgoing messages.
- Message passing overhead is largely eliminated by allowing processes to insert events directly into the input queue of other processes. Here,

the only additional penalty associated with scheduling an event, when compared to a sequential event list implementation, is that for remote memory references, which are generally more expensive than local references, and locking overhead.

- Rather than using anti-messages to cancel erroneous computations, a *direct cancellation mechanism* is used. This mechanism merely dereferences a pointer variable to locate the event that is to be canceled.

The direct cancellation mechanism is the most innovative aspect of the Kernel and critical to reduce the TimeWarp overheads in both forward execution and rollback process. Kernel version 2.2 is based on GIT's source code and has been reengineered to provide more accurate timing data. Also to address the state saving problem, an *Incremental State Saving* (ISS) mechanism at C language level has been implemented on top of the Kernel so that both ISS and *Copy State Savin* (CSS) are supported.

5.2 SMTW architecture

The SMTW being described here has a layered structure. At the bottom is a light weight kernel which provides an interface suitable for event-driven discrete event simulation. An additional layer built on top of the kernel deals with ISS management. The application code interacts with the kernel and ISS via a language layer which gives safe version of system calls and is expected to deal with state saving, I/O, and message passing. The application typically consists of a set of logical processes (LP) interacting by sending messages to each other. The LPs process messages as events scheduled and presented by the kernel. The interactions between the application and the kernel are described below.

User-defined initialization procedures are invoked from the kernel once at the start of a simulation run. An event information data structure instance is created each time a message is to be sent out. This event data lasts the entire life time of the event. Among other things such as send time stamp and receive time stamp, two pointers are stored in the event data structure, one points to the message data and the other is a pointer to a memory block used for saving state, called *state block*. Different procedures are applied to save and restore state. The default procedure is for ISS while CSS can be turned on for individual LPs. The kernel schedules arrived events only based on local information. When an event is due for processing, the kernel calls a message processing procedure which is defined by the user for the destination LP of the message. If CSS is on, The current state of the LP, which is stored along with the last event processed by this LP, is copied into the state block associated with current event and the pointer to that state block is passed as parameter to the message processing procedure. User code can then access its state variables via this pointer. CSS is optimized in the way that only one state copy operation is needed and no copy-back is necessary on rollbacks. The kernel attaches the updated state to each event, the user code is presented with proper state instead of keeping separate state memory space for itself. With ISS however, the kernel keeps trail of state changes using the same state block and interprets the contents of it by calling a state restoration procedure defined in ISS layer. While the size of state block is fixed to the maximum size of state variables for all LPs in CSS, it can be chosen to optimize the utilization of memory for ISS, overflow of ISS is handled by list of state blocks. On rollback, copied states are automatically restored, no explicit state restoration operation is required, for ISS a procedure is invoked to restore a previous state along the back trail starting from current event to the point of rollback. Two down calls are supported by the kernel. The *Send* is called from the user code whenever a message needs to be sent. With ISS, *SaveState* has to be called from user code to save the old value along with the address of a state variable before change is made to that variable. Figures 1, 2, 3 4 and 5 depict the main data structures of and the user interface to the SMTW Kernel.

5.3 SMTW Kernel 2.2 Performance

Extensive experimental simulation runs on SMTW Kernel 2.2 have been carried out for the three benchmark models. Being presented here are some typical results. For the Queueing Network and Pingpong, two sets of tests were chosen, one with average free event list size, another with short free list. The latter can be used to examine Kernel's performance under memory stress, this will be common for large scale simulation models. Each set of tests

