Dynamic Instrumentation of Binaries for Gprof Compatible Profiling

by

Michael Noeth
Jyothish S. Varma
Tao Yang

May 9, 2005

Final Project for CSC791A
Dr. Frank Muller
11:50 AM – 1:05 PM T R

Introduction

Background

A programmer's goal is to write correct, efficient code. Code must initially be written with the goal of being correct. Once correctness has been established, programmers generally strive to make their code as efficient as possible. This project focuses on creating more efficient code through the use of profiling tools. Execution profiling tools are a great asset to programmers creating efficient code. Profiling tools can give programmers insight into what functions are being called and how much time is spent in each during a program's execution. This information can be used to identify bottlenecks in a program that may be a good candidate for reexamination to improve efficiency. One such tool in common use is Gprof.

Problem Statement

Currently, Gprof instruments source code at compile time and tracks what functions were called, how many times each function was called, how long was spent in each function, and a call graph (what function called the function in question). Profiling with Gprof entails the following steps:

1. Compile your source code with the -pg option
2. Run your binary file (during this stage a gmon.out file will be generated containing profiling information)
3. Examine the profile information by running Gprof on the gmon.out file

The current framework that Gprof uses may be too restrictive for certain profiling activities. Gprof requires that the source code be recompiled with the -pg option specified in order for profiling instrumentation to take place. This can be too restrictive in the event that (1) only the binary is available (the source code is not accessible), (2) compilation time is prohibitively long, or (3) the programmer wishes to instrument dynamic libraries for profiling that the program calls during execution. We propose creating a tool for binary instrumentation of Gprof profiling.

Design

We set out to create a tool that inserts instrumentation for Gprof profiling into arbitrary binaries. Dyninst provided the means for adding additional code to an already compiled binary. We extracted the code used by the Gnu C Compiler when the -pg option was specified and inserted it into arbitrary binaries.

When source code compiled with the -pg is run, we discovered that “mcount” is called at the beginning of each function in the program. At the end of execution, the final call to mcount generates the gmon.out file that contains Gprof profiling data. In order to reproduce the gmon.out file for code compiled without the -pg option, we used Dyninst to insert calls to mcount at the beginning of each function.

See the Methodology section for more information.

Results

To test the accuracy of our tool, we compared the resulting profiles of programs compiled with the -pg option specified and profiles of programs instrumented with our tool. We compared three different programs: (1) a simple micro benchmark that made many function calls, (2) the LINPACK benchmark, and (3) the llloop benchmark. We ranked the difficulty of the benchmarks as follows in table 1:

Benchmark	Number of functions	Number of function calls	Overall complexity ranking
Our micro benchmark	5	683234	3
LINPACK Benchmark	10	139183	2
Llloop Benchmark	9	176023595	1

Table 1: Ranking of benchmarks used to verify correctness of binary instrumentation tool

We ranked complexity from 1 to 3 with 1 being the most complex program based on the number of functions in the call graph and the total number of function calls. Since we wrote our own benchmark, this received the lowest complexity rating. Our tool worked well for our micro benchmark as well as LINPACK. For the most complex benchmark (llloop benchmark) we were able to replicate only some of the profile by compiling with the -pg option. We leave the profiling of complex benchmarks to future works.

See the results section for more information.

Methodology

Overview

Our instrumentation of arbitrary binaries took place over the course of four stages. The first stage consisted of a familiarization with the tools we would be using (Gprof, Gnu C Compiler, and Dyninst). The second stage consisted of proving we could insert profiling code directly into the source of an application and ensuring the profile was correct. The third stage extended the second by ensuring that the inserted source code could be called from a dynamic library. This additional stage proved necessary because Dyninst calls its snippets from a dynamic library. The final stage consisted of dynamically inserting profiling instrumentation directly into arbitrary binaries using Dyninst.

Stage 1: Familiarization with tools

The first step of the project was becoming familiar with the tools we would be using to develop a method of instrumenting binaries with Gprof profiling. The tools we focused on were Gprof, the Gnu C Compiler (gcc), and Dyninst. Mike Noeth and Jyothish Varma worked to become team experts on Gprof and gcc while Tao Yang became the expert on Dyninst.

Through use of GDB and examination of the gcc source code, we discovered that a function called mcount was called at the beginning of every function to be profiled by Gprof. On the first call to mcount, all data structures for profiling were setup, and an initial call to profil (a system call to perform a stack walk and collect information every 10 milliseconds – see man profil for more information) was made. Subsequent calls to mcount resulted in tallying calls to each individual function while the profil continued to perform its stack walk every 10 milliseconds. The final call to mcount resulted in all the data collected being written to file (gmon.out) and stopping the profil stack walk. The mcount function was extracted from the gcc source code and compiled into an object file from which we could link the mcount call into our programs.

Familiarization with Dyninst was broken down into three major tasks to be prototyped. The first task was to specify a binary for Dyninst to instrument with code. This would later allow our tool to instrument a target binary with Gprof profiling code. The second task required that we be able to identify all the functions used in a program. This would later allow our tool to identify the functions to be prototyped. The third task required that we be able to insert code at the beginning of a function. This would later allow our tool to instrument the necessary call to mcount at the beginning of each function.

Stage 2: Static compilation of mcount

The second stage consisted of creating a simple application and inserting profiling code into the existing source code by hand. This would ensure that we understood how mcount was working, and that we knew when and where to call it. Two binaries were compiled: one with the -pg option specified and the other with the profiling code inserted by hand. Both binaries generated gmon.out profiles that could be examined using Gprof. To complete this stage, we needed to ensure the resultant profiles of both binaries matched.

We began this stage by creating a micro benchmark that would serve as our application (appendix A – Micro benchmark source code). The benchmark made multiple function calls to create a somewhat complex call graph. We compiled the micro benchmark and ensured it executed as expected.

Next, we modified the micro benchmark with calls to mcount at the beginning of each function. The modified micro benchmark was linked to the object file containing mcount obtained in stage 1.

