CS 15-740, Computer Architecture, Fall 2003
Homework Assignment 1
Solutions

The purpose of this assignment is to develop techniques for measuring code performance, to practice reasoning about low-level code optimization, and to better understand Alpha procedure calling conventions.

Policy

You will work in groups of three people in solving the problems for this assignment. (A group of two may be necessary depending on the class size—groups of one are definitely not allowed.) Turn in a single writeup per group, indicating all group members.

Logistics

Any clarifications and revisions to the assignment will be posted on the "assignments" web page on the class WWW directory.

In the following, *HOMEDIR* refers to the directory:

`/afs/cs.cmu.edu/academic/class/15740-f03/public`

and *ASSTDIR* refers to the subdirectory *HOMEDIR*`/asst/asst1`.

Please hand in your assignment as a hard copy of your **formatted** text.

Using Interval Timers

Measuring performance is fundamental to the study of computer systems. When comparing machines, or when optimizing code, it is often useful to measure the amount of time that it takes (preferably at the resolution of processor clock cycles) to execute a particular operation or procedure. Some machines have special facilities to assist in measuring performance. Even without such facilities, almost all machines provide *interval timers*—a relatively crude method of computing elapsed times. In this assignment, you will investigate how to reason about and control the accuracy of timing information that can be gathered using interval timers. One of the goals is to develop a *function timer* which accurately measures the execution time of any function on any machine.

The overall operation of an interval timer is illustrated in Figure 1. The system maintains a (user-settable) counter value which is updated periodically. That is, once every Δ time units, the counter is incremented by Δ. Using the Unix library routine `getitimer`, the user can poll the value of this counter. Thus, to measure the elapsed time of some operation `Op`, the user can poll the counter to get a starting value T_s, perform the operation, and poll the counter to get a final value T_f. The elapsed time for the operation can be *approximated* as $T_{observed} = T_f - T_s$. As the figure illustrates, however, the actual elapsed time T_{actual} may differ from $T_{observed}$ significantly, due to the coarseness of the timer resolution. Since the value of Δ is around 10 *milliseconds* for most systems, this error can be very significant.

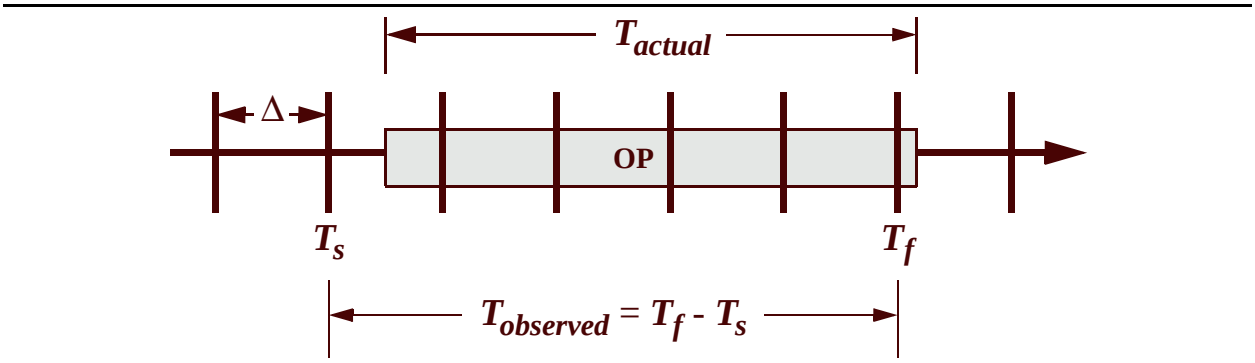

Figure 1: Time Measurement with an Interval Timer

We have encapsulated the Unix interval timer routines for you in a handy timer package called *ASSTDIR*`/etime.c`. You should use this package for all measurements in the assignment. See *ASSTDIR*`/example.c` for a simple example of how to use the package. One notable feature is that it converts the measurements to units of seconds, expressed as a C `double`. The procedure for timing operation `Op` is then:

```
init_etime();
Ts = get_etime();
Op;
Tf = get_etime()
T_observed = Tf - Ts;
```

Problem 1: Bounded Measurement Error

Consider a processor with a 500 MHz clock rate where precisely one addition operation can be performed every clock cycle, and where the value of Δ for the interval timer is 10 milliseconds. You would like to time a section of code (`Op`) consisting purely of a sequence of back-to-back additions.

If your code sequence consists of 10^6 additions, what will the relative measurement error of $T_{observed}$ with respect to T_{actual} be? How about for 10^8 additions? As always, show all of your work.

Solution 1: *(Based on a solution courtesy of A. Nizhner)*

We first demonstrate that E is bounded by Δ. Let T_0 and T_1 be the starting and ending wall-clock time points respectively; that is $T_{actual} = T_1 - T_0$. The values T_f and T_s then correspond to get_etime() measurements at T_1 and T_0. Let $T_f = k_1\Delta$ and $T_s = k_0\Delta$. Then, $T_0 = k_0\Delta + a_0$ and $T_1 = k_1\Delta + a_1$, with $a_0, a_1 < \Delta$. Thus:

$$E = |T_{observed} - T_{actual}| = |(T_f - T_s) - (T_1 - T_0)| = |a_1 - a_0| < \Delta$$

The relative error E_{rel} is then bounded by $\frac{\Delta}{T_{actual}}$. For a 500MHz machine, T_{actual} of an operation with cycle count c is $\frac{c}{500\times10^6}$ seconds; with a Δ of 0.01 seconds, we have $E_{rel} < \frac{0.01\times500\times10^6}{c}$. Thus, for $c = 10^6$, $E_{rel} < \frac{500}{100} = 500\%$; for $c = 10^8$, $E_{rel} < \frac{500}{10000} = 5\%$.

Note, however, that for the particular cases we have we can calculate an even tighter bound. Let's consider the case in which we take consecutive measurements of a sequence of operations. In the absense of

perturbations, if the consecutive measurements differ, they must differ by Δ. The reason for the difference is the position in time of the start of the actual run relative to the timer interrupts. If the measurement run starts right after a timer interrupt, then $T_{observed} = 0$ ns because $T_{actual} = 2\ ms\ <\ 10\ ms\ =\ \Delta$. If the timer interrupt happens in the middle of the measurement period, then $T_{observed} = \Delta$. So the relative errors for the 10^6 additions we consider can be:

$$E^A_{rel} = \frac{|T^A_{observed} - T_{actual}|}{T_{actual}} = \frac{|0-2|}{2} = 100\%$$
$$E^B_{rel} = \frac{|T^B_{observed} - T_{actual}|}{T_{actual}} = \frac{|10-2|}{2} = 400\%$$

Similarly, for the 10^8 additions we can only get 0% relative error because $T_{actual} = 200\ ms$ which is an integer multiple of the Δ. Hence, in this particular case, $T_{observed} = T_{actual}$ and $E_{rel} = 0\%$.

Problem 2: Measuring Δ for Your Timer

Write a C procedure that uses measurements to estimate (as accurately as possible) the value of Δ on any UNIX machine. Provide a listing of your code along with a brief description of your scheme.

Solution 2: *(Based on a solution courtesy of K. Bowers and J. Wales)*

There are numerous subtleties, but the basic idea is to measure something until a timer tick is detected. For example:

```
double get_delta() {
  double end, start;
  init_etime();
  start = get_etime();
  while (start == (end = get_etime()));
  return(end - start);
}
```

Note that with the code above we compute Δ only if the time it takes for etime() to run is smaler than Δ_{actual}. This is a valid assumption, since the timer interrupts are in the order of $msec$, while polling the timer takes several orders of magnitude less time to run. However, even in the case when this is not true, the computed interval will truly be the granularity of measurement we can achieve. For example, say that it takes $4\Delta_{actual}$ to run etime(), and the poll of the timer value takes place $3\Delta_{actual}$ after etime() starts executing. Then, the value we will calculate will be the time after the first call polls the timer value until it ends ($1\Delta_{actual}$), plus the time it takes for the second call to poll the value of the timer again ($3\Delta_{actual}$), which is $4\Delta_{actual}$. Any time measurement will always be off by the time it takes to run the "epilogue" of the first call plus the "prologue" of the second. Since our goal is to account for such disparities, it is appropriate to use the measurement above as the magnitude of the measurement granularity.

However, in the real world it is very difficult to take measurements without background noise (e.g. interrupts). In order to adjust for such perturbations, we propose to use Standard Deviation as a more statistically-robust measure, permitting more rigorous analysis and offering a method that is less affected by the noise of

individual samples. We base this assertion on the assumption that the Δ that is manifest in the running course of the program will not always be exactly the same. If Δ is somewhere between some unknown Δ_{min} and Δ_{max}, what is to ensure that we ever see Δ_{max} experimentally?

