On Net Modeling of OLTP for Parallel Systems

Ludmila Cherkasova, Vadim Kotov
Tomas Rokicki*
Computer Systems Laboratory
HPL-93-16
February, 1993

net modeling, OLTP, TPC-A benchmark, colored Petri nets, scalable systems

In this report, we consider and discuss the first stage in the net modeling of OLTP (*On-Line Transaction Processing*) applications for parallel computers. The goal was to propose a convenient and flexible model, which would be easy to modify with respect to different scheduling algorithms and the number of processes and disks. The parameters may be varied depending on the size of available memory and timing in a model in order to provide the necessary information to analyze the functioning of the system under different transaction rates and conditions, to identify bottlenecks and potential inefficiencies in the system design.

*Computer Science Department, Stanford University, Palo Alto, CA

Contents

1 Introduction

In this report, we consider and discuss the first stage in the net modeling of OLTP (*On-Line Transaction Processing*) applications on parallel computers. The OLTP workload emphasizes update-intensive database services with up to thousands of (concurrent) transactions per second. High performance, in terms of transactions per second, is the main design issue for the OLTP systems. The final goal of the modeling was to evaluate the ability of different proposed architectures to meet the OLTP benchmark requirements. Such an evaluation should be active, that is, it should provide new architectural versions prompted by the results of the modeling.

Effort spent on the (successful) modeling of such large systems resulted in some methodological conclusions which required us to avoid straightforward approaches to the net modeling and to develop a combined technique of using nets together with embedded programming for simulation and analytical modeling.

An ideal modeling environment should include the following basic capabilities:

- means for adequately detailed and comprehensive (for different stages of the design) *specification* of the system,
- means for *formal analysis* of system properties, and
- *simulation* vehicles for performance evaluation.

We used, as a tool, Design/CPN^{TM} [Jensen87, Pinci-Shapiro91, Shapiro91] to model the *TPC-A Benchmark*^{TM} OLTP workload implemented on a concurrent system. Design/CPN is based on *Hierarchical Colored Petri Nets* and it provides all of the mentioned capabilities. However, it is not a trivial task to directly combine these capabilities in one model for a system of this complexity if we want the model to execute reasonably quickly and supply trustworthy results. Our report [Cherkasova-Kotov-Rokicki92] describes a methodology of net modeling of industrial size applications and shows the most typical methods and approaches to speed up the simulation and optimize the simulation engine of Design/CPN.

In this report, we concentrate on a presentation of the uniprocessor model for OLTP applications, its simulation results and validation, as well as the way this model might be used in broader context and, in particular, how it might be used in a context of scalable concurrent computer system.

The next three sections of this report we present an introduction to on-line transaction processing, the key ideas in scalable concurrent systems, and a summary of the Design/CPN system. This background material prepares the way for the discussion of our model that follows.

2 On-Line Transaction Processing

On-Line Transaction Processing characterizes a category of information systems with

- multiple interactive terminal sessions,
- intensive I/O and storage workload,
- large volumes of data stored in databases,
- a requirement for integrity of *transactions.* A transaction is a well-defined unit of activity that satisfies such properties as atomicity, consistency, durability, and serializability.

TPC Benchmark^{TM} A is standardized as a specification of OLTP workload. TPC Benchmark A workload is simple enough, however, that it is sufficient to demonstrate the problems arising in the net modeling of OLTP applications and the proposed solutions to these problems.

TPC Benchmark A (TPC-A) is stated in a form of a hypothetical *bank* that has multiple *branches* with multiple *tellers* at each branch. The bank has many customers, each of which has an *account.* The database represents the cash position of each entity (branch, teller, and account) in correspondent records and a history of recent transactions run by the bank. The transaction represents the work done when a customer makes a deposit or withdrawal against his account. The transaction is performed by a teller at some branch.

TPC-A performance is measured in terms of *transactions per second* (*tps*). An interesting feature of TPC-A Benchmark is a scaling rule that requires that the number of branches, tellers, and accounts be proportional to the reported transactions per second. The main requirements of TPC-A are the following:

- number of terminals (users): 10 per tps.
- one branch, 10 tellers, 100,000 account and 2,592,000 history records per reported tps,
- response time under 2 seconds for 90% of the transactons, and
- a truncated exponential random distribution of transaction arrivals: 1 transaction per 10 seconds per terminal.

Length specifications for records are the following: branch, teller and account records must be at least 100 bytes long, history records are 50 bytes long.

The *database logs* store data on transactions performed in order to allow the system to reconstruct database contents after a crash. In fact, the engineering solution to recover after a failure of any single system component is to use duplexed logging. The size of the log records is not specified by the standard, we use 1000 bytes records [McGrory-Carlton-Askins92], which seems to be reasonable. The system must have a storage to hold 8 hours of actual test system operations (log records) and 90 days of history.

For each transaction, the originating terminal sends data organized as at least four distinct fields, including $Account\_ID$(entifier), $Teller\_ID$, $Branch\_ID$, and (balance update) *Delta*.

Most of the model of the TPC-A benchmark is of the data base and computer system on which the benchmark runs. With multiple concurrent transactions, the system we model generates several different types of concurrent processes. These include:

- top-level application processes,
- processes in the operating system supporting transaction processing
- updating processes in the database storing the accounts tables
- processes supporting I/O and communication activities (which are substantial in this application)

To provide transaction integrity,the system uses locks on the branch, teller, and account records, so that multiple transactions cannot access the same record simultaneously.

After the log information is written, the changes generated by the transaction (or a group of transactions) can be *committed*, and the correspondent records are unlocked.

3 Scalable Systems and Models

Traditional multiprocessor systems are designed to *speed up* the solution of a given problem by breaking it into smaller subproblems to be solved in parallel. For OLTP-like applications with a fixed response time *scaleup* becomes important: If the number of customers and transactions is growing, can a system be scaled up by increasing the number of processes, discs, storage modules, etc., to maintain transaction response time? The system is *scalable* if the ratio of throughput (number of transactions handled per second) to system size (in some appropriate metric) is a constant, for increasing system size and while maintaining response time requirements.

As it is well known, scalability is a major attribute for large scale systems design.

To study scalable systems, we should have scalable models: while the size of the modeled system increases substantially, the size of the model should remain constant as much as possible and modeling time should grow as low as possible. We have here in mind primarily the size of the net specifying the modeled system. We will see later that colored Petri nets are quite scalable with relation to the model size, but they are not well scalable with relation to the modeling time.

4 Colored Petri Nets and Design/CPN

The formalism of Petri nets [Petri62, Peterson81, Reisig85] often is used as a convenient tool to specify the control structure of concurrent systems and processes.

A *Petri net* is a particular kind of directed bipartite graph consisting of two kinds of nodes, called *places* and *transitions* connected with arcs. An initial state of the net is defined by *initial marking* assigning to each place some nonnegative integer. In modeling, using the concept of conditions and events, places represent conditions, and transitions represent events. The presence of a token in a place is interpreted as a condition corresponding to this place is valid. If place has k tokens it might be interpreted as k data items or resources are available.

Graphically, the places are represented by the circles, the transitions – by the boxes or vertical bars, the incidence relation F by the arcs connecting places with transitions and transitions with places and the weight are function defines the multiplicity of arcs.The initial marking is represented by the tokens (dots) inside the circles.

A transition can fire if each its input place has at least one token. A transition fires by removing a token from each input place and by adding a token to each output place.

In colored Petri nets (CPN) [Jensen87], *color* refers to the types of data associated with tokens and is comparable to data types in programming languages. The version of colored Petri nets used in Design/CPN incorporates variables (representing the binding of identifiers to specific colored tokens), arc inscriptions (expressions), and code associated with transitions.

Colored Petri nets become a graphical programming language with rich specification and simulation possibilities. The language through which colored Petri nets specify desired operations in arc expressions and transition codes is ML [Milner-Tofte-Harper90]. This strongly typed functional language provides substantial economy of expressions, is easily extended by the user, and can execute both interpretively and be compiled.

Each net place in a CPN has an associated color set, which constrains the tokens allowed to reside in it. An *arc expression* characterizes the number and color of tokens that may move along the arc. Arc expressions may use functions and variables that describe a scheme for the movement of tokens. A transition is enabled when each of its input places contains the required number of tokens in the right colors. A *guard* may set an additional condition for enabling a transition. After an enabled transition occurs (or *fires*), it removes tokens from its input places and adds tokens to the output places, as specified by the associated arc expressions.

A hierarchical CPN contains a number of hierarchically interrelated subnets called *pages*. In the net, they are represented by hierarchical substitution transitions. A hierarchical transition can be replaced by an associated page, giving a more detailed description of its internal activity or state. The page containing this transition is called a *superpage*, and the refining page is called a *subpage*. The same hierarchical transition may have several different occurences in the net, so the same page may have several instances during the net execution.

Scalability of colored Petri nets can be provided by the use of colored tokens. They provide means to fold multiple transitions that represent identical or similar system components or activities into one transition whose different instances are enabled by tokens assigned different color values. In the similar way, arc expressions resulting in different values during an execution vary the weights of arcs, allowing a single piece of net to treat different but related (by their color set) tokens in different ways. (Some tutorial examples explaining this basic but powerful methodology are found in [Jensen87]).

By parameterizing the size of color sets, one can adjust the same CPN structure to model related systems with different numbers of components. Page instantiation can serve the same purpose.

These few ideas are the foundation of CPN and are powerful enough to develop advanced methodology and techniques to support different stages of distributed system design: specification, rapid prototyping, validation and performance analysis.

5 Simulation Model

The core of our OLTP simulation model consists of 9 pages including:

1) a global declaration node (containing descriptions of types, colors, variables, reference variables, functions, predicates and functions used in the model),
2) an initialization page which allows us to run the model under different parameters,
3) the overall transaction diagram, specifying a logical schema of transaction processing (shown in Figure 1),
4) a record locking and access schema describing the locking and access mechanism for account, teller and branch records (shown in Figure 3),
5) a disk interface page which prepares (augments) disk access requests to be processed, and then collects and distributes ready data,
6) a disk model based on HPUX^{TM} scheduling strategy,
7) a page cleaning model,
8) a group commit and timeout diagram, specifying a strategy for writing log information to disk and commiting the transaction, and
9) a page for collecting results and statistics.

The overall transaction diagram (Figure 1) shows a logical schema of the transaction processing.

