Minimal Instrumentation for Software Feature Location*

Dennis Edwards, Norman Wilde, Sharon Simmons, Eric Golden

Department of Computer Science, University of West Florida

Executive Summary

Feature location is a commonly occurring problem in software reuse and maintenance. In understanding large software systems, Software Engineers need to locate the different software components that work together to provide a specific end user feature. Because of its importance, this problem has generated much interest among software maintenance researchers and a number of different methods of attack have been proposed, many of them involving tracing the execution of the software while the feature is running.

Over the last 15 years, our research group at the University of West Florida has worked with companies in the Software Engineering Research Center (SERC) on a series of studies to apply feature location techniques in practice. These case studies have generally been quite successful; a partnership of industry software engineers and university researchers has usually been able to get useful feature location results. However, recently a problem has appeared at two different SERC affiliates with large *time-sensitive* software systems.

In both cases it appeared that the instrumentation inserted to trace execution was making the system fail to initialize, or to "lock up" after startup. It would appear that less intrusive instrumentation methods are needed for feature location in such systems.

This report describes *mininst*, a "minimal" instrumentation approach, designed to meet the requirements of feature location but with as little impact as possible on system performance. Preliminary results from mininst are quite favorable with only a 10 - 20 % performance impact on several large artificial programs, and only a 1% impact when applied to the well-known Apache httpd web server.

Table of Contents

1 Introduction 2
2 Problems instrumenting time-sensitive software 2
3 Requirements for "minimal" instrumentation 4
4 The mininst runtime engine 4
5 Implementing mininst 8
6 Instrumentation experiments 8
6.1 Gnu compiler gcov options 8
6.2 Modified CodeTEST 9
6.3 Recon3 instrumentation toolset 9
7 Experimental results 9
8 Planned work 10
9 Conclusions 11
10 References 12
Appendix 1 - Generated test programs 14
Appendix 2 - Using mininst with CodeTest 15

* *This report may be cited as SERC-TR-295, Software Engineering Research Center, http://www.serc.net, January, 2009*

1 INTRODUCTION

Modern large software systems offer many different *features* to their end users. A word processor, for example, has user features such as text editing, font setting, document printing, and file saving in different formats.

The software engineers who maintain such software are thus often faced with the problem of feature location: finding where in the code a specific feature is implemented. Because of its importance, this problem has generated much interest among software maintenance researchers and a number of different methods of attack have been proposed.

Many of these methods primarily involve dynamic analysis. The software is traced during execution and traces obtained from tests that exercise the feature are compared to traces obtained from other tests. A number of different dynamic analysis methods have been proposed, e.g. [WILDE1992], [WONG1999] and [EISENBARTH2002].

Other proposals involve pure static analysis or, more recently, make use of a combination of dynamic and static techniques to provide better precision in locating the feature, e.g. [CHEN2000] [EISENBARTH2003] [ZHAO2006] [ROBI2007] [POSH2007] [EADDY2008].

Over the last 15 years, our research group at the University of West Florida has worked with companies in the Software Engineering Research Center (SERC) on a series of studies to apply feature location techniques in practice. We have largely concentrated on variants of the software reconnaissance method [WILDE1995], one of the simpler dynamic analysis approaches. Most of the studies have involved adapting existing tools, from either university or commercial sources, to meet feature location needs in specific industrial environments [WILDE1996] [WILDE1998] [SIMMONS2006] [JIANG2006] [JIANG2007].

These case studies have generally been quite successful; a partnership of industry software engineers and university researchers has usually been able to get useful feature location results. However, recently a problem has appeared at two different SERC affiliates with systems we characterize as *large time-sensitive* software.

Dynamic analysis involves executing the target software system while tracing its execution. Unfortunately, tracing an execution typically requires instrumenting the target software to produce a record of the software components executed. Early experiments indicate that instrumentation at the level of decisions, branches and basic blocks provides the precision necessary for feature location. Instrumentation obviously has a performance impact on the execution of the target. In these SERC time-sensitive systems, the instrumented code was failing to initialize and so traces could not be obtained.

We believe that all feature location methods, except those that are purely static in nature, will encounter instrumentation problems in dealing with this class of software. For this reason we have been seeking a minimally intrusive form of instrumentation that will meet the needs of feature location with as little impact as possible on the execution of the target system. This paper describes our results so far.

2 PROBLEMS INSTRUMENTING TIME-SENSITIVE SOFTWARE

The SERC systems that we are characterizing as *time-sensitive* software had the following characteristics:

- They were large, typically one million lines of code and up, with large development teams.
- They were longstanding fielded products, initially developed some years earlier.
- Developer knowledge was somewhat dispersed, so that it was not easy to answer all questions about their exact workings.
- They were multi-process / multi-thread systems, with a considerable degree of concurrency.
- They were computation intensive, and thus to some extent were pushing the envelope of what their hardware environment could tolerate
- They were long-running event-driven systems so that testing typically involves initializing the system, then running a series of tests one after the other, and then finally shutting down. A test case is thus an interval of time, rather than a complete end-to-end execution (Figure 1).