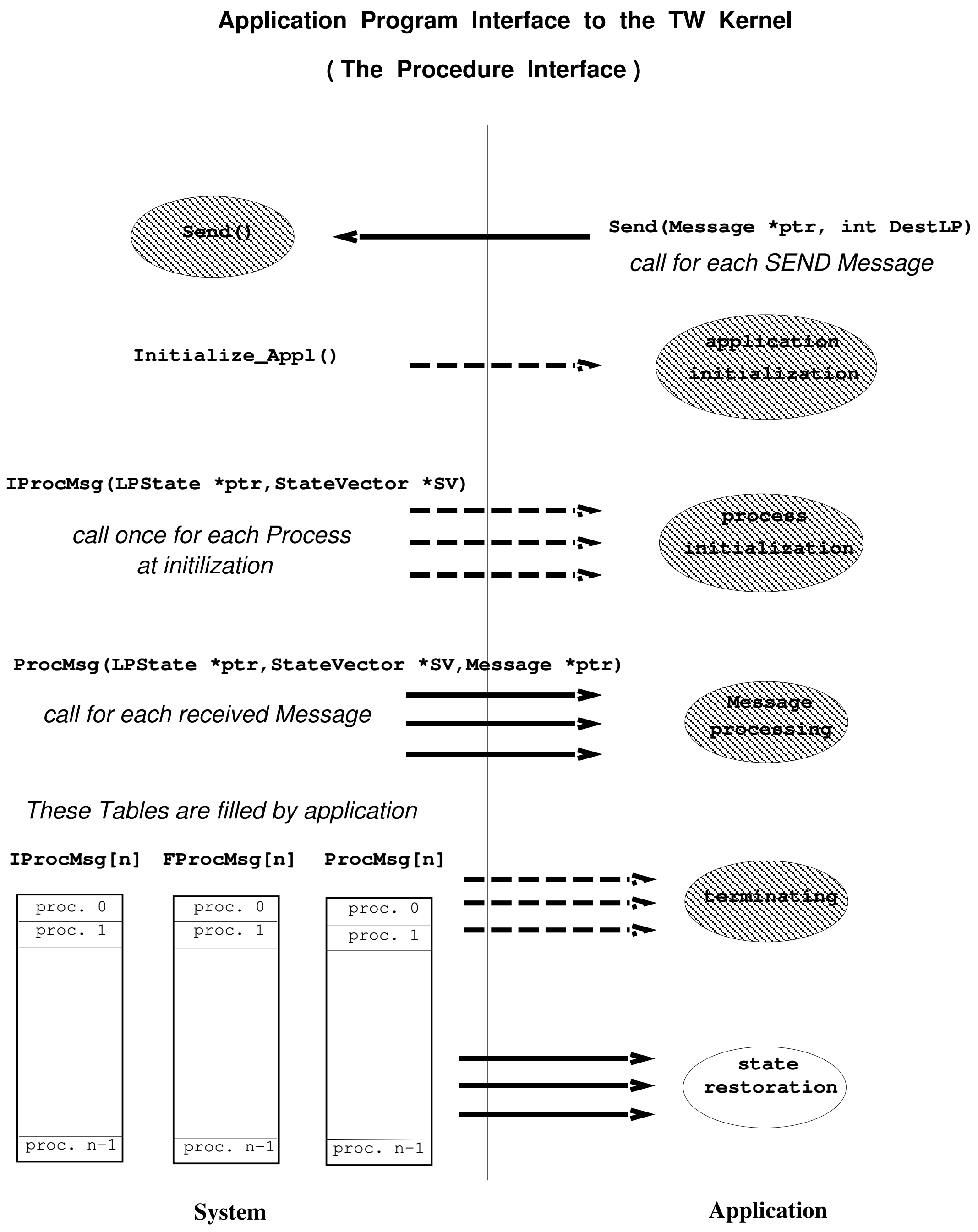


Figure 1: Procedural Interface with SMTW Kernel

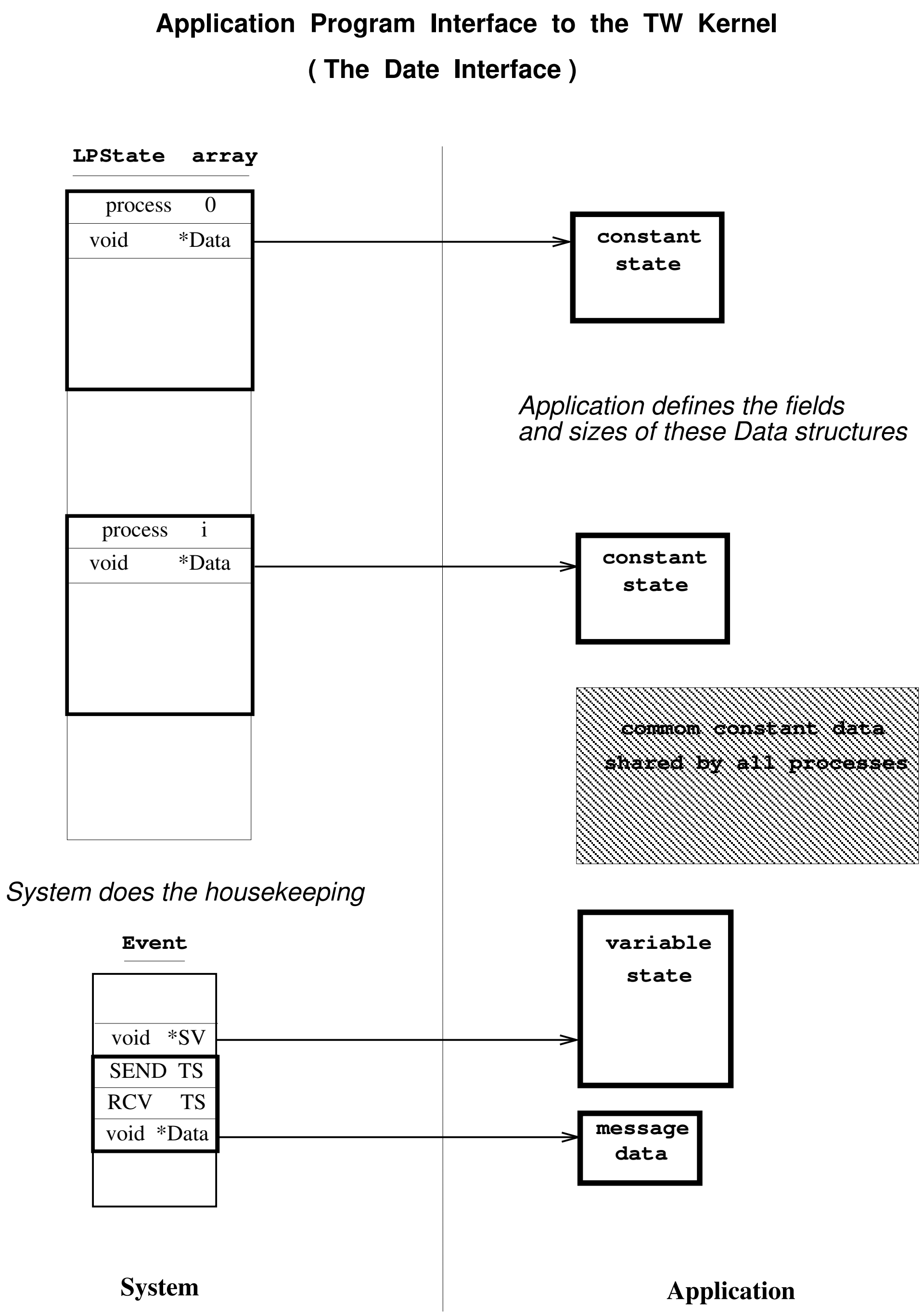


Figure 2: Data Interface with SMTW Kernel

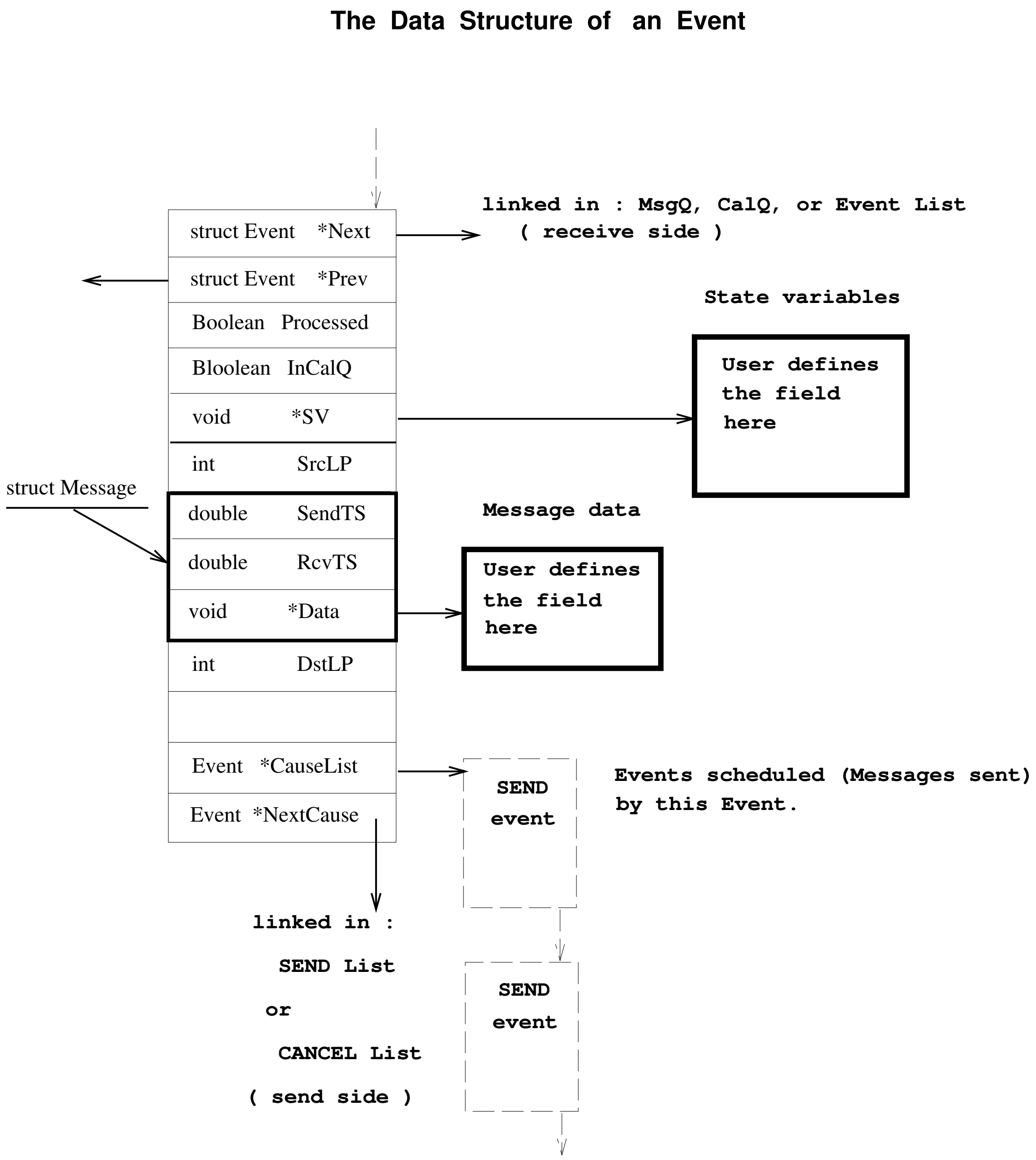


Figure 3: Event Data Structure in SMTW Kernel

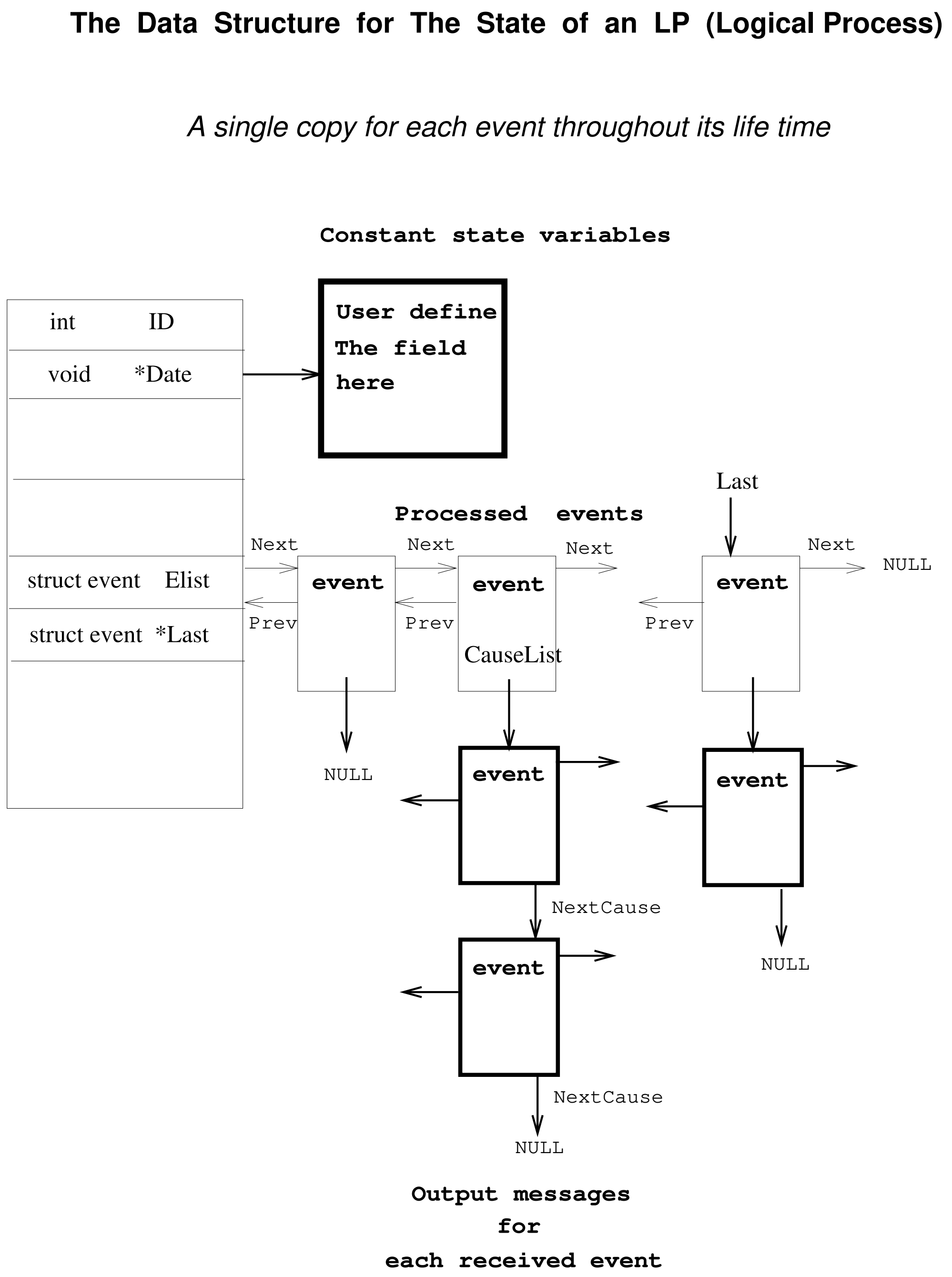


Figure 4: Logical Process Data Structure in SMTW Kernel

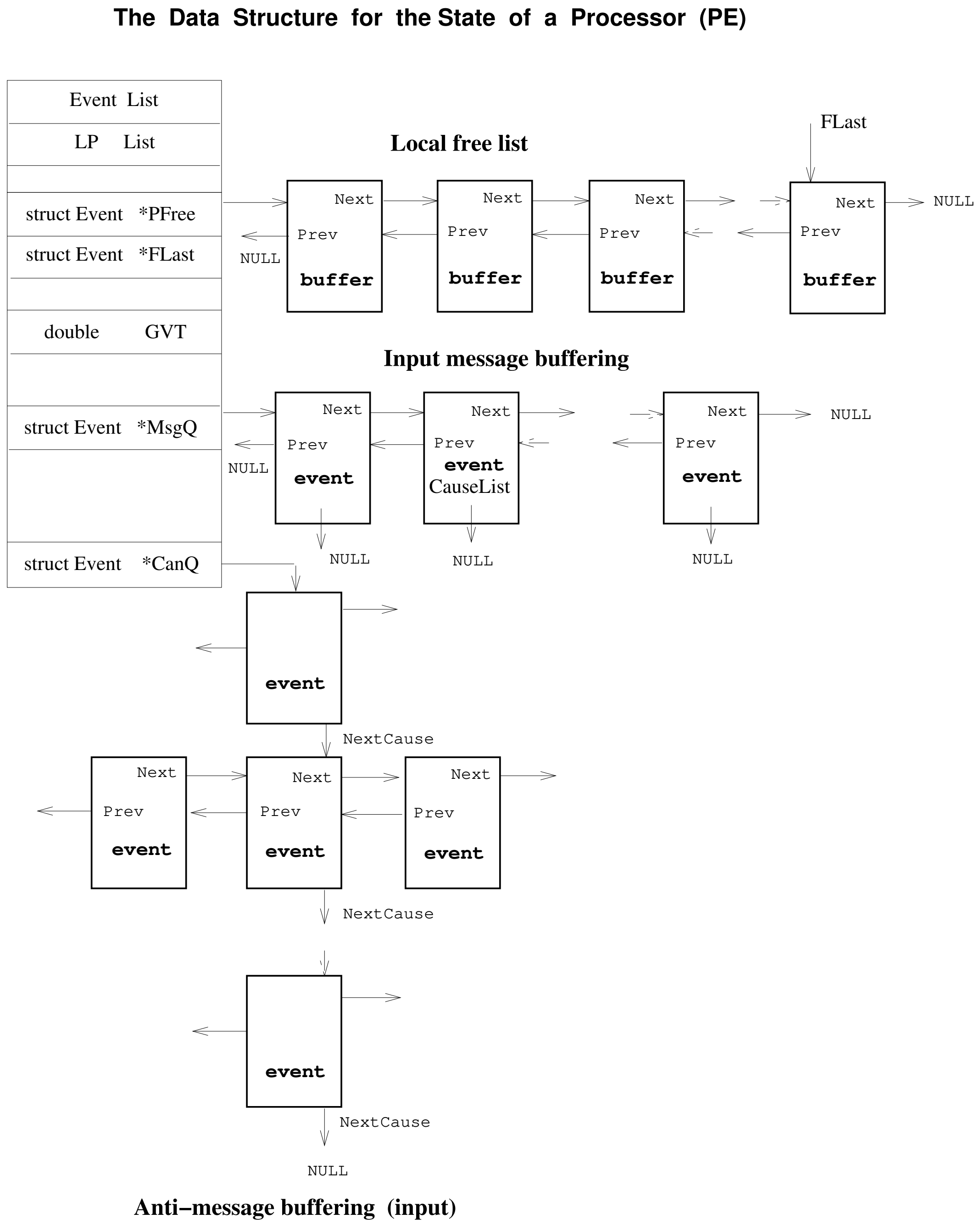
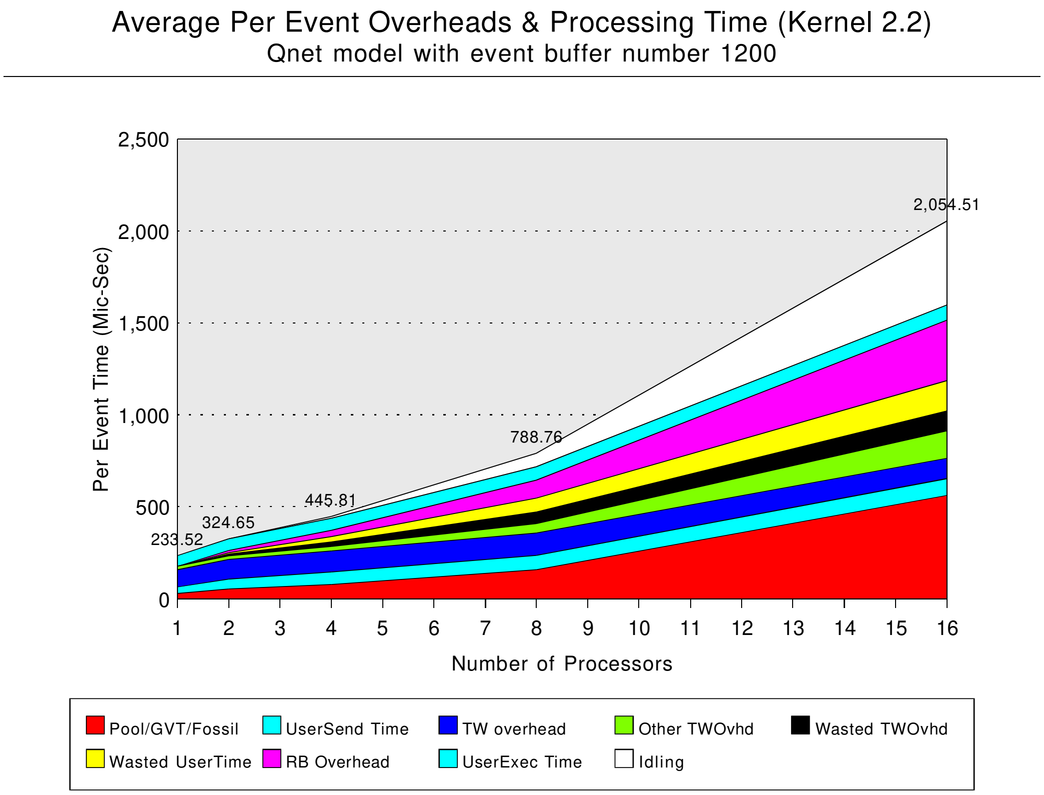


Figure 5: Processor Data Structure in SMTW Kernel

were done on varied number of processors, the largest number of processors was 24 for Pingpong and 16 for Queueing network. Only limited number of runs were done with the SS7-PM model due to the difficulty in obtaining exclusive KSR CPU hours on many processor for such large model and long execution time, but the results are quite consistent. We have collected two sets of data for different application partitioning methods. The state sizes in all these tests are very small except for the SS7-PM, which requires in average two state blocks (256 bytes each) for each event to save state changes (ISS in effect).

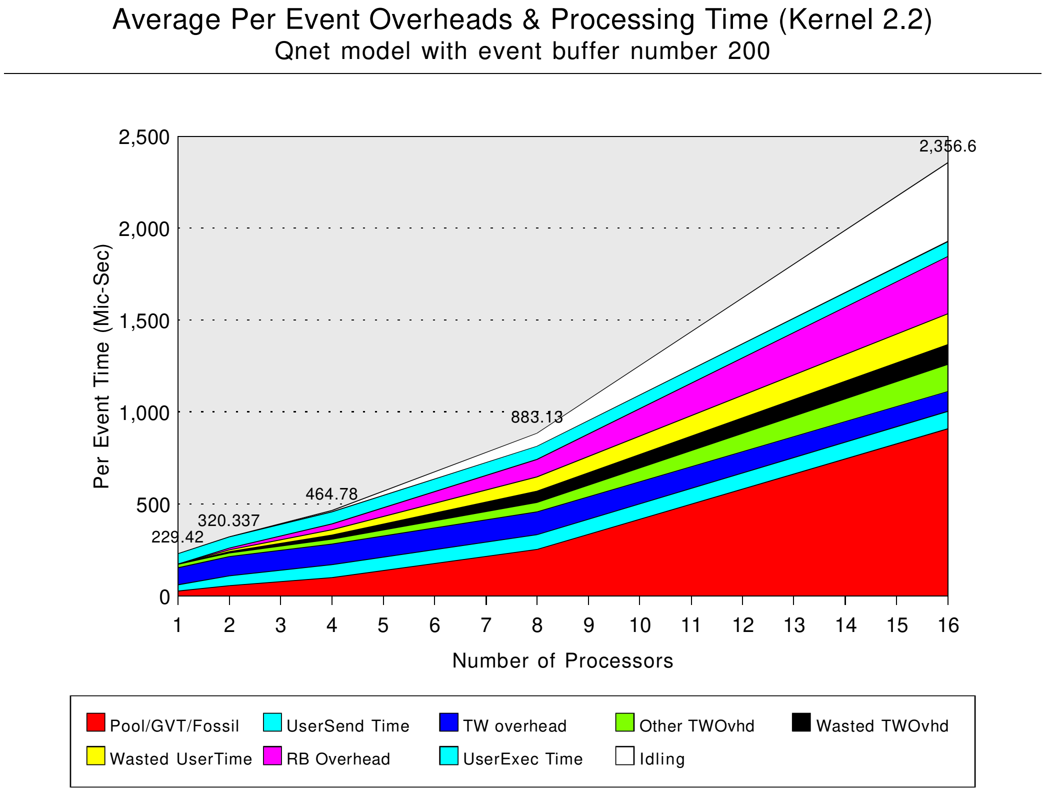
5.3.1 Queueing Network (Qnet)

Figure 6 shows the per event overheads for Queueing Network with four dimensional hypercube. It is typical of models with moderate parallelism (maximum speedup less than 8) and uniform workload. The normal Time-Warp overhead does not change much especially from two to eight processors. The send time increase a bit more with the number of processors, this is caused by contentions around remote input message queues. The so called *other overhead* increases considerably because of rollback incurred cost. User execution time is most stable because in normal case one could expect that locality is well observed in user code. Rollback overheads are obviously depending on rollback rate and will be dealt with in Dynamic load balancing. As other benchmarks will also demonstrate that direct referencing including direct cancellation mechanism, which is only possible with shared memory, gives very good performance results. There is quite an amount of idling time due to limited parallelism. This becomes obvious on sixteen processors while the available parallelism is only eight. The major performance problem here is the global control cost. It increases dramatically, this is an unfavorable property – sensitive to the number of processors. When on sixteen processors, the global control cost reaches 20 time as much as on single processor and becomes the major source of overhead. Figure 7 shows the results of the same benchmark tested with only 200 event buffers instead of 1200 for the results given by Figure 6. The general overhead pattern remains the same, but the global control cost is higher because higher GVT and fossil collection frequency and some invocations of cancelback procedure which is another major source of performance degradation as will be further demonstrated.



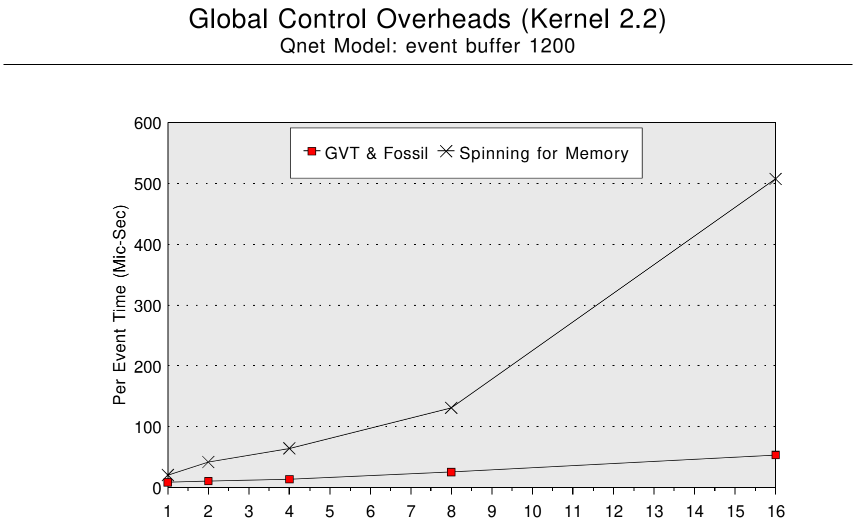
Num. of PEs	Pool GVT Fossil	User Send Time	TW Over-head	Other Over-head	Wasted TW Ovhd	Wasted User Time	RB Over-head	User Exec Time	Idling	Net Event Rate
1	28.71	34.90	94.11	18.37	0	0	0	57.43	0	100%
2	52.08	54.98	105.12	20.71	8.55	9.59	10.47	62.96	0.18	92.58%
4	77.70	67.16	113.75	26.12	24.53	28.58	33.67	65.34	8.96	82.26%
8	156.17	78.43	122.63	49.96	62.81	77.28	96.44	72.46	72.58	66.13%
16	560.51	91.74	110.40	149.67	106.89	168.02	329.07	81.8	456.41	50.81%

Figure 6: Actual Per Event Times (Qnet)



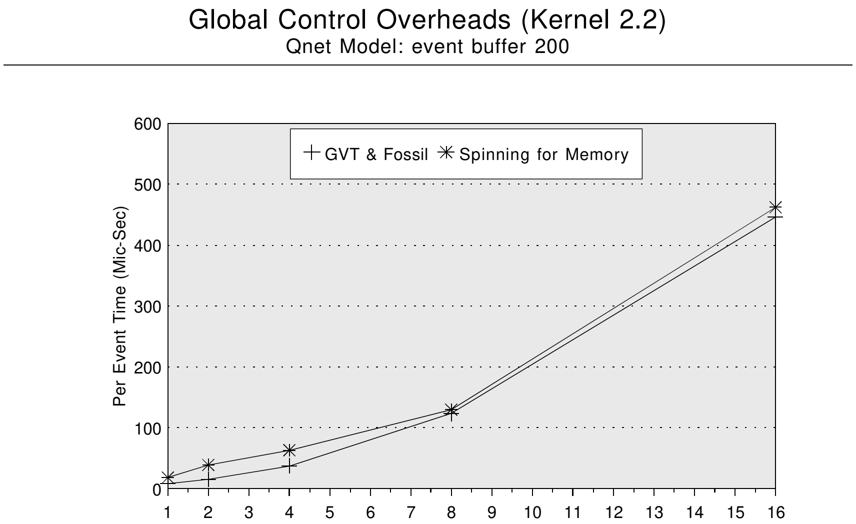
Num. of PEs	Pool GVT Fossil	User Send Time	TW Over-head	Other Over-head	Wasted TW Ovhd	Wasted User Time	RB Over-head	User Exec Time	Idling	Net Event Rate
1	26.35	33.41	93.18	18.65	0	0	0	57.83	0	100%
2	54.29	55.43	102.59	20.62	7.61	8.68	9.45	61.51	0.15	93.09%
4	100.07	69.41	111.87	26.04	23.63	28.25	32.66	64.36	8.49	82.56%
8	252.88	80.73	123.40	50.35	61.80	76.68	94.33	72.58	70.28	66.63%
16	908.15	94.10	111.09	147.23	106.74	168.81	310.12	81.59	428.77	51.0%

Figure 7: Actual Per Event Times (Qnet)



number of PEs	1	2	4	8	16
GVT & Fossil	8.43	10.34	13.45	25.3	53.37
Spinning for Memory	20.37	41.74	64.25	130.87	507.15

Number of processors



number of PEs	1	2	4	8	16
GVT & Fossil	8.25	15.39	37.05	123.22	446.08
Spinning for Memory	18.1	38.9	63.02	129.66	462.07

Number of processors

Figure 8: Global Control Overheads (Qnet)

Take a closer look at the global control cost, it becomes clear that the two main sources of performance degradation are the high cost associated with central free list access and the cost of cancelback. Figure 8 presents the global cost as two parts - GVT computation and fossil collection, spinning for memory. The cost of acquiring a buffer includes the cost of lock, the wait time caused by contention , it is sensitive to the number of processors and the rate of scheduling events (sending messages). With average free list length, the dominant cost is spinning time, GVT and fossil collection cost is relatively low because no cancelbacks were involved. The size of free list has little effect on spinning time, but drastic increase of GVT cost is reported for short free list and the two costs becomes comparable. This is the result of cancelback invocation.