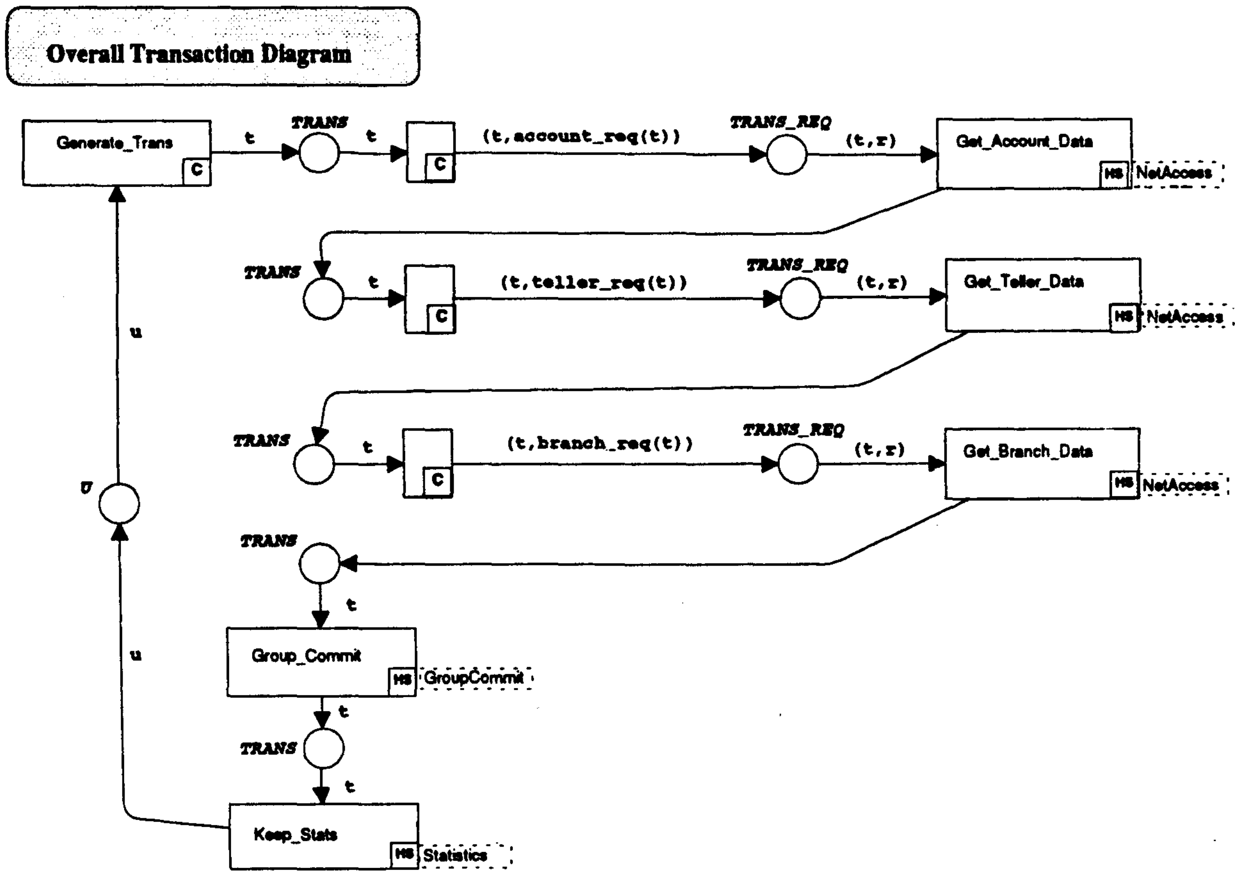

Figure 1: An overall transaction diagram

When a transaction is generated (hierarchical transition *Generate_Trans*) it forms sequentially the following three actions:

- lock and access its account record (hierarchical transition *Get_Account_Data*),
- lock and access its teller record (hierarchical transition *Get_Teller_Data*), and
- lock and access its branch record (hierarchical transition *Get_Branch_Data*).

After this it waits for group commit (hierarchical transition *Group_Commit*). At the end, we collect statistical information (hierarchical transition *Keep_Statistics*).

Here the hierarchical transitions *Get_Account_Data*, *Get_Teller_Data* and *Get_Branch_Data* refer to the same CPN subpage: *RecordLock* (shown in Figure 3), i.e. they represent three different occurrences of the same hierarchical transition *RecordLock*. So this subpage has three different instances during the net execution.

The most complex page, the hierarchical transition *RecordLock* (basic core of which is presented in Figure 2 and a complete model of which is shown in Figure 3), illustrates record access and locking.

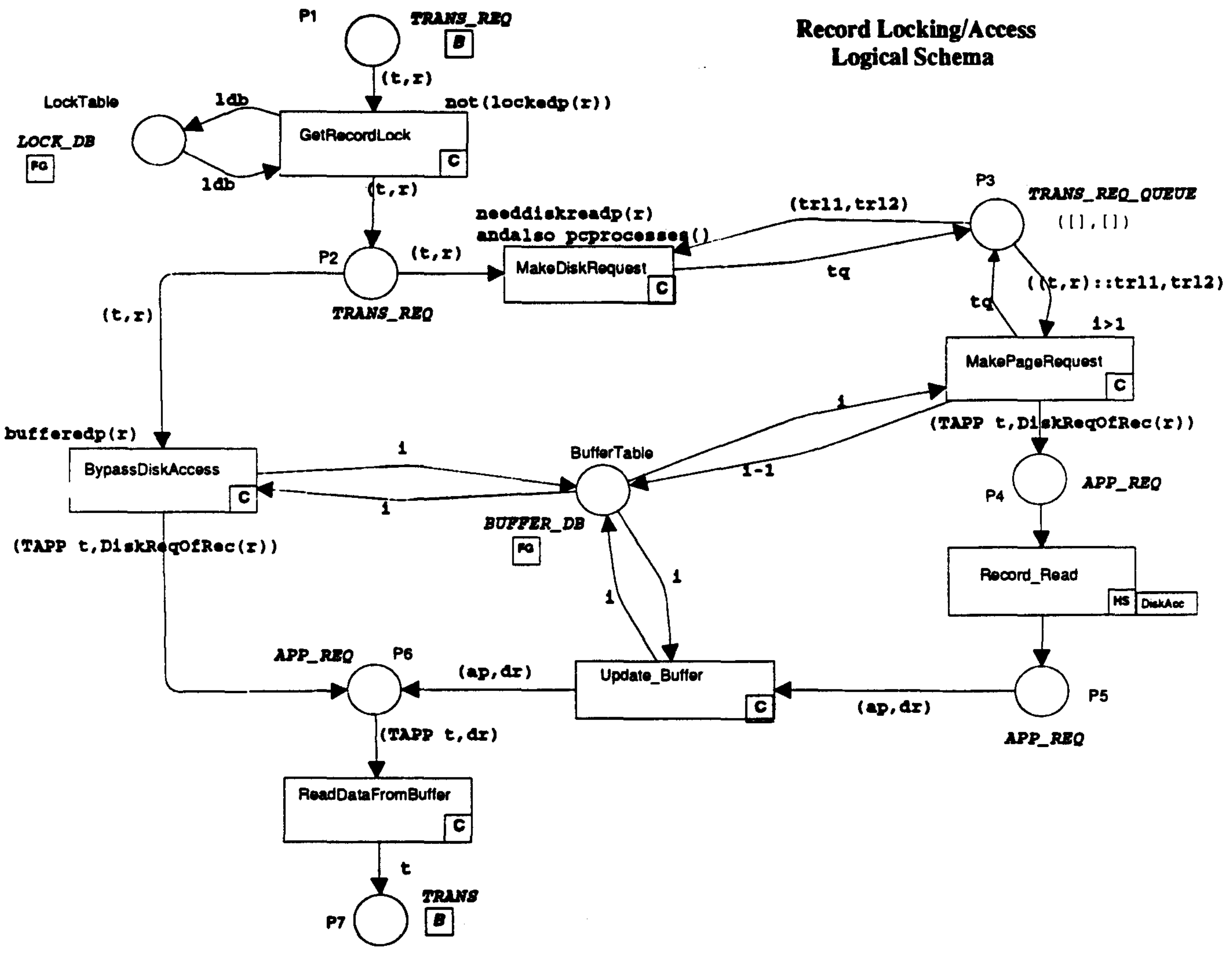

Figure 2: Basic record locking and access schema

At first, a token representing a record request (it will be an account, teller, or branch record request depending which instance of the page it is) arrives in a place *P1*. The transition *GetRecordLock* can fire if its guard is satisfied, i.e. the correspondent record is not locked. When this transition fires, the locking of the data record is modeled by the addition of a corresponding element to the hash table underlying the single token in the place *LockTable*. The place *LockTable* is announced as a *fusion* place that plays a role of a shared resource:

there is only one instance of this place shared between the transitions *Get_Account_Data*, *Get_Teller_Data* and *Get_Branch_Data*, that simulates a mutual exclusive access to a lock table.

Then a token representing a record request proceeds to the place $P2$. If the record is already in memory (i.e. record already has been buffered) then the transition *BypassDiskAccess* will fire, otherwise the transition *MakeDiskRequest* will be chosen. New disk requests are added to a token in a place $P3$ in a special manner: this token implements a FIFO queue. Because of the potentially large number of tokens in this queue, we are again faced with the importance of efficient implementation because a more straightforward approach will dramatically slow down the simulation.

Before a record is read from disk, a free memory page must be reserved for this data by the firing of transition *MakePageRequest.* If there are no available free memory pages then disk requests in place $P3$ are delayed waiting on the page cleaning processes to free a page. Page cleaning is represented in a model by an independent concurrent subnet related to the hierarchical transition *RecordLock* via a fusion place *BufferTable.* A marking in the place *BufferTable* reflects the number of free memory pages and has an underlying hash table relating memory pages to buffered records.

When a memory page is reserved, a token with a corresponding disk request proceeds to a place $P4$ which is an input port of the hierarchical transition *Record_Read.* The transition refers to a CPN subpage *DiskAccess* describing and modeling disk behaviour.

After a record is found and read from the disk (at which point the token will be in place $P5$), the fact that the data is now available in memory is added to the lock database by transition *UpdateBuffer.* It then proceeds to place $P6$, which indicates that the data is buffered. The last stage consists in reading a record from the memory. This stage is represented by firing a transition *ReadDataFromBuffer.*

Figure 2 presents rather a logical model of record locking and access in order easier to explain the logical schema of it. The complete model is shown in Figure 3. New details added in this schema is CPU conflict: because this is a uniprocessor model all the operations requiring CPU time compete for the CPU resource. In order to model this situation, a place with one token (representing CPU) should be added, and all the transitions representing the operations which require CPU time should be connected to this place. To make the picture less messy, we used as such a place a fusion place *BufferTable*, which is already a shared resource for a subset of transitions conflicting as for data and pages in memory as for CPU time. The transitions consuming CPU time have now a correspondent time delays (in CPN Design terms: a correspondent time region): for how long they occupy CPU resource. In graphic notation, the time delay associated with transition is written “inside” the transition after the sign @. Thus for example, the transition *GetRecordLock* has a time delay specified by @ $+CPUlock()$ where the function $CPUlock()$ defines how long it takes for CPU to get a record lock. These time delays as well as some other parameters are input parameters for the model and may be changed depending on concrete system we would like to simulate.

The hierarchical transition *InLineCleaning* referring to a page cleaning model (shortly described below) is included here in order to experiment with the performance evaluation of different page cleaning strategies. Thus, this new hierarchical transition *InLineCleaning* represents the strategy chosen in AllBase where the transaction itself is responsible for finding and cleaning a dirty page.