In light of this consideration, using the "maximum deviation" method may provide misleading results. In view of the desire for statistical robustness, the "average-absolute-value-of-deviation" suggestion is less compelling than Standard Deviation. Formally, if $x_1, x_2, x_3, \cdots$ are individual samples and μ the mean, the standard deviation σ of the samples is given by:

$$\sigma = \sqrt{\sum_{i=1}^{k}(x_i - \mu)^2}$$

In our case, the samples will be subsequent measurements of operation Op. Note that in the absense of perturbations,

$$|T^1_{observed} - T^0_{observed}| = 0 \quad or \quad |T^1_{observed} - T^0_{observed}| = \Delta$$

So, any difference in subsequent measurements can be attributed to the relative position of the start of the measurement run relative to the timer interrupts. In other words, our measurements will be quantized on two values (bimodal), either $(T^{noPerturbations}_{observed})$ or $(T^{noPerturbations}_{observed} + \Delta)$. In the presence of perturbations, other effects will contribute to the difference. However, at the limit, the observed values will converge to the two values above. Hence, with a sufficiently large sample base we will have:

$$\mu_{perturbations} = \mu_{noPerturbations} = \frac{|T^{noPerturbations}_{observed} - (T^{noPerturbations}_{observed} + \Delta)|}{2} = \frac{\Delta}{2}.$$

The standard deviation σ will also converge to the mean μ because of the quantization of $T_{observed}$, so we can approximate $\Delta = 2\sigma$.

The code included after Problem 3 calculates $\Delta = 2\sigma$. The function timer (func_time) first calls a function to calculate the value of delta (get_delta). It accomplishes this by timing a small test function 10,000 times and determining Δ to be twice the standard deviation of the resulting pool of data.

Having a value for Δ, the timer can then calculate the minimum execution duration $T_{threshold}$ for a repeated test function P in order to achieve a certain error upper bound $E_{acceptable}$ (see next problem).

With this value for $T_{threshold}$, the function proceeds to run P a number N times, tracking the total observed time, starting with $N = 1$. If $T_{threshold}$ is not achieved, then N is doubled and the timing operation repeated until $T_{threshold}$ is equaled or exceeded. A value of time for P is calculated as the mean of the observed time, again with the assumption that, as N grows larger, the mean contribution from delta will move toward zero.

The code follows after Problem 3.

We can improve the accuracy of the measurements by making sure that the activity we measure has sufficient duration to overcome the imprecision of interval timers. That is, we can accurately measure the time required by `Op` by executing it n times for a sufficiently large value of n:

```
init_etime();
Ts = get_etime();
for (i=0; i<n; i++) {
    Op;
}
Tf = get_etime()
T_aggregate = Tf - Ts;
T_average = T_aggregate/n;
```

How do we choose a large enough value of n? The idea is that n must be large enough such that $T_{aggregate}$ is larger than the minimum value ($T_{threshold}$) which guarantees a relative measurement error less than the desired upper bound of E. The value of $T_{threshold}$ can be computed based on Δ and E. However, since the elapsed time for `Op` is unknown, we cannot compute the minimum value of n ahead of time.

One approach is to start with $n = 1$, and continue doubling it until the observed $T_{aggregate}$ is large enough to guarantee sufficient accuracy (i.e. it is larger than $T_{threshold}$).

Problem 3: Implementing a Function Timer

Implement a function timer in C that uses the doubling scheme outlined above to accurately measure the running time of any function on any system. Your function timer should have the following interface

```
typdef void (*test_funct)(void);
double func_time(test_funct P, double E);
```

where P is the function to be timed and E is the maximum relative measurement error. These prototypes are already defined for you in *ASSTDIR*`/func_time.h`. Implement your `func_time()` function in a separate file called `func_time.c`.

Your function timer should: (1) determine the timer period Δ using the scheme from the previous problem; (2) calculate $T_{threshold}$ as a function of Δ and E; and then (3) repeatedly double n until $T_{aggregate} \geq T_{threshold}$. It should work for any function on any system, regardless of the running time of the function or the timer period of the system.

Solution 3: *(Based on a solution courtesy of K. Bowers and J. Wales)*

The most common difficulty that people had was in computing the minimum threshold value. Remember that $T_{threshold}$ is expressed in terms of $T_{observed}$, not T_{actual}. More formally, we established in Problem 1 that $|T_{observed} - T_{actual}| < \Delta$ so $T_{observed} = T_{actual} \pm \Delta$. We also showed that the acceptable error (E_{rel}) is bound by $E_{rel} < \frac{\Delta}{T_{actual}}$. Note that $T_{actual} + \Delta \geq T_{observed}$. Hence, in order to guarantee the acceptable error, it must hold that $T_{actual} > \frac{\Delta}{E_{rel}}$, so $T_{threshold} \geq T_{actual} + \Delta > \frac{\Delta}{E_{rel}} + \Delta$.

A common error was to assume that $T_{threshold} = \frac{\Delta}{E_{rel}}$. This mistake becomes especially important when the acceptable error (E_{rel}) is nontrivially large. For example, if $E_{rel} = 0.5$, then the correct value of $T_{threshold}$ is 3Δ, whereas the incorrect formula yields 2Δ. Since $T_{observed} = T_{actual} \pm \Delta$, the maximum error when $T_{threshold} = 2\Delta$ is 100% rather than 50%.

Another potential pitfall is to accidentally double n (the number of timing iterations) an extra time and to fail to take this into account when computing the average time.

