Revisiting the Issue of Performance Enhancement of Discrete Event Simulation Software *

Alex Bahouth, Steven Crites, Norman Matloff and Todd Williamson
Department of Computer Science
University of California at Davis
Davis, CA 95616 USA
matloff@cs.ucdavis.edu

Abstract

New approaches are considered for performance enhancement of discrete-event simulation software. Instead of taking a purely algorithmic analysis view, we supplement algorithmic considerations with focus on system factors such as compiler/interpreter efficiency, hybrid interpreted/compiled code, virtual and cache memory issues, and so on. The work here consists of a case study of the SimPy language, in which we achieve significant speedups by addressing these factors.

1 Introduction

As processing speeds increase, the perceived feasible size of discrete event simulations becomes larger. Indeed, the latter seems to be growing faster than the former. Thus performance of simulation software, say in library form (which for convenience we assume here), is a key issue.

Performance is an even larger issue in that it is now common for simulation analysts to place a premium on programming in languages they consider to be clearer, safer or to have shorter development time. As these typically are interpreted languages, such as Java and Python, a price is paid in terms of execution speed.

There is a vast literature on the general performance issue, most of it dealing with algorithm/data structure-theoretic issues [10]. The typical analysis has been on operation counts, or on timing synthetic versions of the algorithms. The latter "simulations of simulations" are definitely important, and will play a role in our work here, but we also give major importance to aspects which have typically not received much consideration, if any:

- *Compiler/interpreter issues:* The focus here is on the executable code. For example, an innocuous looking array in Python, used for the event set in the SimPy simulation language [12] [4], can actually be the source of much inefficiency, resulting in significant deterioration of performance.

- *Overcoming the slowness of interpreted languages:* Interpreted code tends to be considerably slower than compiled code. The gap has been narrowing, due for example to the use of Just in Time (JIT) compilers [9]. However, much of the benefit of JIT stems from avoidance or reduction of program load time, with there being little gain from JIT in the case of long-running programs.

- *Memory hierarchy issues:* On modern machines, memory access considerations can be just as important as algorithmic ones. A cache miss, for instance, is a major disruption to a program, with line replacement causing a significant delay. A virtual memory page fault is even more damaging to performance, as the mechanical nature of disk read/write implies times on the scale of milliseconds rather than the nanosecond scale of CPUs.

  The aspect of this which is most relevant to simulation is event set processing. Though not in the simulation context, there has been a considerable amount of work in memory hierarchy effects of priority queue and hashing algorithms. There was, for instance, at least one early paper on page fault behavior [6] and more recent ones concerning cache behavior, including [11], [8] and many others.

  In the investigation here, we consider memory behavior of such algorithms in actual simulation en-

*We wish to thank Victor Castillo and the Lawrence Livermore National Laboratory for supporting this research.

vironments.

We address all of these issues through the vehicle of a case study involving the SimPy language. Through a combination of techniques, we are able to achieve runtime speedups of as much as 30 to 60 percent.

The organization of the remainder of this paper is as follows. Section 2 discusses some of the details of SimPy and Python internals, and implications for SimPy performance. It then presents a Python/C hybrid, using SWIG [13] as the "glue," which achieves faster execution speed while being transparent to the SimPy application programmer.

Section 3 then discusses the various algorithms and data structures we investigated in this study. Our focus here is nontraditional, such as our consideration of the number of page faults generated.

Section 4 then presents the results of our empirical investigations. As noted above, we will show that the measures we have taken yield significant speedups. Beyond that, we will compare the effects of using the Python/C hybrid, versus the effects of using more sophisticated algorithms.

Section 5 presents conclusions and future work.

2 A Look at Internals

Python is a very elegant language that allows one to perform complex operations compactly and clearly. This is highly appealing to the simulation programmer, and reduces development time, but it does raise serious issues concerning implementation. For this reason, we looked not only at SimPy's internals but also at those of Python, to explore how the interpreted nature of Python may affect simulation speed.

SimPy's event set consists of two main structures, a Python list **timestamps** and a Python dictionary **events**.