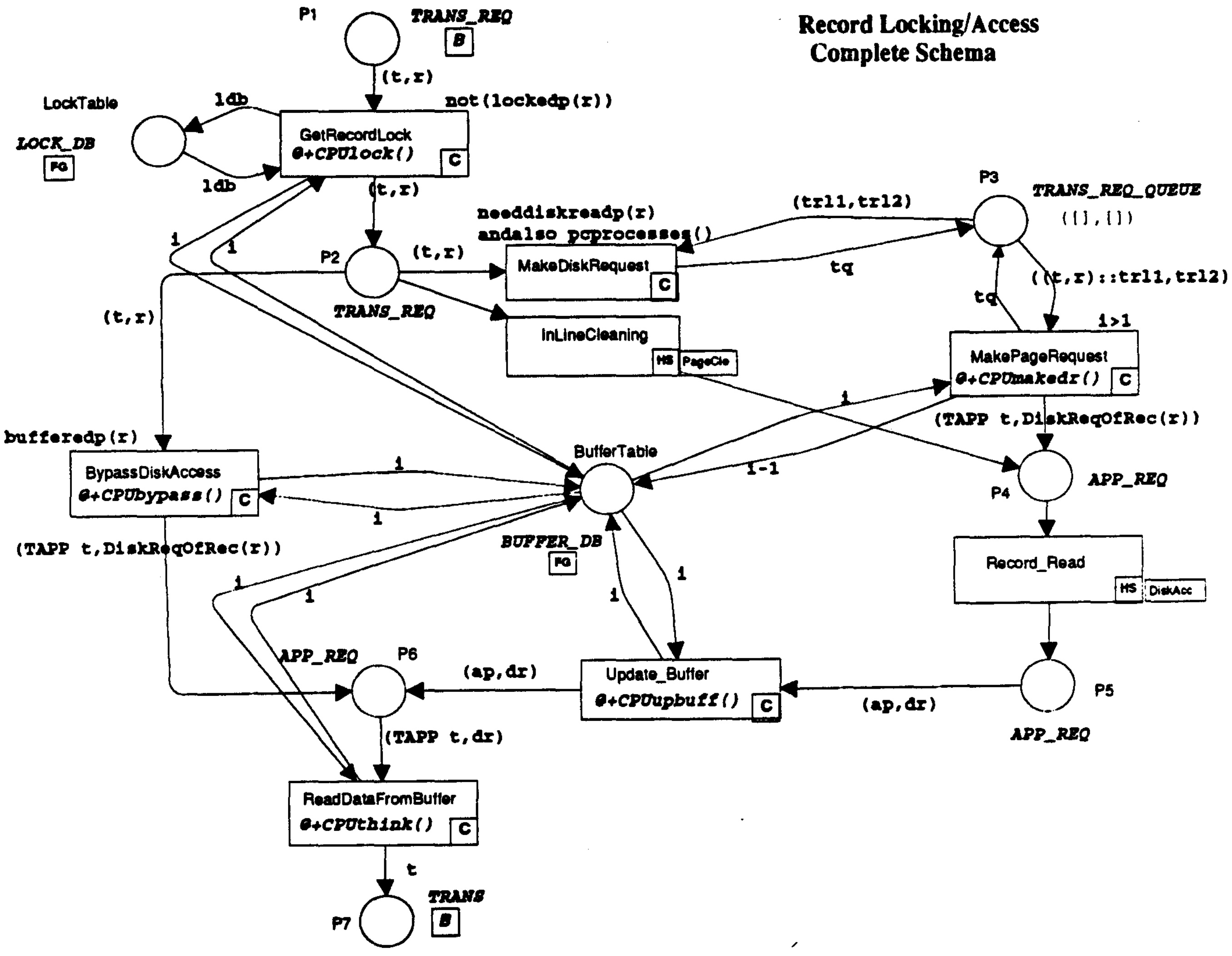

Figure 3: Complete record locking and access schema

A disk model shown in Figure 4 is rather simple: the requests arrive in a place $Q1$. The requests are sent from the different model parts: it might be a request for branch, teller or account record, or it might be writing to log, or writing a dirty page (by page cleaner). In order to know where to return the completed request, all five types of the requests have a different individual ticket-identifier.

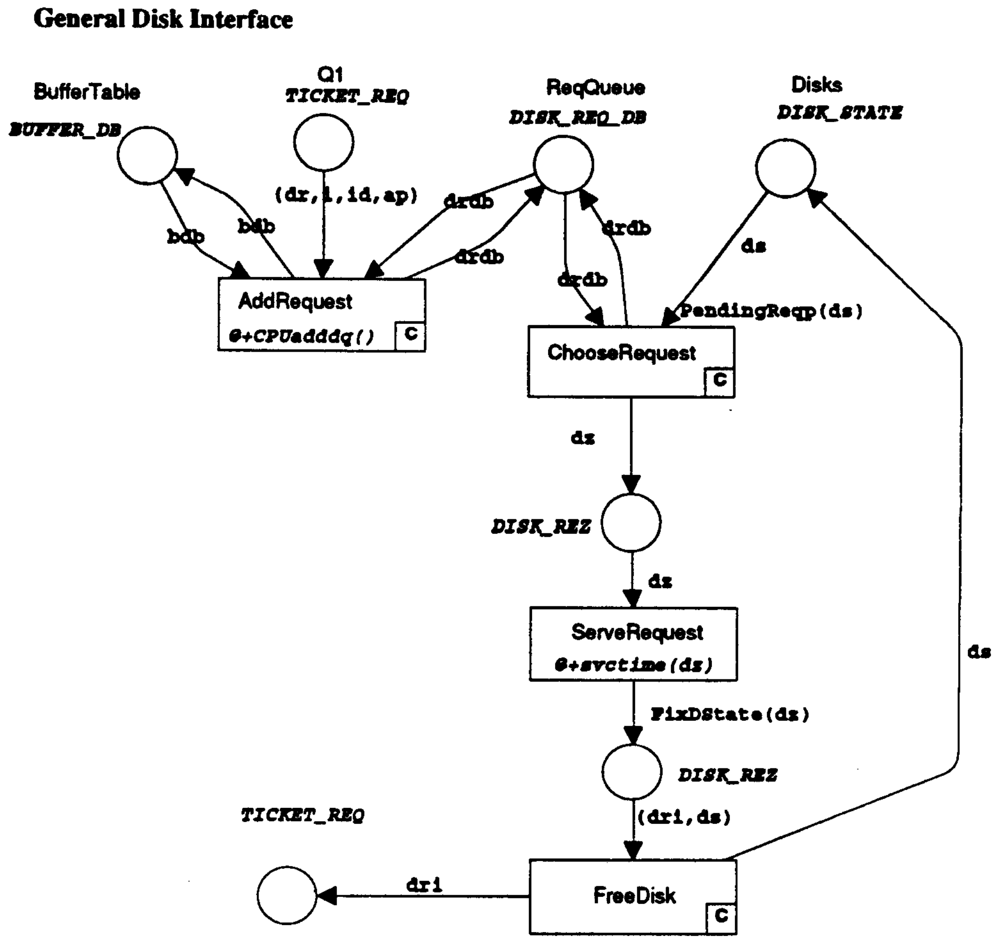

Figure 4: A Disk Model

The firing of transition *AddRequest* puts the request in a queue (a place *ReqQueue*). A place *Disks* has so many tokens as disks available in a system, and disk state is represented by a pair: disk number and its head position. How disks requests are organized in a queue and how they are chosen from this queue might be programed differently depending on the concrete implementation of the system. We used here HPUX^{TM} scheduling strategy (i.e. ordering the request queue in the same way as the correspondent data are ordered on a disk. Thus in such a way, the latest request might be satisfied earlier if its data is located closer to a disk head). However, this strategy might be easily changed to some other different one without changing a structure of the model: only the function defining the scheduling strategy need to be replaced. This is one of the attractive features working with Design/CPN: it has enough flexibility in changing interpretations.

The model of page cleaning, shown in Figure 5, has a similar general structure.

This part of the model works concurrently with the rest of the model: transition *StartCleaning* searches with some available process from a *CleaningProcesses* place over the *BufferTable* representing a memory. After it finds a dirty page which is not in use anymore, it sends this page to a place *S*1. Place *S*1 is an input place of a hierarchical transition *WriteDirty* that

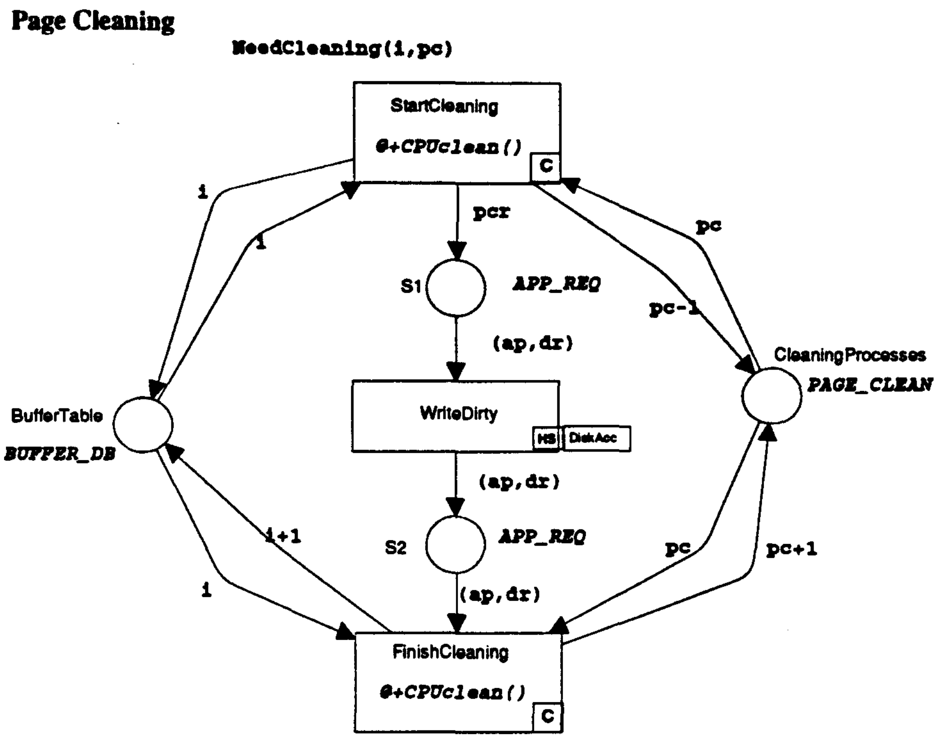

Figure 5: Page Cleaning Model

refers *DiskAccess* page. In such a way, dirty page is queueed to be written to a disk in a manner we described above. When a dirty page is written to a disk, a token is sent to place *S*2. A transition *FinishCleaning* increases the number of available (clean) pages in memory and returns a process back to *CleaningProcesses*. Again, the structure of this part is rather general, the interpretation is hidden inside the code attached to the transitions. It allows to have a set of different models using different, say, page cleaning strategies as well as number of page cleaning processes, and to make a tuning of the model finding the best appropriate parameters and strategies.

In the first version of the OLTP model we used for a page cleaning strategy a dedicated cleaning processes pool working according to the "second chance" algorithm. It concludes in the following: there are certain (fixed) number of page cleaning processes. Whenever the number of free memory pages becomes less than the given minimum (another parameter to tune) all the page cleaning processes start to work. Each memory page has a set of attributes: *busy*, *used*, *dirty* . The *busy* bit shows how many users (transactions) are using this page. The *used* bit reflects if the page is (or was) in use. The *dirty* bit is established whenever the page content was changes but was not written to a disk. Active page cleaning process goes through the memory pages and whenever it finds the page nobody is using (i.e. its *busy* bit is equal to *zero*, *used* bit is *true*, and the *dirty* is also *true*), the page cleaning process changes *used* bit to *false*, and goes to check next page. Whenever it finds the memory page nobody is using and with *used* bit set to *false*, the page cleaning process sends such a page to clean, and becomes passive while this page was written to disk. Thus the pages to clean are selected from those that have not been recently accessed.

