Evaluating a parallel, real-time garbage collector on modern hardware

Ivan Jager Chris Martens

Fall 2009

1 Introduction

Garbage collection is an important memory management device for modern programming languages. As the common architecture in commercially produced computers shifts heavily toward multicore, programming language design is becoming more concerned with paralellism, and, as a consequence, so is garbage collection. The problem of garbage collection that is parallel (multiple GCs can run on different cores), concurrent (the GC can collect at the same time that the program executes), and real-time (program pauses due to collection are sufficiently short and infrequent) has been explored in great depth by Perry Cheng [5, 6].

However, the evaluation of their collector was performed on a Sun Enterprise 10000 server with 64 UltraSPARC II CPUs, and all of the data they present compares between runs on 4 to 32 processors. Our project aims to evaluate this collector on hardware that might more realistically be running on a personal computer today. More specifically, our goal was to test on hardware that is both *modern* (built more recently) and *realistic* according to modern standards (2 or 4 cores, 1 CPU). We present performance results for runs with 1-8 threads in addition to larger numbers.

We approach this problem by compiling some common ML benchmarks [1] with the TILT SML compiler [2] and collecting GC statistics via arguments to TILT's runtime, which we describe in more detail below. Many of the benchmarks described in Cheng's experiments match the name and description of many of our benchmarks, but we could not determine the exact benchmark suite he used, so we do not believe it is valid to compare our data to his directly. However, we can consider the performance we observe in terms of relative speedup (e.g. against sequential performance on the same system) and compare this value to Cheng's results.

This project contributes an approximate reproduction of Cheng's experiments in a distinct and more currently relevant setting. Another worthwhile contribution is the port of TILT to SML/NJ 110.71, the most current version, which consumed the majority of our project time.

2 Background: A parallel, real-time garbage collector

The particular collector we tested is described in detail in Cheng and Blelloch's paper [6]. We give a brief overview here to familiarize the reader with the important parts.

The collector is implemented as part of the runtime [7] for TILT, a type-preserving compiler for the programming language SML. SML is a statically-typed functional language with a module system, making it a prime candidate for testing garbage collection performance – it has a high allocation rate due to things like modules, closures, and datatypes.

Although there are several collector implementations, all with numerous adjectives in their titles, the first idea to understand is a semispace copying collector. Such a collector divides the heap in half, initially designating one half as *from-space* and one half as *to-space*. When memory is requested by the mutator (user program), it is allocated from the from-space. When garbage collection occurs, it begins with a *root set* of live pointers in the program (registers, global variables and stack variables) and computes all reachable heap

memory from those nodes; then it copies the live data (hence *copying collector*) to the to-space and finally updates the root pointers and reverses the roles of the from- and to-space.

Cheng defines a few modifiers as follows (quoted directly): [5]

- Incremental: A single collection is divided into multiple increments whose executions are interleaved with the application on a single processor. Typically, the rate of collection is related to (and greater than) the allocation rate so that the collection is guaranteed to terminate.
- Concurrent: At least one program thread and one collector thread are executing concurrently.
- Parallel: Multiple collector threads are collecting concurrently.

Cheney [4] proposed a practical (non-recursive) implementation of the semispace copying collector for a uniprocessor; Cheng's previous work [3] modified this simple algorithm to make it scalably parallel, and further, added incremental collection and concurrency. The implementation in the TILT compiler extends the algorithm further to eliminate the overhead imposed by unnecessarily fine collection granularity and include *generational collection*, which distinguishes objects' candidacy for collection based on their current lifespan – new objects are allocated in a "nursery" and get promoted to "tenured space" (which gets collected less frequently) only after survining a collection – under the "folk wisdom" assumption that most objects die young.

In total, TILT includes six variants on the copying collector: semispace stop-copy (`semi`), semispace parallel (`semipara`), semispace concurrent (`semiconc`); generational stop-copy (`gen`), generational parallel (`genpara`), and generational concurrent (`genconc`).

3 Measurement approach

Our main measurements were of the execution time and total GC overhead for various benchmarks (described in the Experimental Setup section) while varying the number of processors and collector implementation.

