AN INVESTIGATION OF VARIATIONS IN MEASUREMENTS OF EXECUTION TIMES

A thesis presented to
the faculty of
the College of Arts and Sciences of Ohio University

In partial fulfillment
of the requirements for the degree
Master of Science

Kent H. Hunter
March 2003
This thesis entitled

AN INVESTIGATION OF VARIATIONS IN MEASUREMENTS OF EXECUTION TIMES

BY

KENT H. HUNTER

has been approved

for the Department of Mathematics

and the College of Arts and Sciences by

Carl T. Bruggeman

Assistant Professor of Computer Science

Leslie A. Flemming

Dean, College of Arts and Sciences
HUNTER, KENT H. M.S. March 2003. Mathematics

An Investigation of Variations in Measurements of Execution Times (77pp.)

Director of Thesis: Carl Bruggeman

In Computer Science, particularly with optimizing compilers, ideas are often judged at least partially by their performance on certain benchmarks. The accuracy of these benchmarks is therefore an important concern, but little attention has been given specifically to this issue.

We provide a partial remedy by measuring, under a variety of conditions, the accuracy of a few benchmarks that measure execution times of programs written in a high-level, garbage-collected language. Standard deviations of large sets of execution times are used as a metric.

From such experiments we conclude that, among other things, both running processes and garbage collection can introduce some variation into execution speeds, but an active network connection need not be greatly detrimental. Another result shows that redirecting the output of the benchmark to a file had some interaction with the garbage collector that caused large fluctuations in running times.

Approved: Carl Bruggeman

Assistant Professor of Computer Science
Contents

1 Introduction ........................................... 9

2 Background ............................................ 11
   2.1 Experimental CS .................................. 11
   2.2 Garbage Collection ................................. 12
   2.3 Continuations ...................................... 13
      2.3.1 Activation Records and the Stack .............. 16
      2.3.2 Continuations and the Stack .................. 17
      2.3.3 Using the Heap ................................. 18
      2.3.4 Using a Modified Stack ....................... 21
   2.4 Standard Deviation ................................. 23

3 Measurement ......................................... 25
   3.1 Measuring CPU Time ............................... 25
   3.2 Cycle Counting .................................... 25
   3.3 Real Time ......................................... 27
   3.4 Sources of Variation ............................... 27
      3.4.1 Other Processes ................................ 28
      3.4.2 Operating System ............................... 28
      3.4.3 Network Connection ............................. 28
      3.4.4 Garbage Collection ............................. 29
4 Testing Details

4.1 Environment

4.1.1 Hardware

4.1.2 Operating System

4.1.3 Compiler

4.1.4 Timing Mechanism

4.2 Test Descriptions

4.2.1 Benchmarks

4.2.2 Factors controlled

5 Results

5.1 Single-User Mode

5.2 Network Connections

5.3 Garbage Collection

5.4 Code in the Timer

6 Conclusion

References

Appendices
A Additional Data

B Programs

B.1 The Benchmarks ................................................. 56
B.2 The Benchmarking Scripts ................................. 59
B.3 A Program to Analyze the Data ............................. 75
List of Tables

<table>
<thead>
<tr>
<th></th>
<th>Table Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>TAK: Single-User Mode vs. Normal Booting</td>
<td>36</td>
</tr>
<tr>
<td>2</td>
<td>TAK-LONG: Single-User Mode vs. Normal Booting</td>
<td>36</td>
</tr>
<tr>
<td>3</td>
<td>TAK vs. TAK-LONG (Percent Standard Deviations)</td>
<td>37</td>
</tr>
<tr>
<td>4</td>
<td>TAK: Single-User Mode vs. No Daemons</td>
<td>38</td>
</tr>
<tr>
<td>5</td>
<td>TAK: Beyond Single-User Mode</td>
<td>39</td>
</tr>
<tr>
<td>6</td>
<td>TAK: Network vs. No Network</td>
<td>40</td>
</tr>
<tr>
<td>7</td>
<td>TAK with Different Amounts of Garbage Collection</td>
<td>41</td>
</tr>
<tr>
<td>8</td>
<td>CTAK with Different Amounts of Memory</td>
<td>43</td>
</tr>
<tr>
<td>9</td>
<td>CTAK with Different Amounts of Memory (Percent Standard Deviations)</td>
<td>44</td>
</tr>
<tr>
<td>10</td>
<td>TAK with Different <code>parameterize</code> Placement</td>
<td>45</td>
</tr>
<tr>
<td>11</td>
<td>TAK With and Without an Extra Parenthesis</td>
<td>46</td>
</tr>
<tr>
<td>12</td>
<td>TAK With <code>parameterize</code> Before and After Garbage Collection</td>
<td>47</td>
</tr>
<tr>
<td>13</td>
<td>TAK Using the Inaccurate Script With No Optimization</td>
<td>47</td>
</tr>
<tr>
<td>14</td>
<td>Single-User vs. No Daemons (Long Version)</td>
<td>53</td>
</tr>
<tr>
<td>15</td>
<td>Effect of <code>cron</code> on TAK</td>
<td>54</td>
</tr>
<tr>
<td>16</td>
<td>TAK with Different <code>parameterize</code> Placement (No Daemons)</td>
<td>54</td>
</tr>
</tbody>
</table>
List of Programs

1. Add5-if-positive .................................................. 14
2. call/cc Creating Problems for the Stack .......................... 19
3. Tak ............................................................... 56
4. Ctk ............................................................ 57
5. Benchmarking Script ............................................... 59
6. Benchmarking Script with Extra Parenthesis ...................... 66
7. Different parameterize Placement .................................. 67
8. Yet Another parameterize Placement ............................... 70
9. Scheme Interface for get-time ..................................... 73
10. C Interface for gettimeofday ................................. 74
11. Statistics Collection Script ..................................... 75
1 Introduction

Empirical analysis constitutes an important facet of computer science. Particularly in the field of compiler design, performance on benchmarks is often seen (for good or for ill) as the primary indicator of the worth of an optimizing compiler. Little work has been done, however, to assess the reliability and repeatability of benchmarks. It is well known that a variety of factors, (e.g., hardware, operating system, compiler, coding style) can produce significant variations in timing measurements. This paper analyzes some such factors, with the goal of determining how much variation they can introduce to a measurement (and, where possible, why they introduce such variation).

It was found that running non-idle processes in the background (e.g., cron) can increase the standard deviation of a measurement by an order of magnitude, and that garbage collection can also have a significant effect. Furthermore, the structure of (even seemingly insignificant portions of) the timer itself can have serious consequences for the accuracy of benchmarks.

The work that follows began with the intent of comparing different implementations of first-class continuations, a general and powerful control structure found in Scheme[16] and some implementations of Standard ML[3]. The traditional stack model of function call and return is inadequate when dealing with first-class continuations, and two vastly different implementation strategies have arisen to compensate.[4, 5]

\footnote{This feature also appears in some scripting languages and a “joke language” called Unlambda, but none of these languages currently have an optimizing compiler available.}
12] Some of the benchmarks used in this paper to compare the reliability of computational experiments performed under different conditions will involve first-class continuations, so a brief discussion of this feature is provided, but a complete understanding of this feature is not essential to the work that follows.
2 Background

2.1 Experimental CS

It is relatively uncommon to see experimental numbers in computer science papers. Even less common is any analysis of how accurate those numbers are believed to be. It seems that little research has been devoted to understanding the accuracy of benchmarks. Indeed, it has been argued that experimental validation in the field of computer science is both rarely done and of questionable quality.[19]

This lack of experimentation has been blamed on a mistrust of experimental results in the computer science community resulting from the knowledge that numbers in computer science experiments can vary so greatly based on factors not directly related to the object under discussion. The answer to this, of course, is not to avoid experimentation, but to understand the factors that cause such variations, eliminate them where possible, and include them in the analysis of the experimental results in cases where they cannot be completely avoided.[13]

In attempting to measure first-class continuations, however, it is relatively easy for the overhead of modern implementations to be overlooked in benchmarks (because these implementations are efficient enough that they do not necessarily create enormous differences in running time).[6] This observation led to a desire to make measurements as carefully as possible. To do this, it was necessary to find a timing method that provided as much accuracy as possible (with a small timer granularity).
2.2 Garbage Collection

Most functional programming languages (and indeed, many other modern programming languages) make use of automatic storage management, or garbage collection. In languages without garbage collection, memory to hold complex structures must be explicitly allocated and deallocated by the programmer, and errors in memory management are a common source of bugs. Languages such as Standard ML and Scheme have constructs (such as lexical closures and continuations) that would make manual memory management even more difficult, but such languages almost always provide garbage collection, which eliminates most of the application programmer’s memory management concerns.

One currently popular method of garbage collection (used in Chez Scheme and Standard ML of New Jersey) is generational copying collection. In its simplest form, copying collection makes use of two separate chunks of memory, or semi-spaces: one that is currently used, and one that is not. All allocation is done in the first semi-space, and a collection is performed when that space is nearly full. This collection begins from some root set (usually the registers in the machine and live variables in stack frames—this set generally includes all variables in the program that are currently “live” or accessible) and recursively follows any pointers, copying data to the other semi-space as it goes. Once the collector has followed every pointer in the

---

2At least this is the general perception—there seems to be a dearth of serious research on the subject.\textsuperscript{14}
3Chez Scheme is a trademark of Cadence Research Systems.
root set as far as possible, it will have copied every object in memory that can be accessed by the program to the other semi-space\(^4\). Computation may then resume with the roles of the semi-spaces reversed.[9]

Generational garbage collection adds an extra twist to copying collection\(^5\). The simplest form of generational collector has semi-spaces as above, but the roles of the semi-spaces are not reversed after each collection. Allocation continues to be done in the first space (all data in this space that is still reachable by the program has been copied to the other space, so the first space may be reused), and the next garbage collection will still copy data to the second space. No garbage collection is done on the second space until it is nearly full.[18]

Generational collection can yield huge efficiency gains in the case where many objects have only a short lifetime (which is indeed true of many functional languages, particularly Standard ML of New Jersey \(^6\)). It has been shown that generational garbage collection is useful even in non-functional languages (such as Smalltalk).[24]

### 2.3 Continuations

At the core of most programming languages is a set of control structures that give the programmer power over the order in which parts of the program are executed,

\(^4\)This has the advantage of compacting memory: the copied data will be packed tightly together, reducing cache misses when accessing that data.

\(^5\)There is no particular reason that a generational collector must be a copying collector, but most are.

\(^6\)Because of the technique described in Section 2.3.3
how many times they are to be executed, and whether they should be executed at all. These structures include the familiar if-then statements, loops, and subroutines, as well as more exotic structures (exceptions and coroutines, for example).