5.3.2 Parallel Pingpong

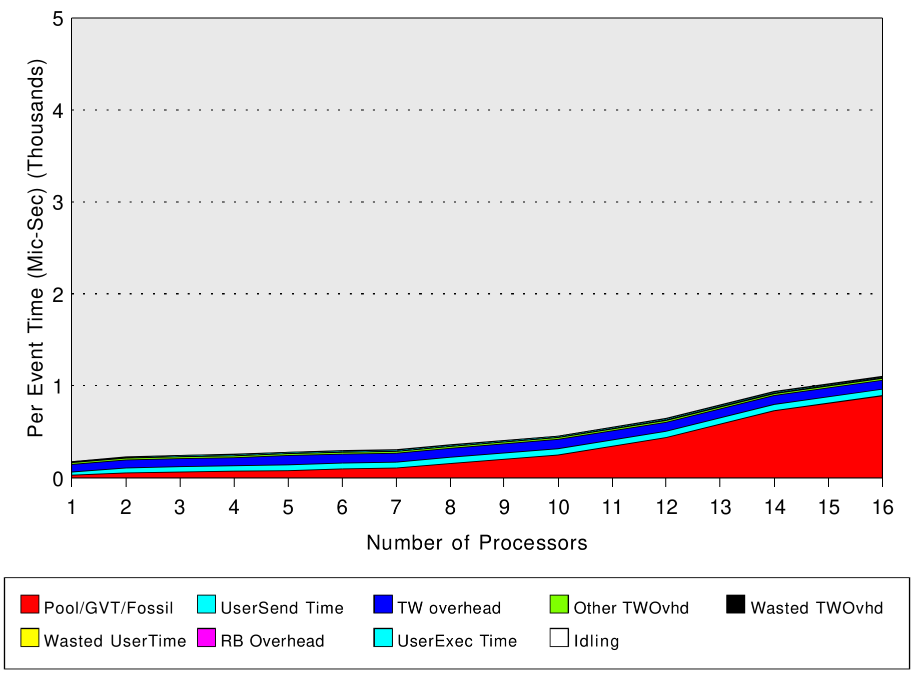
The Pingpong benchmark isolates the possible influence from the user code and provides a good test case for kernel's performance. Figure 9 shows the results of Pingpong without artificial computation on user part and with average free list length 1000. Here rollbacks and their cost are negligible. All the overheads incurred by normal event processing are very stable, increase slightly as more processors are added except for the send time which is just twice as much on 24 processors as on one processor. The global control cost climbs very fast and reaches almost 100 times on 24 processors. With short free list (Figure 10), global cost becomes so high on 24 processors that all the other overheads put together account for just less 2% of global cost. This is caused by frequent invocation of GVT computation, fossil collection and possibly cancelback procedure. Erroneous computation spreads very fast with such a small granularity of event processing that the user code merely read the time stamp and sender of the received message, then send a new message accordingly. While the GVT is still far in the past, some processors may quickly find out that no free buffers are available and have to stop all the processors in order to compute GVT, then easily discover that no buffers can be reclaimed either so cancelback procedure has to be called for rescue. The more processors in the simulation and the smaller the free pool, the more frequently the GVT computation and cancelback will be invoked and the much higher the penalty will be. Figure 11 shows the composition of global control cost, With average free list size, the main cost is on *Spinning for memory*, when the free list gets short, the *GVT and fossil* time becomes dominant.

Figure 12 gives the speedup curves for the two sets of pingpong tests. The real and expected speedup are almost identical, their distance to ideal speedup is such a far cry that the maximum speedup merely reaches 4 for average free list length and just a little higher than 3 for short free list, The curve actually declines after 7 processors – added processing power can not even catch the high shooting cost.

5.3.3 SS7-PM Model

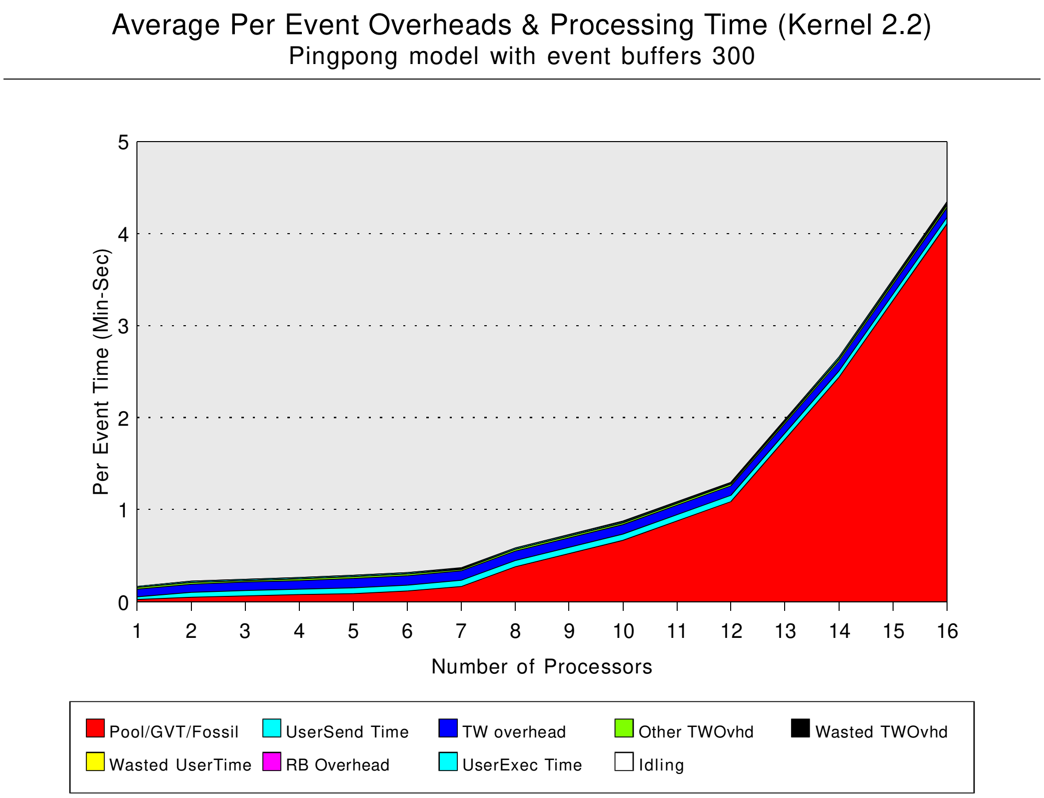
The SS7 performance model is a model of SS7 signaling network simulation model. Experimenting with SS7-PM puts the SMTW Kernel in a more realistic situation. The model has plenty of available parallelism (about 60) and a large message population changing dynamicly (starting with more than 8000), all this makes the rollbacks spread very fast. Large free buffer list had to be used to avoid the invocation of cancelback and the cost associated with it, so that less CPU hours were needed to finish a simulation run. As The overheads charts (Figure **??** and **??**) suggest that the overall overhead pattern is similar to Qnet's. The *TW overhead* and *User Exec. Time* actually decreases with the number of processors because the cache and virtual memory effects (on single processor the model requires more than 1 Gbytes memory space). For the same reason, the absolute value of Time Warp overhead is twice as much as for the Qnet model. The *RB overhead* is higher relative to the *Wasted TW overhead* and *Wasted User Time* suggesting larger amount of events were canceled before being processed. The proportion of global cost is also much higher than that of Qnet. As a result, the maximum speedup with new partitioning method is only 2.64 and 2.92 for real and expected respectively, 1.52 and 2.16 with old partitioning method. The cost of *Spinning for Memory* accounts for most of the global control cost (Figure 15).

Average Per Event Overheads & Processing Time (Kernel 2.2)
Pingpong model with event buffers 1000



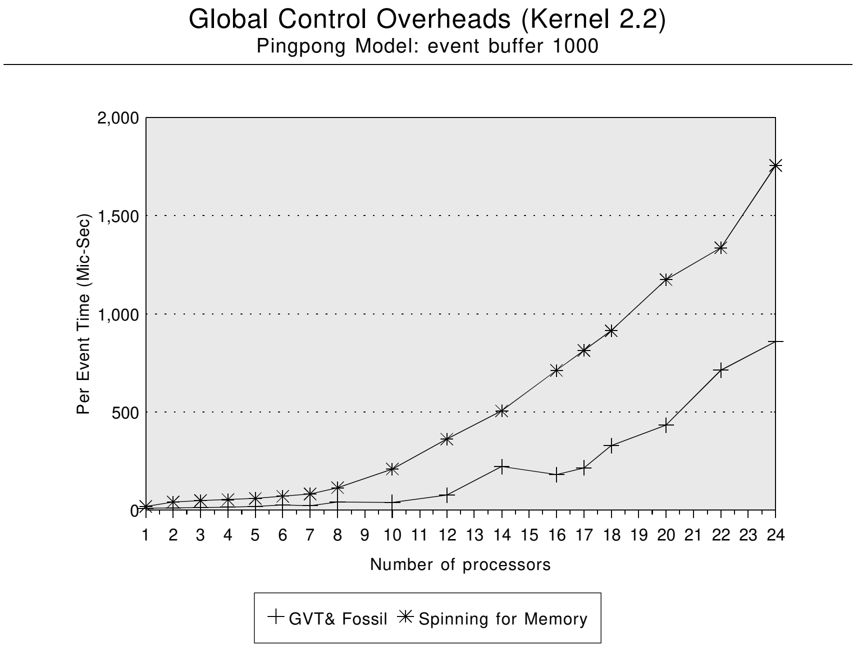
Num. of PEs	Pool GVT Fossil	User Send Time	TW Over-head	Other Over-head	Wasted TW Ovhd	Wasted User Time	RB Over-head	User Exec Time	Idling	Net Event Rate
1	27.86	34.20	82.04	17.60	0	0	0	13.29	0	100%
2	53.02	51.38	87.64	18.44	0	0	0	13.87	0.72	100%
3	61.38	56.41	89.81	19.18	0	0	0.01	14.17	2.52	100%
4	69.17	59.69	90.75	19.06	0.01	0.01	0.03	14.16	2.17	99.99%
5	78.33	61.98	100.01	19.05	0.02	0.01	0.03	14.17	2.54	99.99%
6	95.66	63.94	98.83	19.28	0.02	0.02	0.04	14.19	3.71	99.98%
7	104.76	65.05	98.47	18.79	0.02	0.02	0.04	14.14	2.47	99.98%
8	154.48	66.26	98.64	19.38	0.08	0.06	0.10	14.17	4.50	99.92%
10	248.40	67.08	99.56	19.05	0.10	0.08	0.18	14.19	3.36	99.9%
12	437.25	67.80	98.69	19.76	0.11	0.09	0.22	14.36	5.67	99.89%
14	726.47	68.51	99.56	20.73	0.16	0.14	0.48	14.64	7.27	99.84%
16	891.67	68.95	99.54	20.65	0.03	0.03	0.17	14.76	7.32	99.97%
17	1027.99	68.99	98.96	20.90	0.15	0.13	0.49	14.86	7.79	99.85%
18	1242.09	69.23	100.16	21.58	0.09	0.08	0.22	15.04	8.84	99.91%
20	1606.79	69.75	102.02	22.12	0.21	0.17	1.00	15.37	9.80	99.8%
22	2048.28	70.52	101.79	24.11	0.18	0.16	0.53	15.87	15.1	99.82%
24	2612.97	71.09	102.18	25.97	0.30	0.25	2.42	16.35	20.30	99.71%

Figure 9: Actual Per Event Times (Parallel Pingpong)



Num. of PEs	Pool GVT Fossil	User Send Time	TW Over-head	Other Over-head	Wasted TW Ovhd	Wasted User Time	RB Over-head	User Exec Time	Idling	Net Event Rate
1	21.75	31.71	82.09	17.56	0	0	0	13.33	0	100%
2	48.48	51.07	88.40	18.25	0.01	0	0.01	14.04	0.49	99.99%
3	62.20	58.00	90.16	18.45	0	0	0	14.15	0.61	100%
4	75.13	62.01	91.44	18.45	0.03	0.02	0.03	14.25	0.30	99.97%
5	85.70	64.46	101.04	18.46	0.04	0.03	0.05	14.30	0.52	99.96%
6	113.88	66.27	99.58	18.59	0.02	0.01	0.02	14.29	1.05	99.98%
7	165.23	67.61	99.71	18.63	0.05	0.04	0.06	14.30	1.11	99.95%
8	378.25	68.24	101.30	19.65	0.08	0.07	0.04	14.65	1.54	99.92%
10	663.99	68.53	102.63	19.81	0.31	0.25	0.20	14.85	1.60	99.7%
12	1085.43	68.88	101.60	20.52	0.43	0.36	0.27	15.19	2.07	99.57%
14	2439.61	68.19	103.39	23.69	4.37	3.57	0.39	16.43	3.10	95.95%
16	4107.27	68.78	104.36	26.38	8.48	7.04	0.24	17.84	2.95	92.49%
17	4943.23	69.16	104.47	27.83	10.77	9.01	0.36	18.23	4.05	90.66%
18	5972.74	69.47	105.48	29.47	12.66	10.62	0.32	18.99	4.39	89.28%
19	8634.30	69.94	106.87	34.14	19.65	16.61	0.61	20.38	4.77	84.47%
20	10194.31	70.68	108.64	36.26	22.18	18.72	0.38	21.0	6.33	83.05%
22	13702.15	72.06	108.49	39.63	27.08	23.47	0.40	21.98	5.38	80.03%
24	20345.88	73.44	111.04	45.65	35.82	31.42	1.53	23.96	6.39	75.61%

Figure 10: Actual Per Event Times (Parallel Pingpong)



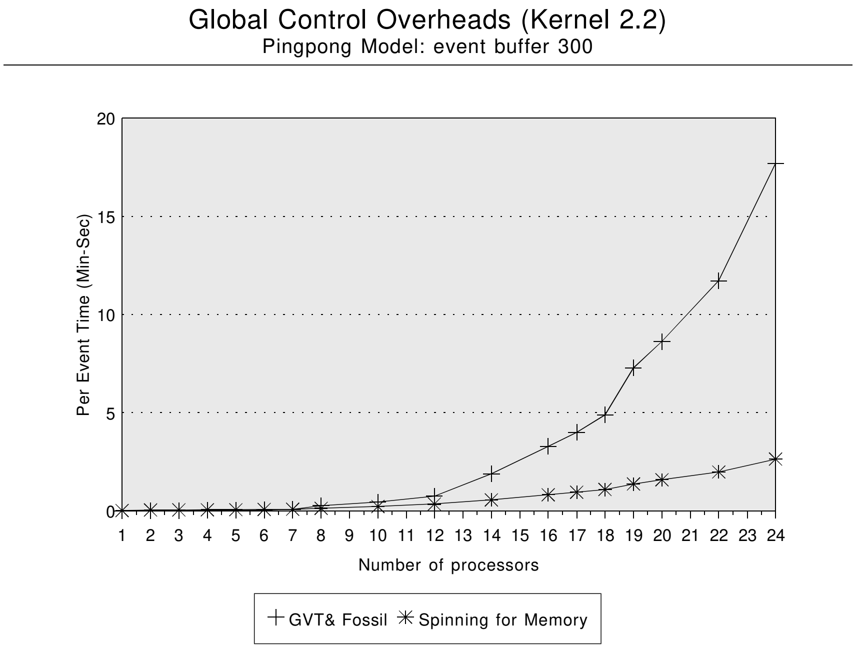
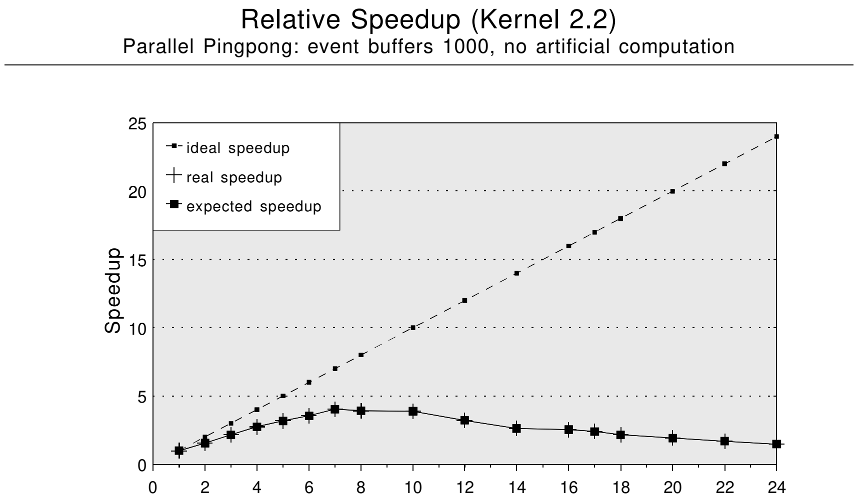
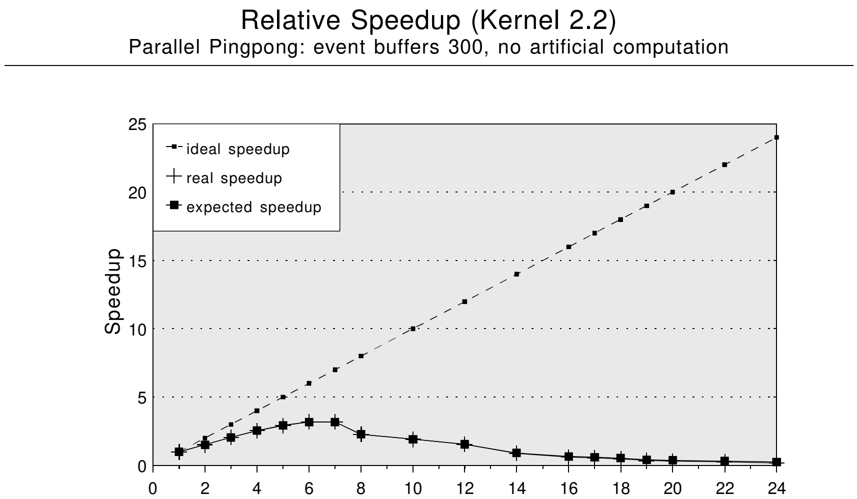