An interesting contribution of Cheng's work is the reconsideration of what it means for a collector to be *real-time*. One possibility is to measure "maximum pause time", which is the longest amount of time for which the collector interrupts mutator execution. However, a series of closely-spaced short pauses can be just as bad as a single long pause, depending on the required response time of the program. Another thing one might like to measure is the total GC overhead, which is just the fraction of overall execution time attributable to collection. The notion of *minimum mutator utilization* (MMU, or just *utilization*) generalizes both of these measurements. The measurement is parametric on a time interval *window size*. For a given window size, the MMU is the fraction of that window that the mutator executes. For example, the two different pause scenarios illustrated below, where each rectangle is a window, are considered to have equal utilization:

because within the given window, the fraction of mutator execution is the same, even though the first sample has shorter pauses. The idea is that different programs will have different response time requirements, so utilization should be evaluated at many different granularities (window sizes).

We collected utilization data for 20 window sizes between 5 ms and 4000 ms. Most of the utilization values we saw were between 0.7 and 0.8, including for all the data presented in our example graphs. Surprisingly, there was not much variation in these numbers with the change in window size.

3.1 Changes from our initial plan

When we first began the project, we wanted to replicate as many of Cheng's experiments as possible. However, due to the surprises described below, we ran out of time to perform many of these tests. In particular, we wanted to find problems that sped up with addition of processors due to being written and run in a parallel manner. However, we couldn't find an implementation of the parallel binding construct (`pval` for TILT), which was a major downfall. Therefore, our experiments changed to running sequential applications in hopes of observing interesting variation based only on the garbage collector's parallelism.

According to the somewhat vague goal tiers set out in our project proposal, we have met our 100% goal to "evaluate [the GC] on a small, modern Sparc with 2 to 4 processors." However, we would have liked to gather more data and do more sensitivity analysis to feel that we truly evaluated the GC. This shortcoming is purely a matter of running out of time due to the complications we met.

4 Experimental setup

4.1 Architecture

Since the GC was only implemented for Alpha and Sparc, and Alpha is discontinued, we decided to try to run our experiments on a modern Sparc machine. To this end, we obtained accounts on a SUN SPARC Enterprise T5220, with a single 1.2 GHz UltraSparc T2 (Niagara 2) and 32 GB of RAM. The T2 is an 8-core CPU, where each core supports 8-way SMT, giving it the ability to run up to 64 thread simultaneously. While this is not a typical desktop machine, it is not too far off from what a typical desktop may be a few years down the line.

We considered porting to x86, but the way TILT targets x86 is through TAL, which has its own GC. So, we would have had to port to a different compiler or write an x86 backend for TILT.

4.2 Building/Porting TILT

TILT is designed to build under SML/NJ 110.07 and under itself. Since our testing machine did not have an installation, our first task was to get SML/NJ to compile. When attempting to build version 110.07, we encountered a bug (segmentation fault during bootstrapping) that was never resolved. After asking for some advice, we decided to try getting SML/NJ 110.71 to build and tweaking TILT to compile under the newer version. This decision turned out to be fruitful, but only with a number of small but tedious, pervasive changes to TILT's source to reflect the change in basis library code between the two versions of SML/NJ. Should one wish to reproduce our experiments from scratch, they can obtain our modified TILT source at

`http://www.cs.cmu.edu/~cmartens/740project/tilt-nj110_71.tar.gz`

A complete list of changes we made is available upon request.

4.3 Benchmarks

To run the benchmarks, which were originally designed for MLton (another Standard ML compiler), we wrote some wrapper code around the tests and a script to generate TILT project files. TILT has some known bugs, so of the 43 benchmarks, 14 failed to compile. 29 benchmarks seems to be more than sufficient for gathering data, so we made no attempt to get the others to compile.

We include graphs showing the runtimes of 3 benchmarks with the generational GC on one thread, as well as the parallel generational GC on 1, 2, 3, 4, 5, 6, 7, 8, 12, 16, 24, and 32 threads. The three benchmarks we include graphs for are `boyer` (which does manipulation over terms), `flat-array` (which creates a large array of pairs, then sums the data in th epairs), and `logic` (from the SML/NJ benchmark suite).