We compiled two binaries: the first was the unmodified micro benchmark compiled with the -pg option specified; the second was the modified micro benchmark linked with the mcount code obtained in stage 1. Both binaries generated gmon.out files which were examined using Gprof. The resultant profiles matched exactly in the number of calls made to each function as well as the time spent in each function. From here we decided we were ready to move onto the next stage.

Stage 3: Dynamic compilation of mcount

Originally, we did not plan on a third stage. We thought that after proving our design via stage 2, we would be able to move on to stage 4 (implementing a tool to instrument binaries with Gprof profiling). After attempting stage 4 and seeing no success, we had to back track. Reviewing Dyninst documentation revealed that the code we were inserting was called from a dynamic library. Thus we decided that we had to call the mcount function from a dynamic library (by hand) to ensure this was not the cause of failure in stage 4.

The mcount function discovered in stage 1 was compiled into a dynamic library. We loaded our dynamic library (appendix A – Wrapper source code) and rather than making static calls to mcount in the modified micro benchmark, we called mcount from the dynamic library.

This additional stage revealed that part of our problem was due to name collisions. When gcc is compiled and installed, it adds an additional library with mcount and its supporting functions. Rather than calling our profiling code from the dynamic library we created, the functions from the library installed with gcc were being invoked. By renaming mcount and all the supporting functions, we were able to avoid the name collisions.

Stage 4: Dynamic instrumentation of mcount

With a new dynamic library created from stage 3 (with new names for mcount and its supporting functions) we attempted to create our tool again. We simply attempted to call the renamed mcount at the beginning of each function for an arbitrary binary. There were two complications at this phase. The first complication was that mcount requires knowing the address range of the code it is profiling, but this information was being corrupted due to the use of dynamic libraries. The second complication was that Dyninst was messing up the stack. Thus, the stack walk code was unable to perform correctly.

The first call to mcount setup data structures for tracking the profiling data. To allocate space for this data to be collected, mcount requires knowing the address range of the program being profiled. In the gcc version of mcount, a hard coded value is used to specify the beginning address of the code segment, and a function etext (we believe this stands for end of text segment) returns the address of the last function in the text segment of a program to be used as the ending address of the code segment. The ranges were coming out incorrect due to another naming collision with etext. The etext function was not returning the correct address anymore. To resolve this issue, we used Dyninst to scan all the functions be profiled, collect their addresses, and pick the largest and smallest address found. The smallest address specified the beginning of the range while the large address specified the end of the range. With this modification, we were able to work around the name collision of etext.

In order to create a call graph, Gprof requires knowledge of which function called mcount (we will refer to this function as "Caller #1"), and knowledge of which function called Caller #1 (we will refer to this function as "Caller #2"). To obtain these values, the mcount code uses a system call __builtin_addr(int level). To get Caller #1, mcount uses __builtin_addr(0), and to get Caller #2, mcount uses __builtin_addr(1). When Dyninst is used to insert calls to mcount, the stack is corrupted (see figure 1). Due to Dyninst's method of inserting code, mcount could no longer determine Caller #2. We tried to use increasing levels of __builtin_addr, but these resulted in memory address from the heap (where Dyninst created its own fake stack). When the top of the stack is reached, any higher levels cause the application to segfault.

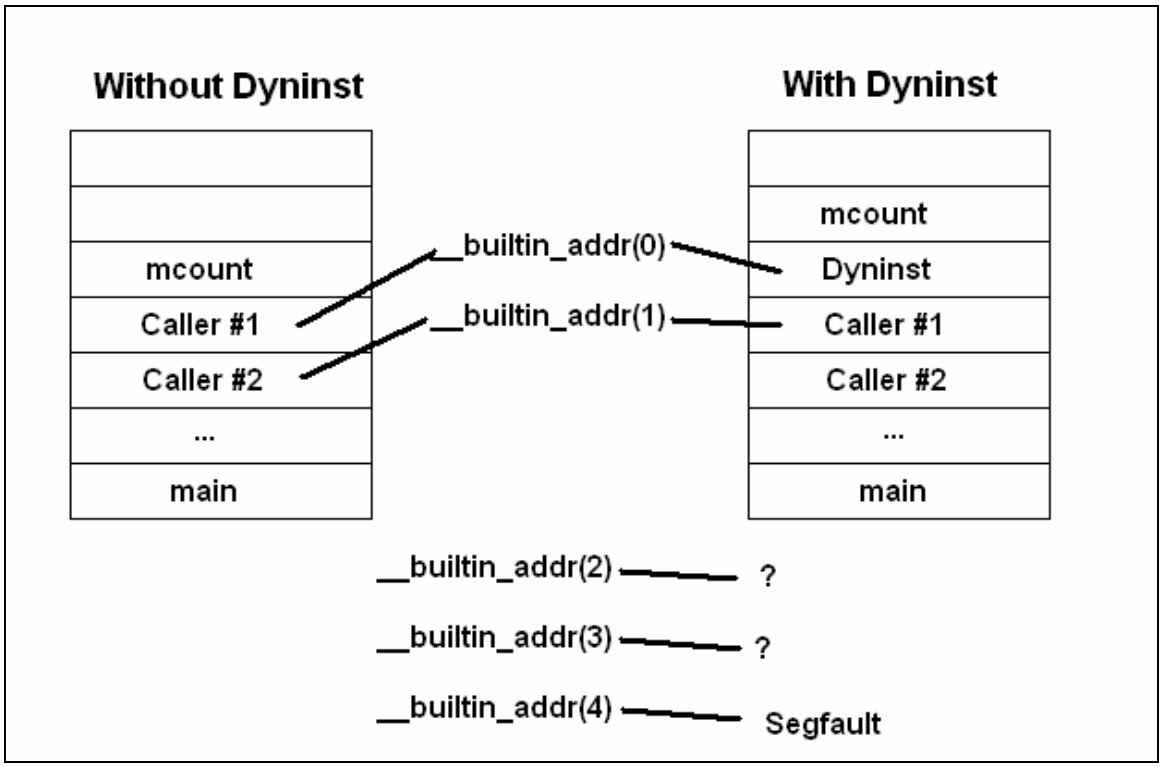

Figure 1: Comparison of program stacks with and without Dyninst

To work around the Dyninst stack, we implemented our own simple stack (appendix A – Fake stack source code). Upon entering a new function, we pushed that functions address onto the stack. Upon leaving a function, we popped the address off of the stack. We modified mcount so that it would use the address of the item on the top of the stack for Caller #1 (rather than __builtin_addr(0)) and the item one away from the top of the stack for Caller #2 (rather than __builtin_addr(1)). With this modification, we were able to work around the stack corruption that Dyninst introduced.