func_time.c

```
#include <stdio.h>
#include <sys/time.h>
#include <math.h>
#include "etime.h"
#include "utils.h"
#include "func_time.h"


#define NUM_REPLICATION_C (int)10000
#define SHOWTEXT 1

double func_time(test_funct P, double E) {

  return get_func_time(P, E, get_delta(NUM_REPLICATION_C));
}

// Time a test_funct P to error tolerance E
double get_func_time(test_funct P,
                     double E,
                     double delta_clf) {
  // VARIABLES AND CONSTANTS

  // Op timing variables.
  double E_clf = E;             // Permissible error.
  double T_threshold_clf;       // Minimum aggregate time to ensure error <= E.
  int num_repetition_c = 0;     // Number of times the operation will be repeated
                                // to obtain a more accurate time.
  int cur_repetition_i;         // Current repetition in loop.

  double Ts_vlf;                // Starting time.
  double Tf_vlf;                // Finishing time.

  double T_aggregate_vlf = 0.0; // Total observed time.



  // BEGIN PROGRAM

  // PART 1: CALCULATE DELTA AND T_THRESHOLD

  // Calculate T_threshold
  T_threshold_clf = delta_clf/E_clf + delta_clf;

#if SHOWTEXT == 1
  // Print out data.
  printf("\ndelta:\t\t%lf\nError bound:\t%lf\nT_threshold:\t%lf\n",
         delta_clf,
         E_clf,
         T_threshold_clf);
#endif

  // PART 2: BEGIN LOOP TO OBTAIN DESIRED MEASUREMENT

  // BEGIN WHILE: T_aggregate < T_threshold
  while(T_aggregate_vlf < T_threshold_clf) {

    if(num_repetition_c < 1)
      num_repetition_c = 1;
    else
      num_repetition_c *= 2;
```

```
    // Set T_aggregate to 0.
    T_aggregate_vlf = 0.0;

    // Initialize timer and warm cache.
    init_etime();

    // Establish starting time.
    Ts_vlf = get_etime();

    // BEGIN LOOP: num_repetition_c
    for(cur_repetition_i = 0;
       cur_repetition_i < num_repetition_c;
       cur_repetition_i++) {
      P();
    }
    // END LOOP: num_repetition_c

    // Establish finishing time.
    Tf_vlf = get_etime();

    // Calculate aggregate observed time.
    T_aggregate_vlf = Tf_vlf - Ts_vlf;
  }
  // END WHILE: T_aggregate < T_threshold


#if SHOWTEXT == 1
  printf("\nnum_repetition:\t%d\nT_aggregate:\t%lf\nT_mean:\t\t%lf ms\n",
        num_repetition_c,
        T_aggregate_vlf,
        (1000*T_aggregate_vlf)/(double)num_repetition_c
        );
#endif

  // Done.
  return(T_aggregate_vlf/(double)num_repetition_c);

}


// Calculate delta for this machine, using sample size num_replication_c
// Delta will be calculated by timing a function a very high number of times.
// Each instance of timing the function is one sample.
// Thus, many samples are taken.
// Delta is calculated as the Standard Deviation, or the Root of
// the Mean of the Sum of the Squared Errors for the samples.
double get_delta(int num_replication_c) {
  // Declare local variables.

  // Index/repetition variables for timing.
  int cur_replication_i;

  // Timer variables.
  double Ts_vlf; // Start time.
  double Tf_vlf; // End time.
  double *T_observed_vlf_a1;

  // Statistics.
  double delta_vlf; // Delta
  double mean_vlf; // Mean

  // Control for erroneous constant values.
  if(num_replication_c < 1)
    num_replication_c = 1;

#if SHOWTEXT == 1
  // Inform the user of how many times the op will be run.
  printf("\nOp timing will be executed %d times for determining delta...\n",
```

```
        num_replication_c);
#endif

  // Allocate memory for observation array.
  T_observed_vlf_a1 = (double *)malloc(num_replication_c * sizeof(double));


  // BEGIN LOOP: replication of timing operation
  for(cur_replication_i = 0;
      cur_replication_i < num_replication_c;
      cur_replication_i++) {

    // PERFORM TIMING OPERATION to DETERMINE DELTA

    // Initialise timing package ("warm the cache").
    init_etime();

    // Record starting time.
    Ts_vlf = get_etime();

    // Execute timed operation.
    do_op();

    // Record ending time.
    Tf_vlf = get_etime();


    // CALCULATE AGGREGATE

    // Calculate total observed time for this instance of running the op.
    T_observed_vlf_a1[cur_replication_i] = Tf_vlf - Ts_vlf;

  } // END LOOP: replication of timing operation.


  // We have now populated the array T_observed_vlf_a1
  // with the values of T_observed
  // for each of num_replication_c replications.

  // BEGIN DELTA-CALCULATION SECTION.

  // We may assume that, as the delta is a +/- additive,
  // it contributes zero to the mean of T_observed_vlf_a1

  // We may calculate delta as the standard deviation of the mean,
  // since mean(T_observed) is what we are taking to be our closest
  // estimate to T_actual.

  // The formula for Standard Dev of the Time Data is as follows:
  // StdDev_vlf = sqrt(variance(T_observed_vlf_a1));

  // We will calculate delva_vlf as the Standard Deviation of the data
  // delta_vlf = calculate_delta(T_observed_vlf_a1, num_replication_c);
  delta_vlf = get_StdDev_lf_a1(T_observed_vlf_a1, num_replication_c);

  // Calculate the mean time just for interest and debugging.
  mean_vlf = get_mean_lf_a1(T_observed_vlf_a1,
                        num_replication_c);

  // PRINT THE INFORMATION
  printf("Mean Time:\t%lf\nDelta:\t%lf\n", mean_vlf, delta_vlf);


  // FREE MEMORY
  free(T_observed_vlf_a1);

  // DONE.
  return(delta_vlf);
```

```
}


/*
 * The basic test operation.
 * The basic test op does not particularly matter in its content.
 * This test op declares 3 ints, sets 2 of them, and declares and
 * sets one double.
 * This test op repeats, 1000 times, an integer compare,
 * 2 integer adds, 1 double add.
 * This test op returns void.
 */

void do_op(void) {
  long num_repetition_c = 1000000;
  long cur_repetition_i;

  long dummy_vd = 0;

  // Repeat a series of operations.
  for(cur_repetition_i = 0;
      cur_repetition_i < num_repetition_c;
      cur_repetition_i++)
  {
    dummy_vd += 1;
  }

}
```

Problem 4: Testing Your Function Timer

Test your function timer using the program *ASSTDIR*/`freq.c`, which uses `func_time()` to estimate the clock frequency of your machine. This routine assumes that your machine executes an integer addition in one clock cycle. This is a safe assumption for most modern processors.

Turn in the output string from freq.c and the type of system you ran it on (e.g., Sparc 5).

Solution 4:

Most people got full credit on this one. Many people ran this on either the class Alphas, or on their Pentium systems.

Problem 5: Alternative Timer Algorithms

Recall that in Problem 3, you repeatedly *doubled* the value of n until it was sufficiently large (i.e. until $T_{aggregate} \geq T_{threshold}$). Now consider the following three algorithms for increasing the value of n:

Algorithm 1: Set $n = 1$ initially, and repeatedly multiply n by a factor of **2** until n is sufficiently large (i.e. the algorithm used in Problem 3).

Algorithm 2: Set $n = 1$ initially, and repeatedly multiply n by a factor of **10** until n is sufficiently large.

Algorithm 3: Set $n = 1$ initially, and repeatedly **add** (*not multiply!*) **100** to n until n is sufficiently large.

Your goal is to minimize the total amount of time that your timing routine takes to accurately time a function. In this problem, you will evaluate the three algorithms described above based on this criteria.

Part 1: Assuming that $T_{threshold} = 100$ milliseconds, and assuming that the timing loop surrounding `Op` involves zero overhead, compute how long it would take for each of the three algorithms for increasing n to accurately time `Op` for each of the following three cases: (i) $T_{actual} = 12.0$ milliseconds, (ii) $T_{actual} = 99.0$ microseconds, and (iii) $T_{actual} = 110.0$ microseconds.

Part 2: Based on a quantitative analysis of their *worst-case* behaviors, evaluate which of the three algorithms for increasing n is most desirable for measuring functions where T_{actual} is an arbitrary value no greater than a microsecond and where Δ for the interval timer is at least 10 milliseconds.

Solution 5: *(Based on a solution courtesy of A. Nizhner)*

Part 1: Let R represent the running time; n is computed as $\lceil \frac{T_{threshold}}{T_{actual}} \rceil$.

(i) $T_{actual} = 12.0\ msec\ (n = 9)$**:**

Algorithm 1: $R = T_{actual}(1 + 2 + 4 + 8 + 16) = 31 \times T_{actual}$ = 372 msec

Algorithm 2: $R = T_{actual}(1 + 10) = 11 \times T_{actual}$ = 132 msec

Algorithm 3: $R = T_{actual}(1 + 101) = 102 \times T_{actual}$ = 1224 msec

(ii) $T_{actual} = 99.0\ \mu sec\ (n = 1010)$**:**

Algorithm 1: $R = T_{actual}(1+2+4+8+16+\ldots+1024) = 2047 \times T_{actual}$ = 202.653 msec (11 iterations)

Algorithm 2: $R = T_{actual}(1+10+100+1000+10000) = 11111 \times T_{actual}$ = 1099.989 msec (5 iterations)

Algorithm 3: $R = T_{actual}(1 + 101 + 201 + 301 + \ldots + 1101) = 6612 \times T_{actual}$ = 654.588 msec (12 iterations)

(iii) $T_{actual} = 110.0\ \mu sec\ (n = 909)$**:**

Algorithm 1: $R = T_{actual}(1+2+4+8+16+\ldots+1024) = 2047 \times T_{actual}$ = 225.17 msec (11 iterations)

Algorithm 2: $R = T_{actual}(1+10+100+1000) = 1111 \times T_{actual}$ = 122.21 msec (4 iterations)

Algorithm 3: $R = T_{actual}(1 + 101 + 201 + 301 + \ldots + 1001) = 5511 \times T_{actual}$ = 606.21 msec (11 iterations)

(**Note**: the number of iterations is not essential—it is included just for your information.)

Part 2: Let n, the lowest number of times required to execute operation OP in order to obtain an accurate estimate of its running time, be given.

Algorithm 1: In the worst case, $n = 2^k + 1$ for some $k \in N$.

$$\begin{aligned} R &= T_{actual}(1 + 2 + 4 + \ldots + 2^{\lceil log_2 n \rceil}) \\ &= T_{actual}(2^{\lceil log_2 n \rceil + 1} - 1 \\ &< T_{actual}(2^{log_2 n + 2} - 1 = T_{actual}(4n - 1) \\ &< T_{actual}(4n) \end{aligned}$$

Algorithm 2: In the worst case, $n = 10^k + 1$ for some $k \in N$.

$$\begin{aligned} R &= T_{actual}(1 + 10 + 100 + \ldots + 10^{\lceil log_{10} n \rceil}) \\ &= T_{actual}(\frac{10^{\lceil log_{10} n \rceil + 1} - 1}{9}) \\ &< T_{actual}(\frac{10^{log_{10} n + 2} - 1}{9}) = T_{actual}(\frac{100n - 1}{9}) \\ &< T_{actual}(\frac{100n}{9}) \end{aligned}$$

Algorithm 3: In the worst case, $n = 100^k + 1$ for some $k \in N$.