A Python list resembles a C array, but is more flexible; it is indexed, but also supports operations such as append, insert, remove, etc., and will grow as needed. In this paper we assume CPython, the popular C-language implementation of Python. In CPython, lists are in fact implemented as C arrays and support the more flexible operations by resizing (with **realloc()**) and shifting elements. A Python dictionary is an associative array, implemented as a hash table in CPython.

The structure **events** stores SimPy event objects by using the events' scheduled times as keys, with each key mapping to a Python list of events scheduled for that time. The structure **timestamps** stores those keys in ascending sorted order. Hence, when an event is to be inserted into SimPy's event set, SimPy uses the event's scheduled time as an index into **events** to find the list of events scheduled for that time (creating this list if the new event is the first one to be scheduled for that time) and adds the event to that list.

If the event's scheduled time is a new time, i.e. there had been no other events scheduled for that time before, the event's time is inserted into the **timestamps** list, using Python's **bisect** library module. This module inserts a new value into a sorted list, using binary search to determine the insertion point.

Accesses to **timestamps** for SimPy enqueue operations would appear to take O(log n) steps, where n is the size of the event set. However, this is not true in a practical sense; the insertion of a new event time causes an internal right-shift of a portion of the C array implementing the Python list **timestamps**, taking O(n) time.

Similarly, a SimPy dequeue operation is also more complicated than would first appear. Theoretically of O(1) complexity, it is actually O(n) as the entire remaining array will be internally left-shifted at the C-array level. Though CPython uses the **memmove()** function for efficiency, the operation is still quite slow for large event sets.

In addition, there is the question of slowdown due to hash table access for the structure **events**. However, rather than investigating the CPython implementation of dictionaries, we noticed that one could dispense with the dictionary.

Specifically, we removed the dictionary, and instead stored both the events and their times in the same structure. This was accomplished by moving the information in **events** into **timestamps**. Each element of the latter now is a Python 2-tuple of the form *(event time, event list for that time)*. The list **timestamps** continues to be maintained in sorted time order via Python's **bisect** module. This is possible since comparison operations on Python tuples are handled lexicographically: First the first elements of the two tuples are compared, and the second elements are compared if the first elements are found to be equal. To properly handle identical event times the less-than operator for the SimPy event object class was overloaded to always return False.

Next, we decided to implement SimPy's event list operations in C. Since Python is dynamically typed it must determine, among other things, how to compare two objects. In the case of the event times it would determine that both are Python floats (stored as C doubles) and then compare them. Not only would it be possible to improve performance by avoiding these kinds of extra steps that Python must take, but it would be a proof of concept that we could achieve a

performance gain by writing this time-critical piece of code in C, possibly opening the door to exploring other algorithms for further performance enhancement.

To achieve this, we turned to SWIG (Simplified Wrapper and Interface Generator). SWIG generates the "glue" code (calls to the Python C API) automatically, so our C functions could be written without having to worry too much about going to and from Python. We did however have to properly manage the reference counts, but this was not difficult.

3 Event Set Algorithms Viewed in a Platform-Dependent Context

As stated in the previous section, SimPy uses a dictionary and a Python list to represent its event list. While this structure is functionally adequate, is it possible for us to do better? First, let us take a look at the inherent problems that may cause complications for the Python list.

We have already noted that due to the right-shift operation in an enqueue action, that action takes O(n) time. But let's take a closer look.

Much previous research, both empirical and theoretical, has shown that if an event set is stored as a linear linked list, which is effectively the case in SimPy, most insertions tend to be near the right-hand end of the list [14]. The degree to which this is true varies from one simulation application to the next, but it is clear that in many cases the O(n) time complexity of the enqueue operation is greatly ameliorated by this consideration. It still will be O(n), but the multiplicative constant may be small.

Dequeue, on the other hand, also of O(n) time complexity, has its multiplicative constant essentially equal 1. Thus if SimPy is to be practical for large event sets, an overhaul of the associated structures is imperative. Again, it should be kept in mind that in selecting new structures, we were taking a rather nonstandard point of view, for instance attempting to minimize memory page faults. We turned first to the calendar queue, a priority queue structure developed by Randy Brown specifically for simulation use [2].