A natural generalization of such constructs is the continuation. The idea of the continuation can be stated fairly simply: given any point in a program’s execution, the continuation of that point is is everything that will be done after that point. In the source code for a program, one can think of a continuation being the sequence of all statements (or expressions) that will be executed after the one in question. If a programming language provides access to continuations, programmers are able to explicitly decide, for any expression they choose, the entirety of the actions to be taken after that expression.

As an example of what a continuation is, take Program 1. This Scheme code implements a simple function that will return either its argument (if that argument is negative) or a number five greater than its argument (if the argument is positive).

<table>
<thead>
<tr>
<th>Program 1 Add5-if-positive</th>
</tr>
</thead>
<tbody>
<tr>
<td>(define (add5-if-positive n)</td>
</tr>
<tr>
<td>(    (+ n 5)</td>
</tr>
<tr>
<td>n))</td>
</tr>
</tbody>
</table>
The continuation of each expression in this function can be examined. The continuation of the \( n \) in the second line is that it will be compared to zero, and returned if it is less than zero or be added to five and then have that sum returned if it is greater than zero. The continuation of the \( + \) in the third line is to be applied to \( n \) and 5, and then return the result to the calling function. The continuation of the \( n \) in line four is simply to be returned to whatever context add5-if-positive was called from.

Continuations have almost always been an important part of the Scheme programming language: the first Scheme compilers were implemented using Continuation-Passing Style, a program transformation in which all continuations are passed and invoked explicitly, so that every function finishes by passing its return-value to its continuation.\cite{23,17} Continuation-Passing Style has also proved useful as a general intermediate form for use in compilers for languages other than Scheme.\cite{2} Continuations can also be useful, however, when the programmer is given direct access to them. Scheme does this, providing programmers with the primitive call/cc operation that allows a program to capture the continuation of the call/cc expression. This feature was included in Scheme compilers with some difficulty, however, because the traditional method of storing activation records on the stack was insufficient to express the full power of continuations, and inefficient workarounds were initially used in order to emit correct code for call/cc.\cite{17}
2.3.1 Activation Records and the Stack

When a program makes a function call, it must store an activation record, which contains both information about the current state of execution, such as a return address (the address of the next instruction after the subroutine call), and information about the function call itself, such as its arguments, so that it can properly execute the function and recover after the called function returns.

A separate activation record must be stored for each function call, not merely for each function that has been defined. At least, this is true if recursive function calls are permitted, which they are in most modern languages, especially functional languages like Scheme and ML.

In most programming language implementations, activation records are pushed onto a stack at the time of a function call and popped off after that function has returned. There are several reasons for the popularity of this method.

First, the stack implicitly keeps track of the order in which function executions have taken place. As a result, it is relatively easy for so-called “stack-crawling programs” (e.g., debuggers) to examine the execution of a program by looking at progressively deeper sections of the stack.

Second, using a stack (usually residing in a chunk of pre-allocated memory) allows activation records to be deallocated with a simple operation: decrementing an address.

---

7Fortran 77, for example, did not allow recursive function calls and thus could use only a single activation record for each function defined in the source code.[22]
Third, as implied above, a stack allows recursion. An instance of a function calling itself is no different from any other function call. The appropriate information is stored on top of the stack without regard to whether other instances of that function may have been called earlier and have information stored farther down the stack, beneath the current call.

There are, however, several disadvantages in using a stack to store activation records for a modern functional language. Such languages make frequent use of closures, which may require some information about a function call to exist even after that function call has finished. In addition (and with more relevance to the current discussion), first-class continuations cannot be implemented efficiently when function calls use the traditional stack model.

2.3.2 Continuations and the Stack

Only a few languages provide continuations as first-class objects (i.e., continuations can be stored in a variable, passed to or returned from a function, and generally used in any situation where, for example, a number or a string could be used in a more traditional language). In such languages, the traditional notion of pushing activation records onto the stack when a function is called, and then popping them off when the function returns, cannot always describe computations involving first-class continuations accurately because, as will be seen shortly, activation records will no longer necessarily follow a last-in first-out ordering. Because the continuation of an
expression is determined by all the activation records on the stack when the continuation is captured, those activation records must exist for as long as the continuation is accessible. Hence no language that provides first-class continuations can rely solely on a traditional stack to maintain activation records for function calls.

Program 2 illustrates this: the continuation of the expression call/cc is stored in the local variable k, which is then returned from the function jmp-into-middle. The line after the definition of jmp-into-middle stores this continuation in the global variable c. After such a statement, the call (c c) would produce the output End 7 by returning through the remainder of the function jmp-into-middle. Note that this accesses a parameter of the particular call to jmp-into-middle that produced c, which would traditionally be stored in the stack frame for that call. Using an unmodified stack implementation, no such frame would exist after the call to jmp-into-middle returned. Also note that the call to (c c) not only returns from jmp-into-middle but also returns into the set! statement, because that call to set! is the continuation of the particular invocation of jmp-into-middle which produced the continuation that is stored in c. Thus the call (c 8) would leave c equal to 8.

2.3.3 Using the Heap

Standard ML of New Jersey (SML/NJ) is an implementation of the Standard ML language that eliminates this problem by using the heap instead of the stack to store activation records. Rather than storing information on top of a stack every time a
Program 2 A program for which the traditional stack cannot be used

(define (jmp-into-middle n)
  (printf "Begin-%n")
  (let ((k (call/cc          ; call/cc returns a function.
        (lambda (k)      ; This lambda expression creates
           k))))       ; a copy of the identity function.
    (printf "End -A-%n" n)
    k))

;; k is the identity function, but when k returns, it returns at
;; the point where call/cc was called above.

(set! c (jmp-into-middle 7))

;; c is now the function k that was returned from
;; (jmp-into-middle 7). A call to c will be like a call
;; to the identity function, except that it will return
;; into the middle of (jmp-into-middle 7).

function is called, SML/NJ allocates an activation record on the heap (e.g., using
brk), and adds it to a linked list of activation records. When a function returns,
the activation record is not “popped off,” and in fact nothing is done with the old
activation record. It is left to the garbage collection mechanism (which must be
present in any ML—and for that matter Scheme—implementation anyway) to safely
dispose of activation records that are no longer useful.[4]

This technique allows the implementation (and hence, the implementor) to avoid
dealing with issues such as determining when a closure may have unlimited extent and
need to be allocated on the heap. It also makes it simple to implement continuations
as first-class objects. When a continuation is captured, the programmer is simply
given a pointer to the appropriate activation record on the heap. As long as this pointer remains active (i.e., bound to a live identifier and potentially usable), that activation record will never be garbage collected and will remain as long as it is needed. Because each activation record is not destroyed upon returning from the call that created it, returning from a single activation record multiple times simply never becomes a problem.

Unfortunately, this approach has disadvantages as well. A stack provides good locality, and thus interacts well with current cache hierarchies, while a linked list in the heap may not [7](though there have been several recent arguments to the contrary[11, 21]). Using the heap-based strategy also means that each activation record must contain the address of the previous activation record in the list (that of the calling function), a space expense that can usually be avoided using a stack. Returning from a function involves following a pointer back to the previous activation record rather than a simple decrement, and garbage collecting activation records has costs both in additional garbage collection time and in tenuring too many objects (storing activation records on the heap will create a need for more frequent garbage collection, and in a generational collector, more frequent garbage collection results in younger objects being promoted to the less-often-collected older generations[25]). More importantly, each function invocation needs to do more work to set up an activation record on the heap (because it will need to store extra pointers, determine
whether there is sufficient heap space to allocate the new activation record, and potentially call the garbage collector or allocate more memory on the heap).

Although Appel claims that a garbage-collected, heap-based list of activation records can achieve speeds equal to (or even better than) that of the traditional stack model[1, 5], this claim depends on the hardware employed (in particular, the hardware must not be slowed down by a cache miss on a write[15]), and is, in any case, not universally accepted.[20]

2.3.4 Using a Modified Stack

As pointed out above, a traditional stack is incapable of dealing with first-class continuations. Because of this, if one wishes to have the speed and cache-locality benefits that supposedly accompany the use of a stack to store activation records, it is necessary to modify the traditional stack model.

Some implementations of Scheme copy the entire stack into another region of memory whenever a continuation is captured, and copy that memory back onto the stack when the continuation is invoked[17], but this makes both operations (continuation capture and continuation invocation) inefficient because stacks can be large and copying memory is slow. A slightly more complicated modification to the traditional stack model, however, results in an efficient implementation of call/cc.[12]

This modification inserts a marker on the stack when a continuation is captured: the return address of the top activation record on the stack is replaced by the address
of a procedure that will make a copy a relatively small portion of the stack. When the procedure in which the continuation was captured returns, rather than popping the activation record off the stack, that activation record and a few beneath it will be copied to some location in memory that will be used as a new stack for the purposes of the computation. This new stack chunk has, in place of its bottom-most activation record’s return address, the address of a procedure that will copy more activation records from the original stack, so this stack can continue to grow downward as much as necessary, while the original stack remains untouched (and ready to produce yet another copy of itself should the need arise).

This method retains for the most part the advantages of using a stack (notably, the cost of making a function call is no greater than in the traditional stack model), but it also has the advantage of making both continuation capture and continuation invocation fairly inexpensive operations. Specifically, the continuation capture will consist primarily of storing the program counter and installing a new return address in the top activation record of the stack, while continuation invocation will involve, in the long run, approximately as much copying as is necessary to maintain the proper semantics for \texttt{call/cc}. The primary disadvantage of this method is that it complicates the implementation of the compiler.

A more limited form of \texttt{call/cc} can be used without significantly altering the stack. So-called “one-shot continuations” are used in the same manner as normal continuations, but raise a runtime error if control passes through them more than
once. As a result, activation records for functions in which such continuations are captured can be popped off the stack normally because returning from the function necessitates returning through the continuation (either by explicitly invoking it or by returning through it normally). Rather surprisingly, this restriction of call/cc suffices for nearly all the common uses of continuations.[6]

2.4 Standard Deviation

Most of the measurements presented in Section 5 are presented in terms of standard deviations. Given a data sample consisting of a number of measurements, the standard deviation is an indicator of the average distance of the sample data from the mean of the sample data. Hence, the smaller the standard deviation, the more accurate the mean of the sample is as an estimator of a “truly average” measurement. The standard deviations of different samples of benchmark runs will be used throughout as a measure of their accuracy.

A good rule of thumb is that, for normal or nearly-normal distributions of data, approximately sixty-eight percent of the measurements are within one standard deviation of the mean, while ninety-five percent of the measurements are within two standard deviations of the mean. This rule is based on analysis of a truly normal distribution, but it works fairly well for most mound-shaped distributions. (The samples under consideration here are approximately “mound-shaped,” though some have multiples “mounds.”)
The standard deviation would be a reasonable way to compare the accuracy of sets of measurements that all have the same mean, but in our case, changing the conditions of an experiment in an effort to improve the accuracy (i.e., obtain a smaller standard deviation) will also alter the mean. As a result, many of the results in this paper incorporate the standard deviation as a percentage of the mean.
3 Measurement

3.1 Measuring CPU Time

The first choice to be made in any timing effort is how exactly one should time events. The obvious solution for a Chez Scheme program would be to use the built-in time function, which prints both real (wall-clock) time and CPU time, as well as memory allocation and garbage collection statistics. Unfortunately, this function cannot provide satisfyingly accurate measurements.