While we debugged the model, we noticed that created OLTP model is sensitive to page cleaning parameters. In the first version of the model, we ignored the TPC-A requirements for history records. Thus we used 4 disks (426 megabytes each) which would be only enough to hold transactions records and 3 disks for logrecords. Thus the access to disks under such parameters had higher latency and certainly has been a restrictive factor in system performance. The OLTP model for 110 tps behaved very differently depending on a number of page cleaning processes. Thus for a number of page cleaning processes less than 20, the latency will increase without bound, and the model will not reach a steady state. The histogram of the latency distribution for 110 tps and 16 page cleaning processes is shown in Figure 6.

Only starting from 20 page cleaning processes the OLTP model for 110 tps after certain period will stabilize (see Figure 7), and for a number of page cleaning processes more than 20 it will show stable behaviour from the beginning.

After that we explored three different strategies for OLTP model:

- AllBase strategy (where the transaction selects and cleans dirty page itself),
- dedicated cleaning processes pool (where we have a fixed number of synchronously working page cleaning processes) and,
- thread strategy (where a page cleaning thread is generated by a transaction whenever a clean page is required, and at the same time, a transaction is continuing its computation without waiting or being delayed for the time this thread cleans a dirty page).

Simulation has shown that the last strategy is the most efficient. The second strategy is very sensitive to a number of cleaning processes, and sometimes it may cause unstable behaviour. AllBase strategy is less efficient, because the page cleaning time is included in transaction path length, while in the last two cases, it is independent activity.

However, the situation and the conclusion about efficiency of one or another strategy is very much dependable on a whole set of system parameters: size of the memory, number of disks, path length and CPU capacity.

The simulation model briefly presented in this section contains a fair amount of programming. The global declaration node specifying types, colors, variables, reference variables, functions and predicates used in the model has 1000 lines of ML code. However, the simulation model we built still is in many ways universal and flexible to use and modify. One of the crucial points in using Design/CPN efficiently is good partitioning and abstraction of the model: what should be represented directly via nets and what should be additionally programmed. For example, in our disk model, we represent the basic features of a disk with nets, but describe

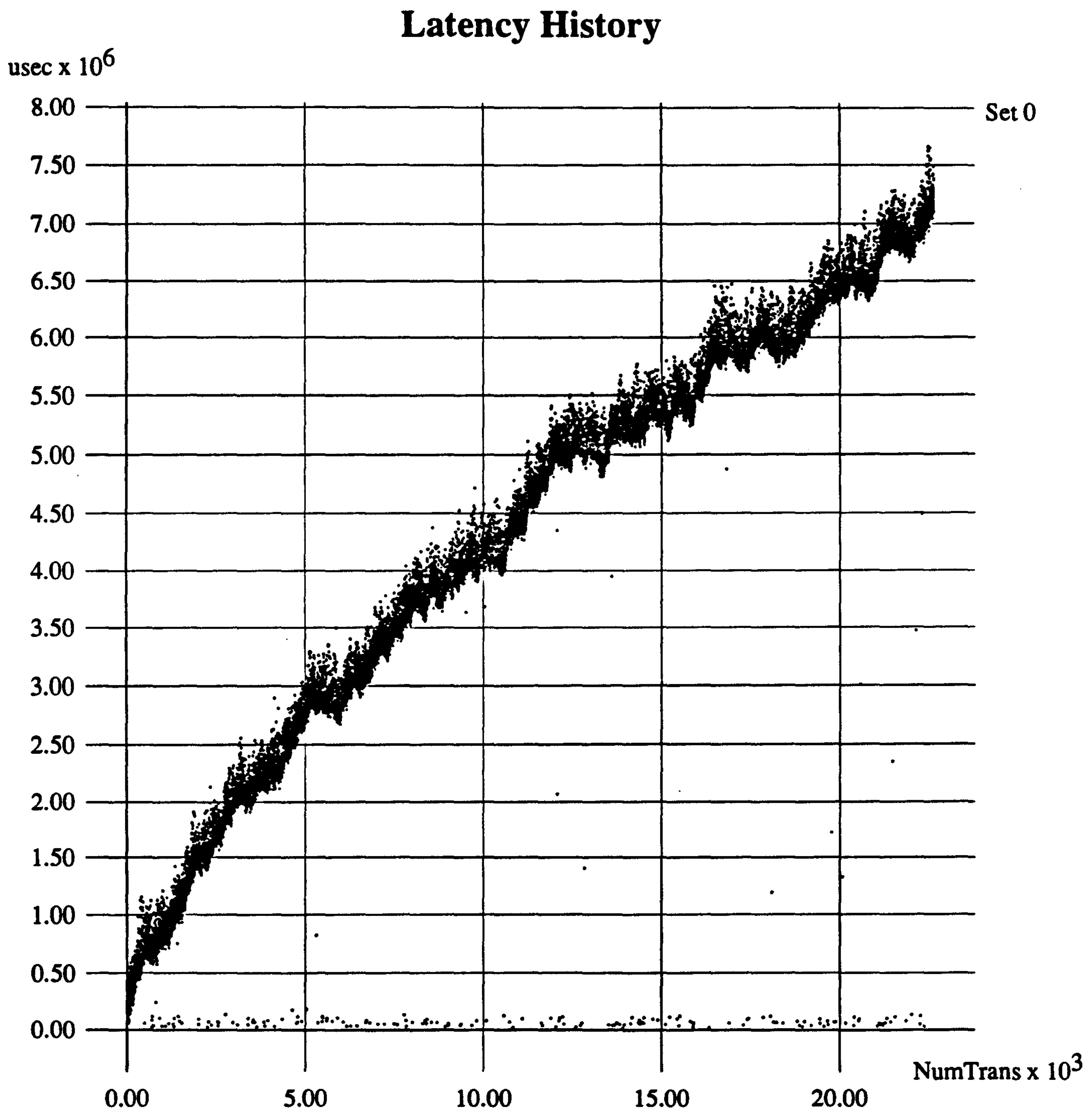

Figure 6: Latency history for 110 tps and 16 page cleaning processes (with 7 disks of storage)

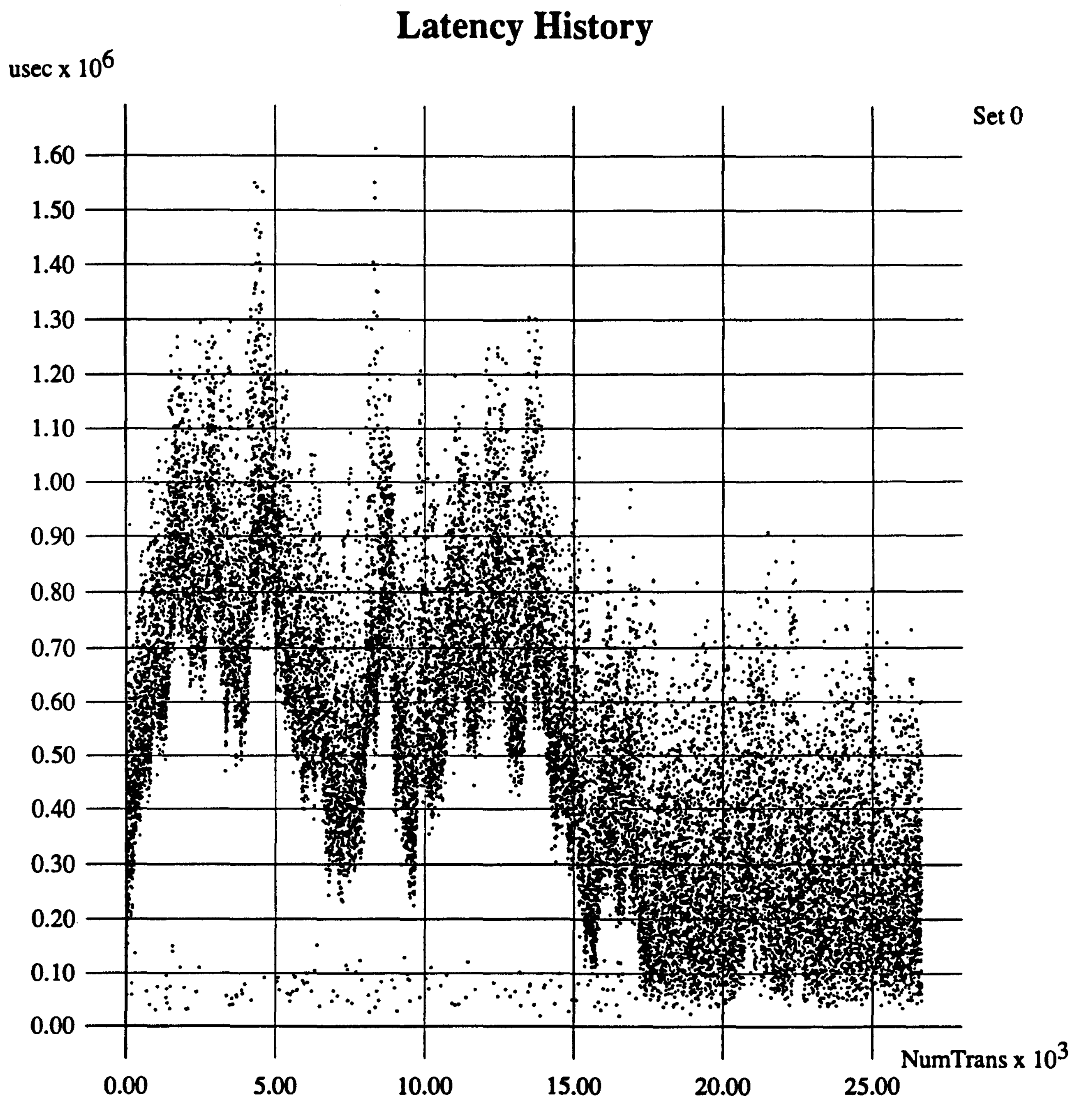

Figure 7: Latency history for 110 tps and 20 page cleaning processes (with 7 disks of storage)

request scheduling with a program. In a case we would like to have a model of a disk with different scheduling strategy we would leave our net model unchanged and only rewrite the scheduling function.

The goal of the OLTP simulation model was to develop a convenient and flexible model, which will be easy to modify with respect to different scheduling algorithms, number of processes and disks, the size of available memory and timing in order to provide the necessary information to analyze the functioning of the system under different transaction rates and conditions, to identify bottlenecks and potential inefficiencies in the system design, and to help guide the design process. In order to accomplish this goal, we collect for each transaction excution detailed history information. This information includes, for example, how long each transaction spent obtaining each record lock, how long it waited for a disk or buffer read, how long it took to group commit, etc. We also collected history information on disk requests, including how long each request spent in the disk queue and how long it took for the physical disk to satisfy the request. We developed a few simple tools to analyze files containing these history traces, perform some simple statistical analysis, and illustrate the results graphically.

6 Simulation Results

The simulation results can be projected showing the latencies sorted in order of increasing length (see Figure 8), or the dynamic history of the latencies (see Figure 9), or plotting a histogram of the latency distribution (see Figure 10).