A calendar queue is a multi-list structure whose name derives from the structure's similarities to a desk calendar. It is implemented as an array of singly linked lists. Each index in the array, referred to as a *bucket*, represents an interval of simulated time, with the interval size referred to as the *bucket width*. Each bucket corresponds to a day in a given calendar year.

When inserting a new event, one uses the simple formula $\lfloor x/w \rfloor \mod n$, where x is the event time, w is the bucket width, and n is the number of buckets. By taking advantage of the modulo operator the structure allows for inserting events from "future years" to wrap around into the appropriate bucket of the calendar.

To prevent long linear searches for events through the linked lists, the calendar queue keeps an average of two events per bucket and dynamically resizes as necessary. Also, to prevent the calendar queue from being too sparsely populated, the number of buckets shrinks by a factor of two when the number of events pending is less than twice the number of buckets.

The other structure that we selected was the splay tree, a self-balancing binary search tree. The splay operation is very similar to that of an AVL tree in that a series of tree rotations is performed to help balance the tree. This allows operations to be executed in O(log n) time, which is much better than the O(n) time in SimPy. We used a modified version of Weiss' splay tree [15] implementation in our experiments.

Aside from the calendar queue and splay tree, we explored the possibilities of sorted lists in the form of arrays and linked lists, as well as binary heaps. Initially our implementations of simple arrays and linked lists suggested performance gain from their C-implementations so long as event sets were sufficiently small. However, once we began testing them with large event sets, the program run times became so long that time did not allow us to wait for them to finish. The binary heap was a great improvement over the arrays and lists. Yet it was not competitive with the calendar queue nor with the splay tree. Also, [3] further indicated that other structures would not have been as competitive, and as such we omitted implementing them.

4 Empirical Results

Our empirical study was conducted mainly on two simulation applications. The first simulated a call center. Here interarrival times for calls were exponentially distributed with parameter λ_1, which varied, and had exponential duration with parameter λ_2, which was fixed at 2.5 in our experiments. The number of call operators varied according to a set protocol which we will not detail here.

The second "application " was actually an abstraction, consisting of an implementation of the classical Hold Model, in which enqueue and dequeue operations strictly alternate. The parameter in this set of experiments was the coefficient of variation, motivated by findings in previous work that smaller values of this quantity are associated with a tendency for most enqueue operations to occur near the tail of the queue.

The timing results for our call center experiments

appear in Figures 1 through 4. We compared all the structures described above (names in the graph labels appear in parentheses):

- the original SimPy (SimPy)
- SimPy modified so that the data in **events** is incorporated into **timestamps**, while retaining an all-Python implementation (SimPyND, for "SimPy no dictionary")
- SimPy modified so that the original event structures are essentially retained but implementated in C; basically, this is SimPyND converted to C (PQArr)
- SimPy modified to use a C-language calendar queue (CQ)
- SimPy modified to use a C-language splay tree (Splay)

In general, the performance rankings of the various structures in the call center experiments was, from fastest to slowest,

$CQ \approx PQArr > SplayTree > SimPyND > SimPy$

In the experiments shown here, CQ and PQArr were essentially tied for the best. However, PQArr's performance does not scale well when one moves to extremely large event sets. For example, when we ran the Hold Model on event sets of size 10,000, we found the following results:

struct	user time	sys. time	event op. time
PQArr	79.47	4.50	57.87
CQ	33.24	3.95	12.69

We came to two tentative conclusions from these experiments: First, implementing event set operations in C does indeed produce a substantial speedup; PQArr greatly outperforms SimPyND. Second, use of more sophisticated data structures does pay off. Something as simple as combining the original **events** and **timestamps** structures produced a worthwhile speedup, and as noted above, CQ really shines for extremely large event sets.