Figure 11: Global Control Overheads (Parallel Pingpong)



number of PEs	1	2	3	4	5	6	7	8	10	12	14	16	17	18	20	22	24
ideal speedup	1	2	3	4	5	6	7	8	10	12	14	16	17	18	20	22	24
real speedup	1	1.55	2.16	2.74	3.17	3.55	4.03	3.91	3.87	3.21	2.61	2.54	2.4	2.16	1.92	1.69	1.47
expected speedup	1	1.55	2.16	2.74	3.17	3.55	4.03	3.92	3.88	3.22	2.62	2.54	2.4	2.16	1.92	1.69	1.48

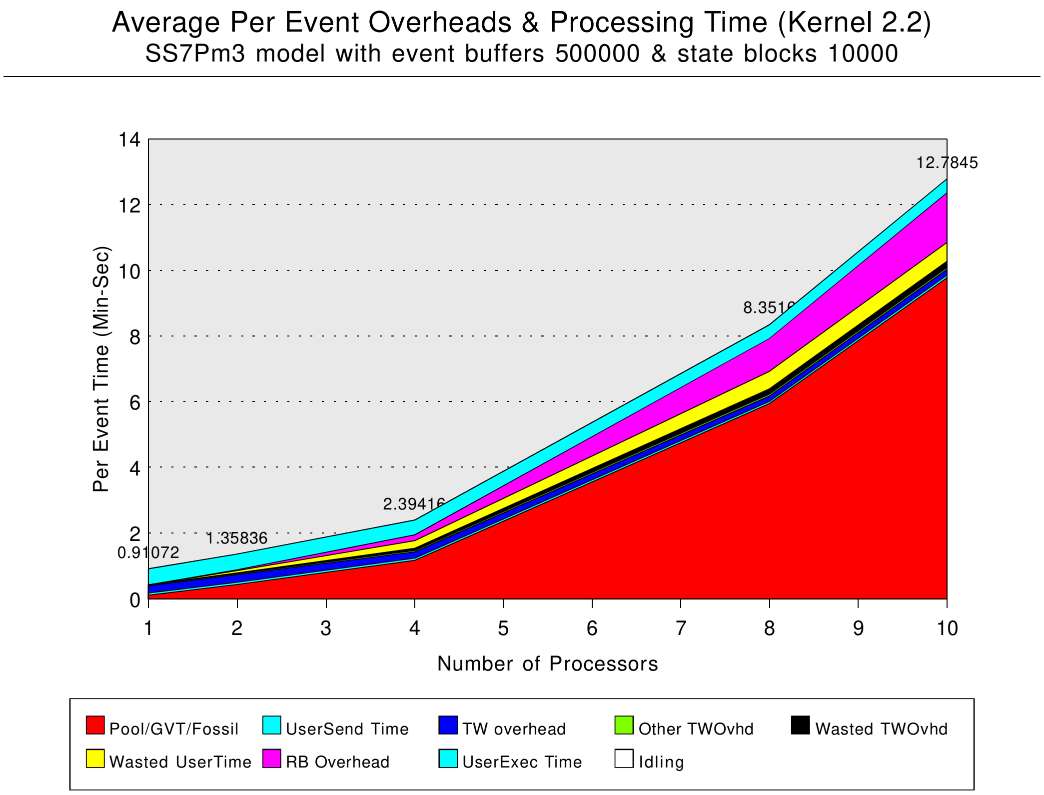
Number of Processors



number of PEs	1	2	3	4	5	6	7	8	10	12	14	16	17	18	19	20	22	24
ideal speedup	1	2	3	4	5	6	7	8	10	12	14	16	17	18	19	20	22	24
real speedup	1	1.51	2.05	2.54	2.92	3.18	3.18	2.28	1.91	1.54	0.88	0.61	0.55	0.48	0.36	0.32	0.26	0.19
expected speedup	1	1.51	2.05	2.55	2.93	3.18	3.18	2.28	1.91	1.55	0.91	0.66	0.6	0.54	0.42	0.38	0.33	0.26

Number of Processors

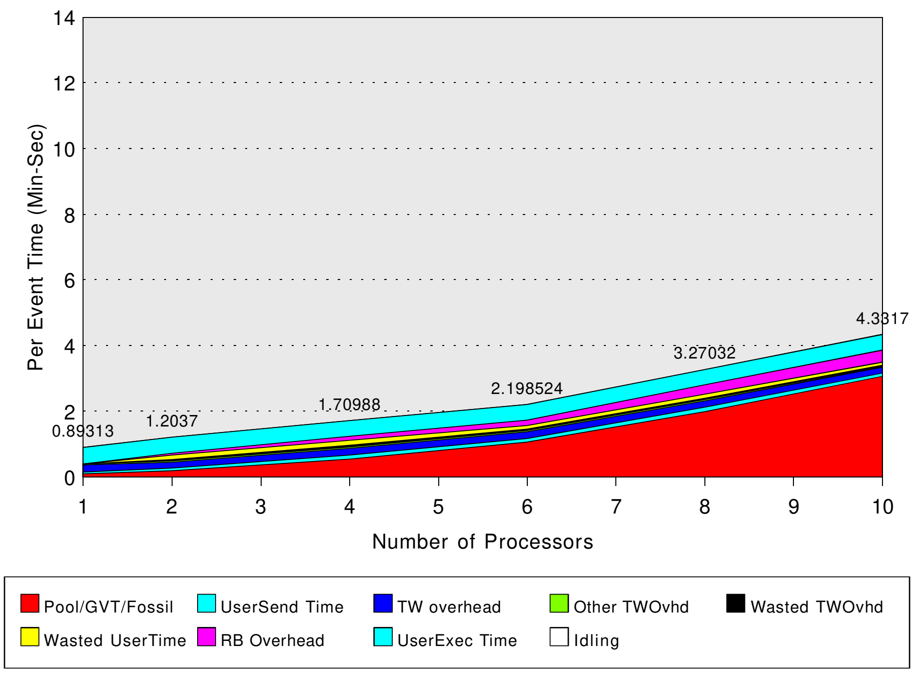
Figure 12: Relative and Expected Speedup (Parallel Pingpong)



Num. of PEs	Pool GVT Fossil	User Send Time	TW Over-head	Other Over-head	Wasted TW Ovhd	Wasted User Time	RB Over-head	User Exec Time	Idling	Net Event Rate
1	111.14	58.25	221.27	24.59	0	0	0	495.47	0	100%
2	432.50	63.80	229.96	28.96	29.52	69.38	26.90	476.52	0.82	88.62%
4	1166.64	71.12	188.36	36.40	78.52	219.40	181.88	449.84	2.0	70.37%
8	5929.04	77.28	166.56	54.0	175.4	521.44	1005.84	417.6	4.4	48.7%
10	9778.5	77.5	170.7	62.7	193.8	566.7	1508.0	421.7	4.9	46.83%

Figure 13: Actual Per Event Times (SS7PM3)

Average Per Event Overheads & Processing Time (Kernel 2.2)
SS7Pm3 model with event buffers 100000 & state blocks 50000

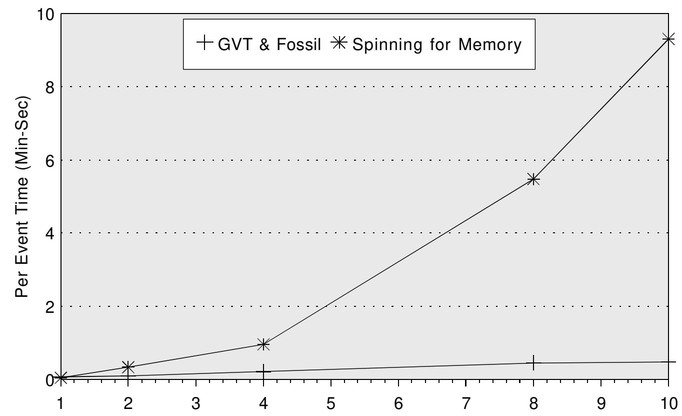


Num. of PEs	Pool GVT Fossil	User Send Time	TW Over-head	Other Over-head	Wasted TW Ovhd	Wasted User Time	RB Over-head	User Exec Time	Idling	Net Event Rate
1	86.24	58.89	217.26	23.10	0	0	0	507.64	0	100%
2	186.3	76.04	188.16	29.06	47.1	139.88	54.18	482.84	0.14	79.98%
4	543.88	112.76	216.04	31.76	57.36	157.2	111.36	479.4	0.12	79.02%
6	1063.68	107.76	204.18	31.98	40.2	115.32	156.6	478.44	0.36	83.56%
8	1989.84	126.72	199.28	32	41.52	124.48	285.52	470.72	0.24	82.76%
10	3061.7	91.4	186.7	30.1	29.7	89.8	369.0	472.9	0.4	86.27%

Figure 14: Actual Per Event Times (SS7PM3)

Global Control Overheads (Kernel 2.2)

SS7PM3 model: old partitioning

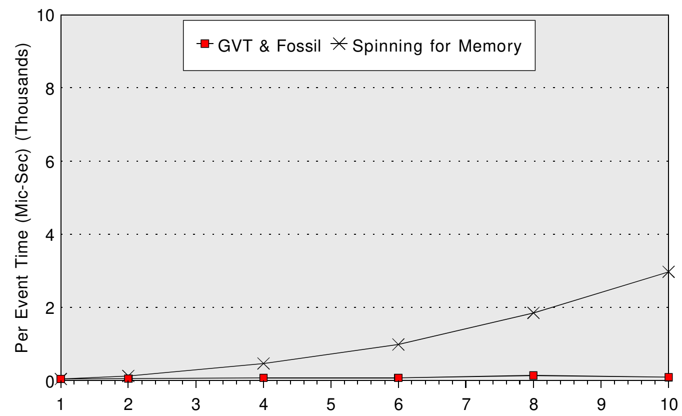


number of PEs	1	2	4	6	8	10
GVT & Fossil	0.06898	0.09968	0.21041		0.44649	0.47226
Spinning for Memory	0.04216	0.33282	0.95624		5.48252	9.30626

Number of processors

Global Control Overheads (Kernel 2.2)

SS7PM3 model: new partitioning



number of PEs	1	2	4	6	8	10
GVT & Fossil	0.04437	0.05731	0.07655	0.07691	0.13978	0.09171
Spinning for Memory	0.04187	0.12899	0.46731	0.98679	1.85004	2.96998

Number of processors

Figure 15: Global Contral Overheads (SS7PM3)

5.4 Critical Path Execution Time

It is very informative to know what is the inherent parallelism a particular application model may provide. Based on this, reasonable expectation can be drawn and efforts can be made to improve the available parallelism.

First we need to find out what it the least time needed to execute a simulation assuming zero overhead and enough processors to allow at any time the execution of any process ready to execute, given causal relations among logical processes. We can also compute the sequential user execution time as the sum of user execution time for all committed events. The ratio of sequential user execution time over the least time is a measure of the parallelism the application can provide. In the following paragraph we give the definition of *Critical Path user Execution Time.*

The least time mentioned above is the user execution time along the critical path. Starting from the initialization phase of each logical process, follow the execution of each event, at each point where an event is scheduled by sending a message the path branches - one continues along with the current process, the other goes to the destination process of the message, finally the path reaches its end at some logical process when the simulation terminates. There can be many such paths and many of them shared a common segment of their path. These paths may be called *Causality Paths.* Among the many causality paths, there is one path that takes longest time to finish execution, we call this causality path the *Critical Path.* Be aware that along each causality path, the time it takes to finish execution also includes those time spent on waiting in addition to the time spent on execution events. If a message arrives at its destination and finds that the logical process is busy processing a previous event then it has to wait until the process finishes processing the event immediately preceding it, or if a logical process has finished processing current event and finds out that the next message hasn't arrived yet then the logical process has to wait until the next due message arrives. Also notice that the time spent on erroneous processing and rollback is not counted. A logical process will eventually process all the events scheduled for it in the correct order.

The algorithm for computing critical path user execution time, which excludes any overheads, can be easily designed following the above description and definition. We will present the algorithm in a separate paper.

5.5 Experimental results with ISS

The following are the results of experiments with ISS conducted with the parallel Pingpong program.

Two sets of tests were carried out:

1. Medium state size of 95 integers per LP,
   per event state change of 20%, 10%, and 2%,
   constant State-Block size of 256 bytes for ISS.

2. Large state size of 500 integers per LP,
   per event state change of 25%, 20%, 10%, 5%, and 2%,
   State-Block size of 2048 bytes, 2048 bytes, 1024 bytes, 512 bytes, and 256 bytes respectively.

State copy is made for each processed event in CSS (Copy State Saving), while only state changes for each event is saved in ISS. The per event copy time in CSS is actually timed, the per event ISS saving time can not be timed directly, it is instead calculated by subtract per event user processing time in CSS context from that in ISS context (in ISS context user-defined event-processing procedure spends extra time on calling SaveInt, a system-defined macro for saving a changed integer). The discrepancy of reported timing data between running on single processor and running on multi-processors is neglectable. The following data were taken on single processor runs, each number, in micro-seconds per event average, is the average of three similar runs,

1. 95 integers per Logical Process:

St. change Rate	20%	10%	2%
CSS	57.52 (90.85)	58.04 (91.43)	58.31 (75.63)
ISS	53.33 (89.95)	23.79 (90.16)	6.24 (77.06)

2. 500 integers per Logical Process:

St. change Rate	25%	20%	10%	5%	2%
CSS	346.12 (87.41)	349.70 (79.86)	361.25 (82.09)	358.76 (85.63)	337.48 (89.26)
ISS	393.98 (87.83)	313.63 (83.21)	143.57 (83.53)	67.00 (89.17)	25.71 (87.48)

The numbers in parenthesis are average per event normal Time Warp overhead. They vary depending on the condition of the processors on which they are running.

As expected, with CSS the time spent on copying state does not really change while with ISS the time spent by user procedure on saving changed state is proportional to the percentage of state changes per event. Somewhere between 20% and 25% state changes per event time needed to save changed state breaks even with the time needed for copy state saving. In the first set of 95 integer state per LP, the State Block size is constant (256 bytes) and this adds some cost for handling State Block overflow. The second set of data is fairer as for both CSS and ISS only a single buffer is used for saving state whatever the way.

The memory space requirement for ISS and CSS is four to one. To save one integer 16 bytes are necessary for ISS compared to only 4 byte for CSS. For the KSR platform, besides 8 bytes is required to save the address of an integer state, an extra 4 byte is wasted for the 8 byte address value must be stored at 8 byte boundary (otherwise resulting bus error). Based on this, the memory space requirement for ISS breaks even with CSS at 25% state changes per event.

With this particular implementation of ISS, there is no point to use ISS if the average state changes per event is 25% or higher. Significant savings both on space and time can be achieved by ISS if average per event state changes are less than 10%. The effect of CSS and ISS on other aspects of the Time Warp behavior may also be different. Experiments with more realistic models are necessary.

6 SMTW Kernel 2.3 and Performance

6.1 SMTW Kernel 2.3

Kernel 2.2 has been very successful in reducing the overheads incurred in both forward and rollback processing of events by direct referencing (including direct cancellation) as discussed previously. Two major problems have been identified in terms of performance. One is the very high cost associated with the central free list scheme, in particular, the sharp increase of cost with the number of processors does not fit in well with the parallel simulation. Second problem is the high cost of cancelback implementation. Whenever the situation arises that cancelback has to be invoked to control the memory consumption, a dramatic increase in global control overhead is observed. These problems are stemmed from the synchronized approach to global control mechanism, the GVT computation, the fossil collection and the cancelback are all carried out in a totally synchronized manner, that is in effect to stop productive simulation execution. The original purpose of Kernel 2.2 was to experiment with cancelback protocol and probably some other mechanisms. To eliminate the overhead bottleneck caused by central free list, a distributed free lists scheme was included in the original source.

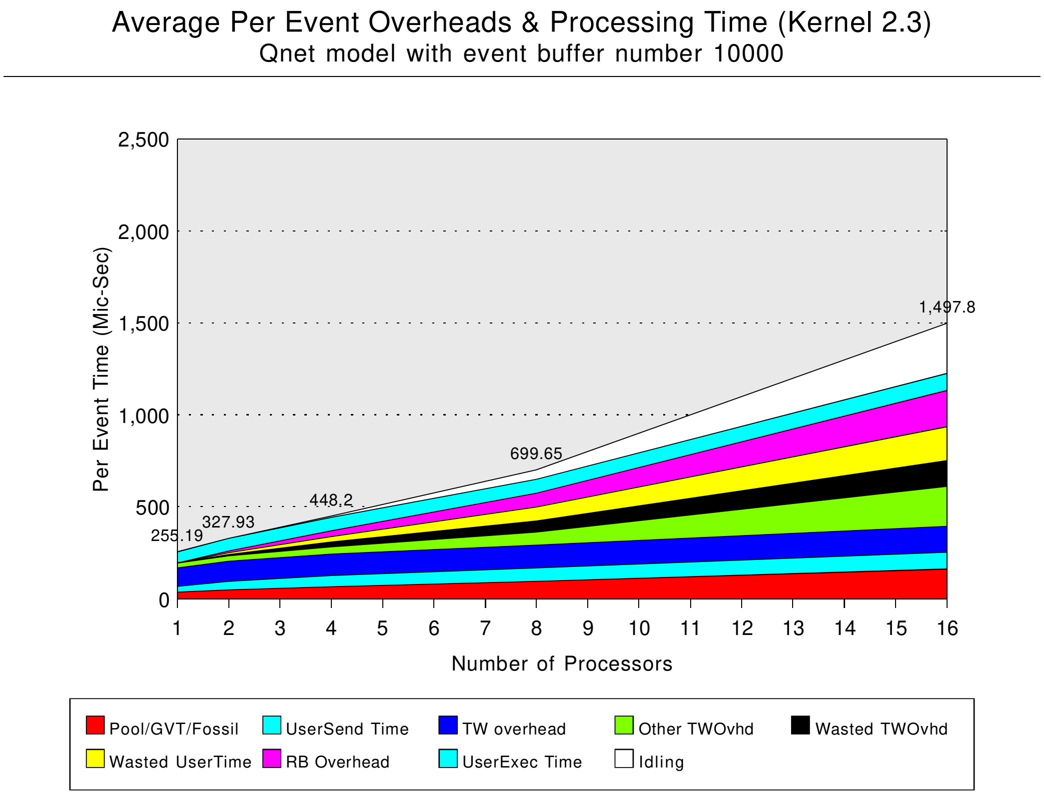
The SMTW Kernel 2.3 is an experimental version using the distributed free lists arrangement. Each processor has two separate free lists, one for event buffers, the other for state blocks. Whenever a message is to be sent out, an event buffer is obtained from the free list on the destination site of the message instead of the local free list. In this way, the buffers remain on the same processor because the event buffer will be reclaimed by fossil collection on the receive site so that the total number of buffers is fixed and there is no concern of either being drained out or flooded over. The global control scheme remains synchronized although the cancelback mechanism which requires a single central free list does not match with the distributed scheme. Next subsection explains the test results.