After implementing our own range finder and stack, we were able to generate the correct profiles for arbitrary binaries. The proof of our design is shown in the next section.

Results

To verify that our tool is able to correctly instrument Gprof profiling, we compared the profiles of three applications instrumented with our tool against the profiles of these applications compiled with the -pg option specified. See table 1 for more details on the applications profiled. For each application profiled we will show (1) a side by side comparison of the Gprof call graph for our tool and a -pg version, (2) a graph of the time spent break down for our tool and a -pg version, and (3) a brief commentary about the profiles.

The first application compared was the micro benchmark originally written to test correctness in stages 2 and 3 of our implementation. We deemed this benchmark the easiest to profile because it did no real work, and was written as a quick test of our concepts.

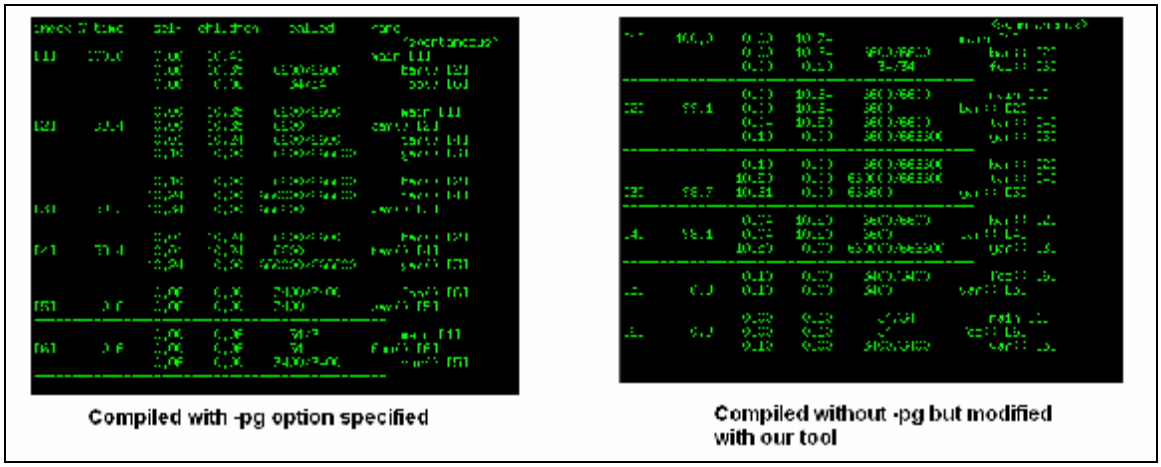

Figure 2: Comparison of call graphs for –pg version of the micro benchmark versus the micro benchmark modified with our tool

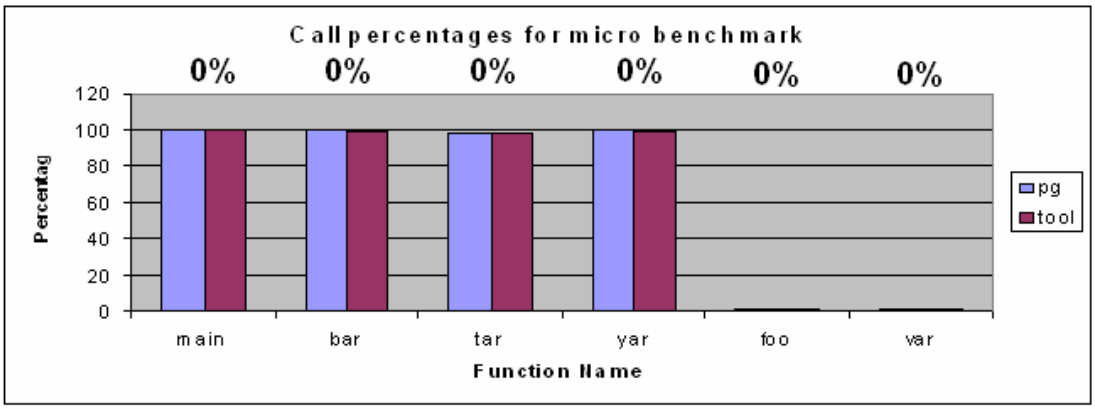

Figure 3: Comparison of time breakdowns for –pg version of the micro benchmark versus the micro benchmark modified with our tool. Note that the percentages at the top indicate the % difference between the two bars.

In both figures 2 and 3 we see that for the micro benchmark, our tool performs flawlessly. The call graphs match in the -pg version and our tool. There is also no difference between the call percentages either.

The second application compared was the LINPACK benchmark. This benchmark was introduced by Jack Donarra as a floating point performance benchmark. It is far more complex than our micro benchmark in that it actually does work. This benchmark is not the most complex because it was written for far slower computers and only runs for less than 1 second.