These three graphs demonstrate the execution of an OLTP application on a uniprocessor system satisfying TPC-A requirements.

We validated a model against the real system using a path length of 142,000 instructions for HP 3000/977 running MPE/XL.

The Figures 8, 9, 10 show simulation results of the model under the rate of 111 transactions per second. In this model, as a page cleaning strategy was used AllBase strategy.

CPU utilization result is 93%. The total run consists of 13000 transactions, and it required overnight simulation (on a 16M 68030-based Unix workstation with HPIB disks.)

The simulation results are consistent with the results of the analytical modeling of [Jacobson92], and coinside with the performance results of the real system (HP 3000/977 running MPE/XL could run 111 tps with CPU utilization of 93%).

The Figures 11, 12, 13 show simulation results of the model under the rate of 115 transactions per second with CPU utilization result is 96.5%.

And the Figures 14, 15, 16 show simulation results of the model under the rate of 118 transactions per second with CPU utilization result is 99.2% which is right at its peak performance while still satisfying the latency requirements.

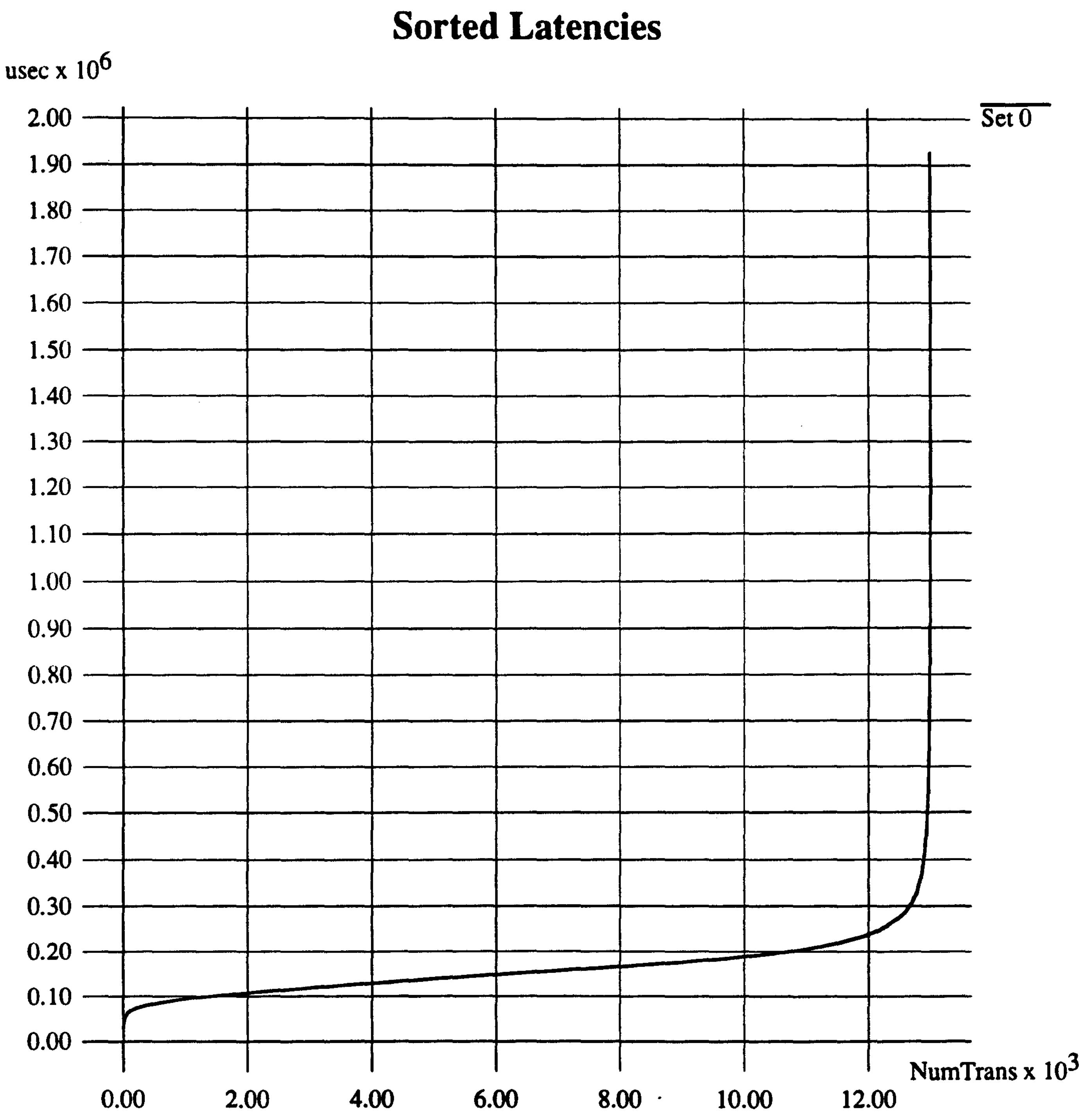

Figure 8: Sorted latencies (111 tps)

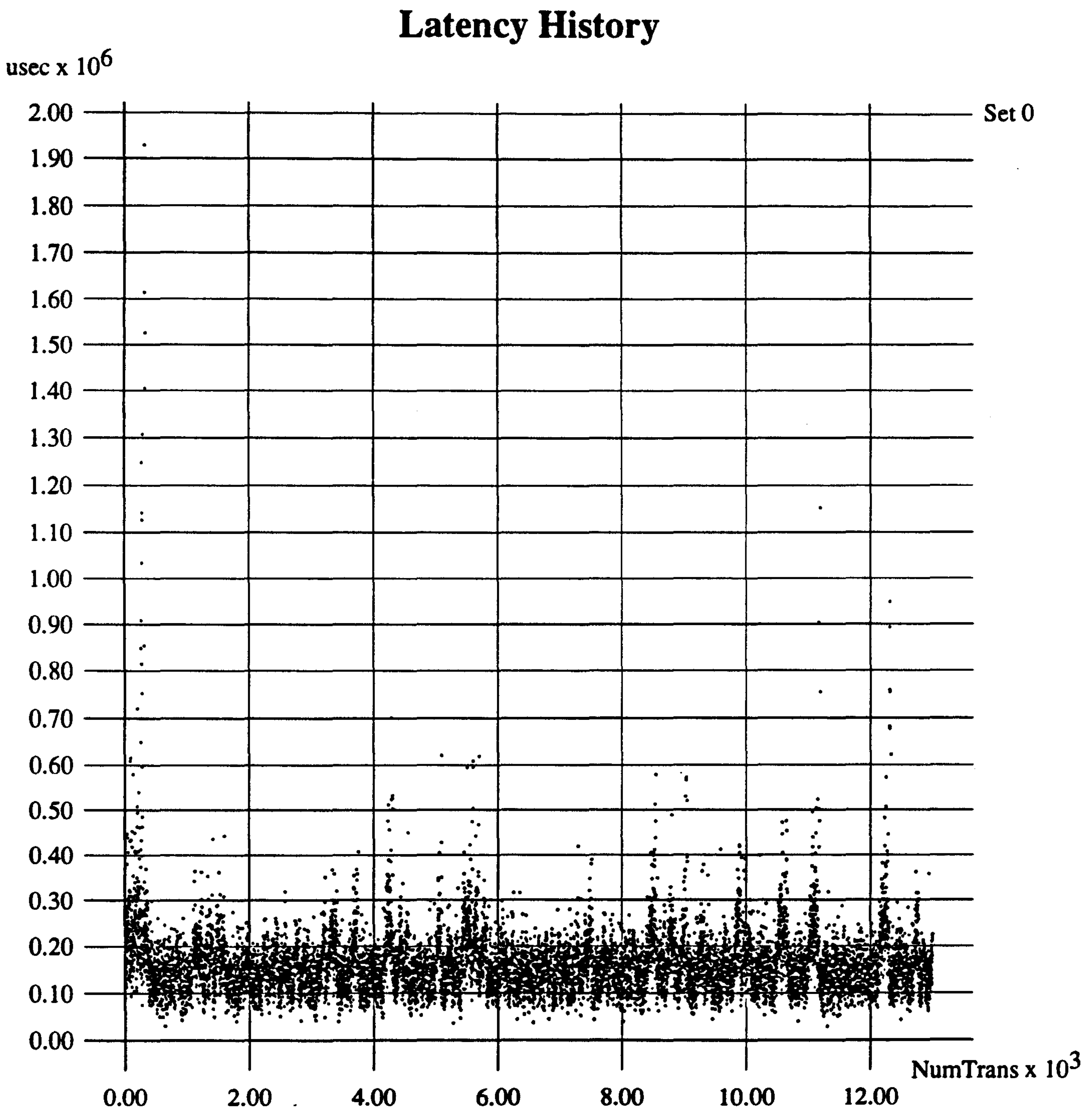

Figure 9: Latency history (111 tps)

Latency Distribution

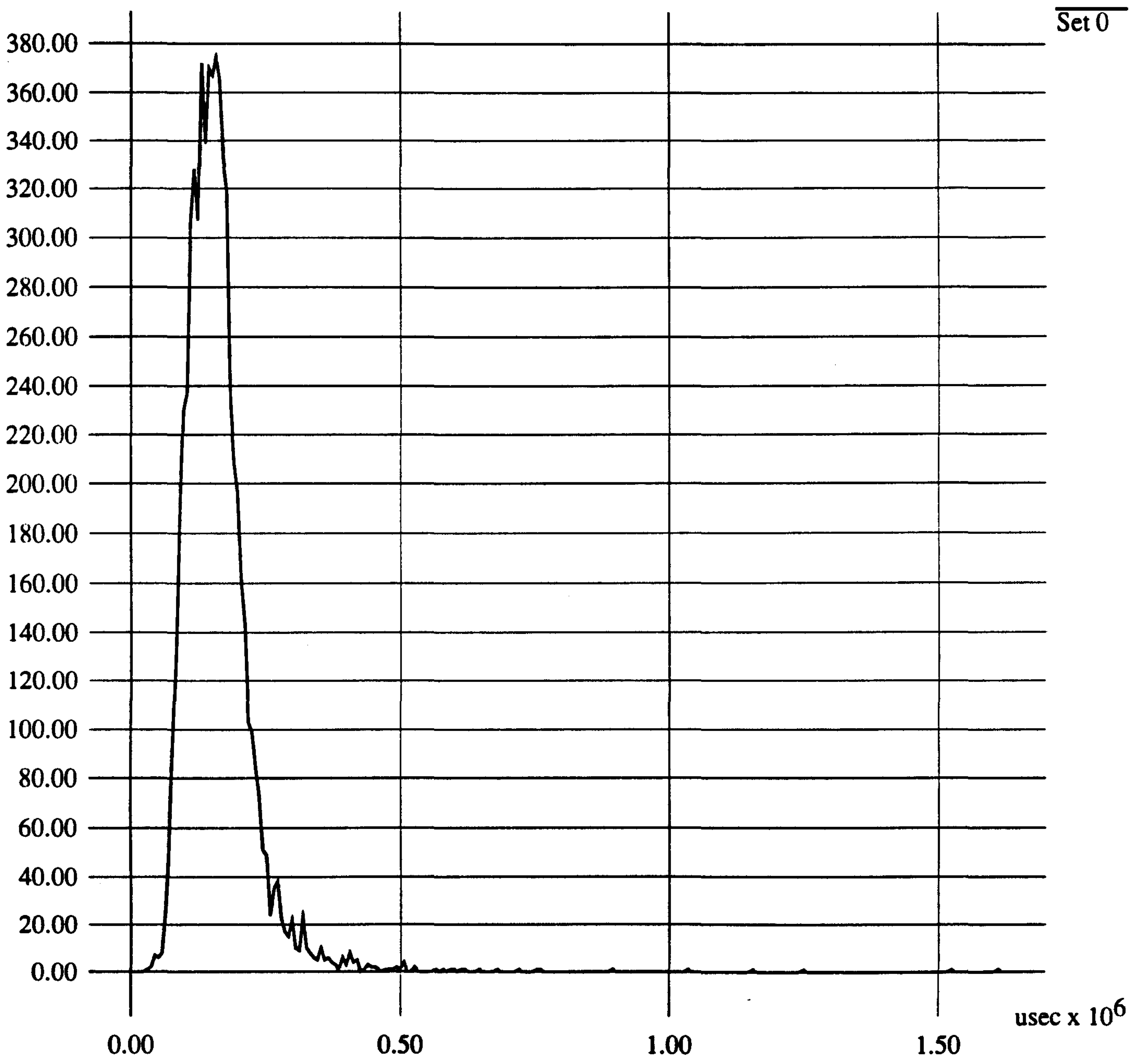

Figure 10: Latency distribution (111 tps)