$$\begin{aligned} R &= T_{actual}(1 + 101 + 201 + \ldots + (n \cdot \lceil \frac{n}{100} \rceil) + 1) \\ &= T_{actual}(\sum_{i=0}^{\lceil \frac{n}{100} \rceil} (1 + 100i)) \\ &= T_{actual}(\lceil \frac{n}{100} \rceil + 1 + 100 \sum_{i=0}^{\lceil \frac{n}{100} \rceil} i) \\ &= T_{actual}(\lceil \frac{n}{100} \rceil + 1 + 50(\lceil \frac{n}{100} \rceil + 1)\lceil \frac{n}{100} \rceil) \\ &= T_{actual}(50\lceil \frac{n}{100} \rceil^2 + 51\lceil \frac{n}{100} \rceil + 1) \\ &< T_{actual}(50(\frac{n}{100} + 1)^2 + 51(\frac{n}{100} + 1) + 1) \\ &= T_{actual}(\frac{n^2}{200} + \frac{151n}{100} + 102) \end{aligned}$$

In the worst case, $n \cdot T_{actual}$ ends up being just short of $T_{threshold}$, and we overshoot as much as possible. The analysis shows that Algorithm 1 and Algorithm 2 are linear. In contrast, the upper bound for Algorithm 3 is quadratic. Without loss of generality, we may set $E_{rel} = 0.1$. Then, with $\Delta \geq 10$ ms and $T_{actual} \leq 1\ \mu s$, we have $n \simeq \frac{T_{threshold}}{T_{actual}} = \frac{\Delta}{E_{rel} \times T_{actual}} \geq 10^5$. Therefore, both Algorithm 1 and Algorithm 2 are superior to Algorithm 3 (given that $T_{threshold}$ is large relative to T_{actual}, which it is in this case). Of the two, Algorithm 1 is the best because it has a smaller constant factor in its upper bound.

Optimizing the `strlen()` Routine

The purpose of these next problems is to get hands-on experience with machine-level programming. Our interest is in being able to understand, measure, and optimize the machine code generated by a compiler. This is a far more useful skill than being able to churn out pages of assembly code by hand. Parts of this assignment involve compiling, disassembling, and running code on one of our Alpha-based machines. For information on how to access these machines, please refer to the link labeled "Information about our Alpha systems" on the class WWW page.

In the next several problems, we will be focusing on the performance of the `strlen()` routine, which is part of the C library. The following paraphrased excerpts from the `strlen()` man page describe its interface and behavior:

```
size_t strlen(const char *s);
```

- The `strlen()` function returns the number of bytes in the string pointed to by the s parameter. The string length value does not include the terminating null character.
- If you pass an out of bounds or NULL pointer to strlen, the function generates a segmentation violation.
- There are no return values reserved to indicate an error.

The file *ASSTDIR*`/strlen_naive.c` contains a straightforward (but naive, from a performance perspective) implementation of `strlen()` in C called "`my_strlen()`". The file *ASSTDIR*`/strlen_naive.s` contains the Alpha assembly code generated using the command: `gcc -O -S strlen_naive.c`

The file *ASSTDIR*`/strlen.dis` contains a disassembled version of the `strlen()` routine taken from the Unix library `/usr/lib/libc.a` on one of our Alpha machines. (This was disassembled with the `x/30i strlen` command of `gdb`.)

Problem 6: Understanding the `strlen()` Assembly Code

Generate an "annotated" version of both *ASSTDIR*`/strlen_naive.s` and *ASSTDIR*`/strlen.dis` using the following conventions:

- Put comments at the top of a code segment describing register usage and initial conditions.
- Put comments along the right hand side describing what each instruction does.

NOTE: Comments of the form:

```
# I won't tell you anything about the registers.
s8addq  r1, r2, r2   # r2 = 8*r1 + r2
ldq     r3, 0(r2)    # r3 = Mem[r2]
```

are useless and will receive little (if any) credit. Instead, we would like to see comments like the following:

```
# Throughout the loop: r1 holds i, r7 holds n
# At the beginning of the loop: r2 = &v[0]
s8addq  r1, r2, r2   # r2 = 8*i + &v[0]
ldq     r3, 0(r2)    # r3 = v[i]
```

In other words, your comments should convey semantic information from the source code, and not simply reiterate what would be obvious to anyone who can read Alpha assembly code.

Solution 6: *(Based on a solution courtesy of A. Nizhner and A. Kannan)*

strlen.s

```
# The only argument (char* s) is in a0.
# The "initialization" section below (through N+28) takes care of the
# case where the string is not aligned on an 8-byte boundary by masking
# out the bytes in the first quadword that do not belong to the string.
# a0 : start position in the string
# v0 : current position in the string
# t0 : scratch register
# t1 : scratch register (byte adjustment for boundary condition)
# t2 : scratch register
# t3 : scratch register

# The remaining code loops through the string, loading 8 bytes at a time and
# testing them in parallel.

# t0 contains the first quadword
0x3ff800d3dd0 <N>:       ldq_u   t0,0(a0)               # load first qword (possibly unaligned)
0x3ff800d3dd4 <N+4>:     lda     t1,-1(zero)            # set all bits in t1
0x3ff800d3dd8 <N+8>:     insqh   t1,a0,t1               # prepare mask (set high, clear low bits)
0x3ff800d3ddc <N+12>:    andnot  a0,0x7,v0              # align the index/ptr to qword (v0 = a0&~0x07)
0x3ff800d3de0 <N+16>:    or      t0,t1,t0               # mask out irrelevant chars (before string start)
0x3ff800d3de4 <N+20>:    cmpbge  zero,t0,t1             # test 8 bytes in parallel
0x3ff800d3de8 <N+24>:    bne     t1,0x3ff800d3e00 <N+48> # if there exist zeros, don't enter loop
0x3ff800d3dec <N+28>:    nop                            # keep going if we haven't found a NULL byte.
                                                        # Inside this loop v0 is iterating 8 bytes at a time.
                                                        # Over the string, we load 8 bytes at a time and use
                                                        # a compbge to check for a zero byte (in which case
                                                        # t1 will not be zero and we'll exit the loop).

# Loop across the qword aligned parts of the string.
# Here, v0 is the quadword pointer and t0 contains the quadword being processed.

0x3ff800d3df0 <N+32>:    ldq     t0,8(v0)               # load next qword of string
0x3ff800d3df4 <N+36>:    addq    v0,0x8,v0              # point to next qword
0x3ff800d3df8 <N+40>:    cmpbge  zero,t0,t1             # test 8 bytes and loop
0x3ff800d3dfc <N+44>:    beq     t1,0x3ff800d3df0 <N+32> # if there exist zeros, exit loop

# If we get here, at least one byte is zero. We need to adjust for boundary condition.
# t0 : holds the quadword from the previous loop
# t1 : result of the previous cmpbge
# v0 : address of the quadword with the NULL character.

# The code below determines the byte offset within the quadword, adds it to
# the quadword address, and subtracts the original byte address to yield the
# string length.

0x3ff800d3e00 <N+48>:    negq    t1,t2                  # flip status bits
0x3ff800d3e04 <N+52>:    blbs    t1,0x3ff800d3e30 <N+96> # branch to final subtraction if no adjustment needed
0x3ff800d3e08 <N+56>:    and     t1,t2,t1               # set bit indicating position of lowest one

# Increment the end position v0 by 1..7 depending on the position of the lowest one in t1

0x3ff800d3e0c <N+60>:    and     t1,0xf,t2              # doubleword mask
0x3ff800d3e10 <N+64>:    addq    v0,0x4,t0              # which doubleword?
0x3ff800d3e14 <N+68>:    cmoveq  t2,t0,v0               # add dword offset
0x3ff800d3e18 <N+72>:    and     t1,0x33,t3             # word mask
0x3ff800d3e1c <N+76>:    addq    v0,0x2,t0              # which word?
0x3ff800d3e20 <N+80>:    cmoveq  t3,t0,v0               # add word offset
0x3ff800d3e24 <N+84>:    and     t1,0x55,t4             # byte mask
0x3ff800d3e28 <N+88>:    addq    v0,0x1,t0              # which byte?
0x3ff800d3e2c <N+92>:    cmoveq  t4,t0,v0               # add byte offset
0x3ff800d3e30 <N+96>:    subq    v0,a0,v0               # length = end post - start pos
0x3ff800d3e34 <N+100>:   ret     zero,(ra),0x1          # return string length in v0
```

strlen_naive.s

```
        .verstamp 3 11
        .set noreorder
        .set volatile
        .set noat
        .file       1 "strlen_naive.c"
gcc2_compiled.:
__gnu_compiled_c:
.text
        .align 3
        .globl my_strlen
        .ent my_strlen
my_strlen:

# Register usage throughout this procedure :
# $0  : String length (len). Initialized to zero.
# $16 : Pointer to the current character to check (s). Initialized to the string address.
# $1  : Scratch register. Holds the current character to check (*s).

my_strlen..ng:
        .frame $30,0,$26,0  # stack pointer is $30, ra is $26
        .prologue 0
        bis $31,$31,$0      # set len to 0
        br $31,$38          # jump to while loop condition check
        .align 4
        .align 5

# this is the body of the while loop.
# len is in $0 and s is in $16 throughout.
$36:
        addq $0,1,$0        # len++
        addq $16,1,$16      # s++

# this is the condition checking part of the
# while loop.  Byte manipulations are performed
# to check the value of *s and jump based on that.
$38:
        ldq_u $1,0($16)     # r1 = *s
        extbl $1,$16,$1     # move *s into byte 0 of r1
        bne $1,$36          # if (*s != 0) jump to $36
        ret $31,($26),1     # return len
        .end my_strlen
```

Problem 7: Measuring the Performance of the `strlen()` Routines

Use your interval timer code to measure the performance of both the `my_strlen()` routine in *ASSTDIR*`/strlen_naive.c` and DEC C library implementation of `strlen()` on the various `strlen()` calls contained in *ASSTDIR*`/strlen_test.c`. Note that you should produce separate timing numbers for each these individual calls to `strlen()`, and be sure to call the initialization routine in this file before you start timing things to ensure that the cache is warm.