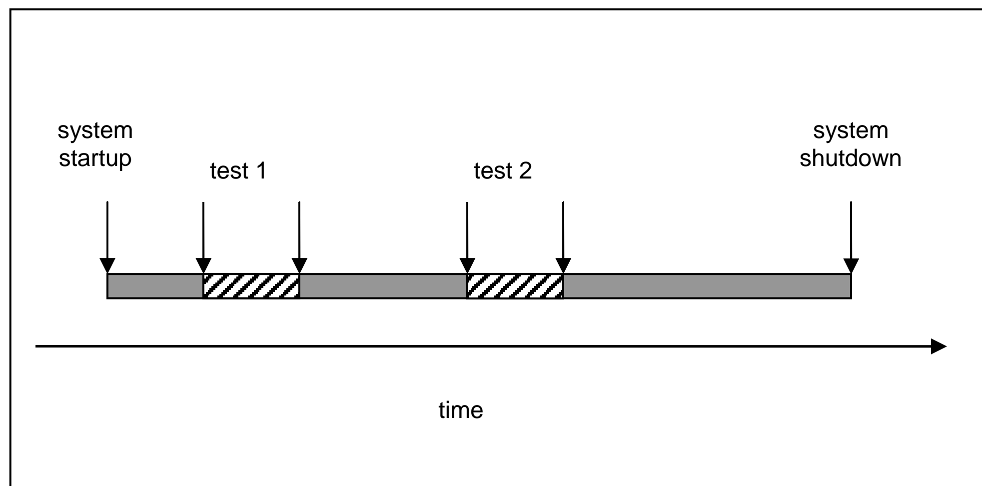

Figure 1 - Testing a Long-Running System

In earlier work, we had generally found that we could get acceptable traces using either commercially available test coverage tools [SIMMONS2006] or our own student-written research tools [RECON3]. However, these tools failed when applied to these time-sensitive systems. Also, ATOM [SRIVASTAVA1994] and Pin [LUK2005] , two industrial object-code instrumentation tools that are designed for efficiency, failed when applied to these systems. In each case the symptoms of the problem were that, while the uninstrumented code would load and initialize within a period of a few minutes, the instrumented code would either fail to initialize or, once started, would be unresponsive to user input.

Normally some degradation in performance is not a severe hindrance in feature location, since the traces are collected in a laboratory test environment and not on a fielded system. In these recent examples it was not clear exactly what was happening (recall that developer knowledge was dispersed) but the hypothesis of some developers was that the inter-process communications were failing due to time-outs, especially during system initialization. Thus the instrumented code would lock up and could not be executed.

3 REQUIREMENTS FOR "MINIMAL" INSTRUMENTATION

This industrial experience illuminates the need for a better way of instrumenting time-sensitive software for feature location. The requirements would include the following:

1. Instrumentation should be fairly low level, so that we can trace the execution of individual branches, decisions or basic blocks in the target software.
2. Instrumentation should provide *time interval* coverage [EDWARDS2004]. For feature location we need to know what components were executed during the time interval in which the feature test was active, and compare that to what was executed in other time intervals (Figure 1). The time interval needs to be adjustable depending on the specific system and test cases.
3. The instrumentation must work in multi-process and multi-thread software, which means that whatever method is used to collect the trace events, it must be designed for thread safety.
4. Instrumentation should have as little impact on performance as possible, especially while the target software is initializing and communication is being set up among the different threads and processes.

In general, instrumentation toolsets consist conceptually of four different parts (Figure 2):

1. An <u>insertion engine</u> that scans the source code (or sometimes the object code) of the target program and inserts additional code at selected points to produce runtime traces.
2. A <u>runtime engine</u> that consists of the inserted trace code plus any support code needed as the target program runs. The runtime engine writes the trace to an output device such as a file, a pipe, shared memory or a socket.
3. Possibly a <u>collection engine</u> that collects the trace from the output device.
4. Possibly an <u>analysis engine</u> that reformats the trace data and produces trace output more suitable for human use or for input to other tools.

For time-sensitive software it is obviously the performance of the runtime components that is most important.

4 THE MININST RUNTIME ENGINE

The requirements given in the previous section, while stringent in some respects, provide an opportunity as compared to instrumentation for normal debugging. For feature location we do not need the actual *sequence* of trace events. The time interval coverage requirement only states that we need to know if a specific code component was executed in a particular time interval; we do not need to know the *ordering* of events within that interval.

This opportunity provides the motivation for our *mininst* design for the runtime instrumentation engine. To keep performance overhead as low as possible we store coverage information for each time interval in a bit string.