We spent considerable time comparing the calendar queue and splay tree structures. We wrote C-language versions of them for our further experiments, shown in Figures 5 through 8. These used the Hold Model, and were designed to explore which of the calendar queue and splay tree methods has better performance. At least in this case, the calendar queue was the better of the two, though the splay tree was still a big improvement over the original SimPy.

However, the picture changes when page faults are accounted for. The splay tree substantially outperforms the calendar queue in this sense, as seen in Figures 9 and 10. (Other figures, not shown here, were similar.) Our experiments were performed on 32-bit PCs running Linux Fedora Core 5. We also performed some limited experiments on a 64-bit machine running the same operating system, with a preliminary indication that there is considerable variation from one machine to another. The fact that the calendar queue displayed relatively poor paging performance suggests that the splay tree may well work better on some systems.

Yet another dimension that can be considered is time spent on system calls. The results of our experiments, typified by Figure 11, indicate that the calendar queue and splay tree structures also reduce system time.

5 Conclusions and Discussion

While more thorough investigation would be fruitful, including on other types of applications, our results here clearly confirm that the hybrid approach—writing most of our simulation in Python or Java, while implementing the event set operations in C for speed—is indeed very effective. This would allow simulation programmers to write in languages they find superior while not having to pay as large a performance penalty as they do now.

We note, for instance, the comments at the beginning of the file **EventList.java** in the package DESMO-J [5]:

> Since each step in the discrete simulation requires searching and manipulating the event-list, this is probably one of the best places in the framework to optimize execution performance. Especially if special models show specific behaviour i.e. primarily inserting new events at the very end of the event-list, other implementations of the event-list might support faster access times.

DESMO-J is intended as a *framework* for developing full simulation packages. Its default event list structure is an array, but as noted above, it is assumed that users will develop their own event list structures and algorithms. Since SWIG can be used for Java as well as Python, our findings here would indicate that the approach holds some promise for performance enhancement of packages built from DESMO-J. Our findings here concerning calendar queues versus splay trees seem to address the above comments too.

Further work is needed to compare this approach with others. For instance, one might try to apply JIT techniques to SimPy. The most commonly used JIT compiler for Python is Psyco [7]. Its Web site claims to offer "2X to 100X speed-ups, typically 4X," and it is quite easy to use. Our preliminary evaluation showed only a very small improvement for unmodified SimPy, but further work is needed here. Work is also needed to compare the effectiveness of our approach here with that of [1].

An aspect which we believe deserves much further attention is that of memory hierarchy performance. Recall that although we found that splay trees did not work quite as well as calendar queues from an overall performance standpoint, the splay trees triggered fewer page faults. As the gap between RAM and disk access continues to widen, it could well be the case that splay trees become more attractive.

References

[1] R. Barr, Z.J. Haas, and R. van Renesse1. JiST: An Efficient Approach to Simulation Using Virtual Machines, *Softw. Pract. Exper.*, 2004, 17.

[2] R. Brown. Calendar Queues: A Fast 0(1) Priority Queue Implementation for the Simulation Event Set Problem, *CACM*, 31(10).

[3] K. Chung, J. Sang, V. Rego. A Performance Comparison of Event Calendar Algorithms: an Empirical Approach, *Software Practice and Experience*, vol. 23(10), October 1993, 1107-1138.

[4] K. Muller. *SimPy—Simulation in SimPy.* Book manuscript.

[5] Lechler, T. and B. Page. DESMO-J: An Object Oriented Discrete Simulation Framework in Java. In *Proceedings of the 11th European Simulation Symposium*, ed. G. Horton, D. Mller, and U. Rde, 1999, 46-50. Erlangen: SCS Publishing House.

[6] D. Naor, C.U. Martel and N.S. Matloff. Performance of Priority Queue Structures in a Virtual Memory Environment, *The Computer Journal*, 34, 5, October 1991, 428-437.

[7] Psyco homepage, `http://psyco.sourceforge.net`.

[8] H. Qi and C.U. Martel. Design and Analysis of Hashing Algorithms with Cache Effects, citeseer.ist.psu.edu/92497.html.