6.2 SMTW Kernel 2.3 Performance

Figure 16 shows the actual per event average overheads of four dimensional hypercube Qnet model with event buffer number 10000. The increase of global control overhead is much slower compared to Figure 6 for the kernel 2.2. According to Figure 17, the cost of *Spinning for Memory* is about 5.6 times as much on 16 processors as that on single processor, compared to 25 times for Kernel 2.2.

For the SS7-PM3 model, the effect of decentralizing free lists are more dramatic that the proportion of global control overhead becomes less important event on 12 processors compared to others such as rollback caused overheads. The pattern of per event overheads has changes considerably (Figure 18 and 19). Figure 20 gives the composition of global overhead.

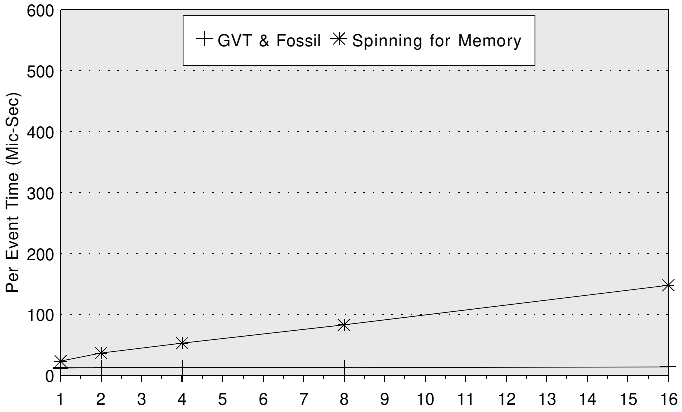
Figure 21 is results of a set of tests for the parallel pingpong without artificial computation inserted in user code (very fine granularity). When the number of processors is small, the global control cost is kept low due to reduced contention on free list. There is a sudden increase in global cost, this happens with the invocation of cancelback. With distributed free lists, each processor becomes less restricted in erroneous processing because the synchronizing effect of a single central free list is relieved. The model contains self-directed messages which makes it very sensitive to the machine condition under which the simulation is running and those processors that are far ahead in virtual time constantly runs out of free lists and engages the entire system in GVT computation. From Figure **??**, it can be seen that spinning for buffer is no longer a big cost problem, but the GVT and memory management cost jumps very high. Long average rollback distance was reported ranging from 100 events to more than 200 events on 10 to 17 processors. The global control overhead is still too high to be useful as a effective control mechanism.



Num. PEs	Pool GVT Fossil	User Send Time	TW Over-head	Other Over-head	Wasted TW Ovhd	Wasted User Time	RB Over-head	User Exec Time	Idling	Net Event Rate
1	34.95	31.16	101.63	26.17	0	0	0	61.28	0	100%
2	48.66	46.18	107.61	29.47	8.96	9.45	10.11	67.33	0.16	92.31%
4	64.68	60.37	117.79	38.11	26.4	29.58	32.35	71.58	7.34	81.69%
8	94.92	72.23	123.09	70.27	62.22	73.9	76.8	73.98	52.24	66.43%
16	160.93	91.08	140.65	217.49	140.79	183.79	198.49	92.53	272.05	49.98%

Figure 16: Actual Per Evet Times (Qnet)

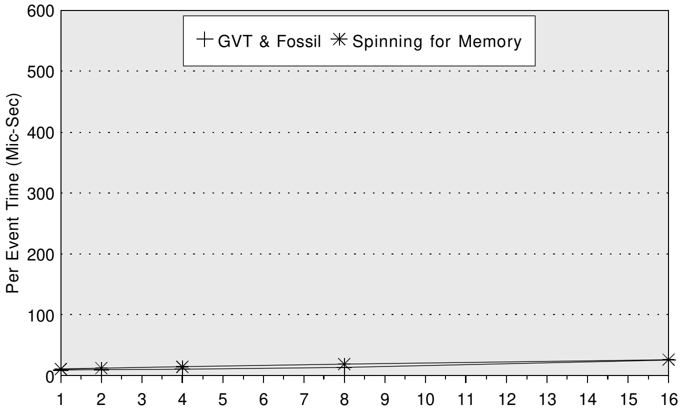
Global Control Overheads (Kernel 2.3)
Qnet Model: event buffer 10000



number of PEs	1	2	4	8	16
GVT & Fossil	12.05	12.39	12.19	12.53	13.38
Spinning for Memory	22.89	36.26	52.48	82.38	147.55

Number of processors

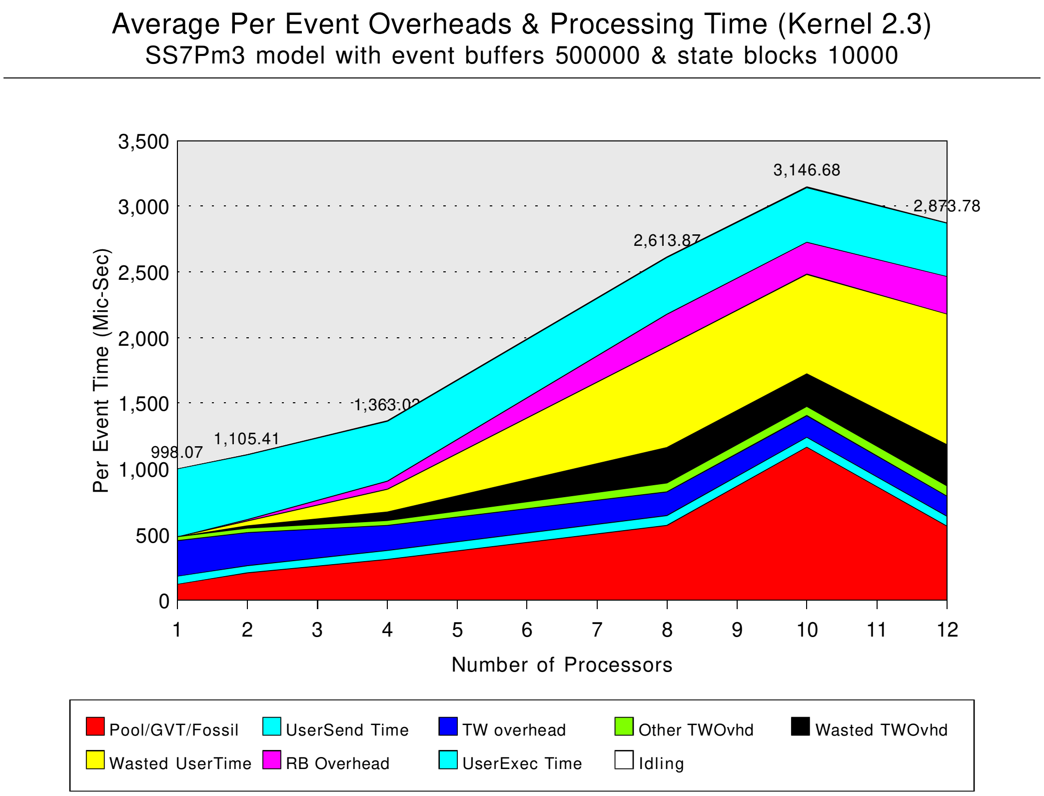
Global Control Overheads (Kernel 3.1)
Qnet Model: event buffer 2000



number of PEs	1	2	4	8	16
GVT & Fossil	9.03	9.64	10.48	13.64	25.41
Spinning for Memory	10.78	12.3	14.41	18.89	26.37

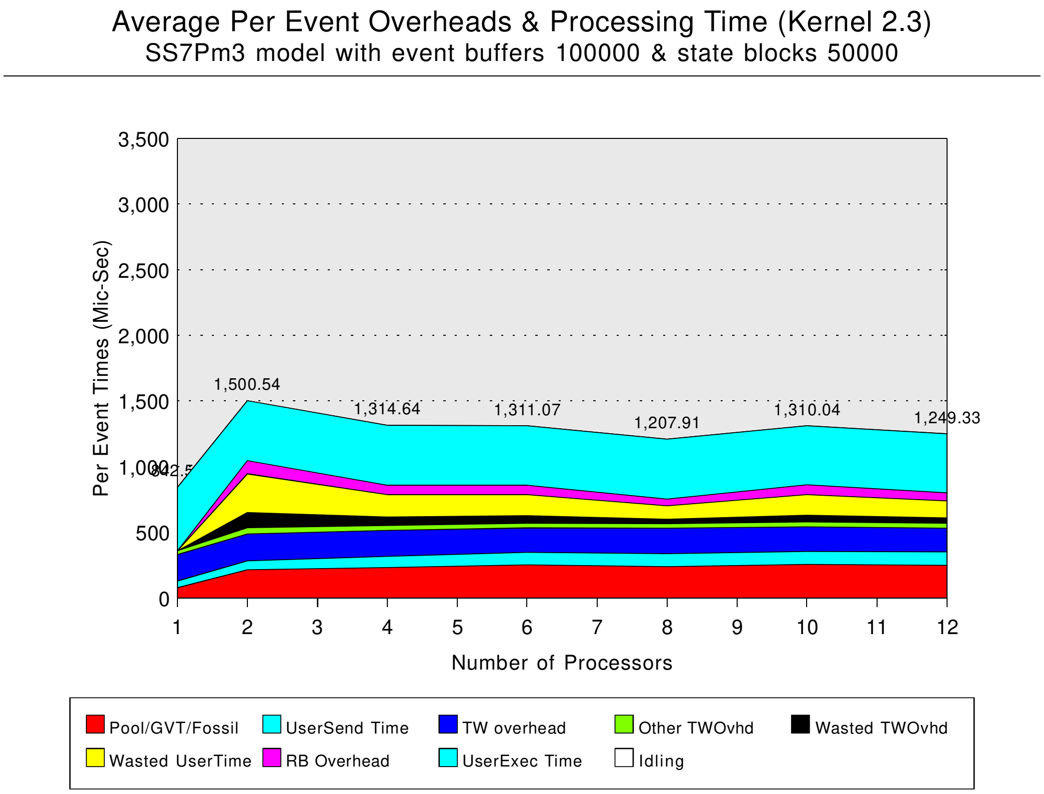
Number of processors

Figure 17: Global Contral Overheads (Qnet)



Num. PEs	Pool GVT Fossil	User Send Time	TW Over-head	Other Over-head	Wasted TW Ovhd	Wasted User Time	RB Over-head	User Exec Time	Idling	Net Event Rate
1	120.04	61.10	270.67	29.24	0	0	0	517.02	0	100%
2	205.43	57.28	251.94	30.72	17.72	38.59	11.71	491.38	0.64	93.43%
4	310.44	65.00	192.53	37.11	64.35	174.97	59.46	458.50	0.67	74.95%
8	567.98	75.49	177.92	71.02	269.00	770.61	244.41	434.19	3.25	39.81%
10	1164.08	75.03	163.70	70.17	252.32	757.70	243.33	416.54	3.81	39.35%
12	561.81	76.66	150.77	81.15	312.26	996.98	285.91	404.71	3.54	32.56%

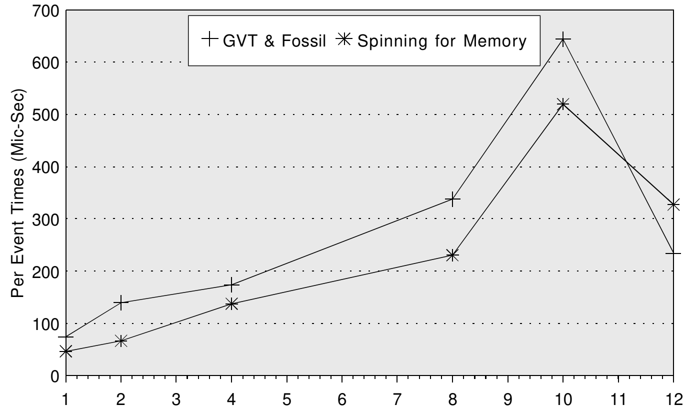
Figure 18: Actual Per Event Times (SS7PM3)



Num. PEs	Pool GVT Fossil	User Send Time	TW Over-head	Other Over-head	Wasted TW Ovhd	Wasted User Time	RB Over-head	User Exec Time	Idling	Net Event Rate
1	78.07	50.10	200.49	28.77	0	0	0	485.13	0	100%
2	211.82	70.09	205.52	45.32	115.23	295.46	99.94	456.88	0.28	64.07%
4	229.14	85.51	198.36	37.73	63.12	172.69	70.79	457.16	0.14	75.86 %
6	251.79	93.89	185.83	37.67	54.50	160.90	71.59	454.76	0.12	77.32%
8	238.06	97.89	193.36	34.41	35.37	101.44	50.58	456.75	0.07	84.54%
10	253.39	100.6	185.36	37.54	52.5	156.25	73.97	450.47	0.16	77.91%
12	248.37	101.1	180.19	36.39	41.93	128.31	62.54	450.32	0.18	81.12%

Figure 19: Actual Per Event Times (SS7PM3)

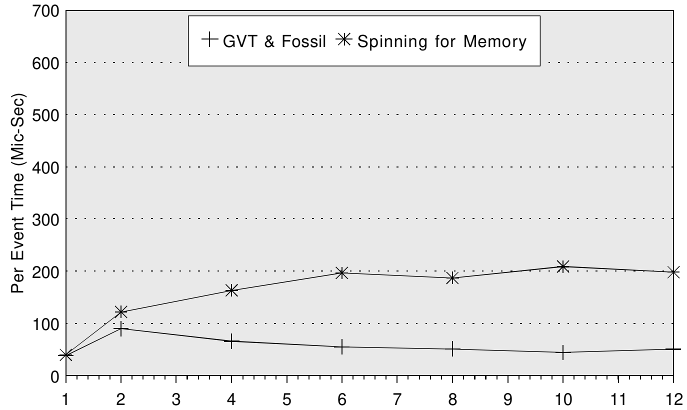
Global Control Overheads (Kernel 2.3)
SS7PM3 model: old partitioning



number of PEs	1	2	4	8	10	12
GVT & Fossil	73.84	139.3	173.22	338.02	644.39	233.84
Spinning for Memory	46.2	66.14	137.22	229.96	519.68	327.68

Number of processors

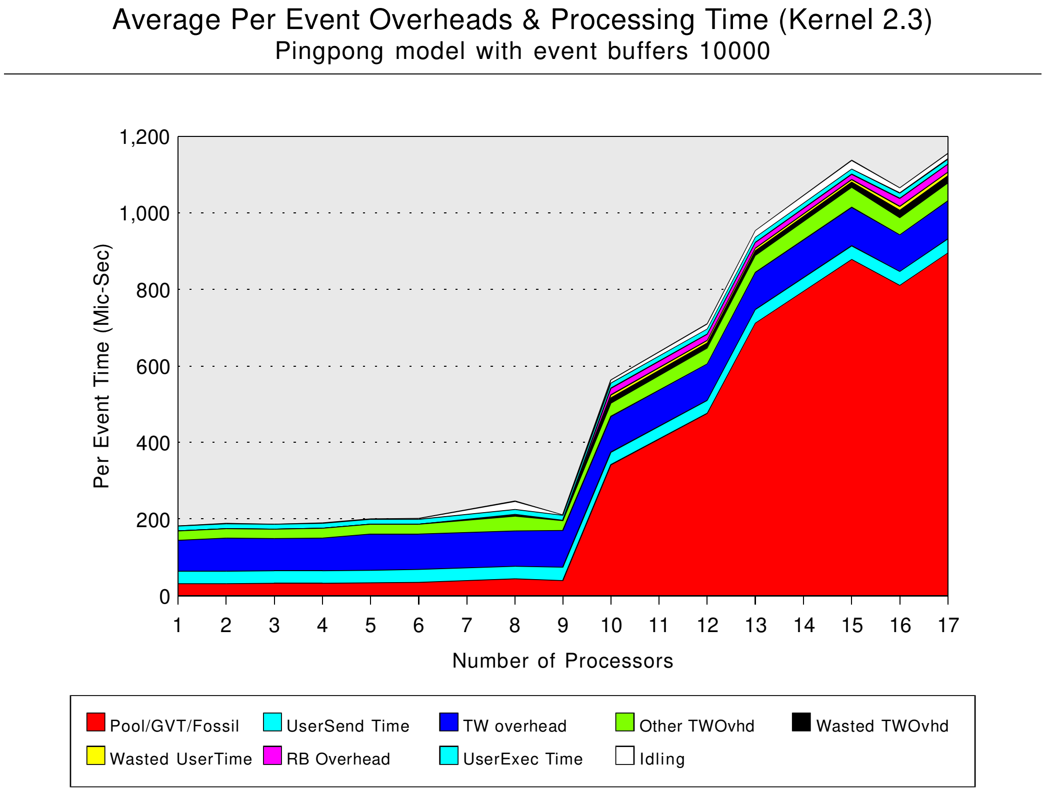
Global Control Overheads (Kernel 2.3)
SS7PM3 model: new partitioning



number of PEs	1	2	4	6	8	10	12
GVT & Fossil	38.2	89.92	65.87	55.28	51.06	44.51	50.46
Spinning for Memory	39.87	121.9	163.27	196.5	187	208.89	197.91