Discuss the relative performance differences between the two versions of the routine, and whether they make sense given your analysis of the assembly code.

Solution 7:

`strlen`:

```
do_test(0, 1): run timed at 0.000096 ms
do_test(0, 4): run timed at 0.000097 ms
do_test(0, 19): run timed at 0.000130 ms
do_test(0, 103): run timed at 0.000232 ms
do_test(0, 4088): run timed at 0.005242 ms
do_test(1, 1): run timed at 0.000097 ms
do_test(1, 4): run timed at 0.000096 ms
do_test(1, 19): run timed at 0.000135 ms
do_test(1, 103): run timed at 0.000218 ms
do_test(1, 4088): run timed at 0.005289 ms
do_test(6, 1): run timed at 0.000097 ms
do_test(6, 4): run timed at 0.000107 ms
do_test(6, 19): run timed at 0.000150 ms
do_test(6, 103): run timed at 0.000234 ms
do_test(6, 4088): run timed at 0.005257 ms
```

`naive_strlen`:

```
do_test(0, 1): run timed at 0.000078 ms
do_test(0, 4): run timed at 0.000143 ms
do_test(0, 19): run timed at 0.000368 ms
do_test(0, 103): run timed at 0.001633 ms
do_test(0, 4088): run timed at 0.061953 ms
do_test(1, 1): run timed at 0.000078 ms
do_test(1, 4): run timed at 0.000143 ms
do_test(1, 19): run timed at 0.000367 ms
do_test(1, 103): run timed at 0.001645 ms
do_test(1, 4088): run timed at 0.061953 ms
do_test(6, 1): run timed at 0.000078 ms
do_test(6, 4): run timed at 0.000144 ms
do_test(6, 19): run timed at 0.000369 ms
do_test(6, 103): run timed at 0.001648 ms
do_test(6, 4088): run timed at 0.061953 ms
```

The naive implementation is very fast for small strings due to minimal overhead. For medium-size or long strings, though, it behaves poorly relative to the library implementation. The reason for this is that the library implementation aligns the pointers to quad-word boundaries, and then it makes use of the cmpbge instruction to compare 8 bytes in parallel with a single load. On the other hand, the naive implementation loads each and every byte. Additionally, the library implementation avoids the unecessary increment of the string's length at every byte check. Instead, it calculates it by subtracting the address of the string's first character from the address of the string's null-terminating character. All these offer a performance advantage to the library implementation, which performs better for strings as small as 4 bytes, and is an order of magnitude faster for very long strings.

Problem 8: Implementing a Better Version of `strlen()` in C

Write your own version of `strlen()` in C. Your code must behave correctly, but at the same time it should be as efficient as possible. You should create a version of your code which only uses C constructs (i.e. no explicit assembly code). In addition, you may optionally create a second version of your code which uses the GCC assembly code directives (i.e. "`ASM`") if it further enhances performance. For further information on how to use assembly code directives in gcc, see the "info" pages on gcc (under "C extensions"). These info pages are reproduced on the *Assignment and exam information* class web page under Assignment 1. Use a minimal number of `ASM` statements—do not simply reproduce large amounts of hand-coded assembly in your C code. Be sure to compile your code using the "`-O`" optimization flag.

Measure the performance of your C-only code and your assembly-augmented code (if applicable). If your assembly-augmented code achieves better performance than your C-only code, discuss why you are not able to achieve comparable performance using only normal C constructs. Also, compare your code with both the naive and UNIX library versions of `strlen()`. If your performance falls short of the UNIX library version, explain why.

Solution 8: *(Courtesy of A. Nizhner)*

Implementation: The main bottleneck of the naive implementation in `my_strlen` is memory bandwidth, since a full word load is performed for each byte comparison. This memory access pattern is extremely wasteful; only one load need be performed for each naturally-aligned run of 8 bytes, or 4 bytes on 32-bit architectures. Our solution, therefore, is to partition the string into naturally-aligned 8-byte sections, perform a single word load for each section, and test each byte within the loaded word, thereby eliminating 7 out of every eight memory references in the naive implementation. The code is shown below.

```
#include <string.h>

/* Loads one native word (8 bytes) at a time and examines each byte in turn.
   This code can be made more portable via compile-time switches; all
   hard-coded dependencies on the Alpha native word width can be eliminated,
   at the cost of more memory references on 32-bit machines.
*/
size_t new_strlen(const char *s)
{
  /* generate aligned long int pointer */
  long int* t = (long int*)(((long int)s)&~7);
  long int acc;
  int u_bits = 8*(((long int)s)&7);

  /* right-align the "unaligned" portion of the string and set the
   high-order bits to prevent false-positives */
  if(u_bits!=0)
    acc = (((*t)>>u_bits) | ((-1L)&(-(1L<<(64-u_bits)))));
  else
    acc = *t;

  /* special-case the first word */
  if((acc&255)=='\0')
    return 0;
  else if(((acc>>8)&255)=='\0')
    return 1;
  else if(((acc>>16)&255)=='\0')
    return 2;
  else if(((acc>>24)&255)=='\0')
    return 3;
  else if(((acc>>32)&255)=='\0')
```

```
    return 4;
  else if(((acc>>40)&255)=='\0')
    return 5;
  else if(((acc>>48)&255)=='\0')
    return 6;
  else if(((acc>>56)&255)=='\0')
    return 7;

  /* the rest, now aligned */
  do {
    t++;
    acc = *t;
    if((acc&255)=='\0')
       return ((size_t)t-(size_t)s);
    else if(((acc>>8)&255)=='\0')
       return ((size_t)t-(size_t)s+1);
    else if(((acc>>16)&255)=='\0')
       return ((size_t)t-(size_t)s+2);
    else if(((acc>>24)&255)=='\0')
       return ((size_t)t-(size_t)s+3);
    else if(((acc>>32)&255)=='\0')
       return ((size_t)t-(size_t)s+4);
    else if(((acc>>40)&255)=='\0')
       return ((size_t)t-(size_t)s+5);
    else if(((acc>>48)&255)=='\0')
       return ((size_t)t-(size_t)s+6);
    else if(((acc>>56)&255)=='\0')
       return ((size_t)t-(size_t)s+7);
  }while(1);

  return 0;

}
```

Evaluation: While clearly faster than the naive implementation, `my_strlen` fails to achieve performance comparable to the DEC C `strlen` routine. The main reason for this is the use of the `cmpbge` instruction in the UNIX library code–by testing each byte in parallel, it allows an average speedup by a factor of approximately 4 for long strings, compared to an equivalent sequential version.

The timing data for our implementation of `strlen` are shown below.

```
do_test(0, 1): run timed at 0.000058 ms
do_test(0, 4): run timed at 0.000073 ms
do_test(0, 19): run timed at 0.000186 ms
do_test(0, 103): run timed at 0.000648 ms
do_test(0, 4088): run timed at 0.023113 ms
do_test(1, 1): run timed at 0.000070 ms
do_test(1, 4): run timed at 0.000086 ms
do_test(1, 19): run timed at 0.000197 ms
do_test(1, 103): run timed at 0.000670 ms
do_test(1, 4088): run timed at 0.023352 ms
do_test(6, 1): run timed at 0.000071 ms
do_test(6, 4): run timed at 0.000127 ms
do_test(6, 19): run timed at 0.000227 ms
do_test(6, 103): run timed at 0.000700 ms
do_test(6, 4088): run timed at 0.023828 ms
```

Problem 9: Measuring `strlen()` on a Different Architecture

Using your interval timer code, measure and compare the performance of both your C-only version of `strlen()` and the native (i.e. UNIX C library) version of `strlen()` on a machine other than an Alpha machine. Discuss whether these results are what you expected, or whether they are surprising.