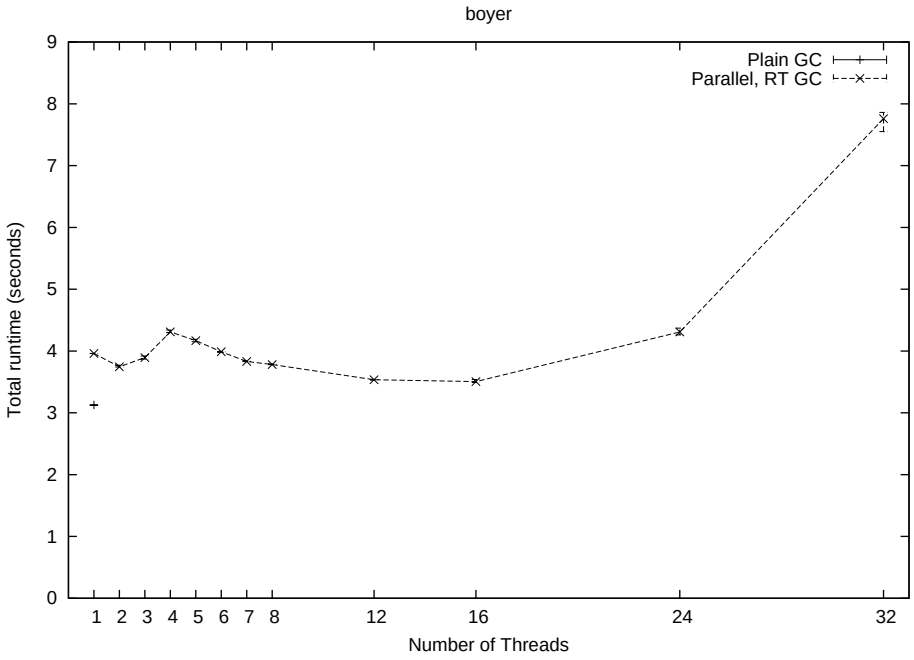


Figure 1: Slowdown when using the parallel GC on the boyer benchmark.

4.4 TILT runtime options

TILT's runtime comes equipped with several options one can pass to the compiled program to instrument its execution. They are invoked as in the following example.

```
$ tilt -o exe project-file
$ ./exe @someboolоption @someintoption=3 [...] <program-args>
```

The above command would set `someboolоption` to `true` and `someintoption` to 3, were said options to exist. A list of runtime options and their descriptions can be obtained with `@help`.

In particular, we ran our code with `@info` to get data for the default generational GC, and `@info @proc=`N `@genpara` for the parallel generational GC with N threads.

5 Experimental evaluation

TILT no longer has the pval extension that was added by Cheng, and appears to be lacking any alternate way to have multiple mutator threads so we were only able to test single-threaded benchmarks. We expected the parallel GC to take advantage of additional threads even when the application couldn't, which we thought would give a better speedup for benchmarks that spent a lot of time in the GC. Surprisingly, in all our tests, the generational parallel GC always did worse than the plain old generational GC. We also noticed an intriguing peak in run times at 4 CPUs, which we have not found a suitable explanation for.

6 Surprises and lessons learned

Our major (pre-experiment) surprise was the difficulty we had building TILT on the Sparc. Perhaps we should have further investigated the bug in compiling the older version of SML/NJ, as it may have cost us

