

# Benchmark `health` Considered Harmful

Craig B. Zilles

Computer Sciences Department, University of Wisconsin - Madison  
1210 West Dayton Street, Madison, WI 53706-1685, USA  
zilles@cs.wisc.edu

## Abstract

*In the past couple of years, a number of software and architectural techniques have been proposed for improving the performance of linked data structures. These research ideas are often evaluated using the Olden benchmark suite [1]. Frequently, in such experiments, the largest speed-up is attained for the benchmark called `health`. This article demonstrates that this benchmark is a micro-benchmark for enormous linked list traversals, and not a good one at that. Given that linked lists of such size are not an efficient data structure, it is unlikely that this benchmark corresponds to any real program. Hence the benchmark should not be used. To demonstrate the inherent inefficiency in its use of linked data structures, the `health` program was modified algorithmically to generate the same output, while improving the execution time by over a factor of 200 on a 500Mhz Pentium II Xeon.*

## 1 Introduction

The program `health` is a “Colombian health care simulator” which uses doubly-linked trees to model a hierarchy of hospitals and a set of doubly-linked lists to track the patients undergoing treatment at each hospital. The simulation is time-based; on every iteration each hospital is processed. Patients periodically show up at hospitals and undergo a multi-step treatment process: 1) they wait for a free physician to attend them, 2) the physician assesses their ailment and either transfers them to the next hospital up the hierarchy (back to step (1) at a new hospital), or 3) the physician treats them. There are fixed latencies for assessment and treatment, but the time spent waiting depends on congestion at the hospital. There is a fixed probability that a patient shows up each cycle and that the patient needs to be referred to the next level of the hospital. The whole benchmark is less than 500 lines of C code.

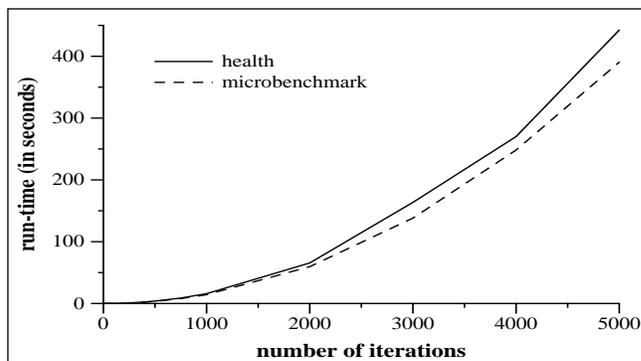
## 2 The Central Problem with `health`

There are many inefficiencies in `health` (others are discussed briefly in Section 3), but the largest problem is due to two properties: 1) the size of the waiter list and 2) the frequency it is traversed. The arrival rate of patients is faster than the hospital system can deal with them. Patients accumulate in the waiting list; the longer the simulation, the longer the list. Hopefully this is not the case in the real Colombian health care system, bringing into question the accuracy of the simulation.

I am not interested in the accuracy of the simulation, but rather whether it is a useful benchmark to use for performance studies. The number of waiters only impacts the benchmark’s run-time behavior if the list is frequently traversed. The list is traversed on two occasions: 1) once for each iteration, to increment the time each waiter has spent waiting, and 2) once for each list insertion, because elements are inserted at the end of the list (and no tail pointer is kept). This means that the monotonically increasing amount of memory used in the simulation is touched (dirtied even) at least once every iteration of the simulation.

Shortly after the simulation begins, the run time is dominated by these linked list traversals, effectively making `health` an enormous linked list traversal micro-benchmark whose behavior is distorted slightly by the rest of the code. The execution time of the benchmark is quadratic with the number of iterations run (list length and number of traversals are both linear with iterations). Figure 1 shows that a parameterizable list traversal micro-benchmark [2], which only performs the list traversals, closely approximates the behavior of the benchmark.

The periodic traversal of the list can be entirely avoided. Rather than increment each element every cycle, it is much more efficient to record when the element is inserted in the list and compute its residency in the list by subtracting that time from the current time. The insertion problem can be simply solved by keeping tail pointers to enable constant time insertions. By making these two changes we remove the quadratic term from the execution time. For the input parameters “5 5000 1 1” this accounts for a speedup over a factor of 100 (from 442.4 seconds to 3.56 seconds).