Solution 9: *(Courtesy of A. Nizhner)*

Measurements. These tests were run on:
`SunOS unix14.andrew.cmu.edu 5.8 Generic_108528-19 sun4u sparc SUNW,Ultra-60`.

`new_strlen`:

```
do_test(0, 1): run timed at 0.000082 ms
do_test(0, 4): run timed at 0.000103 ms
do_test(0, 19): run timed at 0.000174 ms
do_test(0, 103): run timed at 0.000534 ms
do_test(0, 4088): run timed at 0.017090 ms
do_test(1, 1): run timed at 0.000093 ms
do_test(1, 4): run timed at 0.000114 ms
do_test(1, 19): run timed at 0.000210 ms
do_test(1, 103): run timed at 0.000563 ms
do_test(1, 4088): run timed at 0.017395 ms
do_test(6, 1): run timed at 0.000093 ms
do_test(6, 4): run timed at 0.000113 ms
do_test(6, 19): run timed at 0.000200 ms
do_test(6, 103): run timed at 0.000563 ms
do_test(6, 4088): run timed at 0.017700 ms
```

UNIX `strlen`:

```
do_test(0, 1): run timed at 0.000124 ms
do_test(0, 4): run timed at 0.000162 ms
do_test(0, 19): run timed at 0.000226 ms
do_test(0, 103): run timed at 0.000572 ms
do_test(0, 4088): run timed at 0.017395 ms
do_test(1, 1): run timed at 0.000119 ms
do_test(1, 4): run timed at 0.000148 ms
do_test(1, 19): run timed at 0.000236 ms
do_test(1, 103): run timed at 0.000591 ms
do_test(1, 4088): run timed at 0.017395 ms
do_test(6, 1): run timed at 0.000150 ms
do_test(6, 4): run timed at 0.000150 ms
do_test(6, 19): run timed at 0.000234 ms
do_test(6, 103): run timed at 0.000582 ms
do_test(6, 4088): run timed at 0.017395 ms
```

Discussion. When run on a SPARC, `new_strlen` actually exhibits the same level of performance as the `libc strlen`. (In fact, it performs better on short strings!) This is expected, and due to the fact that the SPARC lacks an instruction equivalent to the Alpha's `cmpbge`–in other words, the `libc` implementation enjoys no more parallelism than ours. These results reinforce the conjecture that the memory access pattern was precisely what made the naive implementation naive.

(Note: `new_strlen` obviously had to be modified to account for the 32-bit native word size and the reversed endiannes of the SPARC.)

Problem 10: Stack Frames and Procedure Calls

The file *ASSTDIR*`/structure.c` shows the C code for a function that demonstrates many interesting features of implementing C on an Alpha machine. The code generated by GCC with the `-O` flag is shown in the file *ASSTDIR*`/structure.s`

Document the following:

A. Show the layout of the data structure `list_ele`.

B. Explain how the program implements the returning of a structure by `sum_list`.

C. Describe the register allocation used in `sum_list`.

D. Show the layout of the stack frame used by `sum_list`.

E. Create an annotated version of the assembly code for `sum_list` using our usual formatting conventions.

Solution 10:

A - Data Structure Layout

- offset 0-7: `next`
- offset 8-15: `pt.x`
- offset 16-23: `pt.y`

B - Implementation of Returning Structure The first argument to `sum_list` is a pointer to the `point` result structure located in the caller's stack frame. The function is also changed to return a pointer. Essentially, the function header is changed to `point *sum(point *pResult, list_ptr ls)` and everywhere the variable `result` is used, `*pResult` is used instead. This makes it possible to avoid copying the structure to the function argument, and copying it again to return it after the completion of the function call. Register $16 holds the pointer to the `point` structure when it is passed as an argument, hence it also indicates the place in the caller's stack where the return value is to be placed.

C - Register Allocation

- `$9` holds `result.x`.
- `$10` holds `result.y`.
- `$11` holds `ls` before the recursive call. Subsequent calls overwrite the previous value. The local copy is taken from $17.
- `$12` holds a local copy of the pointer to the caller's `rest` structure. This is where the fuction should place its return value before recursing.
- `$16` holds the `&(rest)` argument to `sum_list`. It indicates the address in the caller's stack where the return structure should be placed. It is overwritten at each recursive call.
- `$17` holds the `list_ptr ls` argument to `sum_list`. It is overwritten by `ls->next` at each recursive call.
- `$1` is used as an integer scratch register.
- `$f1` and `$f10` are used as floating-point scratch registers in the `sum_list` computation.

D - Stack Frame Layout

- offset 0-7: Saved return address `$26`
- offset 8-15: Callee saved `$9`
- offset 16-23: Callee saved `$10`
- offset 24-31: Callee saved `$11`
- offset 32-39: Callee saved `$12`
- offset 40-47: `Padding`
- offset 48-55: `rest.x`
- offset 56-63: `rest.y`
- offset 64-71: `result.x` and later `rest.x`
- offset 72-79: `result.y` and later `rest.y`
- offset 80-87: `Scratch space` used to convert FP to integer
- offset 88-95: `Padding`

E - Annotated Assembly Code

```
       .verstamp 3 11
       .set noreorder            # prevent reordering if assembly instructions
       .set volatile             # loads and stores may not be moved in relation
                                 # to each other or removed by an optimization pass.
       .set noat                 # permit the use of the $at register by the program.
       .file       1 "structure.c"
gcc2_compiled.:
__gnu_compiled_c:
.text
       .align 3                  # quadword alignment
       .globl sum_list           # identify sum_list as an external symbol
                                 # and require the linker to resolve it.
       .ent sum_list             # entry point of the sum_list procedure

# The arguments to sum_list are $16 (&(rest) in the callers stack frame)
# and $17 (list_ptr ls).
sum_list:
       ldgp $29,0($27)           # setup the global pointer gp
sum_list..ng:                    # the local entry point
       lda $30,-96($30)          # point to bottom of current stack frame
       .frame $30,96,$26,0       # describe the stack frame:
                                 # the frame register is $30.
                                 # the stack frame is 96 bytes long.
                                 # the return address is stored in $26.
                                 # there are 0 bytes between the virtual
                                 # frame pointer and the local variables.
       stq $26,0($30)            # save return address
       stq $9,8($30)             # save s0
       stq $10,16($30)           # save s1
       stq $11,24($30)           # save s2
       stq $12,32($30)           # save s3
       .mask 0x4001e00,-96       # set a mask with a bit turned on for each
                                 # register the current routine saved.
                                 # The least significant bit corresponds to $1.
                                 # The offset indicates the distance in bytes
                                 # from the virtual frame pointer to where the
                                 # registers are saved.
       .prologue 1               # make the end of the prologue section of the
                                 # procedure. The flag is set, indicating that gp
                                 # is used by the procedure.

       # argument registers are trashed by subsequent calls
       bis $16,$16,$12           # local copy of pointer to caller's "rest"
       bis $17,$17,$11           # local copy of "ls"
       bne $11,$34               # check if "ls" is NULL

       # "ls" was determined to be NULL
       bis $31,$31,$9            # zero $9  (result.x)
       bis $9,$9,$10             # zero $10 (result.y)
       br $31,$35                # clean up and return
       .align 4                  # octaword alignment

       # "ls" was determined to be nonzero.  Build child arguments and
       # recurse.
$34:
       addq $30,48,$16           # pass &rest in current frame to child
       ldq $17,0($11)            # pass ls->next to child
       bsr $26,sum_list..ng      # recurse
       ldq $1,48($30)            # load rest.x
       stq $1,64($30)            # move rest.x to result.x (why?)
       ldq $1,56($30)            # load rest.y
       stq $1,72($30)            # move rest.y to result.y (why?)
       ldt $f1,8($11)            # load ls->pt.x
       ldt $f10,64($30)          # load rest.x
       addt $f1,$f10,$f10        # $f10 = ls->pt.x + rest.x
       stt $f10,80($30)          # prepare to move fp data to integer reg
```

```
        ldq $9,80($30)            # result.x = ls->pt.x + rest.x
        ldt $f1,16($11)           # load ls->pt.y
        ldt $f10,72($30)          # load rest.y
        addt $f1,$f10,$f10        # $f10 = ls->pt.y + rest.y
        stt $f10,80($30)          # prepare to move fp data to integer reg
        ldq $10,80($30)           # result.y = ls->pt.y

        # Clean up and return.  At this time $9 and $10 contain the bit
        # patterns of the x and y coordinates to be returned to the caller.
$35:
        stq $9,0($12)             # write result.x to rest.x in caller's frame
        stq $10,8($12)            # write result.y to rest.y in caller's frame
        bis $12,$12,$0            # return pointer to "rest" in caller's frame
        ldq $26,0($30)            # restore return address
        ldq $9,8($30)             # restore s0
        ldq $10,16($30)           # restore s1
        ldq $11,24($30)           # restore s2
        ldq $12,32($30)           # restore s3
        addq $30,96,$30           # restore sp (bottom of caller's frame)
        ret $31,($26),1           # return
        .end sum_list
```

Problem 11: Leaf Procedures

The doubly recursive solution to the Fibonacci function is elegant, but inefficient. The file *ASSTDIR*/`fib.c` shows the C code for the function. The code generated by GCC with the `-O` flag is shown in the file *ASSTDIR*/`fib.s`. Without changing the basic implementation, i.e., keep it doubly recursive, try and make is subtantially faster by optimzing the assembly code.