Number of processors

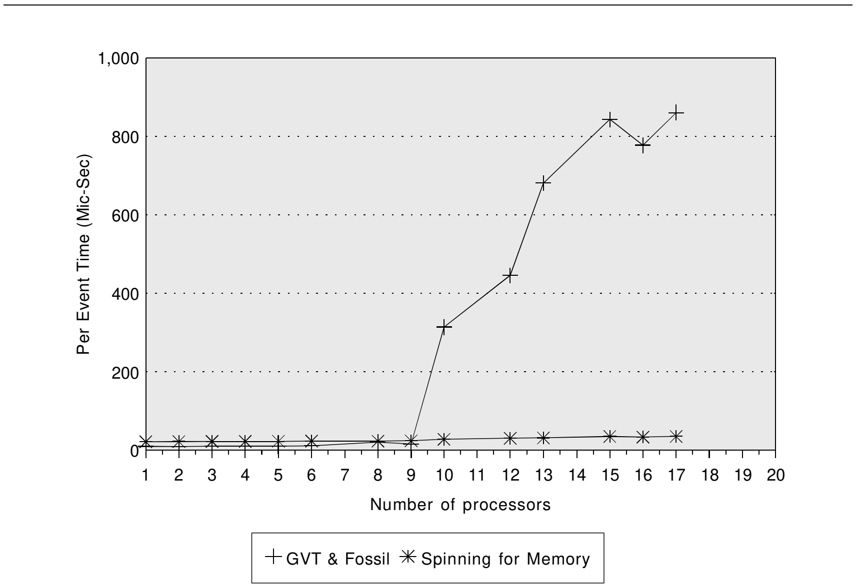
Figure 20: Global Contral Overheads (SS7PM3)



Num. PEs	Pool GVT Fossil	User Send Time	TW Over-head	Other Over-head	Wasted TW Ovhd	Wasted User Time	RB Over-head	User Exec Time	Idling	Net Event Rate
1	31.45	32.47	80.99	24.05	0	0	0	12.98	0	100%
2	31.44	32.56	86.62	24.28	0	0	0	12.98	0.14	100%
3	32.22	32.7	84.09	24.27	0	0	0	13.05	0.27	100%
4	32.8	32.36	85.60	25	0	0	0	13.04	1.28	99.99%
5	33.25	32.7	95.2	24.75	0.05	0	0.05	13	0.9	99.96%
6	34.5	33.66	92.4	25.38	0.30	0.12	0.36	13.02	1.56	99.71%
8	44.08	33.04	91.36	39.44	1.44	0.72	1.76	13.12	21.6	98.47%
9	39.6	34.74	95.85	25.56	0.18	0.09	0.27	13.05	1.35	99.79%
10	341.5	33.1	92.9	35.4	14.7	7.3	17.7	13.2	8.6	86.34%
12	445.35	34.2	96.12	41.04	13.56	6.72	16.44	13.2	13.44	87.61%
13	712.53	35.23	96.98	44.59	12.22	6.11	15.34	13.26	17.42	88.8%
15	878.55	35.55	100.65	52.35	13.35	6.45	14.25	13.2	22.8	88.35%
16	810.88	36.16	96	44.16	19.52	10.24	21.12	13.76	13.76	83.07%
17	895.56	36.72	98.77	47.09	18.7	9.52	20.57	13.77	15.13	84.07%

Figure 21: Actual Per Evet Times (Parallel-Pingpong)

Global Control Overheads (Kernel 2.3)
Pingpong Model: event buffer 10000



Global Control Overheads (Kernel 3.1)
Pingpong Model: event buffer 3000

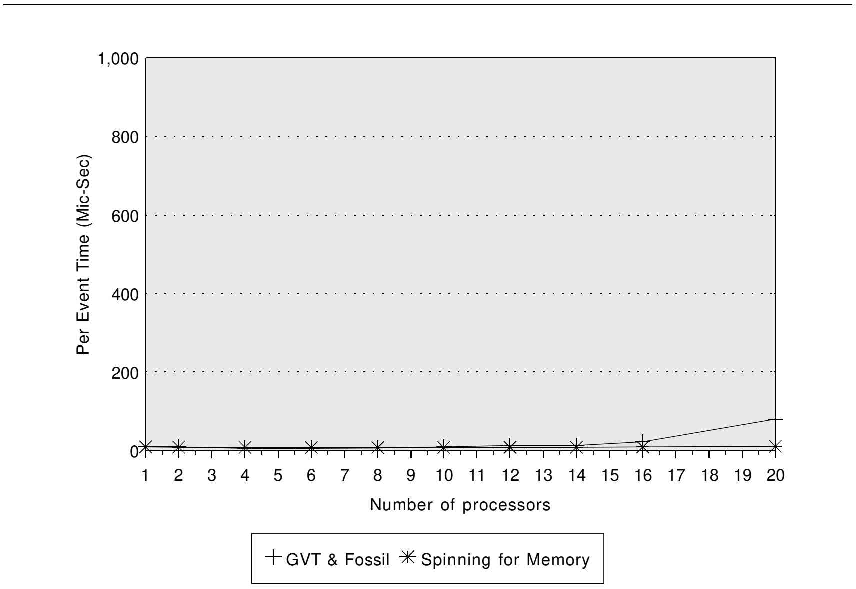


Figure 22: Global Contral Overheads (Parallel-Pingpong)

7 SMTW Kernel 3.1 and Performance

7.1 SMTW Kernel 3.1

The synchronized approach to global control mechanism is still cost too dearly. Besides, the cancelback mechanism does not work properly with the distributed free lists. When the fast running processors run out of free list, the GVT holder processor has plenty of buffers available so there is no point to stop it just for the sake of fast runners. Buffers released through cancelback tend to be quickly picked up by the fast executing processors causing constantly in a back and for situation. Cancelback is necessary with single free list because otherwise the GVT holder processor would not have event buffer available to proceed. With distributed scheme, the fast runner may simply chose to wait until GVT advances. As the bottle neck caused by central free lists suggests that synchronized approach will pose serious performance problem for parallel processing on shared memory architecture.

The SMTW Kernel version 3.1 is a major departure from the version 2.2 and version 2.3. A new GVT algorithm and new memory management scheme have replaced the old ones that requires total synchronization when coming to GVT computation and fossil-collection and cancelback if necessary. The main features of the new method are described briefly in the following subsections.

7.1.1 Fast Asynchronous Continuous (FAC) GVT estimation

FAC algorithm does not involve any synchronization or explicit information requests among processors. Any processor may start GVT estimation at any time (most probably on demand) without the awareness of other processors. Each processor makes contribution to GVT computation continuously at very low cost. The result of GVT estimation is at least the real GVT value at the time a processor signals its intention of computing GVT and at most one step behind the real GVT when it completes its estimation (if the time duration for GVT estimation is less than the granularity of event processing). In the process of GVT estimation, no enquiries are send out and thus no wait for responses are necessary, as a result the computation time is very short and GVT estimation accurate. The GVT value one processor sees as the last

GVT does not necessarily agree with the GVT values other processors see due to the asynchronous nature of this algorithm. This method also lends itself better to a distributed free lists scheme which has been proven to be indispensable in easing the high contentions caused by a single central free list.

7.1.2 Asynchronism

With distributed free lists, The time interval between GVT computation tends to be shorter compared to a single global free list. While some fast running processors have exhausted their free buffers and must call for fossil-collection to reclaim used buffers that are no longer needed, those slow processors still have plenty of buffers to use. A synchronized fossil-collection would ask every processor stop executing events and do fossil-collection. This is very expensive in term of both the cost of fossil-collection itself and the cost of stopping those which are executing along critical path. This situation gets worse with the increase of the number of processors. With asynchronous GVT estimation, it is possible that the fast running processors stops and does fossil- collection without preventing the slow processors from advancing. In some cases this does not really cost the total simulation execution time because the fast runner has to roll back anyway. By properly adjusting the length of free list, over-optimistic execution can be restricted and roll-back rate brought down resulting in better performance. Asynchronism is the core of the new global control mechanism adopted in Kernel 3.1. The asynchronous approach also complies with the Time Warp philosophy.

7.1.3 A much efficient memory management scheme

A comprehensive memory management mechanism is adopted in Kernel 3.1. This mechanism has three components.

1. The first component is a two level free lists structure. Each processor has a private free list and it is not accessible to other processors. When an event is created before sending a message, the event buffer is picked up from local free list instead of from destination processor like in

Kernel 2.3. The main purpose of doing this way is to further reduce the contention for free buffer as no locks are necessary in accessing private free lists. The effect is extensive as shown by the experiments all the costs having to do with free list are brought down. To prevent local free lists from draining out or flooded over, a global free list is used to adjust the buffer level of each processor.

2. The second component is the adjustment of the number of buffers. Longer free list reduces the cost of GVT computation and fossil-collection. But on the other hand, it requires higher memory utilization and less efficient use of the cache, also tends to increase the number of rollbacks, all these would degrade performance. The curve of performance (measured by net event rate) versus number of buffers has a top area. The position of the top and the "flatness" of the top area depends on a given platform very much on the characteristics of the memory management mechanism which has a great impact on the overall performance. It is desirable to run simulation under such a number of buffers that gives best performance. With Kernel 2.2 and Kernel 2.3, the cost of global control is high that we had to run SS7-PM model with several hundreds of thousand buffers to get better performance, the result is much higher overall system overhead and poor performance. In sharp contrast, with Kernel 3.1 best performance of SS7PM model with 4036 logical processes can be obtained with around just eighteen thousand buffers because the global cost is low enough to be able to afford much higher GVT computation frequency to take advantage of the benefit offered by lower memory utilization. The issue here is how to adjust buffer level in a distributed free list scheme. A simple and fair method is adopted – trying to keep the number of buffers on each processor at certain level. Here the number of buffers currently on a processor refers to the number of available free buffers plus the number of buffers occupied by processed events. This number (after fossil-collection) is a reasonable measure of how fast the processor is executing in terms of virtual time. The faster a processor is executing, the less the free buffers are available and the more frequent GVT computation is invoked. This in effect slows down the faster runner without affecting the GVT holder processors. By keeping the number of buffers at reasonable size the cost of GVT and fossil-collection can be kept negligible.

The buffer adjustment is carried out after each call to fossil-collection, the number of buffers as defined above is checked, if it exceeds specified level excessive buffers are dropped to the global free list which is shared by all processors, if it is lower than necessary it will be replenished from global list. If the global list does not have enough buffers to be drawn from, then the total number of buffers is not enough to maintain certain level of buffers for each processor and a call to system for more memory space will be invoked. To reduce the frequency of buffer exchanges with global list, certain tolerance is allowed, that is a lower threshold and a higher threshold are specified, the average buffer level (percentage of maximum number of buffers) is a parameter which user can set in configuration file.

3. The third component is the buffer overflow control. Both the Kernel 2.2 and 2.3 use cancalback to handle buffer overflow. When fossil-collection is unable to reclaim back any free buffer, cancelback procedure is called to try to recover some by cancelling those messages that were sent by faster executing logical processes. Cancelback protocol is designed primarily for single central free list scheme, it requires global knowledge to select messages to be canceled back, so it is in line with the synchronous GVT computation and fossil-collection. Its implementation is very expensive (to find one cancelback message, the entire LP space has to be scanned) and the cost of stopping every processor, especially those executing along critical path, to recover a few buffers is prohibitive as all the experiments have revealed.

   With asynchronous GVT estimation and fossil-collection in place, we also need an asynchronous approach to overflow control. Cancelback in effect is to stop the fast executing processors and to spare buffers for much needing slow runners. With distributed free list scheme, this is not necessary because in most cases the slow runners always have enough free buffers available after fossil-collection, it is only possible for the fast runners to find their free list exhausted and unable to reclaim back used ones as they are far ahead of GVT. It will be useless to cancelback some messages and then send them again. Instead of back and forth this way, simply stop the fast runner, let it wait for GVT to advance. The Kernel 3.1 uses a two-step mechanism, first it tries to slow down the fast runner by supplementing a few buffers from the central free list just like cancelback would try to recover buffers, this will engage the fast runner in high frequency of GVT estimation (again without affecting slow runners), if central list is exhausted the fast runner then have to wait until the processor that will advance the GVT to wake him up. With careful design, the simulation should be able to progress without falling into starvation provided the free lists are longer than certain limit. If the free lists are not sufficient and cause the simulation falls into infinite loop, timeout will come as the last resort. This mechanism is efficient in that the fast runners will immediately resume its execution as soon as the GVT advances without the type of extra overhead incurred in rollback. Experiments have shown that within certain limit a shorter free list over-perform longer one due to restricted over-optimistic execution and low memory management cost which allows for higher GVT computation frequency without incurring high cost.

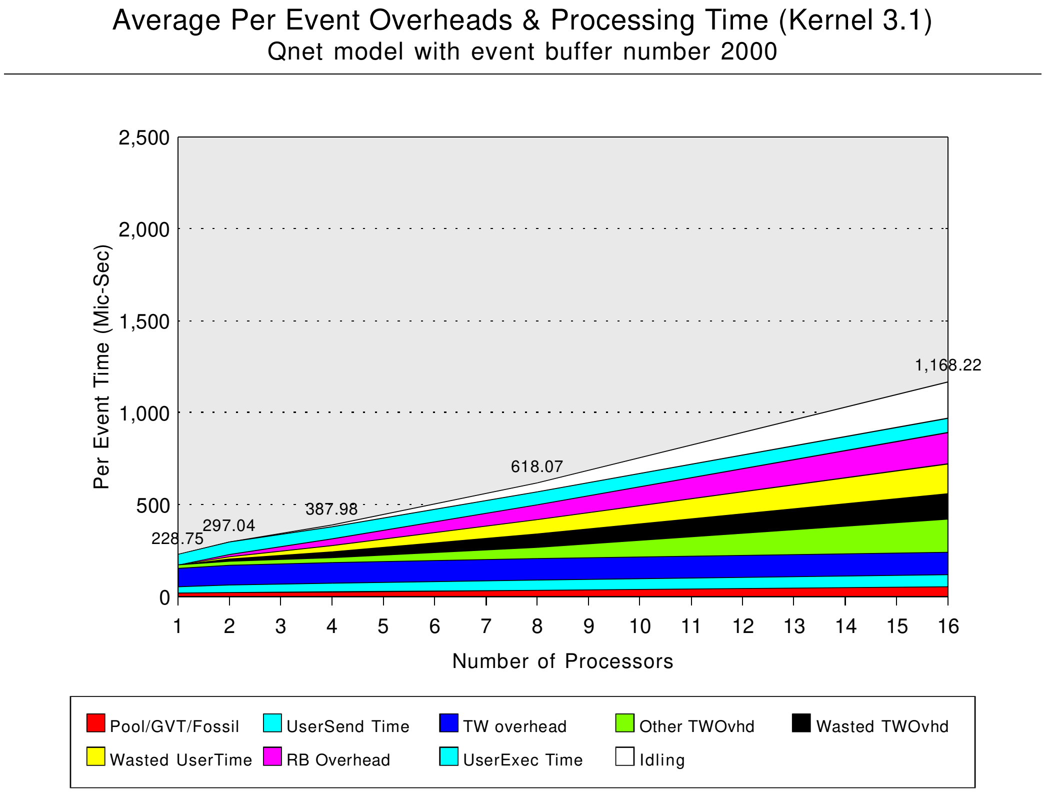
7.2 SMTW Kernel 3.1 performance

Figure 23 presents the performance results of Qnet model running under Kernel 3.1. The overall performance is much improved compared to the results for Kernel 2.3 (Figure 16), even though the rollback rate is higher. Normal Time Warp overhead is lower, but the major contributing fact is the further drop of global control cost. Not only the total cost of global control is much lower (less than half the value on Kernel 2.3), but the cost almost remains the same if the effect of rollback rate is excluded. The biggest cost reduction for this particular model comes from *Spinning for Memory*, a consequence of eliminating locks on local free lists.

The shape of overheads chart shown in Figure 24 is in sharp contrast with Figure 21. The total per event average time is very stable up to 16 processor, then increases on 20 processors but not a sharp climb. Here the difference lies in the cost of GVT and memory management. As we described previously that erroneous processing tends to spread faster with Kernel 3.1 because of the asynchronous approach and much less constraints imposed by the kernel, this can be restricted by using smaller free pools. For the set of tests, the same smaller size of free lists are used for all simulation runs. The faster running processors consume their free list faster and less buffers could be reclaimed as the GVT is log behind, as result they have to call GVT computation and do fossil collection more often. The more processors, the shorter the free list, and the higher the frequency of GVT invocation. This in effect slows down the faster running without affecting the GVT holder processor. Frequent GVT and fossil collection does not necessarily cost the overall execution time as for one thing the overhead is low compared to earlier Kernel versions, more importantly the GVT holder processor is not affected and the faster runner processors spend more time on GVT computation and fossil collection , or even just spin instead of doing rollback all the time. This can be seen from the lower rollback rate on Kernel 3.1. In contrast, the same model running on Kernel 2.3 could not afford to frequent GVT invocation, the more frequent invocation, the slower the overall execution, because the GVT holder has to stop either. This seems getting worse with the invocation of cancelback.