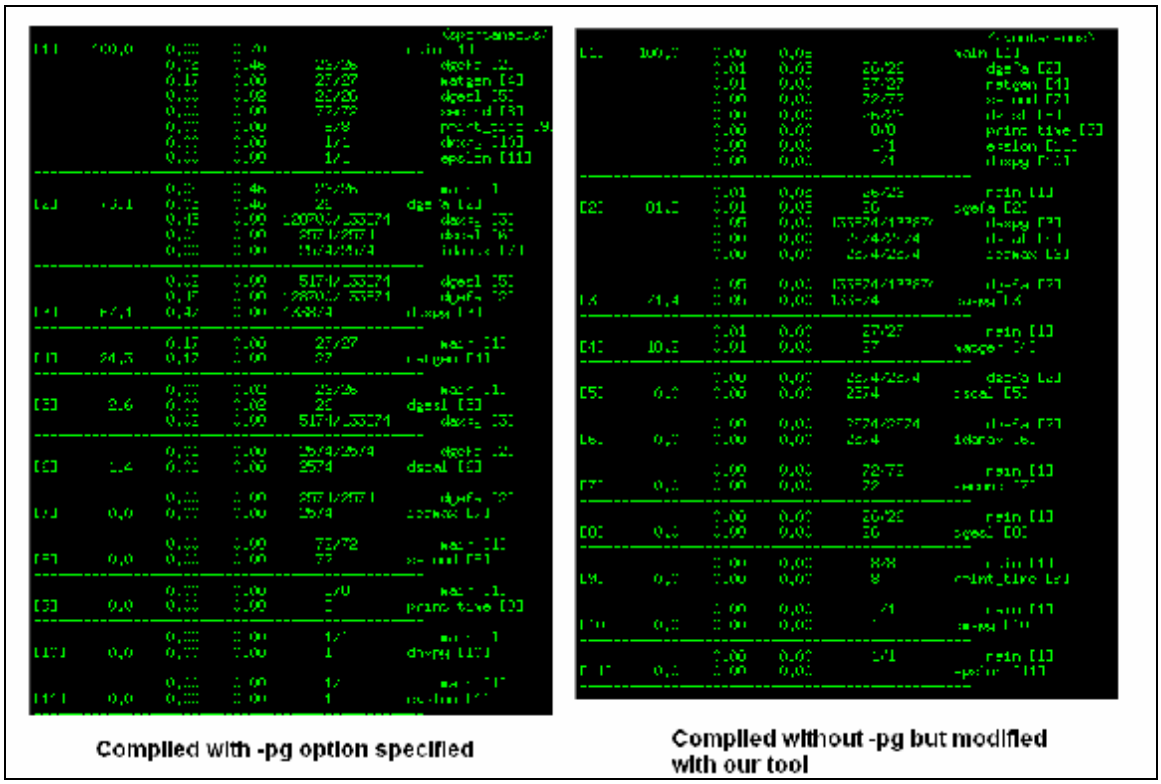

Figure 4: Comparison of call graphs for –pg version of the LINPACK benchmark versus the LINPACK benchmark modified with our tool

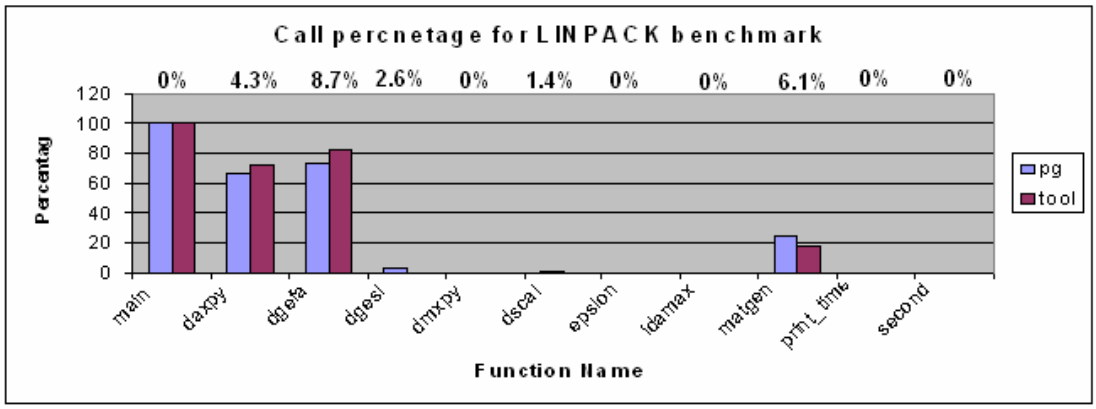

Figure 5: Comparison of time breakdowns for –pg version of the LINPACK benchmark versus the LINPACK benchmark modified with our tool. Note that the percentages at the top indicate the % difference between the two bars.

In both figures 4 and 5 we see that for the LINPACK benchmark, our tool performs well (but not perfect on the percentages). The call graphs match in the -pg version and our tool. There are slight differences between the call percentages. We attribute this percentage difference to machine load and a difference in the cost of the original mcount and our modified mcount.

The third application compared was llloop. This benchmark is the most complex benchmark used. It does real work, has many function calls, and runs for a long time. This was the most strenuous test for our tool.