The time function produces time statistics in milliseconds. In fact, on the chosen platform (Linux\textsuperscript{8} on 32-bit Intel\textsuperscript{9} hardware), time produces times that are accurate only to a hundredth of a second (the length of a jiffy\textsuperscript{10} in the Linux kernel). This appears to be a fairly common problem: tests run on a server running SunOS 5.8\textsuperscript{11} show that it offers only ten-millisecond timer resolution on its “high resolution clock.”

3.2 Cycle Counting

Most modern processor architectures provide dedicated registers for the purpose of obtaining accurate performance measurements. All Intel Pentium and Pentium II processors, for example, have two 40-bit performance counters. These counters are capable of measuring the number of occurrences of a wide variety of events, from TLB

\textsuperscript{8}Linux is a registered trademark of Linus Torvalds.
\textsuperscript{9}Intel is a registered trademark of Intel Corporation.
\textsuperscript{10}the smallest unit of time used in the Linux kernel
\textsuperscript{11}SunOS is a trademark of Sun Microsystems, Inc.
misses to MMX instructions executed. For the purposes of time measurements, these counters can be set to count simply the number of clock cycles that pass while the processor is unhalted. Without special considerations, however, these counters will not be saved when interrupts occur or reset afterward.

With some modification to the operating system, this can be avoided. The Linux kernel can be patched to allow each process on the system to maintain its own “virtual performance counters” that will be unaffected by other processes and the operating system itself. These virtual performance counters have 64-bit accuracy, despite being based on 40-bit registers (this can be done by having the registers sampled with sufficient frequency). The process-specific virtual counters can be sampled from user-space without incurring the cost of an interrupt.[8]

While this method shows great promise for the future, it is not sufficiently widespread or standardized to be useful at present. Though the hardware provides dedicated performance counters, many current operating systems do not provide the means to use them for lengthy benchmarking. Without support from the operating system, reliable results cannot be obtained over a time interval that contains context switches (or at least, such results would be subject to the same variations one observes when making real time measurements).
3.3 Real Time

Wall-clock time measurements, however, can be done under Linux, and many other operating systems based on the UNIX operating system,\textsuperscript{12} using the \texttt{gettimeofday} system call. On the Pentium II processor, this call provides a 32-bit integer representing seconds and a 32-bit integer representing microseconds. Given a function to measure, times can easily be obtained by subtracting the result of a \texttt{gettimeofday} call before the function from the result of a \texttt{gettimeofday} call afterward. A simple interface was constructed to provide this functionality in Scheme.

Unfortunately, using wall-clock time allows outside influences to introduce variations into the times collected, because such a measurement (the total time between the beginning and the end of a function’s execution) does not restrict itself to the current benchmark, but takes into account everything the computer does in that period of time. The aim here is to reduce such variations to whatever extent possible. The next few subsections catalog some of the causes of these variations, and the remainder of this paper studies their significance.

3.4 Sources of Variation

Many factors can cause variations in benchmarking times. Some, such as disk I/O and network bandwidth, are not considered here. What follows deals with the variations that can occur for totally CPU bound procedures. None of the programs that will

\textsuperscript{12}UNIX is a registered trademark of the Open Group.
be timed rely on a hard drive, network connection, or any other slow device. They make use of only the processor and main memory (the command `swapoff` -a was used to ensure that these programs do not swap any pages onto the hard drive), but a number of factors will still create considerable variations in the recorded times.

### 3.4.1 Other Processes

Other processes running on the same system as the benchmark can clearly affect the benchmark’s running time. How much they do so will depend on how frequently other processes interrupt the benchmark process, and by how long such interruptions last.

### 3.4.2 Operating System

The operating system on a computer has enormous potential to interfere with time measurements. The operating system is responsible for allocating time to processes running on a machine, which can result in breaks in a benchmark’s execution that occur at seemingly random times for indeterminate amounts of time.

### 3.4.3 Network Connection

A network connection that is in use by some other process on the machine will create overhead for that process that will in turn affect the running time of the benchmark, but even a network connection that is not explicitly in use can cause unexpected interrupts resulting from incoming traffic. How much effect this has on timing is
closely tied to the efficiency with which the operating system and the network card
handle traffic, particularly broadcast traffic, as well as the nature of the particular
network in question.

3.4.4 Garbage Collection

Garbage collection (a feature of all languages currently offering first-class continu-
actions) can also skew times. If one execution of a benchmark runs the garbage collector
an extra time (compared to the number of times the garbage collector is run during
a different execution of the benchmark), that execution time will be longer by an
amount that may be hard to determine. Garbage collection can also interact in un-
predictable ways with the memory hierarchy and virtual memory subsystem. Some
copying garbage collectors exhibit poor write locality, resulting in an unpredictable
number of cache misses. Unfortunately, as an inherent part of the process being
timed, it is more difficult to eliminate than operating system or network interference.

3.4.5 Timing Environment

Rather surprisingly, the results presented will show that seemingly innocuous alter-
ations to the procedure used to do the timings can introduce enormous variations
into the running times. A proper explanation of this does not appear anywhere in
this paper, but it seems likely that these variations result from some compiler opti-
mization.
4 Testing Details

4.1 Environment

In reporting the results of an experiment, it is imperative that the conditions under which the experiment was conducted are made clear. This should allow others to reproduce the result (or fail in reproducing it), which is essential to scientific study.

4.1.1 Hardware

Except where otherwise noted, times were obtained on a single-processor Intel Pentium II 300 MHz with 64 megabytes of RAM.\footnote{\textsuperscript{13}Intel, Pentium, and Pentium II are registered trademarks of Intel Corporation.}

4.1.2 Operating System

Timing was performed under a Red Hat 7.2 Linux system.\footnote{\textsuperscript{14}Red Hat is a trademark of Red Hat, Inc.} The kernel and system libraries were unmodified. The modules normally running were: sr\_mod, sg, binfo\_misc, ipchains, ide\_scsi, scsi\_mod, ide\_cd, cdrom, sb sb\_lib, uart\_401, sound, soundcore, usb\_uhci, usbcore, ext3, and jbd. Interrupts on the system normally were: timer, keyboard, cascade, soundblaster, rtc, usb\_uhci, PS/2 Mouse, ide0, ide1.
4.1.3 Compiler

The compiler used for all timings in this paper was Chez Scheme. This is an aggressively optimizing compiler that exemplifies one approach to implementing first-class continuations.

4.1.4 Timing Mechanism

All times were determined using a real-time clock (i.e., it included not only the CPU time used by the process in question, but also time taken by the operating system and other processes) with microsecond granularity. In Chez Scheme, this was implemented using a simple interface to the `gettimeofday` system call. Although Chez Scheme implements a native `time` function that reports CPU time used by the application (as opposed to that used by the operating system), this function offers only millisecond granularity on the system that was used. The complete timing mechanism can be found in Appendix B.

4.2 Test Descriptions

4.2.1 Benchmarks

In order to determine the significance of the variations produced by the sources above, tests were conducted using two standard Scheme benchmarks: Tak and Ctak. Tak implements Richard Gabriel’s variant of the triply-recursive Takeuchi function[10],
while Ctk implements a version of Tak that uses first-class continuations in place of recursive function calls. Tak was actually run with two different sets of arguments in this study. Hereafter, the name TAK will be used to refer to the Tak benchmark run with the arguments 22, 9, and 23, which took between one and two seconds to run on the hardware used. The name TAK-LONG will refer to the Tak benchmark run with the arguments 22, 9, and 25, which took just over ten seconds to run on the hardware used. For the sake of uniformity, the name CTAK will be will be used to discuss the specific instance of the Ctk benchmark that was run here.

The Tak benchmark heavily tests function call and return mechanisms, and also includes some arithmetic on small integers and one comparison per function call. Tak has the useful property that it does not use heap-allocated memory in Chez Scheme. This provides a benchmark that can be run without the garbage collector. Ctk, on the other hand, puts a heavy strain on the garbage collector. In Chez Scheme, heavy use of continuations produces many large stack chunks, resulting in frequent garbage collection.

4.2.2 Factors controlled

It is commonly assumed that interference from other processes can cause inaccuracies in benchmarking (particularly when times are obtained using a real-time clock). Taking this a step further, even background processes and some of the operating system’s bookkeeping tasks can be eliminated. To this end, the TAK and TAK-LONG
benchmarks were run on a system that was booted normally (with and without cer-
tain services running), a system in single-user mode, a system in single-user mode running only a minimal set of modules, and a system booted with the argument
init=/bin/sh. Comparing these number should yield some insight into the amount
of variation that is due to various parts of the operating system.

Likewise, variations can clearly result from a network connection. To study this,
numbers were collected on a machine that was connected to a network (though not
explicitly communicating with another computer) and on the same machine with the
network connection disabled. Clearly the results of this comparison depend upon the
specific network to which the computer is connected, but it is hoped that this will
at least provide some information about the level of variation that can result merely
from being connected to a fairly active network.

During another set of runs, garbage collection was done between each timing. Per-
forming this additional garbage collection helps to ensure that each run of a bench-
mark starts under memory conditions that are, as much as possible, identical to those
under which other runs began. This may seem unlikely to affect the accuracy of a
benchmark like Tak, which, under Chez Scheme, never uses any heap memory, but
compacting garbage collection could potentially have beneficial interactions with the
virtual memory subsystem.

Running garbage collection during executions is a necessity for Ctak, but it can
be eliminated for Tak (at least in Chez Scheme). The Tak benchmark will never
actually perform any garbage collection, so one would expect that having garbage collection turned on or off would not affect the accuracy of the benchmark, but garbage collection was explicitly turned off during some runs to determine whether this was actually true.

While Ctk needs garbage collection (a great deal) under most circumstances, it should be able to avoid garbage collection given a sufficiently large heap to begin with. In an effort to determine how much of the variation in Ctk was due to garbage collection, CTAK was run with the Chez Scheme parameter `collect-trip-bytes` set to both the default value (1048576 bytes) and to 40 megabytes (resulting in a program that was able to use much more memory before running the garbage collector).
5 Results

What follows is a discussion of the experimental results that were obtained. These results are presented in a number of tables. In each of these tables, except where explicitly noted, each line in the table represents either TAK or CTAK run 1000 times with a separate 50-run warm up that was not included in the calculations that are presented. The warm-up was intended to ensure that as much of the timing mechanism as possible was in cache. Measurements were made on the first system described in Section 4.1 on page 30, and all values are rounded to the nearest microsecond.