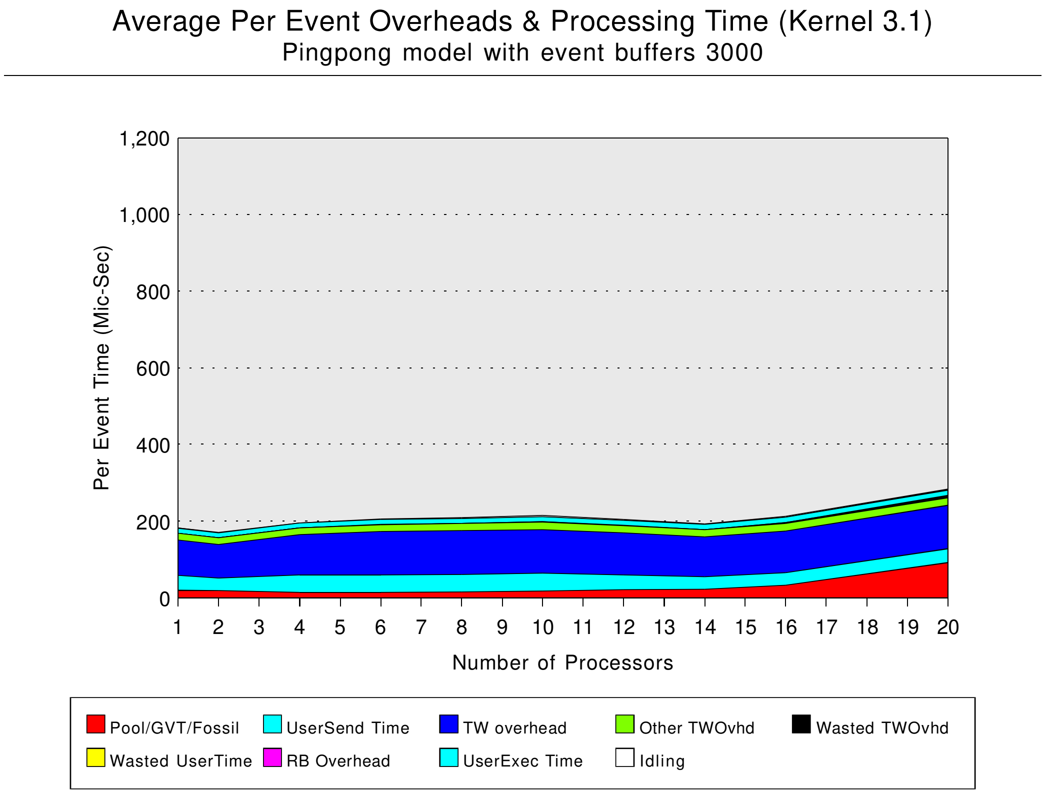
The results for SS7PM model indicate again that kernel 3.1 can handle extremely off-balanced situation exceptionally well. By comparing Figure 26 and Figure 18 (both models use old partitioning method which gives very poor performance above 8 processors), it can be seen that the global control cost is very low and stable. Notice that the rollback rate is much higher with Kernel 3.1. The new partitioning method for SS7PM model works very well except on two processors where it slows down instead of speeding up. Figure 27 show the results on Kernel 3.1. The overall performance seems not as good as on Kernel 2.3 (Figure 19). This is because the much higher rollback rate on Kernel 3.1. The global cost is only about one third the cost for Kernel 2.3. By taking off the effect of rollback, the resulting expected relative speedup is almost linear and very close to ideal speedup as Figure 29 suggests. The loads of processors (total number of committed events) are well balanced so that the execution is not stretched to the limit of free pool and no excessive global cost similar to that for the fine granularity Parallel Pingpong model is reported on kernel 2.3.

In the Kernel 3.1, all the system overheads are not sensitive to the number of processors on which it is running, a desirable property we have been looking for. However, this is not without cost. Under the same condition, simulations running under Kernel 3.1 are more sensitive to the KSR load conditions and tend to have higher rollback rate. This is the result of the asynchronous



Num. PEs	Pool GVT Fossil	User Send Time	TW Over-head	Other Over-head	Wasted TW Ovhd	Wasted User Time	RB Over-head	User Exec Time	Idling	Net Event Rate
1	19.81	33.68	99.63	17.75	0	0	0	57.88	0	100%
2	21.94	40.44	107.58	20.46	12.18	12.42	12.16	69.24	0.62	89.83%
4	24.89	47.75	110.76	28.62	31.7	32.95	35.44	67.37	8.50	77.75%
8	32.53	56.48	116.53	61.32	71.84	77.78	82.03	69.68	49.88	61.86%
16	51.79	66.30	123.02	180.27	137.88	161.18	170.48	77.5	199.8	47.15%

Figure 23: Actual Per Evet Times (Qnet)

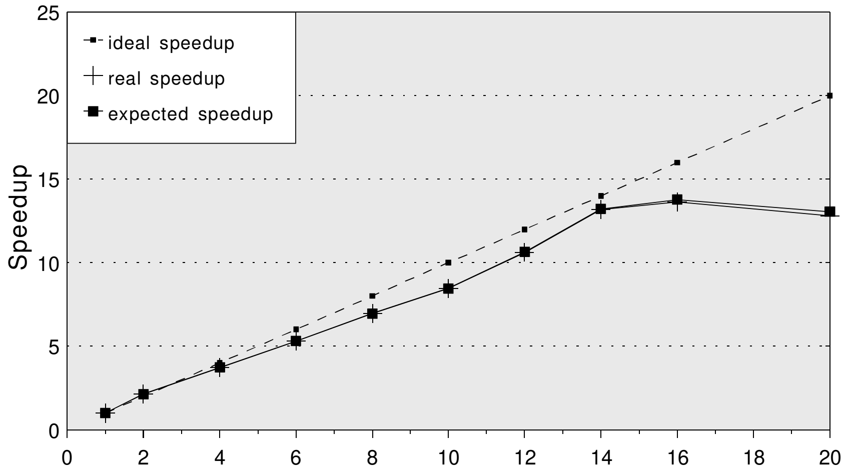


Num. PEs	Pool GVT Fossil	User Send Time	TW Over-head	Other Over-head	Wasted TW Ovhd	Wasted User Time	RB Over-head	User Exec Time	Idling	Net Event Rate
1	18.85	38.95	92.7	17.09	0	0	0	13.56	0	100%
2	18.14	32.73	87.58	17.44	0	0	0	13.35	0.78	100%
4	14.03	45.67	104.11	17.36	0.02	0.01	0.03	13.24	0.55	99.99 %
6	13.88	45.64	112.4	18.23	0.14	0.08	0.16	13.27	1.71	99.87%
8	14.68	45.88	113.87	18.52	0.09	0.05	0.1	13.21	2.24	99.92%
10	17.41	46.35	113.85	19.33	0.55	0.29	0.59	13.22	2.84	99.52%
12	21.2	38.24	109.05	19.39	0.17	0.08	0.17	13.2	2.96	99.84%
14	21.75	32.36	104.22	18.46	0.28	0.12	0.26	13.28	1.48	99.73%
16	31.85	33.41	108.56	19.76	1.13	0.49	1.14	13.41	2.65	98.97%
20	91.43	36.07	112.95	20.34	2.26	1	2.23	13.89	2.71	98.04%

Figure 24: Actual Per Evet Times (Parallel-Pingpong)

Relative Speedup (Kernel 3.1)

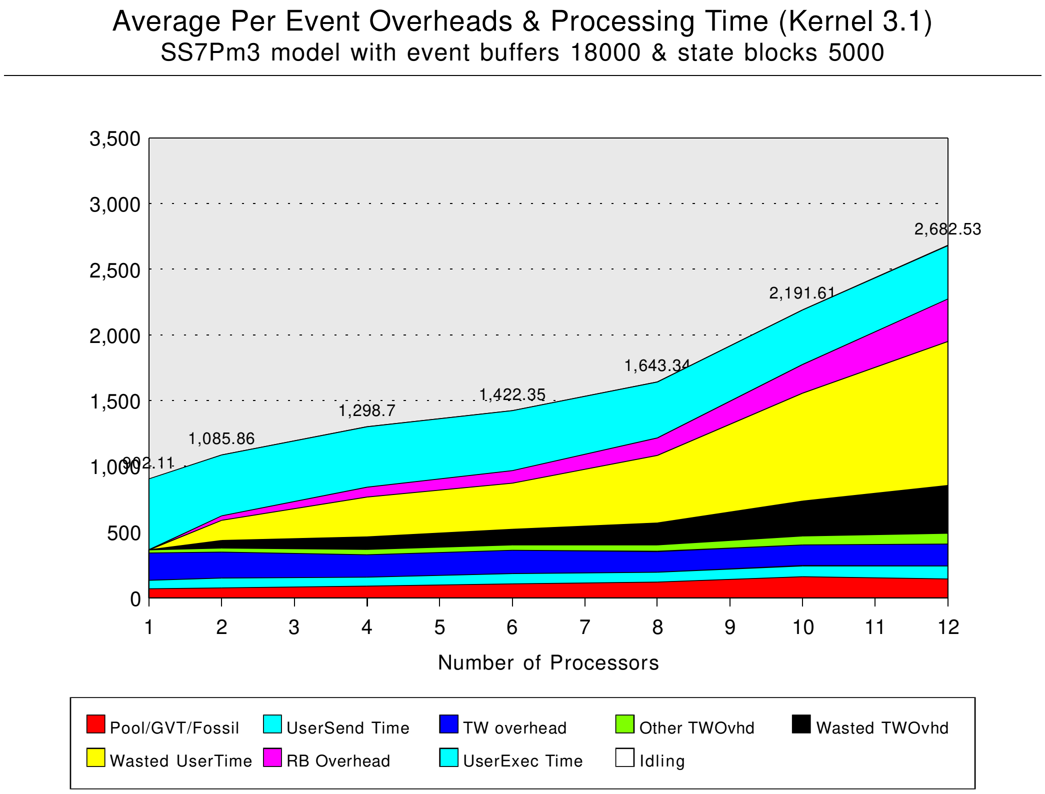
Parallel Pingpong: event buffers 3000, no artificial computation



number of PEs	1	2	4	6	8	10	12	14	16	20
ideal speedup	1	2	4	6	8	10	12	14	16	20
real speedup	1	2.13	3.72	5.29	6.95	8.45	10.63	13.19	13.65	12.81
expected speedup	1	2.13	3.72	5.3	6.95	8.45	10.65	13.23	13.79	13.06

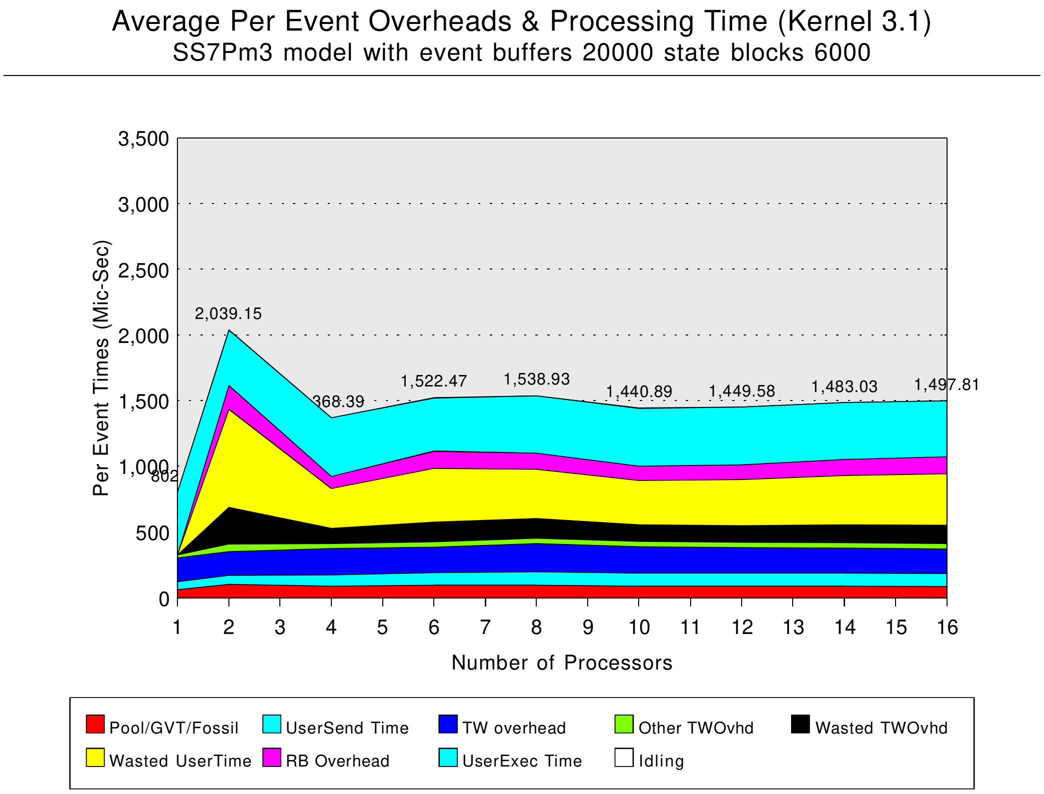
Number of Processors

Figure 25: Relative Speedup (Parallel-Pingpong)



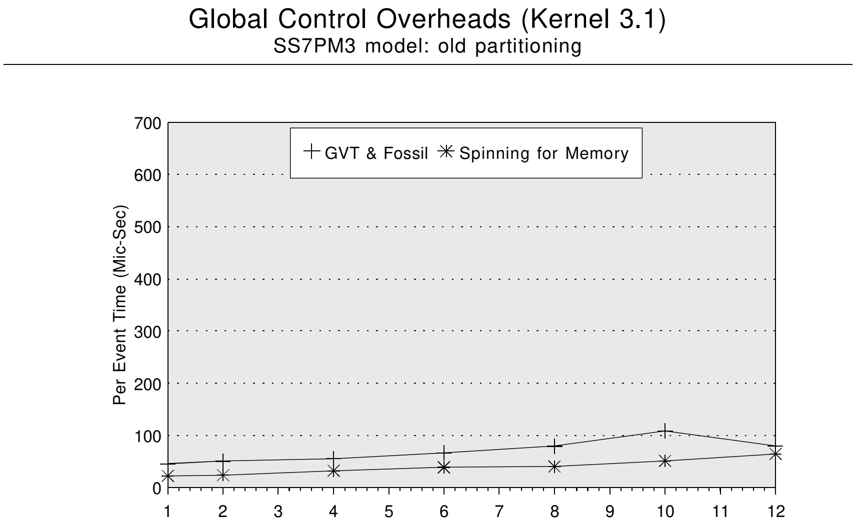
Num. PEs	Pool GVT Fossil	User Send Time	TW Over-head	Other Over-head	Wasted TW Ovhd	Wasted User Time	RB Over-head	User Exec Time	Idling	Net Event Rate
1	67.84	63.41	206.63	25.46	0	0	0	538.77	0	100%
2	73.96	73.48	199.8	29.86	57.46	154.06	34.61	462.2	0.43	77.66%
4	86.91	68.83	169.39	40.07	97.81	303.52	74.64	456.81	0.72	63.39%
6	104.87	77.61	176.31	43.68	115.88	351.12	95.73	456.59	0.56	60.34%
8	119.33	72.99	159.18	50.92	164.48	516.09	132.95	426.47	0.93	49.18%
10	159.42	82.81	160.08	65.72	265.8	823.27	219.35	413.01	2.00	37.59%
12	143.45	97.49	166.22	81.14	363.63	1099.63	322.71	405.16	3.1	31.37%
14	409.74	84.87	158.28	122.77	584.91	1801.68	496.44	402.68	5.93	21.30%
16	639.74	81.15	156.06	144.46	704.95	2147.85	566.02	394.33	7.49	18.13%

Figure 26: Actual Per Event Times (SS7PM3)



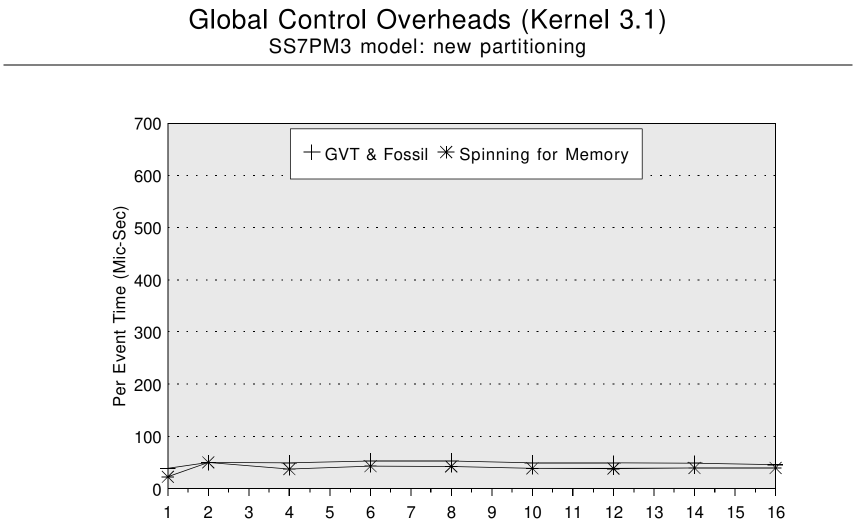
Num. PEs	Pool GVT Fossil	User Send Time	TW Over-head	Other Over-head	Wasted TW Ovhd	Wasted User Time	RB Over-head	User Exec Time	Idling	Net Event Rate
1	61.51	58.72	183.26	21.66	0	0	0	477.16	0	100%
2	100.05	67.8	183.14	55.32	279.35	746.56	184.61	421.65	0.67	39.6%
4	86.65	84.94	200.76	37.02	115.46	305.99	90.10	447.14	0.33	63.49%
6	95.81	92.58	193.77	43.14	148.91	408.08	131.77	438.43	0.33	56.54%
8	95.52	100.07	215.26	42.46	149.08	373.74	122.82	439.58	0.40	59.08%
10	88.41	97.43	200.62	40.32	126.56	338.92	108.35	439.85	0.43	61.32%
12	87.72	99.17	194.23	41.30	125.99	348.68	113.72	438.35	0.42	60.65%
14	88.50	98.04	190.64	42.27	134.88	374.71	122.08	431.57	0.34	58.56%
16	85.21	96.72	186.50	43.70	139.38	391.66	126.96	427.35	0.33	57.23%