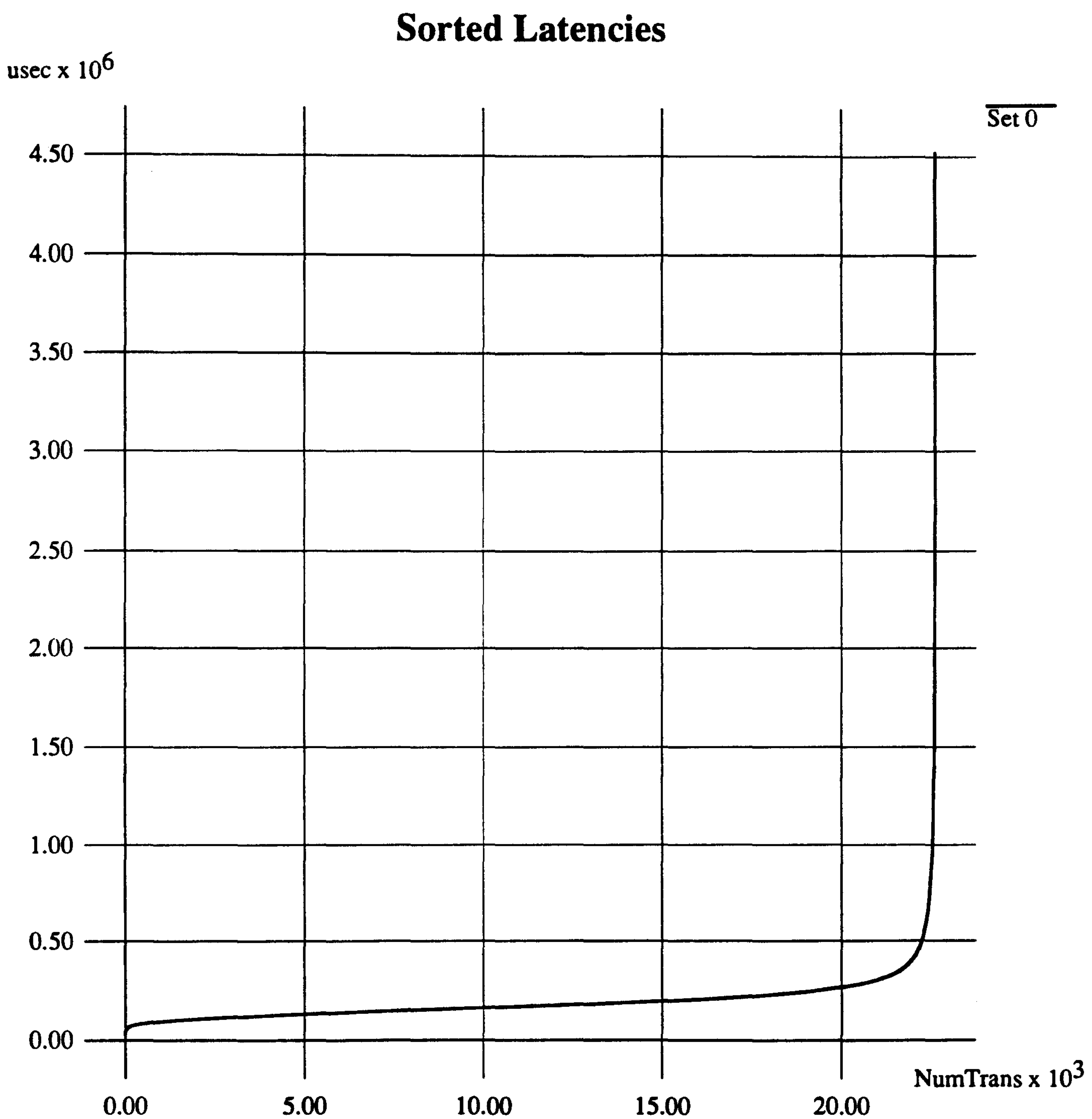

Figure 11: Sorted latencies (115 tps)

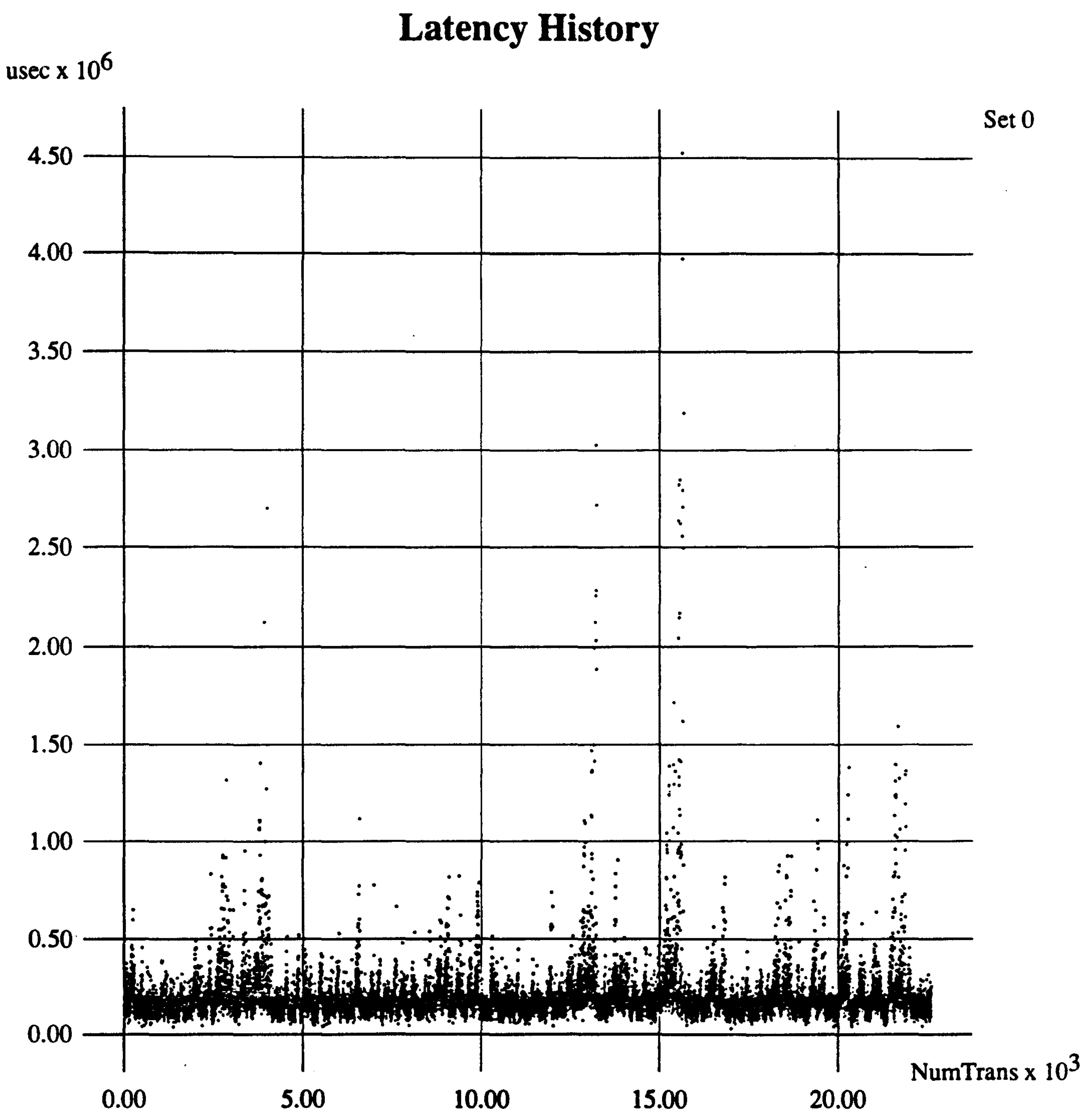

Figure 12: Latency history (115 tps)