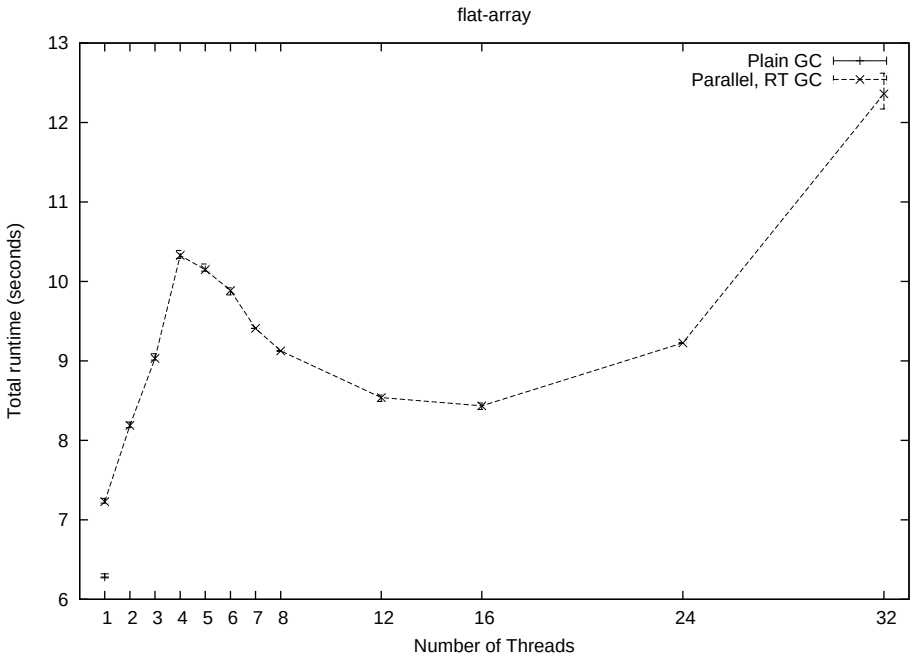


Figure 2: Slowdown when using the parallel GC on the flat-array benchmark.

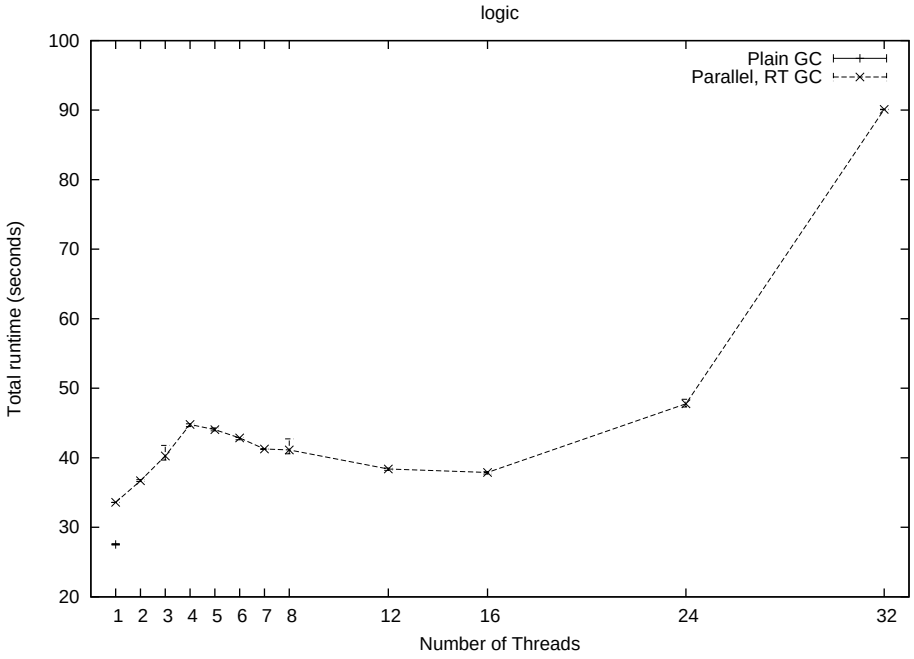


Figure 3: Slowdown when using the parallel GC on the logic benchmark.

less time overall. It may also be a lesson not to use someone else's unmaintained, experimental software for a time-constrained project.

One thing we also found unpleasantly surprising was the lack of care taken in the papers we read to explain their experiments in a reproducible manner. Even once we got TILT to build, there was very little documentation about how to construct project files and no documentation whatsoever about the runtime arguments (which we found by poking around in the runtime source code), and the written reports on the original GC experiments contain no pointers to code written, benchmarks used or explanations of the experimental logistics.

As far as experimental surprises, we did not expect that parallel garbage collection would fare uniformly worse than its sequential counterpart on sequential benchmarks.