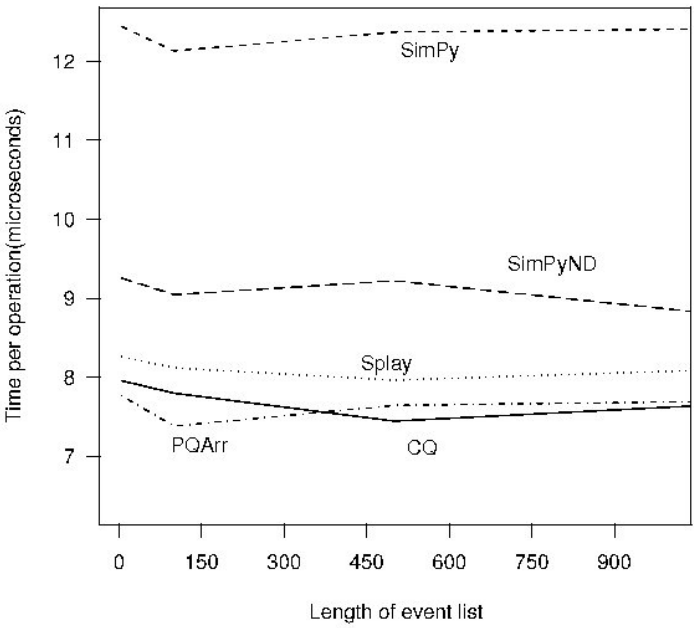

Figure 1. Call Center, $\lambda_1 = 1.0$

[9] R. Radhakrishnan, N. Vijaykrishnan, L.K. John, A. Sivasubramaniam, J. Rubio, J. Sabarinathan. Java Runtime Systems: Characterization and Architectural Implications, *IEEE Transactions on Computers*, 131-146, Feb. 2001.

[10] R. Ronngren and R. Ayani. A Comparative Study of Parallel and Sequential Priority Queue Algorithms, *ACM Trans. on Modeling and Simulation of Computer System*, Vol. 7, No. 2, April 1997, 157209.

[11] P. Sanders. Fast Priority Queues for Cached Memory, in *Proceedings of the 1st Workshop on Algorithm Engineering and Experimentation*, Volume 1619 of Lecture Notes in Computer Science, 1999, 312–327, Springer-Verlag.

[12] SimPy homepage, http://simpy.sourceforge.net.

[13] SWIG homepage, http://www.swig.org.

[14] J. Vaucher. On the Distribution of Event Times for the Notices in a Simulation Event List, *INFOR*, June 1977.

[15] Mark Allen Weiss, *Data Structures and Algorithm Analysis in C++*, Second Edition, 1999.