**Figure 1.** *The quadratic run-time behavior of the benchmark `health` is approximated by a linked list traversal micro-benchmark. The micro-benchmark under-estimates run-time by 10 percent.*



**Figure 2. Run-time contributions of inefficiencies in the benchmark health.** The shaded regions were optimized away through modifications of the benchmark's source code. The white region corresponds to the final run-time, of which 25% is from the random number generator, and 7% is from `printf`s.

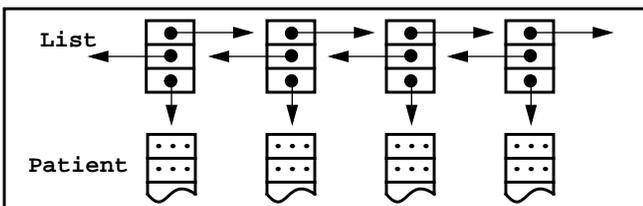
### 3 Other Problems

Although not as egregious as the faults already discussed, `health` is still algorithmically inefficient. The modifications described in this section reduce the execution time of `health` (using the "5 5000 1 1" input) by almost an additional factor of two (from 3.56 to 1.99 seconds). The approximate breakdown of these effects are described in this section and can be seen in graphically Figure 2.

One serious inefficiency in `health` is derived from the fact that separate structures are used for patients and the linked list (as shown in Figure 3) despite the fact that each patient only ever appears in one list. Unifying these into one structure reduces run time by 0.82 seconds. In addition, the function to remove an element from a list traverses the list searching for the matching element, despite the fact that a doubly linked list is maintained. Because removed elements are always at the head of a list, rectifying this (and the memory leak due to not deallocating the list structures) accounts for only an additional 0.09 seconds.

The simulation gathers statistics on how many patients are processed, how long their treatment takes, and how many hospitals they visit. After each patient is treated, he is added to a list of treated patients which is traversed at the end of simulations to collect these statistics. By gathering statistics for a patient when he is treated, we can reuse the structure for a future patient, avoiding the need to allocate additional memory, as well as the traversal of treated patients at the end of the simulation. This change reduces execution time by 0.27 seconds.

Since patients are frequently allocated and (now) deallocated, it is worthwhile to provide our own allocator for patient structures. A list of unused patient structures is maintained (using the forward pointer field already present in the structure), and structures are `malloc()`'ed in batches when the free list is empty. The change reduces run time by 0.18 seconds.



**Figure 3. A pair of data structures is used for maintaining the patient list.** By combining the list and patient structures into a single structure, a ~25% speedup is achieved.

Much as the waiting time for each waiter is incremented each cycle, the time a patient spends at each other stage of treatment is fixed and decremented each cycle; the patient is transferred when the value reaches zero. Since these lists are much shorter (bounded by the number of physicians in the hospital), traversing them every cycle does not affect performance as much, but is unnecessary. Rather than tracking how many more cycles a patient must remain in its current list, we store the iteration when it should be removed, avoiding the decrement. Furthermore, since the lists are always sorted by this iteration number (because the time at each stage is a constant and we insert at the tail), any patients scheduled for removal will always be at the head of the list. Avoiding these unnecessary traversals reduces execution time by 0.11 seconds.

Generally, when an element is removed from a list it is going to be inserted into another one. By grouping the `removeList` and `addList` functionality into a single `moveList` command, and using it when possible we can shave another 0.10 seconds off the execution time.

Of the remaining time, about 25% (0.50 seconds) is used for generating the random numbers used in the simulation. An additional 7% (0.13 seconds) is spent writing to `stderr`. These modifications took about 4 hours to complete.

### 4 Conclusion

Benchmarks are a necessary evil of research in computer architecture, and finding good benchmarks is difficult due to the limitations and slow-downs introduced by simulation-based methodologies. But this difficulty does not relinquish researchers from the burden of verifying that the benchmarks they use are valid. This paper explores the Olden benchmark `health`, and demonstrates that it should not be used; the performance of such an inefficient means of performing a given task is irrelevant. If the behavior of enormous linked list traversals needs to be demonstrated, which is of questionable utility, a micro-benchmark should be used to make clear what is being evaluated.

### 5 References

- [1] A. Rogers, M. Carlisle, J. Reppy, and L. Hendren. Supporting Dynamic Data Structures on Distributed Memory Machines. *ACM Transactions on Programming Languages and Systems*, Mar. 1995.
- [2] <http://www.cs.wisc.edu/~zilles/l1ubenchmark.html>.