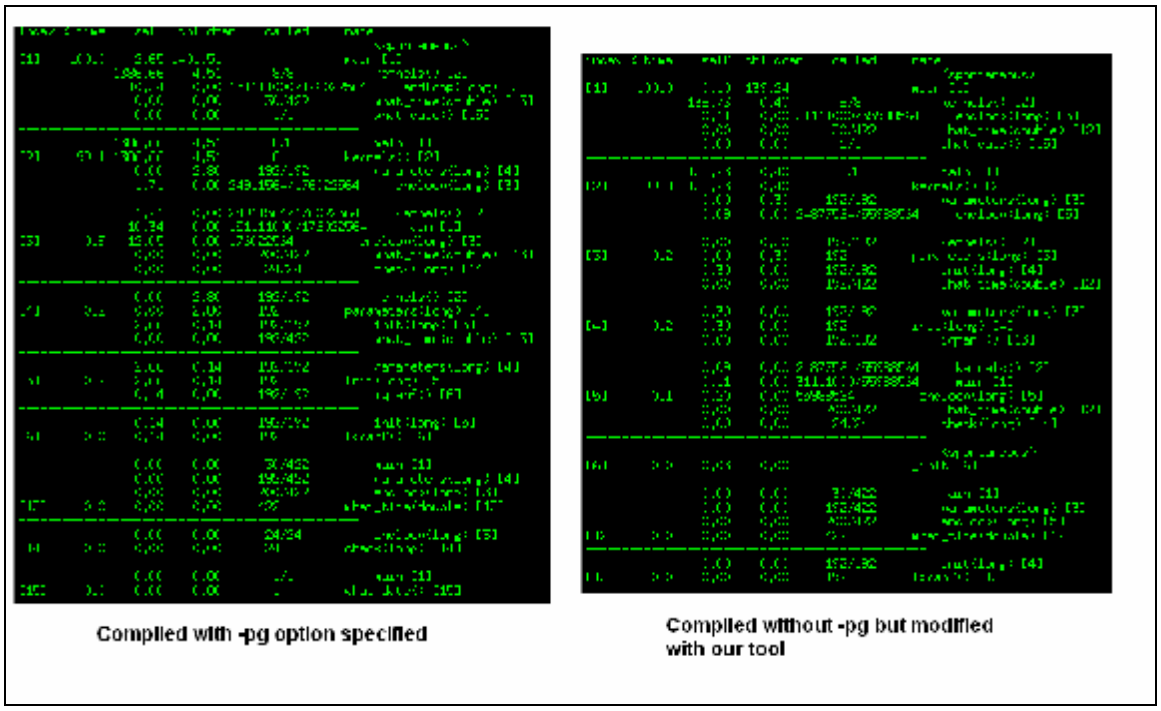

Figure 6: Comparison of call graphs for –pg version of the llloop benchmark versus the llloop benchmark modified with our tool

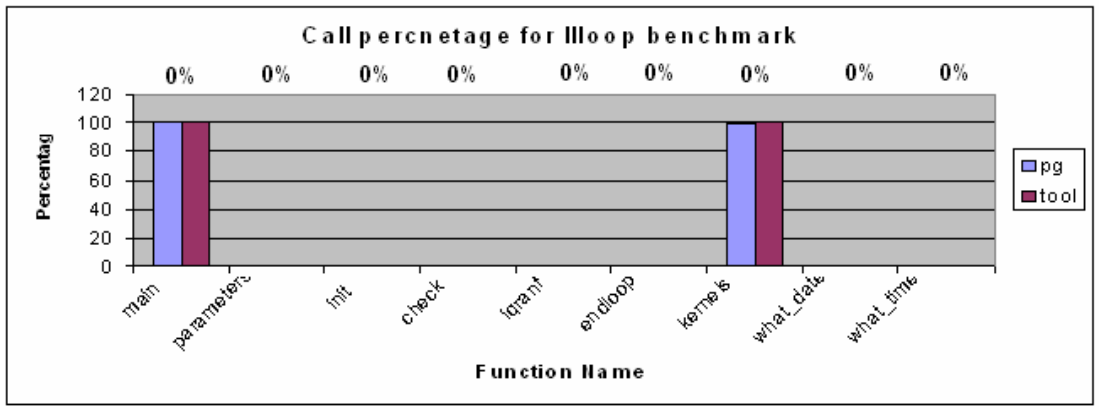

Figure 7: Comparison of time breakdowns for –pg version of the llloop benchmark versus the llloop benchmark modified with our tool. Note that the percentages at the top indicate the % difference between the two bars.

In both figures 6 and 7 we see that for the llloop benchmark, our tool performs well but fails to capture extremely large numbers of calls. The call graphs match in the -pg version and our tool except that for functions endloop, main, and kernels. There appears to be some sort of overflow error occurring. We have put off solving this problem to future works. There is no difference between the call percentages.

Future Works

We have determined that there are 3 modifications / improvements immediately visible for our binary instrumenting tool. They are:

1. Allow for extremely large numbers (i.e. llloop benchmark) to not cause differences in the -pg call graph and the tool's resultant call graph.
2. When Dyninst modifies a binary, it is able to write the resultant image to disk. We were unable to get the resultant image to run. There was a problem with loading the mcount code from a dynamic library.
3. Allow the user to expand the range of memory profiled. This might enable a user to profile not just his / her application, but also the dynamically loaded libraries that the application calls as well. We noticed while working with Dyninst that we were able to capture the addresses of these functions in the dynamically loaded libraries, but never attempted to profile them.

Work Breakdown

Michael Noeth

- Wrote all progress reports
- Familiarized with gcc and Gprof
- Extracted mcount code
- Helped implement etext and fake stack solution

Jyothish S Varma

- Web page
- Familiarized with gcc and Gprof
- Extracted mcount code
- Helped implement etext and fake stack solution

Tao Yang

- Familiarized with Dyninst
- Wrote most of initial code for Dyninst mutator
- Helped implement etext and fake stack solution

References

http://www.dyninst.org/
http://www.gnu.org/software/binutils/manual/gprof-2.9.1/
http://www.dyninst.org/papers/apiPreprint.pdf
http://docs.freebsd.org/44doc/psd/18.gprof/paper.html
http://docs.freebsd.org/44doc/papers/kerntune.pdf
http://docs.hp.com/en/B3909-90002/ch06s01.html
http://www.netlib.org/benchmark/top500/lists/linpack.html

Appendix A

Micro benchmark source code

```
#include <stdio.h>

void foo();
void bar();
void tar();
void var();
void yar();

//extern void (*ptr_internal_mcount)(void);
//extern int load_play();

int main(int argc, char *argv[])
{
  int i,j;

#ifdef PG_NOTON
  load_play();
#endif

  for(i = 0; i < 100; i++)
  {
    if(i % 3 == 0)
      foo();
    else
      for(j = 0; j < 100; j++)
      bar();
  }
}

void foo()
{
  int i;

#ifdef PG_NOTON
  ptr_internal_mcount();
#endif

  for(i = 0; i < 100; i++)
    var();
}

void bar()
{
#ifdef PG_NOTON
  ptr_internal_mcount();
#endif


  tar();
  yar();
}

void tar()
```

```
{
  int i;

#ifdef PG_NOTON
  ptr_internal_mcount();
#endif

  for(i = 0; i < 100; i++)
    yar();
}

void var()
{
  int i;

#ifdef PG_NOTON
  ptr_internal_mcount();
#endif

  for(i = 0; i < 1000; i++);
}

void yar()
{
  int i;

#ifdef PG_NOTON
  ptr_internal_mcount();
#endif

  for(i = 0; i < 1000; i++);
}
```