5.1 Single-User Mode

The first issue to be addressed was the effect of the operating system and other running processes. As can be seen in Table 1, the difference in accuracy between single-user mode and a normal boot-up is significant. The sample standard deviations of measurements made under single-user mode are smaller than those made under normal running conditions by between one and two orders of magnitude, and the difference in the means is insufficient to justify this by itself.

It should also be noted that the standard deviation of the times collected in single-user mode under these conditions does not appear to increase as the mean runtime increases. Table 2 contains the results of measurements done using arguments to TAK that resulted in a running time almost ten times that measured in Table 1. Despite
Table 1 TAK: Single-User Mode vs. Normal Booting: TAK with full optimization turned on, garbage collection between each run, and no garbage collection during individual runs in both single-user and normal modes (with no network connection); each sample contains 1000 executions of TAK

<table>
<thead>
<tr>
<th>Single-User Mode</th>
<th>Mean</th>
<th>Standard Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1161108μs</td>
<td>98μs</td>
</tr>
<tr>
<td></td>
<td>1161111μs</td>
<td>98μs</td>
</tr>
<tr>
<td></td>
<td>1161113μs</td>
<td>96μs</td>
</tr>
<tr>
<td></td>
<td>1161111μs</td>
<td>97μs</td>
</tr>
<tr>
<td></td>
<td>1161114μs</td>
<td>98μs</td>
</tr>
<tr>
<td>Normal Mode</td>
<td>Mean</td>
<td>Standard Deviation</td>
</tr>
<tr>
<td></td>
<td>1163961μs</td>
<td>7302μs</td>
</tr>
<tr>
<td></td>
<td>1170785μs</td>
<td>9844μs</td>
</tr>
<tr>
<td></td>
<td>1161436μs</td>
<td>1633μs</td>
</tr>
<tr>
<td></td>
<td>1161383μs</td>
<td>90μs</td>
</tr>
<tr>
<td></td>
<td>1171755μs</td>
<td>30518μs</td>
</tr>
</tbody>
</table>

Table 2 TAK-LONG: Single-User Mode vs. Normal Booting: TAK-LONG with full optimization turned on, garbage collection between each run, and no garbage collection during individual runs in both single-user and normal modes (with no network connection); each sample contains 1000 executions of TAK-LONG

<table>
<thead>
<tr>
<th>Single-User Mode</th>
<th>Mean</th>
<th>Standard Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10986470μs</td>
<td>124μs</td>
</tr>
<tr>
<td></td>
<td>10986484μs</td>
<td>122μs</td>
</tr>
<tr>
<td></td>
<td>10986372μs</td>
<td>93μs</td>
</tr>
<tr>
<td></td>
<td>10986392μs</td>
<td>95μs</td>
</tr>
<tr>
<td></td>
<td>10986408μs</td>
<td>94μs</td>
</tr>
<tr>
<td>Normal Mode</td>
<td>Mean</td>
<td>Standard Deviation</td>
</tr>
<tr>
<td></td>
<td>10998123μs</td>
<td>55596μs</td>
</tr>
<tr>
<td></td>
<td>11025195μs</td>
<td>412456μs</td>
</tr>
<tr>
<td></td>
<td>10986685μs</td>
<td>105129μs</td>
</tr>
<tr>
<td></td>
<td>11020832μs</td>
<td>434201μs</td>
</tr>
<tr>
<td></td>
<td>10986656μs</td>
<td>8282μs</td>
</tr>
</tbody>
</table>

the much higher running times, the standard deviations are still very nearly the same as those in the previous data set.

This means, of course, that the standard deviation of the longer runs is much less significant when compared to the size of the mean. In Table 3, the percent standard deviations of the long and short runs are compared. In the longer runs, the standard deviation is a smaller percentage of the mean by almost a factor of ten because the
Table 3 TAK vs. TAK-LONG (Percent Standard Deviations): A comparison of the percent standard deviations for TAK and TAK-LONG with garbage collection between each run, and no garbage collection during individual runs in both single-user and normal modes (with no network connection); each sample contained 1000 executions.

<table>
<thead>
<tr>
<th></th>
<th>Single-User Mode</th>
<th>Normal Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>TAK</td>
<td>TAK-LONG</td>
<td>TAK</td>
</tr>
<tr>
<td>0.008832%</td>
<td>0.001128%</td>
<td>0.627310%</td>
</tr>
<tr>
<td>0.008659%</td>
<td>0.001109%</td>
<td>0.840763%</td>
</tr>
<tr>
<td>0.008832%</td>
<td>0.000843%</td>
<td>0.140566%</td>
</tr>
<tr>
<td>0.008924%</td>
<td>0.000864%</td>
<td>0.007707%</td>
</tr>
<tr>
<td>0.009026%</td>
<td>0.000857%</td>
<td>2.604448%</td>
</tr>
</tbody>
</table>

standard deviation has not changed, while the mean has increased by nearly a factor of ten.

Further investigation indicates that similar accuracy can be obtained simply by shutting down all unnecessary services and daemons with no need to boot into single-user mode (see Table 4). Here, the services shut off include cron, sshd, and gpm, of which only cron is likely to create very large variations (the system is not connected to a network, so sshd should simply sleed, while gpm will respond to the occasional mouse event, but probably has little effect—no experiments were done to verify this, however). Much of the variation introduced by cron in Table 1 results from the fact the cron sometimes started other processes, but other results (Table 15 on page 54) indicate that cron can adversely affect accuracy even when it does not run anything else.
Table 4 TAK: Single-User Mode vs. No Daemons: TAK with full optimization turned on, garbage collection between each run, and no garbage collection during individual runs in both single-user mode and normal mode with all standard services (in this case `cron`, `sshd`, and `gpm`) disabled (with no network connection); each sample contains 1000 executions of TAK.

<table>
<thead>
<tr>
<th>Cron et al. turned off</th>
<th>Single-User Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>Standard Deviation</td>
</tr>
<tr>
<td>1161385μs</td>
<td>91μs</td>
</tr>
<tr>
<td>1161384μs</td>
<td>88μs</td>
</tr>
<tr>
<td>1161385μs</td>
<td>90μs</td>
</tr>
<tr>
<td>1161385μs</td>
<td>91μs</td>
</tr>
<tr>
<td>1161390μs</td>
<td>90μs</td>
</tr>
</tbody>
</table>

In practice, however, it may be difficult to determine which services are introducing variations into the benchmarking, and the most reliable (and perhaps simplest) way to ensure that no unnecessary processes are running is to boot into single-user mode.

It is interesting to note that the mean times in Table 4 are slightly shorter than the times under single-user mode in Table 1. While it seems as if benchmarks run under single-user mode should run faster, it appears that this intuition is incorrect. Because this result is contrary to common intuition, Tak was run under these conditions several additional times, and those runs performed under single-user mode were consistently slightly slower than the others (see Table 14 on page 53). No explanation for this is immediately apparent (perhaps some feature of the Linux kernel causes extra overhead in single-user mode).

On the other hand, it seems difficult to improve on the approximately 100μs standard deviation achieved with single-user mode. Two additional methods beyond
Table 5 TAK: Beyond Single-User Mode: Percent standard deviations of TAK with full optimization turned on, garbage collection between each run, and no garbage collection during individual runs in “normal” single-user mode, as well as single-user mode with most modules turned off and with the boot-time argument `init=-/bin/sh` (with no network connection); each sample contains 1000 executions of TAK.

<table>
<thead>
<tr>
<th>Single-User Mode</th>
<th>Minimal Modules</th>
<th>With <code>init=-/bin/sh</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>0.008832%</td>
<td>0.008464%</td>
<td>0.007641%</td>
</tr>
<tr>
<td>0.008659%</td>
<td>0.008750%</td>
<td>0.007531%</td>
</tr>
<tr>
<td>0.008832%</td>
<td>0.008699%</td>
<td>0.007761%</td>
</tr>
<tr>
<td>0.008924%</td>
<td>0.008391%</td>
<td>0.007658%</td>
</tr>
<tr>
<td>0.009026%</td>
<td>0.008211%</td>
<td>0.007493%</td>
</tr>
</tbody>
</table>

simply running in single-user mode were explored: removing modules from the running kernel image, and booting the kernel with the additional option `init=-/bin/sh`.

Table 5 shows that, while each of these options may offer some improvement over the standard deviations found using only single-user mode, neither yields a drastic improvement.

5.2 Network Connections

Another potential source of variation is a network connection. Even if no process on the machine is actively using the network connection, the mere act of being on a network may introduce variations into program runtimes. For example, the benchmarking machine may need to deal with broadcast traffic that is not intended for it, resulting in context switches with potentially large overheads.
Table 6 TAK: Network vs. No Network: Percent standard deviations for TAK with full optimization turned on, garbage collection between each run, and no garbage collection during individual runs in single-user mode both with an active network connection and without; each sample contains 1000 executions of TAK

<table>
<thead>
<tr>
<th>Network Off</th>
<th>Network On</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.032372%</td>
<td>0.074705%</td>
</tr>
<tr>
<td>0.017926%</td>
<td>0.070292%</td>
</tr>
<tr>
<td>0.033955%</td>
<td>0.334129%</td>
</tr>
<tr>
<td>0.031395%</td>
<td>0.071142%</td>
</tr>
<tr>
<td>0.034159%</td>
<td>0.072130%</td>
</tr>
</tbody>
</table>

Table 6 shows that merely being part of a real network can more than double the standard deviation of a benchmark’s running time relative to its mean, but the effect of a network connection on benchmarking will be highly dependent upon the individual network on which the machine finds itself. The above numbers should therefore be taken with a grain of salt, but they do demonstrate the severity of variations that can be introduced by an active but unused network connection. Furthermore, the very fact that the variations introduced by a network connection depend on the specific network acts as an argument not to have any network connection active while conducting experiments. Repeatability is a hallmark of good experimentation, and network conditions may be difficult to reproduce not only on different networks, but even on the same network at different times.

15This data was collected on a 1 gigahertz AMD Athlon processor with 512 megabytes of RAM running Linux 2.4.18rc4 specifically compiled for the AMD Athlon. AMD Athlon is a trademark of Advanced Micro Devices, Inc.
### Table 7 TAK with Different Amounts of Garbage Collection

Percent standard deviations of TAK in single-user mode (no network connection) with full optimization turned on, and different amounts of garbage collection; each sample contains 1000 executions of TAK.