7 Conclusions and future work

There were many tests that we ran out of time to do that could be performed as future work. The main thing missing from our test suite was parallelism in mutator code. Cheng augmented the TILT compiler with a `pval` construct for writing multiple declarations to run in parallel for the sake of his tests, and our version of TILT did not have this addition.

Many of Cheng's interesting results were about the effects of load balancing. Reproducing these tests with load balancing would also be worthwhile future work. Additionally, there were a few variables for which we did not run sensitivity analysis, such as collection rate and heap size.

One of the goals of this project was to determine why such a full-featured parallel garbage collector was not in more widespread use. We have learned a number of possible answers to this question. The first is, as mentioned above, the experiments and usage of the compiler in general are difficult to reproduce. Furthermore, they are implemented for specific obsolete hardware, and it has proved nontrivial to build the source on supposedly-supported hardware which is not much different from the original experiments' environment. Finally, TILT's runtime is buggy and crashes unpredictably with some collectors on some inputs. These are primarily problems with using TILT, not the collector itself, but as it is currently impossible to use the collector without TILT, we believe this is a major factor in its lack of widespread popularity.

Additionally, our data seems to be telling us that parallel garbage collection is not worthwhile for sequential programs, at least on the particular benchmarks we tested. This could be another reason this GC has not met much demand, as most programs today are not explicitly parallel. However, we do not believe that this means the GC has no use except for explicitly parallelized programs. Cheng performs several tests [5] on the collector's performance under *coarse parallelism* tests, which involved running a distinct permutation of the same sequence of benchmarks on each thread in parallel. It seems conceivable that one might find a way to perform such tests without a linguistic parallel binding construct, so that would be an obvious next vector for future work.

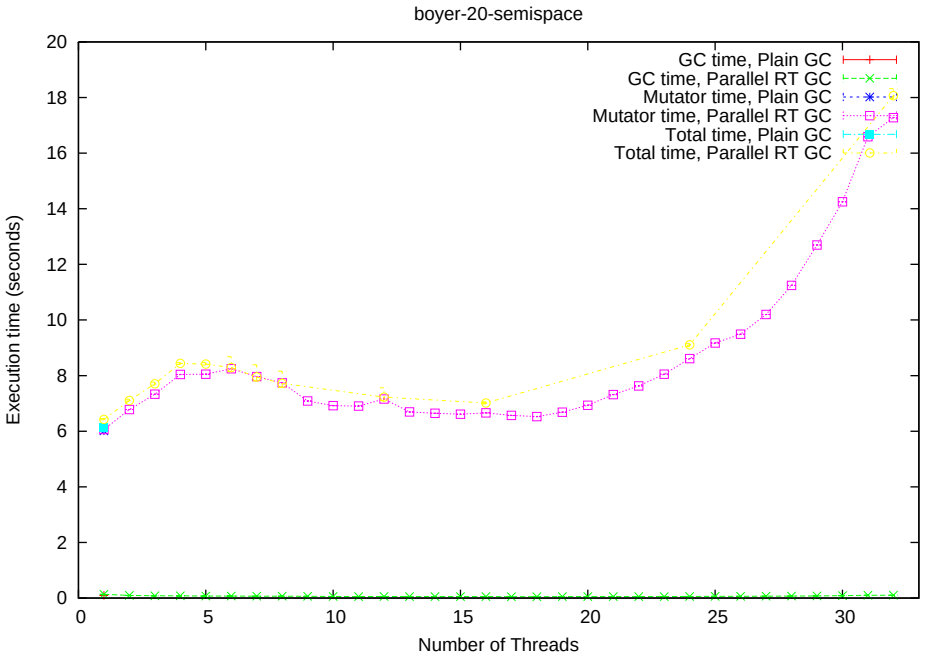
8 Addendum, December 6, 2009

New Results

To attempt to determine the cause of the collector's seeming non-scalability for sequential benchmarks, we gathered and plotted more data after the initial project submission deadline. Our hypothesis based on this data is that the *mutator* causes the increase in running time proportional to the number of threads; the GC itself actually scales fairly well.