Figure 27: Actual Per Event Times (SS7PM3)



number of PEs	1	2	4	6	8	10	12
GVT & Fossil	45.52	50.46	54.94	66.35	79.23	108.33	79.24
Spinning for Memory	22.32	23.49	31.97	38.52	40.1	51.09	64.2

Number of processors



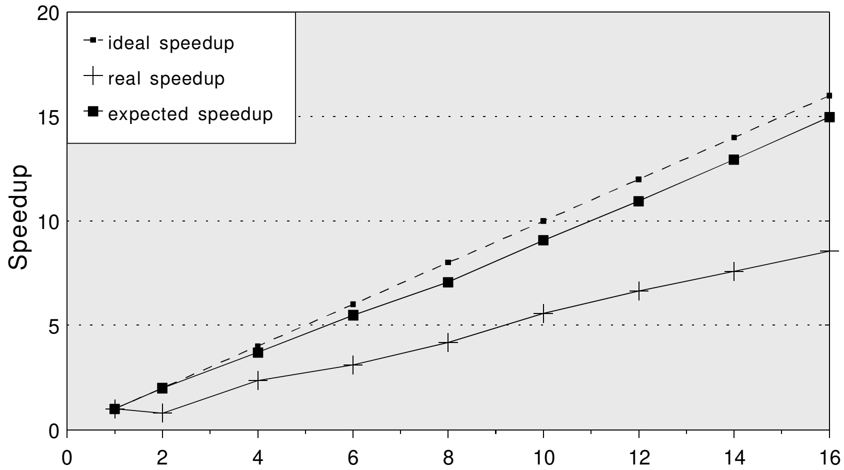
number of PEs	1	2	4	6	8	10	12	14	16
GVT & Fossil	38.91	49.77	49.49	52.93	53.01	49.52	49.18	48.9	45.58
Spinning for Memory	22.6	50.28	37.16	42.89	42.51	38.89	38.54	39.6	39.63

Number of processors

Figure 28: Global Contral Overheads (SS7PM3)

Relative Speedup (Kernel 3.1)

SS7PM3 model: event buffers 20000, state blocks 6000.



number of PEs		1	2	4	6	8	10	12	14	16
ideal speedup		1	2	4	6	8	10	12	14	16
real speedup		1	0.79	2.35	3.1	4.17	5.57	6.64	7.58	8.57
expected speedup		1	1.99	3.7	5.48	7.06	9.08	10.95	12.94	14.98

Number of Processors

Figure 29: Relative Speedup (SS7PM3)

approach to GVT computation and memory management, such things as synchronized GVT computation and fossil-collection, frequent access to shared lists would have some kind of synchronizing effect. With much less such restraints each processor tends to go as far as it can. When running applications with irregular workload, it is very likely that some processors will be executing far ahead in terms of virtual time. Experiments with SS7PM3 and Parallel pingpong models show that when running on more than 8 processors the committed event rate varies widely depending on the machine condition, lower event rate is always accompanied by higher rollback rate and longer rollback distance. Fortunately, this is not a serious problem in the sense that we will be implementing some load balancing mechanism to restrain rollback. Even with higher rollback rate the overall performance is still much better for such applications as Qnet model.

Appendix

A Statistics collection of the SMTW Kernel

The SMTW Kernel instrumentation has been designed specifically for the purpose of performance study for both the application model and the Kernel itself. Extensive statistics and timing data are recorded for each Logical Process (LP) and each Processor (PE) during a simulation run. A thorough understanding of the statistics data is fundamental to the performance study. Here we first explain the set of basic data recorded by the Kernel, followed by definitions for each item of the three levels of statistics output.

Basic statistics data collection

The following data concerning events are collected for each LP:

NEvent number of events processed.

NAbort number of aborted events (preempted).

NCancel number of processed events canceled.

NStraggler number of straggler induced rollbacks.

NRBStraggler number of processed events rolled back by stragglers.

NSecondary number of cancellation induced rollbacks.

NRBSecondary number of processed events rolled back by cancellation.

NArt number of artificial rollbacks (cancelback induced).

NRBArt number of processed events rolled back artificially.

LongestRB longest rollback in events.

EvExecTime accumulation of user event processing time including time spent on sending messages and possible GVT calculation and Fossil-collection for acquiring event buffers or State Blocks.

The following timing data are recorded for each PE (in Sec.)

TWoverhead accumulated Time Warp overhead on scheduling events, copying state (if applicable), ideling and rollbacks, excluding send time.

Idletime accumulated time of idling, it is recorded as part of TWoverhead.

StrglRBtime accumulated overhead on straggler induced rollbacks, it is part of TWoverhead.

ScndRBtime accumulated overhead on cancellation induced rollbacks, it is part of TWoverhead.

Searchtime accumulated insert time of the schedule list (one for each PE), it is part of TWoverhead.

Copytime accumulated time spent on copying state, in effect only if copying state saving is turned on, zero With Incremental State Saving on. It is part of TWoverhead.

Goodtime accumulated successful event processing time.

Badtime accumulated aborted event processing time. The relationship between the event processing time recorded here and the *EvExectime* recorded for each LP is:

$$Goodtime + Badtime = \sum_{i=0}^{n-1} EvExectime$$

where n is the number of LPs on the same PE.

NumFC number of calls to Fossil-collection.

FCtime(0) accumulated time spent on Fossil-collection requested by other PEs, not applicable to Kernel 3.1 and later versions.

FCtime(1) accumulated time spent on Fossil-collection entered within `Send()` procedure when no more event buffers available from event buffer free pool.

FCtime(2) accumulated time spent on Fossil-collection called when possible termination is detected.

FCtime(3) accumulated time spent on Fossil-collection entered within `SaveInt()` procedure when no more free State Blocks available from State Block free pool.

FCWait-time accumulated waiting time on barrier-synchronization for Fossil-collection, which is part of the $FCtime[]$s. It is always zero with Kernel 3.1 and later versions.

Spintime accumulated time spent on acquiring a free event buffer or State Block, which is part of the user event processing time and which includes $FCtime[1]$ and $FCtime[3]$.

Proctime accumulated event processing time spent on executing application code only.

StartTime the user clock time when the scheduler starts running (after all the schedulers have been bound to a physical PE.

EndTime the user clock time when the scheduler finishes execution. Since both the start point and the end point are synchronized among PEs, the elapsed time ($EndTime - StartTime$) should be very close across all PEs.

InitTime the time spent on initialization – from after all the threads (executing scheduler) having been bound to their own physical PE until all of them having finished executing user-defined `IProcMsg()`, excluding time spent within `IProcMsg()`.

NPEs number of PEs involved in a simulation session, each PE runs a single thread that executes the procedure `scheduler()`.

Note that GVT calculation is invoked only when Fossil-collection is necessary to reclaim event buffers and State blocks occupied by obsolete events. A procedure named `FossilCollect()` incorporates GVT calculation, fossil-collection and possible cancelback into a single module (Cancelback is not used in Kernel 3.1 and later versions).

Statistics output

The statistics output from each simulation run has three levels which can been turned on/off by setting the statistics level (0, 1, 2) option in the kernel configuration file .tw-config1.

Summary Time Warp Statistics

Level 0 is the "Summary Time Warp Statistics", which always gets printed out after a successful run. These are the summary data and essential timing data calculated based on the sets of basic statistics data given above. In the following definitions, nlp denotes the total number of LPs. Each definition starts with the text describing the item which actually appears in the statistics output followed by the formula used to calculate the value.

"Event processed"

$$TNEvent = \sum_{i=0}^{nlp-1} NEvent$$

"Events aborted"

$$TNAbort = \sum_{i=0}^{nlp-1} NAbort$$

"Processed Events Cncl'd"

$$TNCancel = \sum_{i=0}^{nlp-1} NCancel$$

"Straggler Induced RBs"
"Processed Events RB'd"

$$TNStraggler = \sum_{i=0}^{nlp-1} NStraggler$$

$$TNRBStraggler = \sum_{i=0}^{nlp-1} NRBStarggler$$

"Cncl Evnt Induced RBs"
"Processed Events RB'd"

$$TNSecondary = \sum_{i=0}^{nlp-1} NSecondary$$

$$TNRBSecondary = \sum_{i=0}^{nlp-1} NRBSecondary$$

"Artificial RBs"
"Processed Events RB'd"

$$TNArt = \sum_{i=0}^{nlp-1} NArt$$

$$TNRBArt = \sum_{i=0}^{nlp-1} NRBArt$$

"Longest rollback"

$$\text{Maximum } \{LongestRB \text{ of all LPs}\}$$

"Net Event processed" : (and the percentage over TNEvent)

$$TNetEvent = TNEvent - TNCancel - TNRBStraggler$$
$$-TNRBSecondary - TNRBArt$$

$$\frac{TNetEvent}{TNEvent} * 100$$

"Number of rollbacks" :

$$TNStraggler + TNSecondary + TNArt$$

"Proc. Evnts RB'd" : (and the percentage over TNEvent)

$$TNRBStraggler + TNRBSecondary + TNRBArt$$
$$\frac{TNRBStraggler + TNRBSecondary + TNRBArt}{TNEvent} * 100$$

"Proc. Evnts Cncl'd" : (and the percentage over TNEvent)

$$TNCancel$$
$$\frac{TNCancel}{TNEvent} * 100$$

In the following definitions, $NPEs$ denotes the number of **processors**.

"Average RB Distance" :

$$\frac{TNRBStraggler + TNRBSecondary + TNRBArt}{TNStraggler + TNSecondary + TNArt}$$

"Actual execution time (Sec)" : (the average per PE execution time)

$$Exectime = \frac{\sum_{i=0}^{NPEs-1}(EndTime - StartTime)}{NPEs}$$

"Initialization time" : (part of $Exectime$)

$$\frac{\sum_{i=0}^{NPEs-1} InitTime}{NPEs}$$

"Event Rate (NetEvnts/Sec) :

$$\frac{TNetEvent}{Exectime - InitTime}$$

"Critcl Path user execution time" : (explained below)

$$\sum_{critical-path} ProcTime$$

"Sequential user execution time" :

$$\sum_{i=0}^{NPEs-1} ProcTime * \frac{TNetEvent}{TNEvent}$$

This is followed by the sequential user time with send time included:

$$(\sum_{i=0}^{NPEs-1} (Goodtime + Badtime - Spintime) * \frac{TNetEvent}{TNEvent})$$

"Effective per net event time" : (in micro-seconds)

$$\frac{Exectime - InitTime}{TNetEvent} * 1000000$$

What follows are the break-down of this per net event time and all the items used in expressions below are average per PE. Please notice that the actual per net event time should be multiplied by the Number of PEs.

" per event TW overhead" :

$$PerEventTWoverhead = \frac{PerPETWoverhead}{TNEvent} * 1000000$$

$$PerPETWoverhead = \frac{\sum_{i=0}^{NPEs-1}(TWoverhead - Idletime - StrgRBtime - ScndRBtime)}{NPEs}$$

" (copy time)" :

$$\frac{\sum_{i=0}^{NPEs-1} Copytime}{NPEs} * \frac{1000000}{TNEvent}$$

" other" : (see explanation below)

$$\frac{dis}{TNetEvent} * 1000000$$

" wasted TW overhead" : (distributed to per Net Event)

$$\frac{PerEventTWoverhead * (TNEvent - TNetEvent)}{TNetEvent}$$

" idling" :

$$\frac{\sum_{i=0}^{NPEs-1} Idletime}{NPEs} * \frac{1000000}{TNetEvent}$$

" RB overhead" :

$$\frac{\sum_{i=0}^{NPEs-1}(StrglRBtime + ScndRBtime)}{NPEs} * \frac{1000000}{TNetEvent}$$

" user execution time" :

$$UserProcTime = \frac{\sum_{i=0}^{NPEs-1} Proctime}{NPEs} * \frac{1000000}{TNEvent}$$

" user send time" :

$$UserExecTime - UserProcTime$$

$$UserExecTime = \frac{\sum_{i=0}^{NPEs-1}(Goodtime + Badtime - Spintime)}{NPEs} * \frac{1000000}{TNEvent}$$

" wasted user time" : (distributed to per Net Event)

$$\frac{UserExecTime * (TNEvent - TNetEvent)}{TNetEvent}$$

" Pools, GVT and termination" :

$$\frac{\sum_{i=0}^{NPEs-1}(Spintime + FCtime[0] + FCtime[2])}{NPEs} * \frac{1000000}{TNetEvent}$$

The number given under "Pools, GVT and termination" includes the total of the first category plus the "spinning for memory" of the third category in Monitoring Statistics. It comprises the time spent on obtaining a buffer or a State Block from the free pools and the time on executing procedure `FossilCollet()` ($FCtime[0]$, $FCtime[1]$, $FCtiem[2]$, $FCtime[3]$). This is the major portion of system overhead on Kernel 2.2 and 2.3 and it is subject to the load condition of the processors executing the simulation. It is also where improvement can be made in terms of the kernel performance. GVT computation and fossil-collection are very expensive on Kernel 2.2 and 2.3, a centralized free pool becomes the bottle-neck of total system overhead.

A Critical Path computation has been installed, it calculates the user execution time along Critical Path on the fly. The result of this computation is given in "Summary Time Warp Statistics" under the item of "Critcl Path user execution time". Critical Path user execution time is the least time needed to execute the simulation given zero overhead, causal relations among LPs and enough processors to allow at any time the execution of any process ready to execute. It indicates the maximum possible parallelism provided by the application. Along with Critical Path execution time also calculated is the "Sequential user execution time" which is the sum of user execution time (user code only)for all committed events. These are the execution time spent on application code excluding the time necessary for sending messages.

The ratio of Sequential execution time over Critical Path execution time can be used as a measure of the parallelism inherent in the application. That is the maximum speedup that the application is able to provide.

Monitoring Statistics

Level 1 is the "Monitoring Statistics" which gives average per PE timing data in three categories with each broken down into more details.

"GVT Computation, Fossil Collection and Cancelback" –

"Total spin time" :

$$\frac{\sum_{i=0}^{NPEs-1} FCWait\_time}{NPEs}$$

"Total Computation time" :

$$\frac{\sum_{i=0}^{NPEs-1}(FCtime[0] + FCtime[1] + FCtime[2] + FCtime[3] - FCWait\_time)}{NPEs}$$

"Termination time" : (included in "Total Computation time")

$$\frac{\sum_{i=0}^{NPEs-1} FCtime[2]}{NPEs}$$

"No. of calls" :

$$\frac{\sum_{i=0}^{NPEs-1} NumFC}{NPEs}$$

"total" :

$$\frac{\sum_{i=0}^{NPEs-1}(FCtime[0] + FCtime[1] + FCtime[2] + FCtime[3])}{NPEs}$$

"Time Warp Overhead" –

"Queue search time" :

$$\frac{\sum_{i=0}^{NPEs-1} Searchtime}{NPEs}$$

"Idling time" :

$$\frac{\sum_{i=0}^{NPEs-1} Idletime}{NPEs}$$

"StragglerRB time" :

$$\frac{\sum_{i=0}^{NPEs-1} StrglRBtime}{NPEs}$$

"SecondaryRB time" :

$$\frac{\sum_{i=0}^{NPEs-1} ScndRBtime}{NPEs}$$

"real overhead" :

$$\frac{\sum_{i=0}^{NPEs-1}(TWoverhead - Searchtime - Idletime - StrglRBtime - ScndRBtime)}{NPEs}$$

"other" :

the discrepancy between recorded (Endtime - StartTime) and the total of three categories is denoted by "dis", which make up for those code statements that are left out in timing, may also be considered as TW overhead.

"total" :

$$\frac{\sum_{i=0}^{NPEs-1} TWoverhead}{NPEs}$$

"Event processing" –

"Computation time" :

$$\frac{\sum_{i=0}^{NPEs-1}(Goodtime + Badtime - Spintime)}{NPEs}$$

"Spinning for memory" :

$$\frac{\sum_{i=0}^{NPEs-1}(Spintime - FCtime[1] - FCtime[3])}{NPEs}$$

"total" :

$$\frac{\sum_{i=0}^{NPEs-1}(Goodtime + Badtime - FCtime[1] - FCtime[3])}{NPEs}$$

"Total execution time" :

sum of the above three categories, equal to Exectime minus InitTime given in "Summary statistics".

Time Warp Statistics

Level 2 prints out for each LP the following:

```
"LP"          LP Id.
"Events"      NEvent.
"MS/Ev"       EvExecTime*1000/NEvent (in mini-sec).
"Abort"       NAbort.
"Cancel"      NCancel.
"NStrgl"      NStraggler.
"RB:Strl"     NRBStraggler.
"NSecnd"      NSecondary.
"RB:scnd"     NRBSecondary.
```

```
"NArt"        NArt.
"RB:Art"      NRBArt.
"LRB"         LongestRB.
"NetEvt"      NEvent - NCancel - NRBStraggler -
              NRBSecondary - NRBArt.
```

and it also prints out for each PE the same categories of timing data as those given in "Monitoring Statistics" plus the last column which shows the initialization time on each PE.

References

[1] M. Arlitt, Y. Chen, R. Gurski and C. Williamson, "Traffic Modeling in the ATM-TN Telesim Project: Design, Implementation, and Performance Evaluation", Proceedings of the 1995 Summer Computer Simulation Conference (SCSC'95), Ottawa, Ontario, pp. 847-851, July 1995.

[2] S. Das, R. Fujimoto, "A Performance Study of the Cancelback Protocol for Time Warp", Proceedings of the 7th Workshop on Parallel and Distributed Simulation, pp. 135-142, July 1993.

[3] S. Das, R. Fujimoto, K. Panesar, D. Allison, M. Hybinette, "A Time Warp System for Shared Memory Multiprocessors", Proceedings of 1994 Winter Simulation Conference, December 1994.

[4] R. M. Fujimoto, "Time Warp on a Shared Memory Multiprocessor", Proceedings of the 1989 International Conference on Parallel Processing, July 1989.

[5] D. Jefferson, "Virtual Time", ACM Transactions on Programming Language and Systems, pp. 405-425, July 1985.