Modified gmon.c source code

```
/*-
 * Copyright (c) 1991 The Regents of the University of California.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in
the
 *    documentation and/or other materials provided with the
distribution.
 * 3. All advertising materials mentioning features or use of this
software
 *    must display the following acknowledgement:
 *    This product includes software developed by the University of
 *    California, Berkeley and its contributors.
 * 4. Neither the name of the University nor the names of its
contributors
```

```
 *    may be used to endorse or promote products derived from this
software
 *    without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS''
AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE
LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE
GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY
OF
 * SUCH DAMAGE.
 */

/*
 * This is a modified gmon.c by J.W.Hawtin <oolon@ankh.org>,
 * 14/8/96 based on the original gmon.c in GCC and the hacked version
 * solaris 2 sparc version (config/sparc/gmon-sol.c) by Mark Eichin. To
do
 * process profiling on solaris 2.X X86
 *
 * It must be used in conjunction with sol2-gc1.asm, which is used to
start
 * and stop process monitoring.
 *
 * Differences.
 *
 * On Solaris 2 _mcount is called by library functions not mcount, so
support
 * has been added for both.
 *
 * Also the prototype for profil() is different
 *
 * Solaris 2 does not seem to have char *minbrk whcih allows the
setting of
 * the minimum SBRK region so this code has been removed and lets pray
malloc
 * does not mess it up.
 *
 * Notes
 *
 * This code could easily be integrated with the original gmon.c and
perhaps
 * should be.
```

```
 */

#ifndef lint
static char sccsid[] = "@(#)gmon.c  5.3 (Berkeley) 5/22/91";
#endif /* not lint */

#include <stdio.h>

#if 0
#include <unistd.h>

#endif
#ifdef DEBUG
#include <stdio.h>
#endif

#if 0
#include "i386/gmon.h"
#else

struct phdr {
                char    *lpc;
                char    *hpc;
                int     ncnt;
};


#define HISTFRACTION 2
#define HISTCOUNTER unsigned short
#define HASHFRACTION 1
#define ARCDENSITY 2
#define MINARCS 50
#define BASEADDRESS 0x8000000 /* On Solaris 2 X86 all executables start
here
                         and not at 0 */

struct tostruct {
  char *selfpc;
  long count;
  unsigned short link;
};
struct rawarc {
    unsigned long       raw_frompc;
    unsigned long       raw_selfpc;
    long                raw_count;
};
#define ROUNDDOWN(x,y)  (((x)/(y))*(y))
#define ROUNDUP(x,y)    ((((x)+(y)-1)/(y))*(y))
#endif

/* char *minbrk; */

#ifdef __alpha
extern char *sbrk ();
#endif

    /*
```

```
     *      froms is actually a bunch of unsigned shorts indexing tos
     */
//IMP:: static int            profiling = 3;
static int        profiling = 0;
static unsigned short   *froms;
static struct tostruct  *tos = 0;
static long       tolimit = 0;
static char       *s_lowpc = 0;
static char       *s_highpc = 0;
static unsigned long    s_textsize = 0;

static int  ssiz;
static char *sbuf;
static int  s_scale;
    /* see profil(2) where this is describe (incorrectly) */
#define           SCALE_1_TO_1      0x10000L

#define     MSG "No space for profiling buffer(s)\n"

extern int errno;
//extern int test();


new_monstartup(lowpc, highpc)
    char    *lowpc;
    char    *highpc;
{
    int                 monsize;
    char          *buffer;
    register int  o;

      /*
       *    round lowpc and highpc to multiples of the density we're
using
       *    so the rest of the scaling (here and in gprof) stays in
ints.
       */
    lowpc = (char *)
          ROUNDDOWN((unsigned)lowpc, HISTFRACTION*sizeof(HISTCOUNTER));
    s_lowpc = lowpc;
    highpc = (char *)
          ROUNDUP((unsigned)highpc, HISTFRACTION*sizeof(HISTCOUNTER));
    s_highpc = highpc;
    s_textsize = highpc - lowpc;
    monsize = (s_textsize / HISTFRACTION) + sizeof(struct phdr);
    buffer = (char *) sbrk( monsize );
    fflush(stdout);

    if ( buffer == (char *) -1 ) {
      write( 2 , MSG , sizeof(MSG) );
      return;
    }
    froms = (unsigned short *) sbrk( s_textsize / HASHFRACTION );

    if ( froms == (unsigned short *) -1 ) {
      write( 2 , MSG , sizeof(MSG) );
      froms = 0;
```

```
      return;
    }
    tolimit = s_textsize * ARCDENSITY / 100;
    if ( tolimit < MINARCS ) {
      tolimit = MINARCS;
    } else if ( tolimit > 65534 ) {
      tolimit = 65534;
    }
    tos = (struct tostruct *) sbrk( tolimit * sizeof( struct tostruct )
);

    if ( tos == (struct tostruct *) -1 ) {
      write( 2 , MSG , sizeof(MSG) );
      froms = 0;
      tos = 0;
      return;
    }
/*    minbrk = (char *) sbrk(0);*/
    tos[0].link = 0;
    sbuf = buffer;
    ssiz = monsize;
    ( (struct phdr *) buffer ) -> lpc = lowpc;
    ( (struct phdr *) buffer ) -> hpc = highpc;
    ( (struct phdr *) buffer ) -> ncnt = ssiz;
    monsize -= sizeof(struct phdr);
    if ( monsize <= 0 )
      return;
    o = highpc - lowpc;
    if( monsize < o )
#ifndef hp300
      s_scale = ( (float) monsize / o ) * SCALE_1_TO_1;
#else /* avoid floating point */
    {
      int quot = o / monsize;

      if (quot >= 0x10000)
            s_scale = 1;
      else if (quot >= 0x100)
            s_scale = 0x10000 / quot;
      else if (o >= 0x800000)
            s_scale = 0x1000000 / (o / (monsize >> 8));
      else
            s_scale = 0x1000000 / ((o << 8) / monsize);
    }
#endif
    else
      s_scale = SCALE_1_TO_1;
    new_moncontrol(1);
}



new_mcleanup()
{
    int                 fd;
    int                 fromindex;
    int                 endfrom;
```

```
    char          *frompc;
    int                 toindex;
    struct rawarc rawarc;

    new_moncontrol(0);
    fd = creat( "gmon.out" , 0666 );
    if ( fd < 0 ) {
      perror( "mcount: gmon.out" );
      return;
    }
#   ifdef DEBUG
      fprintf( stderr , "[mcleanup] sbuf 0x%x ssiz %d\n" , sbuf , ssiz
);
#   endif

    write( fd , sbuf , ssiz );
    endfrom = s_textsize / (HASHFRACTION * sizeof(*froms));
    for ( fromindex = 0 ; fromindex < endfrom ; fromindex++ ) {
      if ( froms[fromindex] == 0 ) {
          continue;
      }
      frompc = s_lowpc + (fromindex * HASHFRACTION * sizeof(*froms));
      for (toindex=froms[fromindex]; toindex!=0;
toindex=tos[toindex].link) {
#         ifdef DEBUG
            fprintf( stderr ,
                  "[mcleanup] frompc 0x%x selfpc 0x%x count %d\n" ,
                  frompc , tos[toindex].selfpc , tos[toindex].count );
#         endif
          rawarc.raw_frompc = (unsigned long) frompc;
          rawarc.raw_selfpc = (unsigned long) tos[toindex].selfpc;
          rawarc.raw_count = tos[toindex].count;
          write( fd , &rawarc , sizeof rawarc );
      }
    }
    close( fd );
}


/* Solaris 2 libraries use _mcount.  */
asm(".globl _mcount; _mcount: jmp new_internal_mcount");
/* This is for compatibility with old versions of gcc which used
mcount.  */
asm(".globl mcount; mcount: jmp new_internal_mcount");

#ifdef USING_DYNINST
new_internal_mcount(unsigned my_pc, unsigned my_caller_pc, unsigned
max_addr, unsigned min_addr)
#else
new_internal_mcount(int para)
#endif
{
      register char                *selfpc;
      register unsigned short      *frompcindex;
      register struct tostruct     *top;
      register struct tostruct     *prevtop;
      register long                toindex;
```

```
	static char already_setup;

	/*
	 *	find the return address for mcount,
	 *	and the return address for mcount's caller.
	 */

#ifdef USING_DYNINST
	selfpc = (void *)my_pc;
	frompcindex = (unsigned short *)my_caller_pc;
#else
	selfpc = (void *) __builtin_return_address (0);
	frompcindex = (void *) __builtin_return_address (1);
#endif

	if(!already_setup) {
	  already_setup = 1;
	  new_monstartup((char *) min_addr, (char *) max_addr);

#ifdef USE_ONEXIT
	  on_exit(new_mcleanup, 0);
#else
	  atexit(new_mcleanup);
#endif
	}
	/*
	 *	check that we are profiling
	 *	and that we aren't recursively invoked.
	 */
	if (profiling) {
		goto out;
	}
	profiling++;
	/*
	 *	check that frompcindex is a reasonable pc value.
	 *	for example:	signal catchers get called from the
stack,
	 *			not from text space.  too bad.
	 */
	frompcindex = (unsigned short *)((long)frompcindex -
(long)s_lowpc);
	if ((unsigned long)frompcindex > s_textsize) {
	  goto done;
	}
	frompcindex =
	    &froms[((long)frompcindex) / (HASHFRACTION *
sizeof(*froms))];
	toindex = *frompcindex;
	if (toindex == 0) {
		/*
		 *	first time traversing this arc
		 */
		toindex = ++tos[0].link;
		if (toindex >= tolimit) {
			goto overflow;
		}
		*frompcindex = toindex;
```

```
		top = &tos[toindex];
		top->selfpc = selfpc;
		top->count = 1;
		top->link = 0;
		goto done;
	}
	top = &tos[toindex];
	if (top->selfpc == selfpc) {
		/*
		 *	arc at front of chain; usual case.
		 */
		top->count++;
		goto done;
	}
	/*
	 *	have to go looking down chain for it.
	 *	top points to what we are looking at,
	 *	prevtop points to previous top.
	 *	we know it is not at the head of the chain.
	 */
	for (; /* goto done */; ) {
		if (top->link == 0) {
			/*
			 *	top is end of the chain and none of the chain
			 *	had top->selfpc == selfpc.
			 *	so we allocate a new tostruct
			 *	and link it to the head of the chain.
			 */
			toindex = ++tos[0].link;
			if (toindex >= tolimit) {
				goto overflow;
			}
			top = &tos[toindex];
			top->selfpc = selfpc;
			top->count = 1;
			top->link = *frompcindex;
			*frompcindex = toindex;
			goto done;
		}
		/*
		 *	otherwise, check the next arc on the chain.
		 */
		prevtop = top;
		top = &tos[top->link];
		if (top->selfpc == selfpc) {
			/*
			 *	there it is.
			 *	increment its count
			 *	move it to the head of the chain.
			 */
			top->count++;
			toindex = prevtop->link;
			prevtop->link = top->link;
			top->link = *frompcindex;
			*frompcindex = toindex;
			goto done;
		}
```

```
      }
done:
      profiling--;
      /* and fall through */
out:
      return;           /* normal return restores saved registers */

overflow:
      profiling++; /* halt further profiling */
#   define  TOLIMIT     "mcount: tos overflow\n"
      write(2, TOLIMIT, sizeof(TOLIMIT));
      goto out;
}

/*
 * Control profiling
 *    profiling is what mcount checks to see if
 *    all the data structures are ready.
 */
new_moncontrol(mode)
    int mode;
{
    if (mode)
    {
      /* start */
      profil((unsigned short *)(sbuf + sizeof(struct phdr)),
           ssiz - sizeof(struct phdr),
           (int)s_lowpc, s_scale);

      profiling = 0;
    } else {
      /* stop */
      profil((unsigned short *)0, 0, 0, 0);
      profiling = 3;
    }
}
```