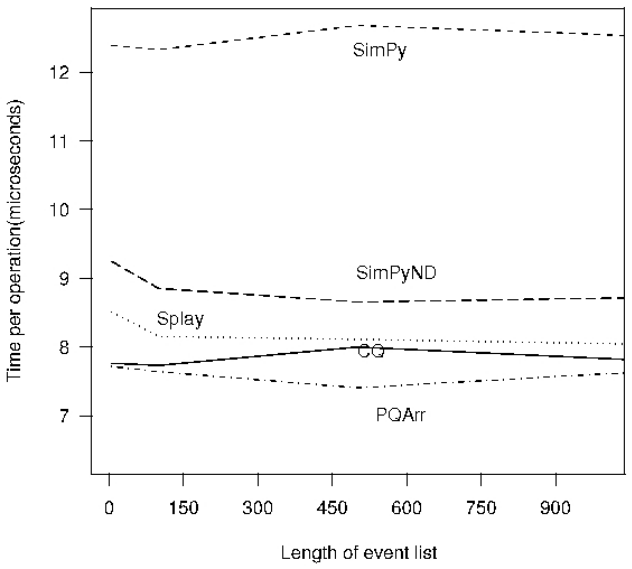

Figure 2. Call Center Model, $\lambda_1 = 1.5$

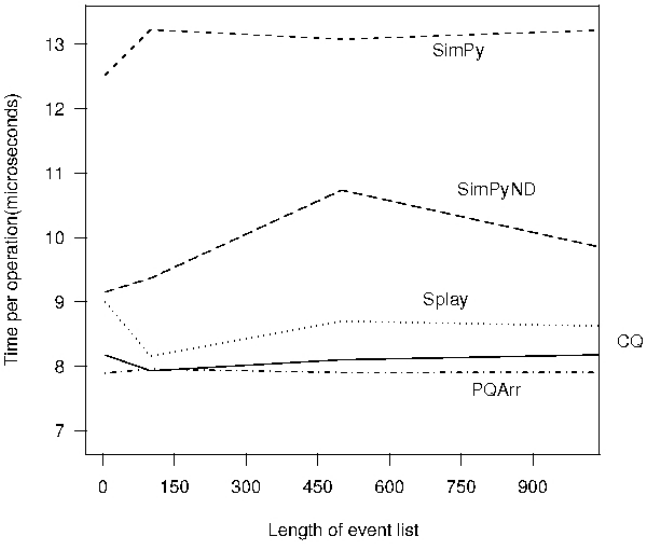

Figure 4. Call Center Model, $\lambda_1 = 3.5$

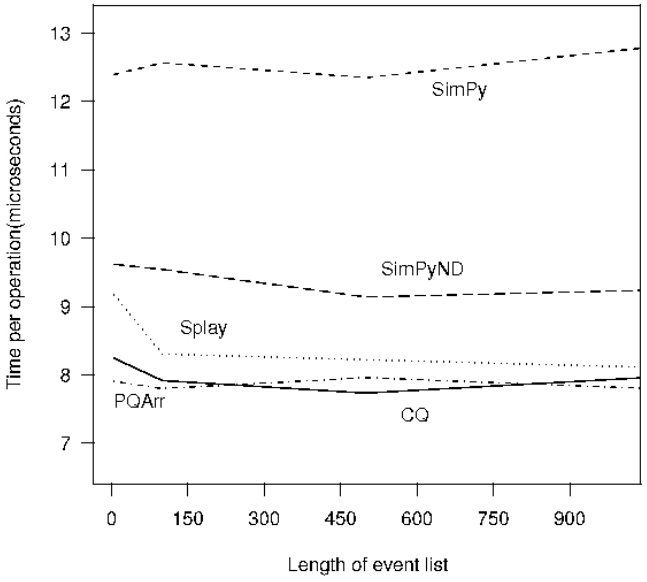

Figure 3. Call Center Model, $\lambda_1 = 2.0$

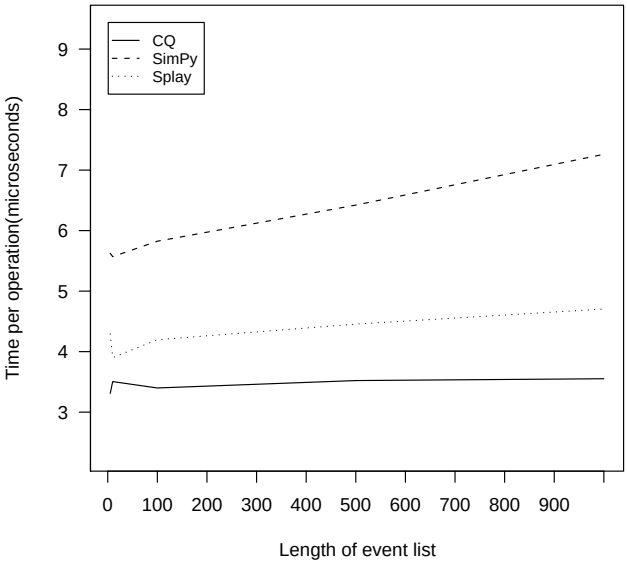