<table>
<thead>
<tr>
<th>None At All</th>
<th>During Runs Only</th>
<th>Between Runs Only</th>
<th>Between and During</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.007934%</td>
<td>0.008840%</td>
<td>0.008832%</td>
<td>0.008523%</td>
</tr>
<tr>
<td>0.009136%</td>
<td>0.008689%</td>
<td>0.008659%</td>
<td>0.009040%</td>
</tr>
<tr>
<td>0.008310%</td>
<td>0.008701%</td>
<td>0.008832%</td>
<td>0.008422%</td>
</tr>
<tr>
<td>0.008049%</td>
<td>0.008606%</td>
<td>0.008924%</td>
<td>0.008735%</td>
</tr>
<tr>
<td>0.007868%</td>
<td>0.008695%</td>
<td>0.009026%</td>
<td>0.008366%</td>
</tr>
</tbody>
</table>

### 5.3 Garbage Collection

It was mentioned in Section 4.2.2 on page 33 that some consideration was given to the potential effects of garbage collection on the Tak benchmark. Although Tak should not perform any garbage collection under Chez Scheme, running a thorough garbage collection before each Tak execution and explicitly turning off garbage collection during Tak runs may have beneficial results. To explore this, the following measurements were made: Tak with no garbage collection at all during the entire set of a thousand runs, Tak with garbage collection on (though not used) during each run, but without additional garbage collection between runs, Tak with garbage collection between each run, and Tak with garbage collection between each run but with garbage collection explicitly turned off during runs. Table 7 presents the results of these experiments. None of these four approaches to garbage collection in seem to have an effect on the TAK benchmark, in which no garbage collection is done.
A program like TAK can be run without any garbage collection (and garbage collection seems to have little if any effect on it), thus avoiding any variation that garbage collection might introduce into the times collected for such a program. Most programs in modern functional languages rely heavily on garbage collection, however, and not performing garbage collection during a benchmarking run will likely crash the program or result in thrashing the virtual memory subsystem, which may introduce enormous variations into the times collected.

The CTAK program, for instance, requires garbage collection during each run. This naturally means that a complete garbage collection should be performed between each run, so that individual runs start with as similar a memory configuration as possible. (Otherwise, a different number of garbage collections could be performed during different runs, resulting in two subsets of the sample that may have very different means.) Doing this before the first run will also put objects used by the timing system itself into the highest generation, so that they need not be considered by most garbage collection performed during an individual run.

For programs in which garbage collection must be done, it is often desirable to include the cost of garbage collection in the measured running time for the benchmark. There seems to be no simple way to do this without significant variations resulting from the garbage collection. On the other hand, if the garbage collection is considered irrelevant, the variation associated with it can be reduced by increasing the amount of memory that the program will use before doing a garbage collection.
Table 8 CTAK with Different Amounts of Memory: CTAK in single-user mode (no network connection) with full optimization turned on, and full garbage collection between runs. Different amounts of memory allocation permitted between garbage collections. With 1 megabyte, CTAK performed 454 garbage collections, while with 40 megabytes, CTAK performed 13; each sample contains 1000 executions of CTAK

<table>
<thead>
<tr>
<th>Mean (µs)</th>
<th>Mean (µs)</th>
<th>Standard Deviation (µs)</th>
<th>Standard Deviation (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6412058</td>
<td>6213977</td>
<td>13092</td>
<td>3600</td>
</tr>
<tr>
<td>6433178</td>
<td>6213455</td>
<td>11484</td>
<td>3430</td>
</tr>
<tr>
<td>6417742</td>
<td>6213116</td>
<td>11923</td>
<td>3390</td>
</tr>
<tr>
<td>6411722</td>
<td>6213104</td>
<td>12498</td>
<td>3385</td>
</tr>
<tr>
<td>6425532</td>
<td>6213056</td>
<td>11000</td>
<td>3388</td>
</tr>
</tbody>
</table>

Obviously, if the process can be given enough memory that garbage collection is no longer necessary, this will eliminate any inaccuracy resulting from garbage collection. Allowing the process to use more memory, however, can be an effective way to reduce garbage collection inaccuracy even if some garbage collection must still be done. Table 8 provides a comparison of CTAK under Chez Scheme with the normal collect-trip-bytes and CTAK under Chez Scheme with collect-trip-bytes set to forty megabytes. It should be noted here that, for this discussion, one megabyte has the usual meaning of 1,048,576 (two raised to the twentieth power) bytes, while 40 megabytes means exactly 40,000,000 bytes. There is, of course, no reason to think that 41,943,040 bytes would produce significantly different results from 40,000,000 bytes.

This is one example in which it is important to pay attention to the mean as well as the standard deviation. Allowing CTAK to use 40 megabytes of memory between
Table 9 CTAK with Different Amounts of Memory (Percent Standard Deviations): Standard deviations as percent of the mean for CTAK in single-user mode (no network connection) with full optimization turned on, and full garbage collection between runs, and different amounts of memory allocation permitted between garbage collections; with 1 megabyte, CTAK performed 454 garbage collections, while with 40 megabytes, CTAK performed 13; each sample contains 1000 executions of CTAK.

<table>
<thead>
<tr>
<th>One Megabyte</th>
<th>40 Megabytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.204177%</td>
<td>0.057928%</td>
</tr>
<tr>
<td>0.178513%</td>
<td>0.055200%</td>
</tr>
<tr>
<td>0.185778%</td>
<td>0.054562%</td>
</tr>
<tr>
<td>0.194931%</td>
<td>0.054483%</td>
</tr>
<tr>
<td>0.171199%</td>
<td>0.054529%</td>
</tr>
</tbody>
</table>

garbage collections reduces the mean by nearly two-tenths of a second. This is to be expected since the garbage collector will run less frequently, but, because of the structure of the CTAK program, will not necessarily do more work per collection. This may, however, cause one to question whether the reduction in standard deviations simply corresponds to the reduction in the means. This is not the case: Table 9 shows that ratio of standard deviations to means is smaller for those runs that were allowed to use 40 megabytes at a time.

5.4 Code in the Timer

The script used to run these benchmarks (Program 5 on page 59) originally contained a typographical error. The closing parenthesis of the call to parameterize occurred too early—after the value of collect-trip-bytes is printed—and does not include the actual running of the benchmark.
Table 10 TAK with Different parameterize Placement: TAK in single-user mode (no network connection) with full optimization turned on, and full garbage collection between runs, slightly different timing scripts. Each sample contains 1000 executions of TAK.

<table>
<thead>
<tr>
<th>Script with parameterize Ended Early</th>
<th>parameterize inside micro-timer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>Standard Deviation</td>
</tr>
<tr>
<td>1161416 µs</td>
<td>103 µs</td>
</tr>
<tr>
<td>1161417 µs</td>
<td>101 µs</td>
</tr>
<tr>
<td>1161414 µs</td>
<td>103 µs</td>
</tr>
<tr>
<td>1161413 µs</td>
<td>104 µs</td>
</tr>
<tr>
<td>1161409 µs</td>
<td>105 µs</td>
</tr>
</tbody>
</table>

It was noticed early on that this script, when told to print out garbage collection messages, printed those messages to the screen rather than into the file. Somehow, the actual source of the problem was not immediately noticed, and this bug was solved by adding an extra parameterize call into the micro-timer macro, so that console output was redirected immediately before the garbage collection that was done before the first call to get-time in each run (a unified diff of these two files appears on page 67). Rather surprisingly, this simple change caused enormous variations in the times collected. Table 10 compares the results of running TAK under single-user mode with the original script with the results of running it with the “fixed” script. (Similar results occurred with a normal boot-up and most services disabled. These results can be found in Table 16 on page 54.)

\[16\] The diff makes this change look much larger than it actually was because all of the indentation in micro-timer following parameterize was changed.
Table 11 TAK With and Without an Extra Parenthesis: TAK in single-user mode (no network connection) with full optimization turned on, and full garbage collection between runs, slightly different timing scripts. Each sample contains 1000 executions of TAK.

<table>
<thead>
<tr>
<th>Script with Extra Parenthesis</th>
<th>Script with the Extra Parenthesis Deleted</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>Standard Deviation</td>
</tr>
<tr>
<td>1161416 µs</td>
<td>103 µs</td>
</tr>
<tr>
<td>1161417 µs</td>
<td>101 µs</td>
</tr>
<tr>
<td>1161414 µs</td>
<td>103 µs</td>
</tr>
<tr>
<td>1161413 µs</td>
<td>104 µs</td>
</tr>
<tr>
<td>1161409 µs</td>
<td>105 µs</td>
</tr>
</tbody>
</table>

Table 10 shows that the placement of the parameterize call in the code for the timing script had an effect not only on the standard deviation, but also on the mean. This effect cannot possibly be attributed to anything other than the timing script, so it must be assumed that the insertion of this additional parameterize is responsible for an enormous amount of inaccuracy.

This problem was later fixed by moving parameterize back to its original position and deleting the extraneous parenthesis (a unified diff for this change appears on page 66). This seemed to have little effect on the means of the numbers collected (it actually made them slightly smaller), and no noticeable effect on the standard deviations. Table 11 shows the times for TAK with and without the additional parenthesis.

Furthermore, it was discovered that the results obtained using a script with the parameterize between the garbage collection and the first call to get-time are similar to the results obtained with the incorrect parameterize placement (and with the one-parenthesis-deletion correction). This is shown in Table 12. Therefore, the cause
Table 12 TAK With \texttt{parameterize} Before and After Garbage Collection: TAK in single-user mode (no network connection) with full optimization turned on, full garbage collection between runs, and \texttt{parameterize} both before and after the garbage collection; each sample contains 1000 executions of TAK.

<table>
<thead>
<tr>
<th>\texttt{parameterize} Before GC</th>
<th>\texttt{parameterize} After GC</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{Mean}</td>
<td>\texttt{Standard Deviation}</td>
</tr>
<tr>
<td>1199258,\mu s</td>
<td>22467,\mu s</td>
</tr>
<tr>
<td>1199539,\mu s</td>
<td>22532,\mu s</td>
</tr>
<tr>
<td>1198941,\mu s</td>
<td>22287,\mu s</td>
</tr>
<tr>
<td>1199367,\mu s</td>
<td>22406,\mu s</td>
</tr>
<tr>
<td>1199198,\mu s</td>
<td>22372,\mu s</td>
</tr>
</tbody>
</table>

Table 13 TAK Using the Inaccurate Script With No Optimization: TAK in single-user mode (no network connection) with no optimization at all, and full garbage collection between runs, using both the correct script and the inaccurate script with the \texttt{parameterize} call before the garbage collection in micro-timer; each sample contains 1000 executions of TAK.

<table>
<thead>
<tr>
<th>\texttt{Good Script}</th>
<th>\texttt{Inaccurate Script}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{Mean}</td>
<td>\texttt{Standard Deviation}</td>
</tr>
<tr>
<td>2058008</td>
<td>104</td>
</tr>
<tr>
<td>2058015</td>
<td>101</td>
</tr>
<tr>
<td>2058007</td>
<td>105</td>
</tr>
<tr>
<td>2058016</td>
<td>101</td>
</tr>
<tr>
<td>2058010</td>
<td>97</td>
</tr>
</tbody>
</table>

of the inaccuracy in the numbers collected using the script in which \texttt{parameterize} appears before the garbage collection seems as if it must stem from some interaction between the \texttt{parameterize} and the garbage collection.