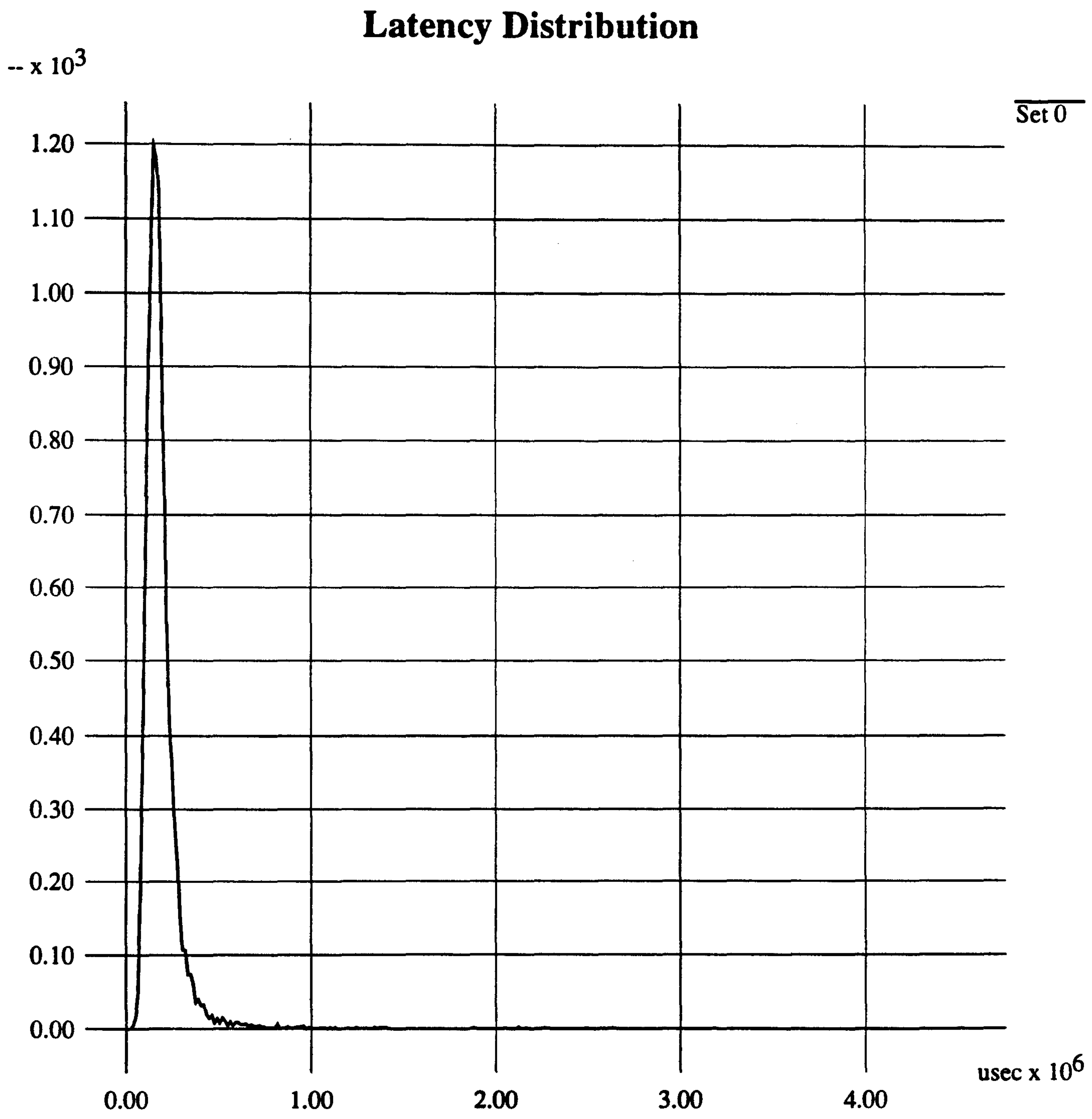

Figure 13: Latency distribution (115 tps)

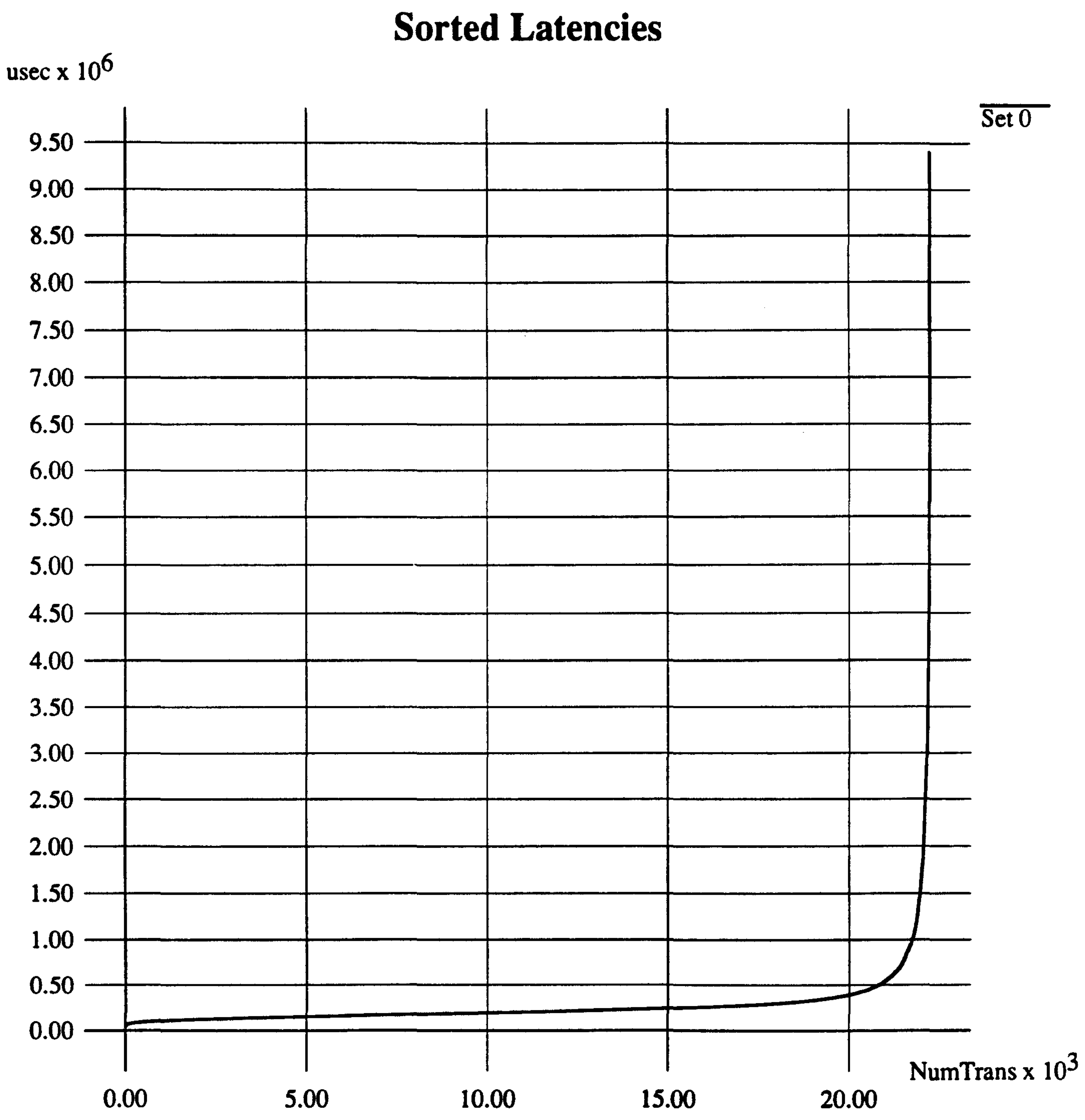

Figure 14: Sorted latencies (118 tps)

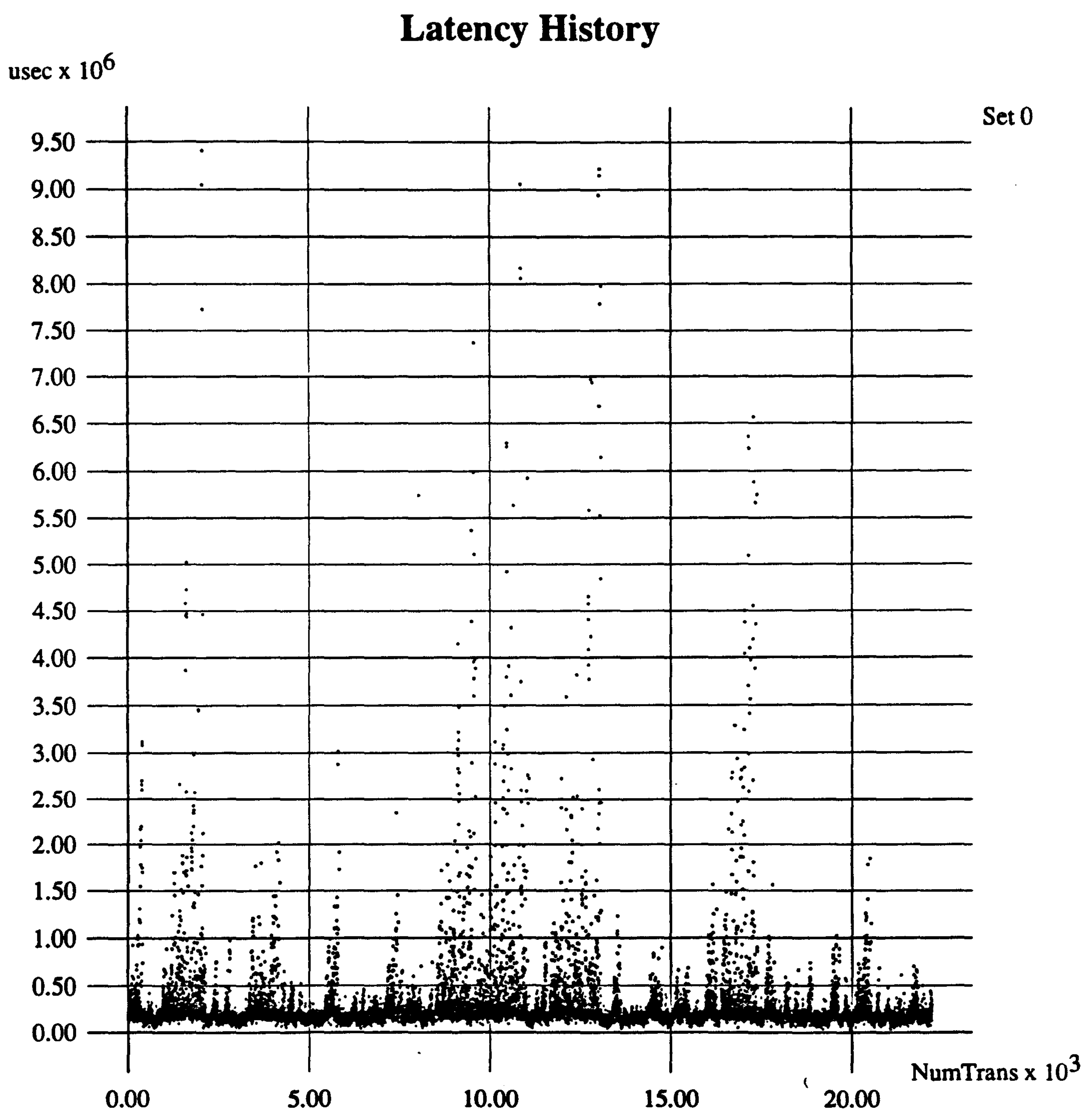

Figure 15: Latency history (118 tps)