Figure 5. Hold Model, COV = 0.1

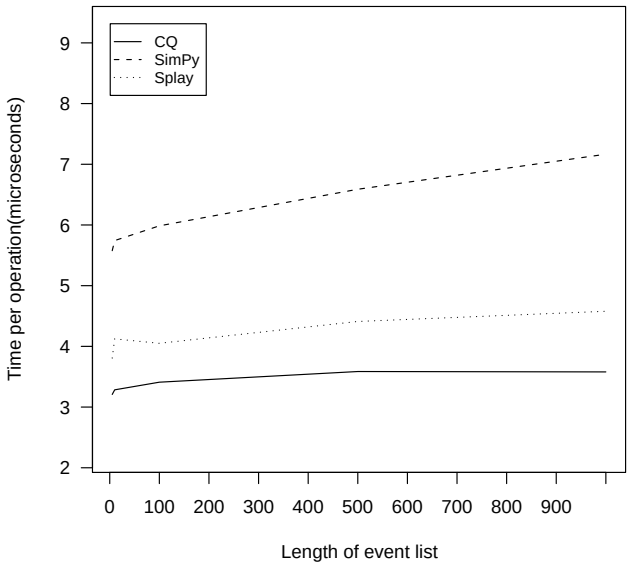

Figure 6. Hold Model, COV = 1.0

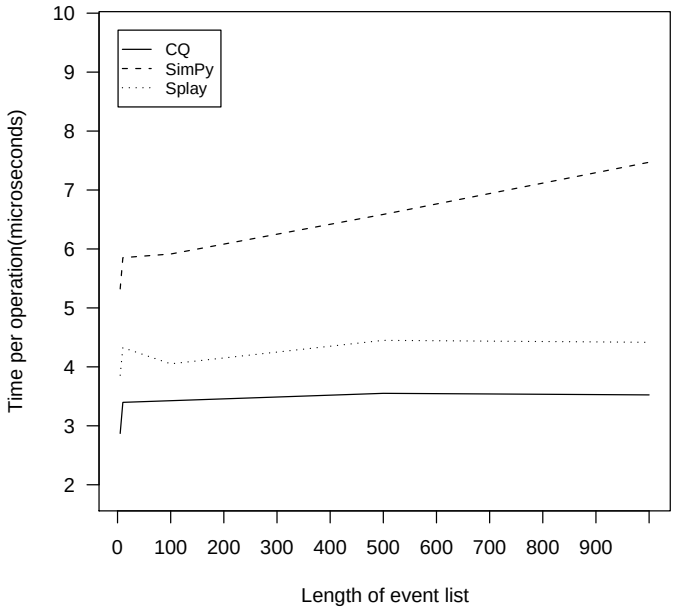

Figure 7. Hold Model, COV = 2.6

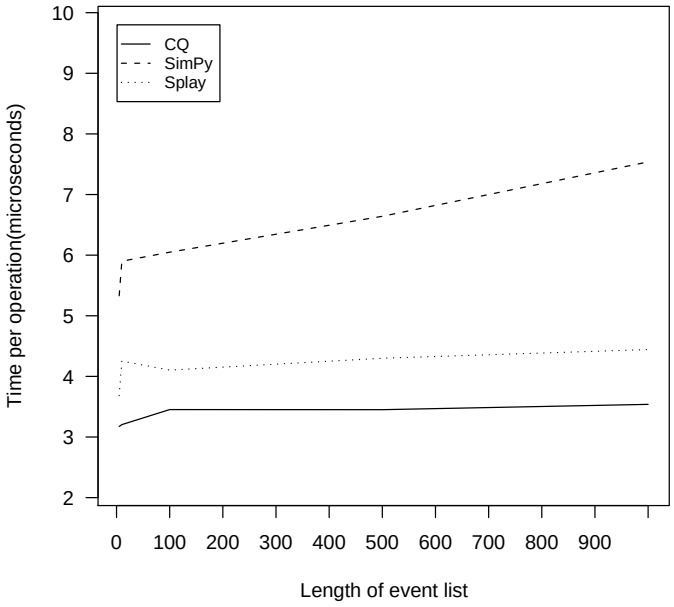

Figure 8. Hold Model, COV = 4.0