One clue to this behavior appears when both the corrected script and the script with the \texttt{parameterize} before the garbage collection are run with no optimization at all (that is, \texttt{optimize-level} is set to zero and all the \texttt{cp0} parameters that control partial evaluation are also zero). In this case, both scripts produce results with a low
standard deviation (see Table 13), but the script with the `parameterize` before the garbage collection runs somewhat slower (nearly a 10% decrease in speed) than the script with the `parameterize` in the `micro-timer` macro. It seems possible that some optimization was introduced with the intent of improving this situation in the former case (which it does), but that this optimization somehow produces great variation in running times.
6 Conclusion

Several sources of variations in the measurement of the running time of Tak and Ctak under Chez Scheme have been examined. One potential source of variation, the operating system, seems to have little if any effect. The results seem to indicate that almost all of the inaccuracy that may seem to result from the operating system comes instead from other processes (which the operating system may start automatically at boot time).

Other sources of variation can be eliminated completely using obvious methods. A network connection, for example, can simply be removed (provided it is not integral to the benchmark), and interference from other processes can be minimized relatively easily by shutting down those processes. Garbage collection, however, may easily be the subject (or at least a significant part) of measurements being made, so that eliminating the variation it introduces would necessarily skew the results of the measurement. Even in cases where garbage collection is not the subject of the experiment, it may be an unavoidable part of the language or environment.

Finally, there is rather startling evidence that something as simple and natural as redirecting the output of an experiment to a file from within the timing script can introduce significant variations if not done carefully and with some knowledge of the inner workings of the compiler or interpreter. It appears that any timing script requires testing to ensure that it is not introducing variations on its own.
This underscores the importance of testing any environment before using it to make measurements, and of testing it again after even a slight change.
References


Appendices

A Additional Data

Table 14 Single-User vs. No Daemons (Long Version): TAK with full optimization turned on, garbage collection between each run, and no garbage collection during individual runs in both single-user mode and normal mode with all standard services (in this case `cron`, `sshd`, and `gpm`) disabled (with no network connection), including all times recorded (because they consistently run against the expectation that benchmarks should run faster in single-user mode); each sample contained 1000 executions of TAK.

<table>
<thead>
<tr>
<th>Cron et al. turned off</th>
<th>Single-User Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>Percent Std. Dev.</td>
</tr>
<tr>
<td>1161416 µs</td>
<td>0.008832%</td>
</tr>
<tr>
<td>1161410 µs</td>
<td>0.008234%</td>
</tr>
<tr>
<td>1161411 µs</td>
<td>0.008676%</td>
</tr>
<tr>
<td>1161417 µs</td>
<td>0.008659%</td>
</tr>
<tr>
<td>1161414 µs</td>
<td>0.008832%</td>
</tr>
<tr>
<td>1161413 µs</td>
<td>0.008924%</td>
</tr>
<tr>
<td>1161409 µs</td>
<td>0.009026%</td>
</tr>
<tr>
<td>1161414 µs</td>
<td>0.008201%</td>
</tr>
<tr>
<td>1161413 µs</td>
<td>0.008065%</td>
</tr>
<tr>
<td>1161412 µs</td>
<td>0.008581%</td>
</tr>
<tr>
<td>1161405 µs</td>
<td>0.008175%</td>
</tr>
</tbody>
</table>
Table 15 Effect of `cron` on TAK: TAK with full optimization turned on, garbage collection between each run, and no garbage collection during individual runs in normal mode with `cron` running starting new processes, running but doing nothing, or not running (with no network connection); each sample number represents the percent standard deviation of 1000 executions of TAK

<table>
<thead>
<tr>
<th>cron Running other Processes</th>
<th>cron by Itself</th>
<th>No cron</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.122912%</td>
<td>0.008336%</td>
<td>0.007835%</td>
</tr>
<tr>
<td>0.130855%</td>
<td>0.071369%</td>
<td>0.007598%</td>
</tr>
<tr>
<td>0.022838%</td>
<td>0.007971%</td>
<td>0.007310%</td>
</tr>
<tr>
<td>0.108312%</td>
<td>0.008346%</td>
<td>0.007634%</td>
</tr>
<tr>
<td>0.131241%</td>
<td>0.070876%</td>
<td>0.007934%</td>
</tr>
</tbody>
</table>

Table 16 TAK with Different parameterize Placement (No Daemons): TAK with full optimization turned on, garbage collection between each run, and no garbage collection during individual runs in normal mode with all standard services (in this case `cron`, `sshd`, and `gpm`) disabled (with no network connection), run using both the original benchmarking script and the script with `parameterize` inside `micro-timer`; each sample contained 1000 executions of TAK

<table>
<thead>
<tr>
<th>Original Script</th>
<th>Script with <code>parameterize</code> in <code>micro-timer</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>Mean</td>
</tr>
<tr>
<td>1161385(\mu s)</td>
<td>1264807(\mu s)</td>
</tr>
<tr>
<td>1161384(\mu s)</td>
<td>1248572(\mu s)</td>
</tr>
<tr>
<td>1161385(\mu s)</td>
<td>1248558(\mu s)</td>
</tr>
<tr>
<td>1161385(\mu s)</td>
<td>1248586(\mu s)</td>
</tr>
<tr>
<td>1161390(\mu s)</td>
<td>1248570(\mu s)</td>
</tr>
<tr>
<td>Standard Deviation</td>
<td>Standard Deviation</td>
</tr>
<tr>
<td>91(\mu s)</td>
<td>163428(\mu s)</td>
</tr>
<tr>
<td>88(\mu s)</td>
<td>76089(\mu s)</td>
</tr>
<tr>
<td>90(\mu s)</td>
<td>76092(\mu s)</td>
</tr>
<tr>
<td>91(\mu s)</td>
<td>76099(\mu s)</td>
</tr>
<tr>
<td>90(\mu s)</td>
<td>76090(\mu s)</td>
</tr>
</tbody>
</table>
B Programs

In order for computer science experiments to be repeatable the source code for programs used in the experiment must be made available to whatever extent possible. This allows the code to be re-run by others and scrutinized for bugs. In the pages that follow, one can find the benchmarks used, the benchmarking script that drives them (including a number of variations and the timing code), and a script used to extract statistical information about sample data. (Some of these programs have been modified to fit the page.) Chez Scheme, the compiler used in all the measurements, is a commercial product, and it is not possible to produce the source code here. Information about Chez Scheme can be obtained from Cadence Systems (http://www.scheme.com), and information about (and source code for) the Linux kernel included in Red Hat Linux 7.2 is available Red Hat Inc. (http://www.redhat.com).
B.1 The Benchmarks

Program 3 Tak

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; File: tak.sch
; Description: TAK benchmark from the Gabriel tests
; Author: Richard Gabriel
; Created: 12-Apr-85
; Modified: 12-Apr-85 09:58:18 (Bob Shaw)
; 22-Jul-87 (Will Clinger)
; 23-Jun-02 (Kent Hunter)
; Language: Scheme
; Status: Public Domain
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;; TAK -- A vanilla version of the TAKEuchi function
;;;; This defines a lexical scope in which 'tak' must
;;;; keep a constant definition, allowing efficient
;;;; inlining optimizations.
(module (tak)
  (define (my-tak x y z)
    (if (not (< y x))
      z
      (my-tak (my-tak (- x 1) y z)
        (my-tak (- y 1) z x)
        (my-tak (- z 1) x y)))
  (define (tak x y z)
    (my-tak x y z))
)
Program 4 Ctak

File: ctak.sch
Description: The ctak benchmark
Author: Richard Gabriel
Created: 5-Apr-85
Modified: 10-Apr-85 14:53:02 (Bob Shaw)
24-Jul-87 (Will Clinger)
5-Jul-02 (Kent Hunter)
Language: Scheme
Status: Public Domain

; The original version of this benchmark used a continuation
; mechanism that is less powerful than call/cc and also
; relied on dynamic binding, which is not provided in standard
; Scheme. Since the intent of the benchmark seemed to be to
; test non-local exits, the dynamic binding has been replaced
; here by lexical binding.

; For Scheme the comment that follows should read:
;;; CTAK -- A version of the TAK procedure that uses continuations.

;;; CTAK -- A version of the TAK function that uses the
;;; CATCH/THROW facility.

(module (ctak)
  (define (ctak x y z)
    (call-with-current-continuation
     (lambda (k)
      (ctak-aux k x y z))))

  (define (ctak-aux k x y z)
    (cond ((not (< y x)); xy
      (k z))
      (else (call-with-current-continuation
        (ctak-aux
         k
         (call-with-current-continuation
          (lambda (k)
(ctak-aux k
   (- x 1)
   y
   z))
(call-with-current-continuation
 (lambda (k)
   (ctak-aux k
      (- y 1)
      z
      x)))
(call-with-current-continuation
 (lambda (k)
   (ctak-aux k
      (- z 1)
      x
      y)))))))))))
B.2 The Benchmarking Scripts

The final, correct script appears first here. All of the diffs appearing later are between this correct version and some other version.

Program 5 Benchmarking Script

```bash
#!/bin/bash

# This script actually uses bash-specific features.

# Number of times to run the full benchmarking function
TIMES_TO_RUN=${TIMES_TO_RUN_DEFAULT:-5}

# Number of times to run the expression during each benchmark
WARMUP=${WARMUP_DEFAULT:-50}
NUM=${NUM_DEFAULT:-1000}

# Garbage collection:
TRIP_BYTES=${TRIP_BYTES_DEFAULT:-"(collect-trip-bytes)"
GC=${GC_DEFAULT:-"no"}
GC_MSG=${GC_MSG_DEFAULT:-"(collect-notify)"
GC_RADIX=${GC_RADIX_DEFAULT:-"(collect-generation-radix)"
GC_OFF=${GC_OFF_DEFAULT:-""}

# Optimizations:
OPTIMIZE=${OPTIMIZE_DEFAULT:-"(optimize-level)"
INSPECTOR=${INSPECTOR_DEFAULT:-"(generate-inspector-information)"
CP0_EFFORT=${CP0_EFFORT_DEFAULT:-"(cp0-effort-limit)"
CP0_SCORE=${CP0_SCORE_DEFAULT:-"(cp0-score-limit)"
CP0_OUTER_UNROLL=${CP0_OUTER_UNROLL_DEFAULT:-"(cp0-outer-unroll-limit)"

# Address to which summaries should be sent
REPORT_ADDR=${REPORT_ADDR_DEFAULT:-""}

# File containing the timer function
TIMER_FILE=${TIMER_FILE_DEFAULT:-"gettime.ss"

LOAD_FILES=${LOAD_FILES_DEFAULT:-""}

# F_variables are for the filenames. Their default values are
# the default values Chez Scheme uses for these things (obtained
# by hand, so they may be incorrect).
F_TRIP_BYTES=${TRIP_BYTES_DEFAULT:-"-tb1048576"}
```
F_GC={GC_DEFAULT:-""}
F_GC_MSG={GC_MSG_DEFAULT:-""}
F_GC_RADIX={GC_RADIX_DEFAULT:-"-gcRadix4"}
F_GC_OFF={GC_OFF_DEFAULT:-""}
F_OPTIMIZE={OPTIMIZE_DEFAULT:-"-opt0"}
F_INSPECTOR={INSPECTOR_DEFAULT:-""}
F_CPO_EFFORT={CPO_EFFORT_DEFAULT:-"-cp0-200"}
F_CPO_SCORE={CPO_SCORE_DEFAULT:-"-20"}
F_CPO_OUTER_UNROLL={CPO_SCORE_DEFAULT:-"-0"}