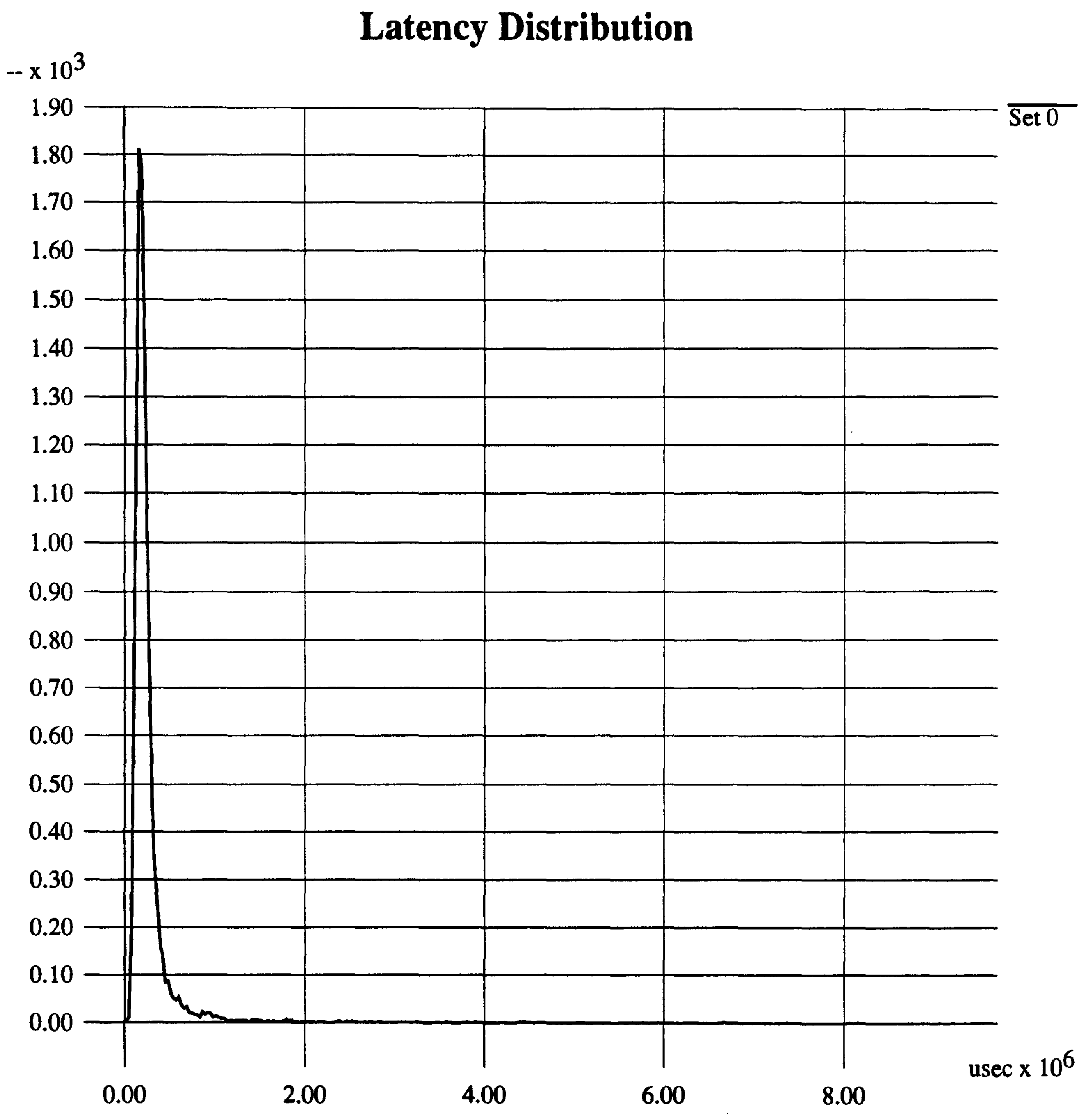

Figure 16: Latency distribution (118 tps)

Figure 17 shows additional statistics about the sorted latencies for 10%, 50%, 90% of transactions as well as minimum, maximum and average latencies for 111 tps (Figure 17a), 115 tps (Figure 17b), and 118 tps (Figure 17c), latencies are given in microseconds.

As one can see, for the cases of 111 tps and 115 tps (CPU utilization: 93% and 96.5% correspondingly) the latency increases insignificantly. However, the closer to the upper limit of the CPU utilization the latency changes more drastically. Thus for 118 tps (CPU utilization is 99.2%) the latency increases 1.5 times comparing with 111 tps and 115 tps.

a) 12997 data points

Minimum	10%	50%	90%	Maximum	Average
29,666	98,327	152,378	225,627	1,928,328	161,079

b) 22614 data points

Minimum	10%	50%	90%	Maximum	Average
29,380	109,678	170,662	275,527	4,524,631	192,291

c) 22120 data points

Minimum	10%	50%	90%	Maximum	Average
34,977	123,074	199,583	392,831	9,407,719	274,698

Figure 17: Statistical analysis of sorted latencies a) for 111 tps, b) for 115 tps, and c) for 118 tps

The simulation results support the following conclusion: TPC-A throughput is approximately the ratio of the processor speed to the average path length. This conclusion is consistent with the results of [Jacobson92] where the analytical model for TPC-A was developped and the performance analysis of TPC-A has been done. [Jacobson92] also concludes that the CPU is certainly such a bottleneck that everything else has an insignificant influence on a performance.

7 Refinement

One of the crucial factors in efficiently using Design/CPN, as we mentioned earlier, is the correct partition of the model into the net part and the programming part. Another important goal is to use CPN hierarchy efficiently. The hierarchical pages might be considered as submodels composing the whole model. In the ideal case, different subpages (there might be several in one group) represent different parts of the designed system. Whenever we need to change or substitute some part with a new or different one, it would be convenient to substitute or redesign a correspondent hierarchical page instead of changing the whole model.

We follow this methodology for modeling OLTP applications on multiprocessor systems by extending our OLTP model for a uniprocessor. The difference in the model is that a transaction during its processing might require a data from different computer nodes. These nodes are connected to each other via a specially designed interconnect. Thus we introduce one more level of hierarchy in our overall transaction diagram (see Figure 1) where the hierarchical transitions *Get_Account_Data*, *Get_Teller_Data* and *Get_Branch_Data* refer to a new CPN subpage: *NetAccess*. A new hierarchical page *NetAccess* is shown in Figure 18.

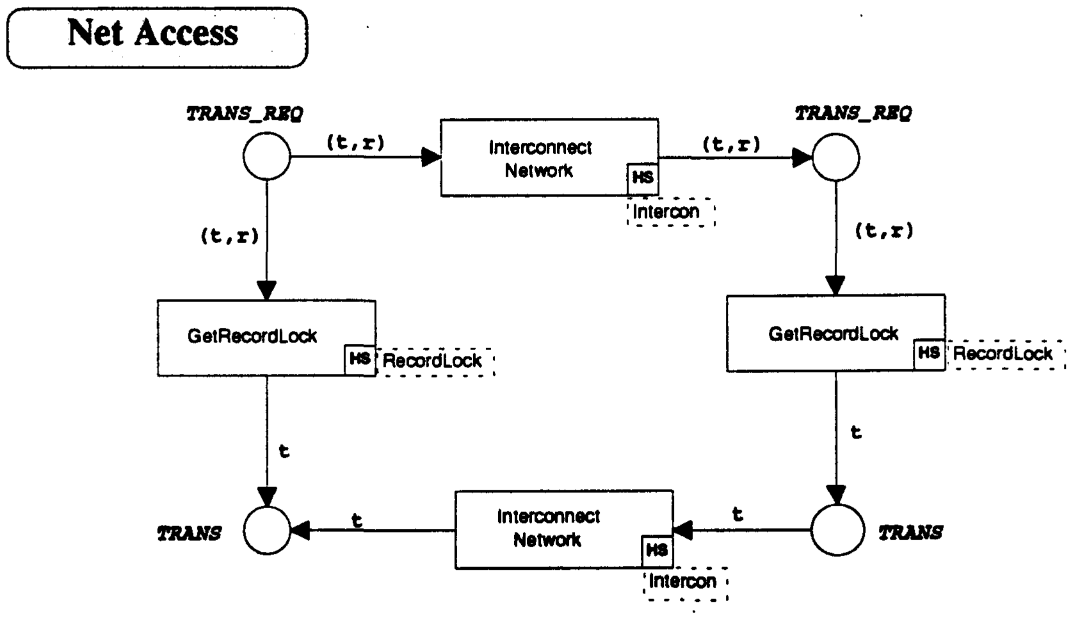

Figure 18: A net access page

The purpose of this additional page is that in case a transaction requires data from a different computer node, a request and a reply containing the data should be transmitted through the interconnect. Here is another choice to either model the internal structure of the interconnect in detail (for this purposes it is represented in Figure 18 as a hierarchical transition) or to represent it as a simple transition with an average time delay. Adding one more level of hierarchy in modeling the interconnect net gives us the ability to identify its properties and potential bottlenecks under the specific OLTP workload. We can also experiment with it, changing the parameters to improve its performance.

Another page that requires a modification is the group commit page, because the group commit protocol is different for a multiprocessor systems.

8 Conclusion

We have presented our experience in modeling an on-line transaction processing system using colored Petri nets and the Design/CPN tool. The developed model might be used for simulating the systems with different timing and size parameters, the scheduling strategies might be modified and incorporated too.

The model with parameters satisfying the TPC-A benchmark was validated against the real sytem (HP 3000/977 running MPE/XL) with the same parameters. The simulation results correspond to the real data within 5%.

The simulation results support the following conclusion: TPC-A throughput is approximately the ratio of processor speed to average path lengh (it is consistent with the results of the analytical model in [Jacobson92]). In current systems, the CPU is certainly such a bottleneck that everything else has an insignificant influence on performance. As we explore scalable systems, however, other bottlenecks will arise, and our simulation model will become more important in predicting and measuring such bottlenecks. Even with conventional multiprocessor hardware, faster CPUs and larger disks (and thus fewer spindles) will, at some point, shift the bottleneck from the CPU cycles available to the disk system throughput.

Acknowledgements

Many thanks are due to Bill Worley for the constant support and active participation during the design stage of the model.
We also thank John McGrory who provided us with helpful insight into OLTP concepts and David Jacobson for his exceptional help in getting path length data from Cupertino for validation of the model and fruitful discussions.
Special thanks are due to Mike Lemon who consulted us on graphic output for simulation results and actively participated at all the stages of Design/CPN installation.

References

[Baldassari-Bruno91] Baldassari M., Bruno G. PROTOB: An Object-Oriented Methodology for Developing Discrete Event Dynamic Systems. In *High-Level Petri Nets*, Springer Verlag, 1991, pp. 624-648.

[Cherkasova-Kotov-Rokicki92] Cherkasova L., Kotov V., Rokicki T. On Net Modeling of Industrial Size Concurrent Systems. HP Laboratories Report No. HPL-92-154, December, 1992.

[Genrich87] Genrich H.J. Predicate/Transition Nets. In *Petri Nets: Central Models and Their Properties*, Springer Verlag, LNCS 254, 1987, pp. 207-247.

[Gray91] The Benchmark Handbook for Database and Transaction Processing Systems. Edited by Jim Gray, Morgan Kaufmann Publishers Inc., San Mateo, California, 1991.

[Jacobson92] A Performance Analysis of the TPC-A Benchmark. HPL-OSR-91-2, October, 1992.

[Jensen87] Jensen K. Coloured Petri Nets. In *Petri Nets: Central Models and Their Properties*, Springer Verlag, LNCS, vol. 254, 1987, pp. 248-299.

[Jensen87] Jensen K. Coloured Petri Nets: A High Level Language for System Design and Analysis. In *Advances in Petri Nets 1990*, Springer Verlag, LNCS, vol.483, 1991, pp. 342-416.

[Kohler-Raab92] Kohler W., Raab F. TPCV Benchmark^{TM} C: The order-Entry Benchmark.

[McGrory-Carlton-Askins92] McGrory II J.J., Carlton A., Askins B.J. Transaction Processing Performance on PA-RISC Commercial Unix Systems. In *Proceedings of the IEEE COMPCON'92* , 1992.

[Pinci-Shapiro91] Pinci V., Shapiro R.M. An Integrated Software Development Methodology Based on Hierarchical Colored Petri Nets. In *High-Level Petri Nets*, Springer Verlag, 1991, pp. 649-666.

[Shapiro91] Shapiro R.M. Validation of a VLSI Chip Using Hierarchical Colored Petri Nets. In *High-Level Petri Nets*, Springer Verlag, 1991, pp. 667-690.

[Milner-Tofte-Harper90] Milner R., Tofte M., Harper R. The Definition of Standard ML. MIT Press, 1990.