Dyninst mutator source code

```
#include <stdio.h>
#include<stdlib.h>
#include<sys/types.h>
#include<unistd.h>
#include <fcntl.h>
#include <assert.h>

#include "BPatch.h"
#include "BPatch_Vector.h"
#include "BPatch_thread.h"
#include "BPatch_snippet.h"

BPatch bpatch;

int main(int argc, char **argv)
{
```

```
BPatch_image *appImage;
BPatch_Vector < BPatch_point * >*points;
BPatch_Vector < BPatch_function * >*functions;

BPatch_Vector < BPatch_function * >snippetFunc;
BPatch_Vector < BPatch_snippet * >snippetArgs;

const char *arg[5];
char tmpstr[80];
int i, j;
bool isSharedLib;
FILE *fp;
char text1[80];
char text2[80];
char text3[80];
char command[80];
int  tmpint;
int range_max = -1;
int range_min = -1;

if(argc != 2)
{
  printf ("./dynMutator appPathName\n");
  exit (0);
}

for (i = 0; i < 5; i++)
  arg[i] = 0;

BPatch_thread *appThread = bpatch.createProcess(argv[1], arg, NULL);
appImage = appThread->getImage ();

if(!appThread->loadLibrary("./gmonSharedLib.so", false))
{
  printf ("load library failed!\n");
  exit (0);
}

// get the entry (which calls mcount) and exit handler pointers
BPatch_Vector<BPatch_function *>entry_handler_vec;
BPatch_Vector<BPatch_function *>exit_handler_vec;

appImage->findFunction ("update_pc_stack", entry_handler_vec);
assert(entry_handler_vec.size() == 1);

appImage->findFunction ("pop_pc_stack", exit_handler_vec);
assert(exit_handler_vec.size() == 1);

BPatch_Vector <BPatch_module *>*allmodules;
allmodules = appImage->getModules();

// find the min and max address of the text segment
for (j = 0; j < allmodules->size (); j++)
{
  tmpstr[0] = 0;
    if ((*allmodules)[j]->getName (tmpstr, 80))
    {
```

```
    functions = (*allmodules)[j]->getProcedures();
    for(i = 0; i < (*functions).size(); i++)
    {
      tmpstr[0] = 0;
      (*functions)[i]->getName (tmpstr, 80);
      BPatch_function *targ_func=(*functions)[i];

      // attempting to profile shared library module - break
      if((*functions)[i]->isSharedLib ())
      {
        isSharedLib = true;
        break;
      }

      void *base_addr=targ_func->getBaseAddr();
      assert(base_addr != NULL);

      if(range_max == -1)
      {
        range_max = (int) base_addr;
        range_min = (int) base_addr;
      }
      else
      {
        if((int) base_addr > range_max)
          range_max = (int) base_addr;

        if((int) base_addr < range_min)
          range_min = (int) base_addr;
        }
    }
    }
}

// instrument entry and exit points with stack code
for (j = 0; j < allmodules->size (); j++)
{
  tmpstr[0] = 0;
  if((*allmodules)[j]->getName(tmpstr, 80))
  {
    if (!strcmp (tmpstr, "DEFAULT_MODULE"))
    continue;

    functions = (*allmodules)[j]->getProcedures();

    for(i = 0; i < (*functions).size(); i++)
    {
    tmpstr[0] = 0;
    (*functions)[i]->getName (tmpstr, 80);

    if ((*functions)[i]->isSharedLib ())
      break;

    BPatch_function *targ_func=(*functions)[i];

    // 1. get base addr
    void *base_addr=targ_func->getBaseAddr();
```

```
      assert(base_addr != NULL);
      BPatch_constExpr func_addr_expr(base_addr);
      snippetArgs.push_back(&func_addr_expr);

      // 2. pass parameters min address and max address (found above)
      BPatch_constExpr max_addr_expr(range_max);
      BPatch_constExpr min_addr_expr(range_min);
      snippetArgs.push_back(&max_addr_expr);
      snippetArgs.push_back(&min_addr_expr);

      // 3. define expr for entry & exit
      BPatch_funcCallExpr entryCall(*entry_handler_vec[0],
snippetArgs);
      BPatch_funcCallExpr exitCall(*exit_handler_vec[0], snippetArgs);

      // 4. get entry points & insert
      BPatch_Vector < BPatch_point * >*entry_points = (*functions)[i]-
>findPoint (BPatch_entry);
      appThread->insertSnippet (entryCall,*entry_points);

      // 5. get exit points & insert
      BPatch_Vector < BPatch_point * >*exit_points = (*functions)[i]-
>findPoint (BPatch_exit);
      appThread->insertSnippet (exitCall,*exit_points);
      }
    }
  }

  appThread->continueExecution ();

  while (!appThread->isTerminated ())
    bpatch.waitForStatusChange ();
}
```