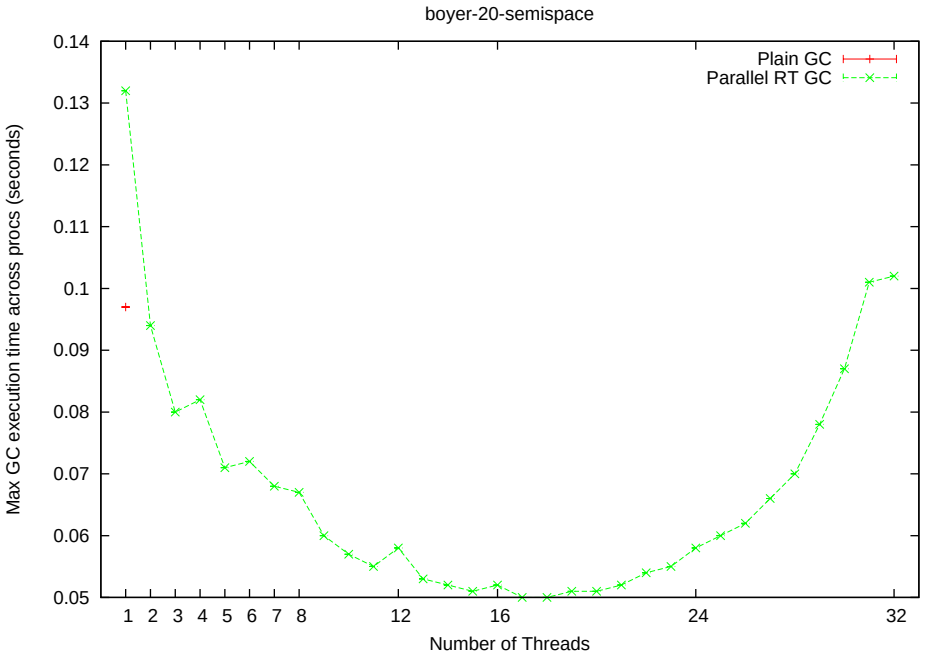
To reach this conclusion, we varied our methodology rather than blindly gathering more data. The first thing we changed is that we started using the semispace collector (rather than generational) as our primary data source, since its results seemed more regular and easy to predict. More importantly, we decided to plot the time spent in garbage collection and the time spent in mutator code (as reported by TILT's profiler) as separate entities. Therein lay the insight.

When we look at total time, mutator time, and GC time separated out and on the same scale for the boyer benchmark, it looks something like this:



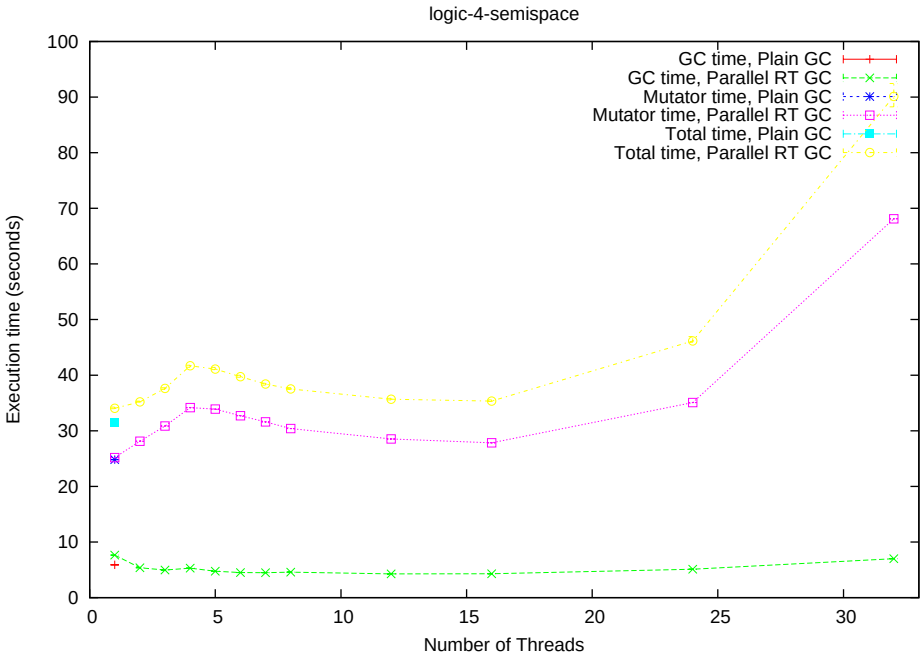
("semispace" indicates that we ran using the semispace collector as opposed to generational and 20 is the number of times the benchmark ran in sequence.)