Document the following:

A. Explain your optimization and why it speeds up fib

B. Plot the time taken to run fib for fib 1 to fib 40 with and without your optimization.

C. Determine the cost of allocating a frame for a procedure using this data.

Solution 11:

A - Optimization We have applied several optimizations. First of all, we eliminate all memory references associated with stack manipulation in the base case (that is, the leaf codepath), since the computation is trivial enough to be performed entirely in caller-saved registers. While negligible for small values of n, the performance improvement is significant for $n >= 40$ due to the asymptotic growth characteristics of fib. This optimization alone provided most of the performance improvement we observed. Additionally, we use $0 to hold both the return value of the cmple, and the return value of the fib() call, therby eliminating one lda instruction. Note that $0 always holds the return value of the function call. Then, we change the way we save registers in the stack. Originally we load the address of our frame to $30 and then we store the callee saved registers we modify. However, the stores depend on the value of the previous lda, potentially stalling the pipeline. Instead, we directly manipulate the offsets and store the registers to the appropriate location at the stack frame, and load $30 at the end. Also, we choose to compute the argument to the fib(n-2) call first. That way, we can reuse $10 to hold the result of fib(n-1). This eliminates the usage of $9, and removes the stq/ldq instructions needed to save/restore its value at every function call. In addition, by using one less

register we can get rid of the space allocated for it at the stack frame, and we no longer need the padding to cache-align the frame. Thus, the new stack frame is only 16 bytes long, and exactly 4 of them fit in a cache line, effectively doubling its density from the original size of 32 bytes. Finally, we remove the L3 label since it leads to dead code, and eliminate the branch to L4. All the additional optimizations besides the elimination of stack frame allocation for the base case result in a 23 % performance increase. The code has been modified as follows:

```
        .verstamp 3 11
        .set noreorder
        .set volatile
        .set noat
        .arch ev4
.text
        .align 5
        .globl fib
        .ent fib
fib:
        .frame $30,16,$26,0     # the stack frame is only 16 bytes
        .mask 0x4000600,-16     # correct mask: only $10 and $26 saved
        ldgp $29,0($27)

# If the argument to fib() is less than 2, return 1
# As an additional optimization, use $0 to hold both
# the return value of the cmple, and the return value
# of the fib() call, therby eliminating one lda instruction.

$fib..ng:
        cmple $16, 2, $0        # ret = ((n<=2) ? (1:0))
        bne $0, fastret

# Move lda to the end so that the stq instructions don't
# depend on it. Change the offsets from $30 appropriately.

        stq $10,-8($30)
        stq $26,-16($30)
        lda $30,-16($30)
        .prologue 1

        mov $16,$10
        subl $10,1,$16
        bsr $26,$fib..ng

# Compute the argument to the next fib(n-2) call first.
# Then, reuse the register to hold the result of fib(n-1).
# This eliminates one register, and removes the stq/ldq
# instructions. The stack frame is also denser, as it can
# be compressed to 16 bytes with no padding necessary to
# cache-align it. There are exactly 4 frames per cache line.

        subl $10,2,$16
        mov $0,$10
        bsr $26,$fib..ng
        addl $10,$0,$0
        .align 4

# Remove the L3 label since it is dead code, and eliminate
# the branch to L4.

$L4:
        ldq $26,0($30)
        ldq $10,8($30)
        lda $30,16($30)

fastret:
        ret $31,($26),1
        .end fib
```

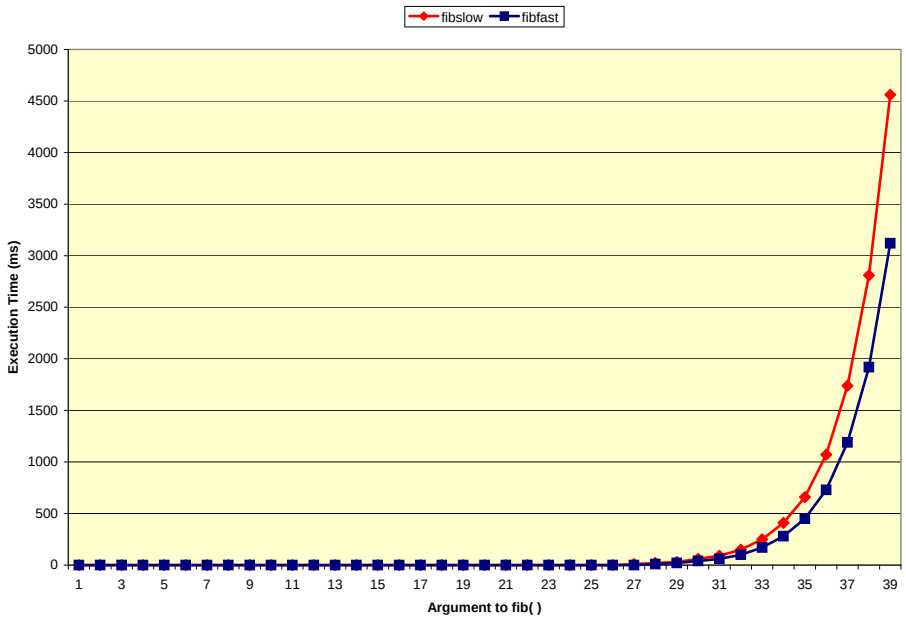

Figure 2: The running times of fibfast and fibslow

B - Performance Evaluation See Figure 2.

C - Discussion An invocation of `fib(n)` results in precisely $fib(n)$ executions of the leaf codepath. The above plot gives execution times of 7377.584 ms and 6476.736 ms for `fibslow 40` and `fibfast 40`, respectively. Since `fibfast` and `fibslow` differ only in the implementation of the base case (we are ignoring the effect of the optimization on the recursive step), and the cost difference is due entirely to stack manipulation, the cost of allocating a stack frame is given by $\frac{7377.584\ ms - 6476.736\ ms}{fib(40)} = 8.803\ ns$. Note that the measurements were done with a version of the code that optimizes only the frame allocation for the base case as discussed above.

Long Jumps

(NOTE: this last section is for extra credit only.)

In this section, you will enhance your understanding of Alpha calling and returning conventions by examining the Unix `setjmp` library, which in effect cheats on the standard procedure calling conventions.

`setjmp` provides a mechanism for nonlocal procedure exits. By using `setjmp` and `longjmp`, a procedure can exit to a function a long way up the procedure chain, bypassing its calling procedure.

This is typically used for error handling, in the following style:

```
jmp_buf env;
```

```
main()
{
   int jval;
   if ((jval = setjmp(env)) != 0) {
     /* Do error recovery and cleanup. */
     /* And possibly try to execute again. */
   } else {
     /* Do normal case. */
     ...
   }
}
```

The first time `setjmp()` is called, it sets up the buffer env with enough information to reconstruct the state of the registers and the stack. It returns 0 to the calling function.

The calling function then does other code, and deep within some complicated hierarchy of other functions, it may find some error:

```
deep_within_some_complicated_hierarchy ()
{
  if (error_detected) {
     longjmp(env, ERROR_CONSTANT);
  }
  ...
}
```

`longjmp()` restores the state that was saved in `setjmp()` and resets the stack pointer to the position it was in when `setjmp()` was called.

The file `ljdemo.c` contains a somewhat contrived example of the use of `setjmp()` and `longjmp()`. Study this code to better understand the behavior of these constructs. Try compiling and running it (it should work on any Unix machine).

Problem 12: *(For Extra Credit)* **Understanding the Implementation of Long Jumps**

The file *ASSTDIR*`/longjmp.dis` contains a disassembly of the code for `___setjmp()` and `_longjmp()`, which are simplified versions of `setjmp()` and `longjmp()`. In addition, the file also includes `__longjump_resume()`, which is called by `_longjmp()` This code was extracted from the Unix library `/usr/lib/libc.a` on one of our Alpha machines. Your job is to create annotated versions of these three procedures. Show clearly what state is being saved and restored, and how longjmp subverts the Alpha calling routines to appear to 'return' to a place other than that from which it was called.