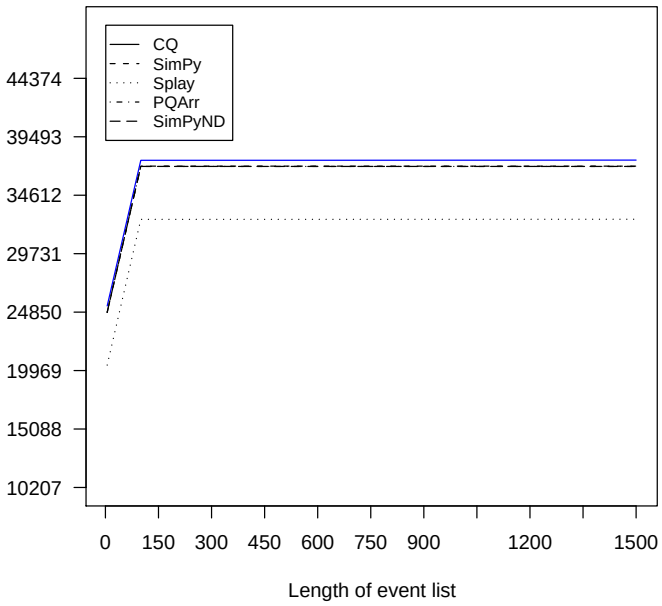

Figure 9. Number of Page Faults: Call Center Model, $\lambda_1 = 1.0$

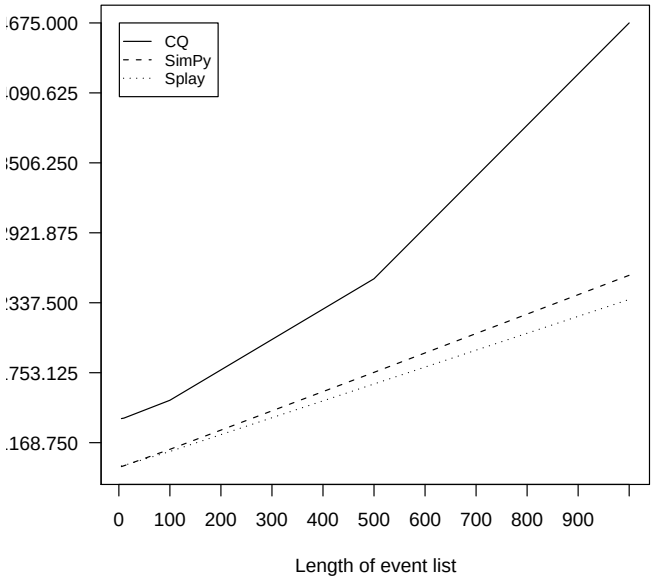

Figure 10. Number of Page Faults: Hold Model, COV = 1.5

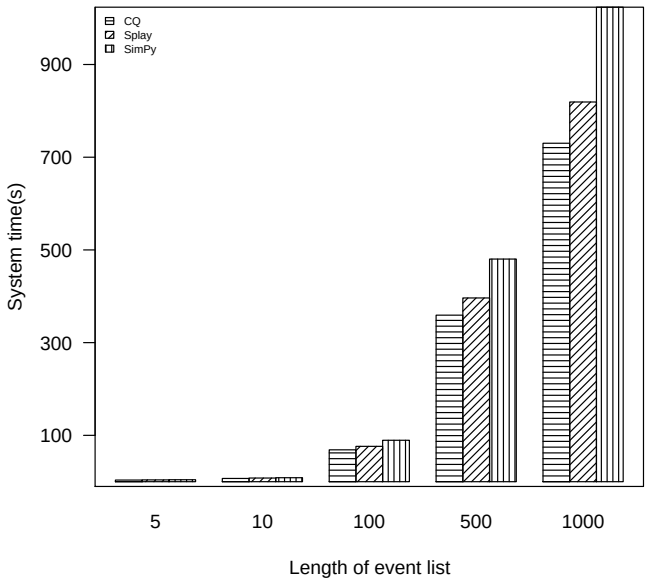

Figure 11. Amount of System Time: Hold Model, COV = 1.5