F_SINGLE_USER={SINGLE_USER_DEFAULT:-""}

function usage {
  echo 'bench [-t times-to-run ]'
  echo '   [-w warmup-runs ]'
  echo '   [-n number-of-real-runs ]'
  echo '   [-b collect-trip-bytes ]'
  echo '   [-g generation-to-collect-between-runs ]'
  echo '   "no" for no collections between runs’
  echo '   [-G] (turns off collection DURING runs)'
  echo '   [-m] (turns on garbage collection messages)’
  echo '   [-o optimize-level ]’
  echo '   [-i] (turns off inspector)’
  echo '   [-l file-to-load ]* ’
  echo '   [-e value-for-cpo-effort-limit ]’
  echo '   [-s value-for-cpo-score-limit ]’
  echo '   [-S ] (Marks files created in single-user mode)’
  echo '   [-u value-for-cpo-outer-unroll-limit ]’
  echo '   [-r value-for-collect-generation-radix ]’
  echo '   [-R address-to-which-summary-should-be-sent ]’
  echo '   [-h] (print this message and exit)’
}

# Process options
while getopts "t:w:n:b:g:Gmo:il:e:s:S:Su:r:R:h" opt; do
  case $opt in
  t ) TIMES_TO_RUN="$OPTARG" ;;
  w ) WARMUP="$OPTARG" ;;
  n ) NUM="$OPTARG" ;;
  b ) TRIP_BYTES="$OPTARG"
F_TRIP_BYTES="-tb$OPTARG" ;
g ) GC="$OPTARG"
    F_GC="-gc$OPTARG" between" ;
G ) GC_OFF="(collect-request-handler void)"
    F_GC_OFF="-NoGC"
    GC_OFF_MSG="(begin
        (display "NO GC during benchmarks")
        (newline))" ;
m ) GC_MSG="#" F_GC_MSG="-messages" ;
o ) OPTIMIZE="$OPTARG"
    F_OPTIMIZE="-opt$OPTARG" ;
i ) INSPECTOR="#" F_INSPECTOR="-noInspector" ;
l ) LOAD_FILES="$LOADFILES (load "$OPTARG")" ;
e ) CPO_EFFECT="$OPTARG"
    F_CPO_EFFECT="-cpo-$OPTARG" ;
s ) CPO_SCORE="$OPTARG"
    F_CPO_SCORE="-$OPTARG" ;
S ) F_SINGLE_USER="-single" ;
u ) CPO_OUTER_UNROLL="$OPTARG"
    F_CPO_OUTER_UNROLL="-$OPTARG" ;
r ) GC_RADIX="$OPTARG"
    F_GC_RADIX="-gcRadix$OPTARG" ;
R ) REPORT_ADDR="$OPTARG" ;
h ) usage
    exit 0 ;
? ) usage
    exit 1
esac
done
shift $((OPTIND - 1))

if [ $GC = "no" ]; then
    DO_GC=""
    GC_LEVEL="No"
else
    DO_GC="(collect ${GC})"
    GC_LEVEL="Level ${GC}"
fi
EXP="\$*"
F_EXP=${EXP}/-\}
F_EXP=${F_EXP}/()\}
F_EXP=${F_EXP}/()\}

FILENAME="\$F_EXP\$F\_SINGLE\_USER\$F\_OPTIMIZE"
FILENAME="\$FILENAME\$F\_GC\$F\_GC\_MSG\$F\_TRIP\_BYTES"
FILENAME="\$FILENAME\$F\_GC\_RADIX\$F\_GC\_OFF"
FILENAME="\$FILENAME\$F\_INSPECTOR"
FILENAME="\$FILENAME\$F\_CP0\_EFFORT\$F\_CP0\_SCORE\$F\_CP0\_OUTER\_UNROLL"
FILENAME="\$\{FILENAME\}-fixed-parameterize"
if /sbin/ifconfig | grep "eth"; then
  FILENAME="\$\{FILENAME\}-network"
fi

# Put in file numbers and iterate
declare -i iter=0
while [ $iter -lt $TIMES\_TO\_RUN ]; do
  FNAME="\$\{FILENAME\}.\$((iter += 1))"

  scheme <<\EQF
  (collect-notify \$\{GC\_MSG\})
  (optimize-level \$\{OPTIMIZE\})
  (generate-inspector-information \$\{INSPECTOR\})
  (cp0-effort-limit \$\{CP0\_EFFORT\})
  (cp0-score-limit \$\{CP0\_SCORE\})
  (cp0-outer-unroll-limit \$\{CP0\_OUTER\_UNROLL\})
  (collect-generation-radix \$\{GC\_RADIX\})
  (collect-trip-bytes \$\{TRIP\_BYTES\})
  \$\{GC\_OFF\}

  ;;; this file contains the get-time function
  (load "$\{TIMER\_FILE\}"")

  ;;; Load whatever files we need to
  ;;; (probably containing the benchmark)
  \$\{LOAD\_FILES\}

  ;;; calls the expression e n t i m e s
  (define-syntax ntimes
    (syntax-rules ()
((_ n e) (do ((i n (- i 1)))
  ((= 0 i)
   e])))

;;; evaluates the expression 'e' and returns the amount of
;;; time the execution took.
(define-syntax micro-timer
  (syntax-rules ()
    ((_ e) (begin
      ${DO_GC}
      (let ((long-before (get-time)))
        (let ((before (get-time)))
          e
          (let ((after (get-time))
            (overhead (- before long-before)))
            (- (- after before) overhead))))))))

;;; runs an expression n times using the micro-timer to time the
;;; execution speed each time, and prints the time it took.
(define-syntax batch-micro-timer
  (syntax-rules ()
    ((_ n e) (begin
      (ntimes n
        (begin
          (display (date-and-time))
          (printf "%Real Time: %.1f"
            (micro-timer e))))))))

;;; This macro will warm-up and the time execution of 'e'.
;;; It evaluates 'e' 'warm-up' times and throws those times away.
;;; After that, it evaluates 'e' 'n' times, recording the number
;;; of microseconds required for each evaluation.
(define-syntax bench-micro
  (syntax-rules ()
    ((_ warm-up n e)
      (begin
        (batch-micro-timer warm-up e)
        ;; Warm up (including the timer)
        (with-output-to-file "${FNAME}"
          (lambda ()
            (parameterize
              .....
            )
            (micro-timer e)))))))
(printf "Listing microsecond runtimes in ~% $\{NAME\}-\%")
(printf "Warm-up: ~A-%Iterations: ~A-%" warm-up n)
$\{GC\_OFF\_MSG\}
(printf "GC done approximately every ~A-byte~%"
(collect-trip-bytes))
(printf "Collect radix: ~A-%" (collect-generation-radix))
(if (collect-notify)
 (printf "GC info collected~%") ())
(printf "$\{GC\_LEVEL\} GC done between timings~%")
(if (not (generate-inspector-information))
 (printf "No inspector info generated~%") ())
(printf "Optimized at level ~A-%" (optimize-level))
(printf "CP0 effort/score/unroll limits: ~A/~A/\-A-%"
 (cp0-effort-limit)
 (cp0-score-limit)
 (cp0-outer-unroll-limit))
(printf "Summary emailed to: $\{REPORT\_ADDR\}-\%")
(printf "Expression: ~S-%" 'e )
(batch-micro-timer n e)))))))

(bench-micro $\{WARMUP\} $\{NUM\} $\{EXP\})

EOF

# The following portion of the script works well on Ohio
# University's Computer Science Department's primary
# server. The uncommented text after this works on my
# machine. You may need to switch these around depending on
# how mail works for you.
#   # Email statistics (Currently assumes the network is up)
#   if [ -n "$\{REPORT\_ADDR\}" ]; then
#     mail "$\{REPORT\_ADDR\}" <<EOF
#Subject: Summary of $\{NAME\}
#
#$\{./header $\{NAME\}}
#$\{./stats $\{NAME\}}
#.
#EOF
#
fi

if [ -n "$REPORT_ADDR" ]; then
    mail -s "Summary of ${FNAME}" $REPORT_ADDR <<EOF
$(./header ${FNAME})

$(./stats ${FNAME})
EOF
fi
done
Program 6 Unified Diff Displaying the Benchmarking Script with an Extra Parenthesis

```bash
--- bench Fri Aug 2 00:55:49 2002
+++ bench-typo Fri Aug 2 00:56:52 2002
@@ -125,7 +125,7 @@
     FILENAME="$FILENAME$F_GC_RADIX$F_GC_OFF"
     FILENAME="$FILENAME$F_INSPECTOR"
     FILENAME="$FILENAME$F_CP0_EFFORT$F_CP0_SCORE$F_CP0_OUTER_UNROLL"
-FILENAME="$\{FILENAME\}\-fixed-parameterize"
+FILENAME="$\{FILENAME\}\-typo"
     if /sbin/ifconfig | grep "eth"; then
         FILENAME="$\{FILENAME\}\-network"
     fi
@@ -205,7 +205,7 @@
     ${GC_OFF_MSG}
     (printf
         "GC done approximately every \"A bytes\""
-        (collect-trip-bytes))
+        (collect-trip-bytes))
     (printf
         "Collect radix: \"A\"" (collect-generation-radix))
     (if (collect-notify)
```
Program 7 Unified Diff placing the parameterize call inside micro-timer in the benchmarking script but before the garbage collection

--- bench Fri Aug 2 00:55:49 2002
+++ bench-bad-param Fri Aug 2 00:57:43 2002
@@ -125,7 +125,7 @@
  FILENAME="$FILENAME$F_GC_RADIX$F_GC_OFF"
  FILENAME="$FILENAME$F_INSPECTOR"
  FILENAME="$FILENAME$F_CP0_EFFORT$F_CP0_SCORE$F_CP0_OUTER_UNROLL"
-FILENAME="${FILENAME}-fixed-parameterize"
+FILENAME="${FILENAME}-really-bad-param-sans-paren"
  if /sbin/ifconfig | grep "eth"; then
    FILENAME="${FILENAME}-network"
  fi
@@ -165,13 +165,15 @@
 (define-syntax micro-timer
   (syntax-rules ()
     ((_ e) (begin
-       ${DO_GC}
-       (let (((long-before (get-time)))
-             ((before (get-time)))
-             e
-             ((after (get-time)))
-             (overhead (- before long-before)))
-             (- (- after before) overhead)))))
+       (parameterize
+         ([console-output-port (current-output-port)])
+         ${DO_GC}
+         (let (((long-before (get-time)))
+             ((before (get-time)))
+             e
+             ((after (get-time)))
+             (overhead (- before long-before)))
+             (- (- after before) overhead)))))