It may help you understand the code to write down what data is stored in which places in the `jmp_buf`. If you write this information down, turn that in too.

Hints:

- The `ldt` and `stt` instructions load and store 64-bit numbers into/from the *floating-point* register file. Their behavior is analogous to `ldq` and `stq` for integer data.
- The value `0xacedbade` is used as a "magic token" to identify a buffer that has (probably) been set by `setjmp()` (A moment's thought should convince you that doing a `longjmp()` to a buffer that has not been set up by `setjmp()` is likely to result in errors that are very difficult to track down.) This value is generated by a combination of three instructions:

```
ldah r1, 22135(r31)
lda  r1, -8849(r1)
addq r1, r1, r1
```

- The `ldah` instruction, which stands for Load Address High, multiplies a signed 16-bit constant by 65536, and then adds this value into the destination register. The `lda` instruction, which stands for Load Address, generates the effective address address in the destination register (in this case, it subtracts 8849 from `r1`).

Solution 12: *(Courtesy of J. Hendricks, D. Koes, D. Malayeri)*

```
# The env structure stores the certain information at certain offsets:
# offset info
# 16 r26
# 104 r9
# 112 r10
# 120 r11
# 128 r12
# 136 r13
# 144 r14
# 152 r15
# 240 r26 (again)
# 264 gp
# 272 sp
# 280 0xacedbade

# r16 is the first argument (the env pointer to the jmpbuf)
# This routine saves a bunch of state.  It doesn't need to save
# temporary registers since they can't have important values in them
# anyway (since this is a function call).

___setjmp:
  0x10: 241f5677 ldah r0, 22135(r31) # r0 := 0x56770000
# why does the gp get stored?
# it isn't preserved across function calls...
# hmm..
  0x14: b7b00108 stq gp, 264(r16) # *(r16+264) := gp
# r27 is the current procedure's address
  0x18: 243b0001 ldah r1, 1(r27) # r1 := r27+1
# store a permanent (callee save) register to env
  0x1c: b5500070 stq r10, 112(r16) # *(r16+112) := r10
  0x20: 2000dd6f lda r0, -8849(r0) # r0 := r0 - 8849 = 0x5676dd6f
# store another perm register to env
  0x24: b5700078 stq r11, 120(r16) # *(r16+120) := r11
  0x28: 202180e0 lda r1, -32544(r1) # r1 := r1-32544
# store another perm register to env
  0x2c: b5900080 stq r12, 128(r16) # *(r16+128) := r12
  0x30: 40000400 addq r0, r0, r0 #r0 := 2*r0 = 0xacedbade
# store the magical value 0xacedbade into env
  0x34: b4100118 stq r0, 280(r16) # *(r16+280) := r0
# store another perm register to env
  0x38: b5b00088 stq r13, 136(r16) # *(r16+136) := r13
# set our return value to 0
  0x3c: 47ff0400 bis r31, r31, r0 # r0 := 0
# store another perm register to env
  0x40: b5d00090 stq r14, 144(r16) # *(r16+144) := r14
  0x44: 443f041d bis r1, r31, gp # gp := r1
# store another perm register to env
  0x48: b5f00098 stq r15, 152(r16) # *(r16+152) := r15
# store the return address
  0x4c: b7500010 stq r26, 16(r16) # *(r16+16) := r26
# store all the floating point permanant (callee save) registers
```

```
  0x50: 9c500138 stt $f2, 312(r16)
  0x54: 9c700140 stt $f3, 320(r16)
  0x58: 9c900148 stt $f4, 328(r16)
  0x5c: 9cb00150 stt $f5, 336(r16)
  0x60: 9cd00158 stt $f6, 344(r16)
  0x64: 9cf00160 stt $f7, 352(r16)
  0x68: 9d100168 stt $f8, 360(r16)
  0x6c: 9d300170 stt $f9, 368(r16)
# store the return address (again) in a different location
  0x70: b75000f0 stq r26, 240(r16) # *(r16+240) := r26
# store the stack pointer
  0x74: b7d00110 stq sp, 272(r16) # *(r16+272) := sp
# store another perm register to env
  0x78: b5300068 stq r9, 104(r16) # *(r16+104) := r9
  0x7c: 6bfa8001 ret r31, (r26), 1 # return to whence we were
                                                # called


# this routine does the actual restore from the env pointer
# so all the state saved in setjmp gets put back where it came from

__longjump_resume:
# load the saved return address (from the setjmp) into r26
 0x110: a75000f0 ldq r26, 240(r16) # r26 := *(r16+240)
# the return value needs to get set to the second argument to longjmp (r17)
 0x114: 47f10400 bis r31, r17, r0 # r0 := r17
 0x118: a7b00108 ldq gp, 264(r16) # gp := *(r16+264)
 0x11c: 461f0412 bis r16, r31, r18 # r18 := r16 who knows why
# load a callee save integer register from saved state
 0x120: a5300068 ldq r9, 104(r16)
# if the second argument to longjmp is zero, return value is actually 1
# although in this disassembled code it appears to be 3, strange
 0x124: 44003480 cmoveq r0,0x11, r0

# restore the callee save integer registers
 0x128: a5500070 ldq r10, 112(r16)
 0x12c: a5700078 ldq r11, 120(r16)
 0x130: a5900080 ldq r12, 128(r16)
 0x134: a5b00088 ldq r13, 136(r16)
 0x138: a5d00090 ldq r14, 144(r16)
 0x13c: a5f00098 ldq r15, 152(r16)
# restore the callee save floating point registers
 0x140: 8c500138 ldt $f2, 312(r16)
 0x144: 8c700140 ldt $f3, 320(r16)
 0x148: 8c900148 ldt $f4, 328(r16)
 0x14c: 8cb00150 ldt $f5, 336(r16)
 0x150: 8cd00158 ldt $f6, 344(r16)
 0x154: 8cf00160 ldt $f7, 352(r16)
 0x158: 8d100168 ldt $f8, 360(r16)
 0x15c: 8d300170 ldt $f9, 368(r16)

# set the stack pointer to the saved value
 0x160: a7d00110 ldq sp, 272(r16)
# return to the caller of the previous setjmp (not _longjmp)
 0x164: 6bfa8001 ret r31, (r26), 1


# wrapper around __longjump_resume that does error checking
_longjmp:
 0x17c: 23defff0 lda sp, -16(sp) # create a small stack frame
                                                # (16 bytes)
 0x180: 27bb0001 ldah gp, 1(r27) # setup the gp
 0x184: 23bd8184 lda gp, -32380(gp)
 0x188: b75e0000 stq r26, 0(sp) # store the return address onto
                                                # the stack

# a correctly setup env will have 0xacedbade at this position in env
 0x18c: a4100118 ldq r0, 280(r16) #r0 := *(r16+280)
```

```
# next three lines put 0xacedbade into r28
 0x190: 279f5677 ldah r28, 22135(r31)
 0x194: 239cdd6f lda r28, -8849(r28)
 0x198: 439c041c addq r28, r28, r28

 0x19c: 401c05a0 cmpeq r0, r28, r0 # make sure env has right
                                                  # magic number
 0x1a0: e4000002 beq r0,0x1ac # jump ahead if r0 is false

# apparently the addres of __longjump_resume is stored in global
# data somewhere, probably to facilitate shared objects or something
 0x1a4: a77d8010 ldq r27, -32752(gp)
# call __longjump_resume, this will never return since it
# resets the return address to be the original setjmp ra
 0x1a8: 6b5b4000 jsr r26, (r27), __longjump_resume

# branch to next instruction, put the next instruction's address into r1
 0x1ac: c0200000 br r1,0x1b0
# do some funky gp stuff - we apparently have position independant data
# that is pc relative (we've loaded the pc into r1)
 0x1b0: 27a10001 ldah gp, 1(r1)
 0x1b4: 23bd8150 lda gp, -32432(gp)
 0x1b8: a77d8018 ldq r27, -32744(gp)
# call off to an error handling routine..
 0x1bc: 6b5b4000 jsr r26, (r27), __longjmperror
# do some more funky gp stuff to get the address of abort
 0x1c0: 27ba0001 ldah gp, 1(r26)
 0x1c4: 23bd8140 lda gp, -32448(gp)
 0x1c8: a77d8020 ldq r27, -32736(gp)
# we had problems apparently and have to abort
 0x1cc: 6b5b4000 jsr r26, (r27), abort
 0x1d0: 6bfa8001 ret r31, (r26), 1
```