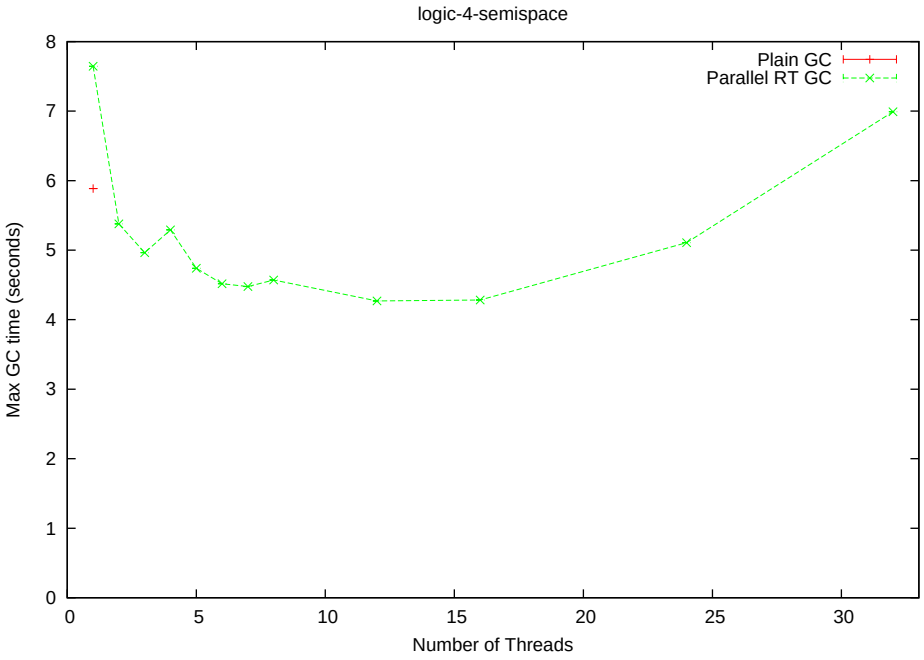
The mutator running time nearly echoes the total running time trend, while GC running time is miniscule. And if we take a closer look at how GC time varies:



It turns out that, up to a certain point (and after the addition of one thread), more threads means less time is spent doing GC – this is much closer to the kind of data we were originally expecting from overall running time.

Below, we present a few more benchmark graphs that support the hypothesis that the unscalable running times comes from the mutator:





Again, although parallel GC does not uniformly scale, the running times are clearly dominated by the mutator code.