Fake stack source code

```
#include <cstdio>
#include <cstdlib>
#include <vector>

using namespace std;

vector<unsigned>pcstack;

extern "C" void new_internal_mcount(unsigned, unsigned, unsigned,
unsigned);

void update_pc_stack(unsigned pcval, unsigned max, unsigned min)
{
  pcstack.push_back(pcval);

  if(pcstack.size() == 1)
    new_internal_mcount(pcstack[0],0, max, min);
  else
    new_internal_mcount(pcstack[pcstack.size()-
1],pcstack[pcstack.size()-2], max, min);
```

```
}

void pop_pc_stack(unsigned pcval)
{
  // sanity check
  if(pcstack[pcstack.size()-1] != pcval)
  {
    fprintf(stderr,"\nERROR:pop_pc_stack sanity check failed. top of
stack =%x and argument is %x\n\n",pcstack[pcstack.size()-1],pcval);
    exit(-1);
  }

  pcstack.pop_back();
}
```

Wrapper source code

```
#include <stdio.h>
#include <dlfcn.h>
#include <stdlib.h>
#include <assert.h>

void *lib_handle;
void (*ptr_internal_mcount)(int)=NULL;


int load_play()
{
  char *error_msg;

  lib_handle = dlopen ("./gmonSharedLib.so", RTLD_NOW);
  if (!lib_handle)
    {
      printf ("Error during dlopen()\n");
      exit (1);
    }

  ptr_internal_mcount = dlsym (lib_handle, "new_internal_mcount");
  assert(ptr_internal_mcount != NULL);

  //dlclose (lib_handle);

  return 0;
}
```