;;; runs an expression n times using the micro-timer to time the
;;; execution speed each time, and prints the time it took.
@@ -181,7 +183,7 @@
  (ntimes n
    (begin
      (display (date-and-time))
```
- (printf "%-Real Time: "A-"
+ (printf "%-Real Time: "A-"
      (micro-timer e))))))

;;; This macro will warm-up and the time execution of 'e'.
@@ -196,32 +198,32 @@
    ;; Warm up (including the timer)
    (with-output-to-file "${NAME}"
      (lambda ()
-    (parameterize
+    (parameterize
        ([console-output-port (current-output-port)])
-      (printf
-          "Listing microsecond runtimes in %A-%${NAME}-%"
-      (printf
-          "Warm-up: A-%Iterations: A-% warm-up n"
-          (printf
-              "GC done approximately every A bytes-%"
-              (collect-trip-bytes))
-      (printf
-          "Collect radix: A-% (collect-generation-radix))
-      (if (collect-notify)
-          (printf "GC info collected-%") ()
-      (printf "$GC_LEVEL" GC done between timings-%")
-      (if (not (generate-inspector-information))
-          (printf "No inspector info generated-%") ()
-      (printf "Optimized at level A-% (optimize-level))")
-      (printf "CP0 effort/score/unroll limits: A/A/A-%"
-          (cp0-effort-limit)
-          (cp0-score-limit)
-          (cp0-unroll-limit))
-      (printf "Summary emailed to: ${REPORT_ADDR}-%")
-      (printf "Expression: S-% 'e "
-          (batch-micro-timer n e)))))
+      (printf
+          "Listing microsecond runtimes in %A-%${NAME}-%")
+      (printf
+          "Warm-up: A-%Iterations: A-% warm-up n"
+          $GC_OFF_MSG)
(printf
  "GC done approximately every \textasciitilde A bytes\textasciitilde \%
  (collect-trip-bytes)))
(printf
  "Collect radix: \textasciitilde A\textasciitilde \%
  (collect-generation-radix))
(if (collect-notify)
  (printf "GC info collected\textasciitilde \%") ()
  (printf "${GC\_LEVEL} GC done between timings\textasciitilde \%")
(if (not (generate-inspector-information))
  (printf "No inspector info generated\textasciitilde \%") ()
  (printf "Optimized at level \textasciitilde A\textasciitilde \%
  (optimize-level))
  (printf "CP0 effort/score/unroll limits: \textasciitilde A/\textasciitilde A/\textasciitilde \%
  (cp0-effort-limit)
  (cp0-score-limit)
  (cp0-outer-unroll-limit))
  (printf "Summary emailed to: ${REPORT\_ADDR}\textasciitilde \%"
  (printf "Expression: \textasciitilde S\textasciitilde \%
  (batch-micro-timer n e))))))
(bench-micro ${WARMUP} ${NUM} ${EXP}))
EOF
Program 8 Unified Diff placing the parameterize call inside micro-timer in the benchmarking script and after the garbage collection

--- bench Fri Aug  2 00:55:49 2002
+++ bench-bad-param-after-gc Fri Aug  2 00:59:36 2002
@@ -125,7 +125,7 @@
 FILENAME="$FILENAME$F_GC_RADIX$F_GC_OFF"
 FILENAME="$FILENAME$F_INSPECTOR"
 FILENAME="$FILENAME$F_CP0_EFFORT$F_CP0_SCORE$F_CP0_OUTER_UNROLL"
-FILENAME="$FILENAME-fixed-parameterize"
+FILENAME="$FILENAME-really-bad-param-sans-paren-after-gc"
 if /sbin/ifconfig | grep "eth"; then
   FILENAME="$FILENAME-network"
 fi
@@ -166,12 +166,14 @@
 (syntax-rules ()
   ((_ e) (begin
     ${DO_GC}
-    (let ((long-before (get-time)))
-      (let ((before (get-time)))
-        e
-        (let ((after (get-time))
-          (overhead (- before long-before)))
-          (- (- after before) overhead)))))
+    (parameterize
+      ([console-output-port (current-output-port)])
+      (let ((long-before (get-time)))
+        (let ((before (get-time)))
+          e
+          (let ((after (get-time))
+            (overhead (- before long-before)))
+            (- (- after before) overhead)))))

;;; runs an expression n times using the micro-timer to time the
;;; execution speed each time, and prints the time it took.
@@ -181,7 +183,7 @@
 (ntimes n
   (begin
     (display (date-and-time))
-    (printf "-%Real Time: "%n"
+    (printf "-%Real Time: "%n"
(micro-timer e)))))))

;;; This macro will warm-up and the time execution of 'e'.
@@ -196,32 +198,32 @@
 ;; Warm up (including the timer)
 (with-output-to-file "${FNAME}"
   (lambda ()
-    (parameterize
+    (parameterize
      ([console-output-port (current-output-port)])
-      (printf
-        "Listing microsecond runtimes in % $${FNAME}-%"")
-      (printf
-        "Warm-up: -A-%Iterations: -A-%" warm-up n)
-      ${{GC_OFF_MSG}}
-      (printf
-        "GC done approximately every -A bytes-%"
-        (collect-trip-bytes))
-      (printf
-        "Collect radix: -A-%" (collect-generation-radix))
-      (if (collect-notify)
-        (printf "GC info collected-%")())
-      (printf "${{GC_LEVEL}} collected-%" GC done between timings-%")
-      (if (not (generate-inspector-information))
-        (printf "No inspector info generated-%")())
-      (printf "Optimized at level -A-%" (optimize-level))
-      (printf "CP0 effort/score/unroll limits: -A/-A/-A-%"
-        (cp0-effort-limit)
-        (cp0-score-limit)
-        (cp0-outer-unroll-limit))
-      (printf "Summary emailed to: ${REPORT_ADDR}-%")
-      (printf "Expression: -S-%" 'e )
-      (batch-micro-timer n e))))))))
+    (printf
+      "Listing microsecond runtimes in % $${FNAME}-%")
+    (printf
+      "Warm-up: -A-%Iterations: -A-%" warm-up n)
+    ${{GC_OFF_MSG}}
+    (printf
+      "GC done approximately every -A bytes-%"
(collect-trip-bytes))
+
+  (printf
+    "Collect radix: A-%" (collect-generation-radix))
+  (if (collect-notify)
+    (printf "GC info collected-%") ()
+  (printf "$GC_LEVEL$ GC done between timings-%")
+  (if (not (generate-inspector-information))
+    (printf "No inspector info generated-%") ()
+  (printf "Optimized at level A-%" (optimize-level))
+  (printf "CP0 effort/score/unroll limits: A/A/A-%"
+    (cp0-effort-limit)
+    (cp0-score-limit)
+    (cp0-outer-unroll-limit))
+  (printf "Summary emailed to: $REPORT_ADDR-%")
+  (printf "Expression: S-%" 'e )
+  (batch-micro-timer n e))))))
+
+  (bench-micro $WARMUP$ $NUM$ $EXP$)
+EUF
Program 9 Interface allowing access to the gettime function from Chez Scheme

;; init_time MUST be called before get_sec and get_usec

;; load the compiled C code
;; compile with 'gcc -fPIC -shared -o gettime.so gettime.c'
(load-shared-object "./gettime.so")

;; returns the current time in microseconds
(define get-time
  (lambda ()
    (let ((start (foreign-procedure "init_time" () void))
          (get-sec (foreign-procedure "get_sec" () integer-32))
          (get-usec (foreign-procedure "get_usec" () integer-32)))
      (start)
      (+ (* (get-sec) 1000000) (get-usec))))
Program 10 A Collection of C functions designed to be called from Scheme that provide access to the times produced by the gettimeofday system call

/* This file contains a simple function to return the current time
 * in microseconds.
 */

#include <stdio.h>
#include <errno.h>
#include <sys/time.h>

/*
 * Minutes west of Greenwich and timezones don’t matter for
 * the purpose of timing a process.
 */
static struct timezone tmz = {0, 0};

static struct timeval tv = {0, 0};

void
init_time()
{
    if (gettimeofday(&tv,&tmz) < 0) {
        perror("gettime: ");
    }
}

long
get_sec()
{ return tv.tv_sec; }

long
get_usec()
{ return tv.tv_usec; }
B.3  A Program to Analyze the Data

Program 11 Statistics Collection Script

#! /usr/bin/perl
use Getopt::Std;

if(not getopts("b:")) {
    die "Invalid Arguments";
}

foreach $file (@ARGV) {
    open FILE, "$file" or die "Unable to open $file";
    @lon = ();
    while(<FILE>) {
        if(/Real Time: \(\d+/) {
            push @lon, $1;
        }
    }
    close FILE;

    $mean = mean(@lon);
    $median = median(@lon);
    $variance = variance(@lon);
    $stddev = stddev(@lon);
    $percent_stddev = $stddev / $mean * 100;

    print "Statistics for $file:\n";
    print "Mean: $mean\n";
    print "Median: $median\n";
    print "Variance: $variance\n";
    print "Standard Deviation: $stddev\n";
    print "Percent Std. Dev.: $percent_stddev\n";

    if($opt_b) {
        my %b = buckets(@lon);
        foreach $key (sort @{$a <=> $b} keys %b) {
            print "$key $b{$key}\n";
        }
    }
}
sub mean {
    my $sum = 0;
    foreach $num (@_) {
        $sum += $num;
    }
    return $sum / (scalar(@_));
}

sub median {
    sort { $a <=> $b } @_; 
    $len = scalar(@_);
    if (($len % 2) == 0) {
        return ($_[int($len / 2)] + $[int($len / 2)+1]) / 2;
    }
    return $_[($len / 2];
}

sub variance {
    my $m = mean(@_);
    my $sumsq = 0;
    foreach $num (@_) {
        $sumsq += ($num - $m) * ($num - $m);
    }
    return $sumsq / (scalar(@_) - 1);
}

sub stddev {
    return sqrt(variance(@_));
}

sub buckets {
    # $opt_b gives the size of each bucket
    my %buckets = ();
    my @lon = sort { $a <=> $b } @_; 
    my $index = $lon[0];
    for (@lon) {
        if (($_ > $index) && ($_ < $index + $opt_b)) {
            if($buckets{$index}) {
                ++$buckets{$index};
            } else {
        }
$buckets{$index} = 1;
}
} else {
    $index = $_;
    $buckets{$index} = 1;
    next;
}

return %buckets;