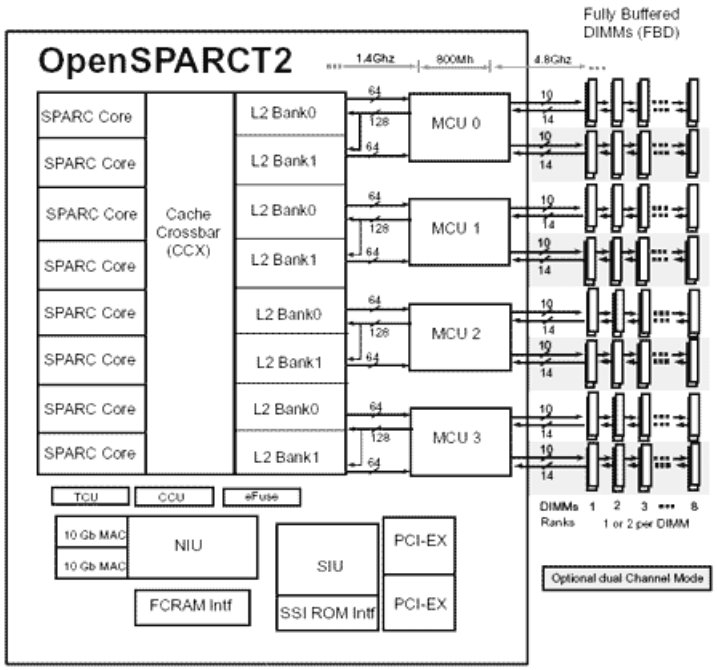


Figure 4: UltraSPARC T2 Block Diagram

Mutator non-scalability

This new data tasks us with explaining the increase in mutator time. We found no good way to more deeply profile the mutator runtime, but we tried to at least make an informed speculation based on what we know about the hardware we ran on. For example, it could be explained by cache lines being invalidated from the core running the mutator. However, the T2 has a shared L2, and the individual L1s are significantly smaller than the GC heap size. While each of the 8 cores is physically arranged closest to certain banks of L2, as you can see from Figure 8, the cores are connected to L2 through a crossbar and there does not appear to be any core-L2 affinity.

Additional (unfruitful) attempts and conclusions

The above summarizes our informative findings about the behavior of TILT-compiled code. We also performed several sensitivity analyses that did not lend much insight, but we thought it was worth looking at them in case they did.

For instance:

- There are runtime options (`@minheap`, `@maxheap`, `@fixheap`) for setting the program's heap size; we wondered if varying heap size would change scalability at all. Results were not surprising: shrinking heap size increased time spent in GC, but heap size had no noticable effect on the shape of the graph when plotted against number of threads.
- Varying the numerical argument passed to the benchmark simply runs it more times; larger numbers produced smoother graphs, but produced no surprising changes.
- The MMU (described earlier in the report) varied with window size in approximately the expected way (bigger windows had better utilization); we thought this data was less important to solving the central

mystery or else we would have included them. One thing we did not get to that might be interesting is to fix a window size and look at utilization across number of processors, but we expect that this would vary similarly to GC time.

At a high priority on our "future work" wishlist from before this addendum was to run parallel benchmark code. We managed to check out a copy of TILT with the parallel binding construct `pval` from the project repository and spent considerable time trying to get it to build, but after encountering significantly more software rot than the slightly newer version we worked with originally, we decided that the amount of effort necessary would exceed the increase in utility over data we could more readily gather.

Despite not running parallel benchmarks, we believe the question we have made progress toward answering is a valuable one: is a parallel garbage collector only useful for parallel code? Perhaps not – the time spent doing garbage collection seems to scale favorably with additional processors. However, this particular implementation on this particular machine generates mutator code that runs more slowly with additional processors, causing its whole parallelism enterprise to be a net loss.

9 Acknowledgments

We would like to acknowledge James Hoe and Ippokratus Pandis for the usage of the T2 Niagara we used for testing, David Swasey for his advice on compiling TILT, and William Lovas for his help with shell scripting.

10 Credit

Equal credit for the project should be given to each group member.

References

[1] http://mlton.org/cgi-bin/viewsvn.cgi/mlton/trunk/benchmark/.

[2] http://www.tilt.cs.cmu.edu/.

[3] Guy E. Blelloch and Perry Cheng. On bounding time and space for multiprocessor garbage collection. *SIGPLAN Not.*, 39(4):626–641, 2004.

[4] C. J. Cheney. A nonrecursive list compacting algorithm. *Commun. ACM*, 13(11):677–678, 1970.

[5] Perry Cheng. *Scalable real-time parallel garbage collection for symmetric multiprocessors*. PhD thesis, Pittsburgh, PA, USA, 2001. Adviser-Blelloch, Guy and Adviser-Harper, Robert.

[6] Perry Cheng and Guy Blelloch. A parallel, real-time garbage collector. In *Proc. PLDI 2001*, 2001.

[7] Perry Cheng, Robert Harper, and Peter Lee. Generational stack collection and profile-driven pretenuring. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 162–173, New York, NY, USA, 1998. ACM.