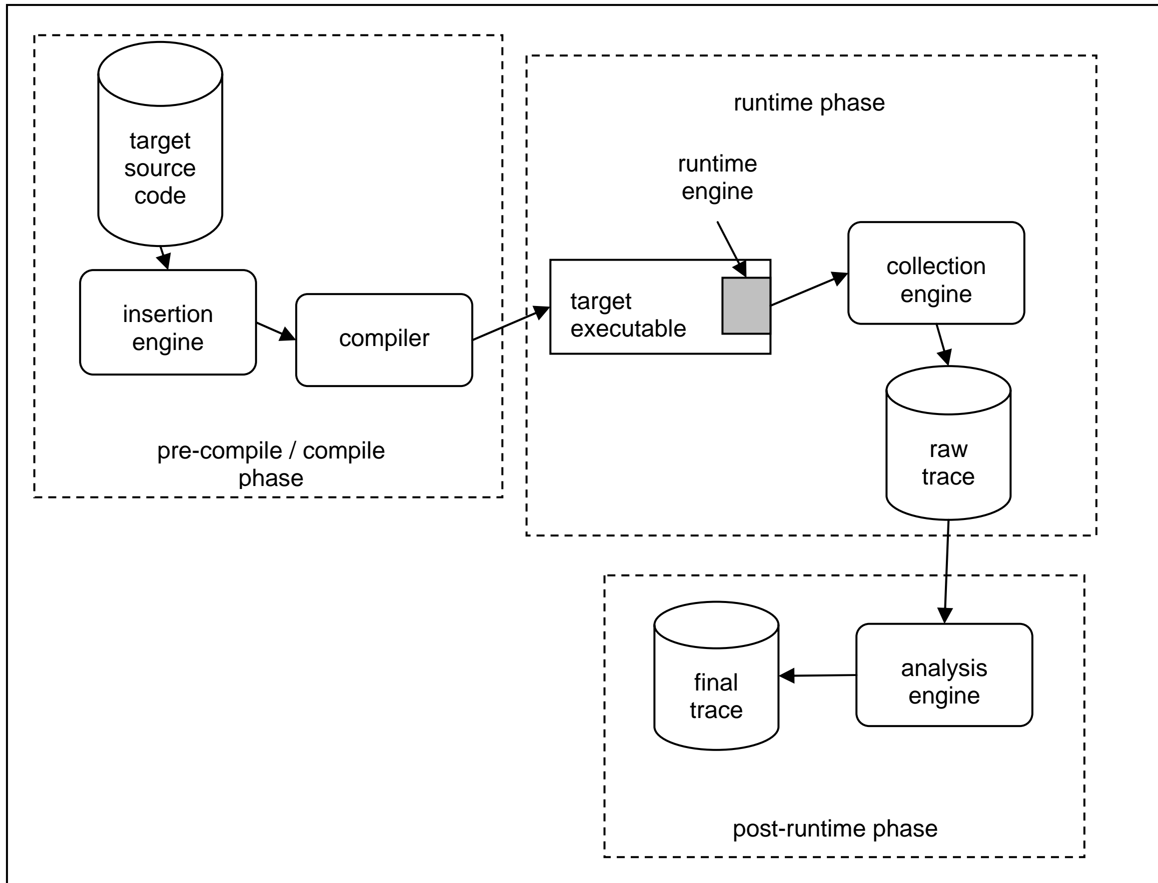

Figure 2 - Tracing Software Execution

During the compilation process, instrumentation points (usually basic blocks) are assigned a sequential index as they are located. The index, ι, is used as an offset into a bit string, stored at memory location α, where each bit represents the execution of the code at the instrumentation point. From ι, the byte number, β, and bit offset inside the byte, δ, are computed as follows:

$$\beta = \iota / 8$$

$$\delta = \iota \text{ MOD } 8$$

From these two numbers, instrumentation code is produced that is inserted at the appropriate instrumentation point. To set bit number δ in byte number β, code similar to the following is inserted. Note that the values of β and δ are known at compile time and therefore, are inserted as constant values.

```
*(α + β) |= (1 << δ)
```

As an example, consider the simple program in Table 1. A function is used to implement the multiplication of two small positive integers by iterative adding.

Instrumentation points are identified as the beginning and ending of basic blocks. Table 2 shows the resulting instrumentation code that is sent to the compiler. Had another instrumentation point been found, it would receive the code

```
*(α + 1) |= 1;
```

since the value of ι would indicate the first bit in the next byte.

Table 1 - Simple multiplication program

```
int multiply(int a, int b)
{
  int product;
  product = a;
  while( --b > 0 )
  {
    product += a;
  }
  return product;
}

int main(int argc, char ** argv)
{
  int answer, i;
  for ( i = 1; i < 10; i++ )
  {
    answer = multiply( 10, i );
    printf("%d x %d = %d\n",
        10, i, answer);
  }
  return 0;
}
```

It is important to preserve *locality* in the code so that it will execute efficiently on a modern cached processor [BRYANT2003]. To check locality, we compiled the instrumented and uninstrumented programs using the gcc compiler on Linux and compared the resulting executable code. Seven machine instructions were inserted for each instrumentation point. Assuming that the bit string located at memory address 0x80496c0, that is, α = 0x80496c0, then the inserted instructions are as shown in Table 3.

Table 2 - Simple instrumented program

```
int multiply(int a, int b)
{*(α + 0) |= 1;{
  int product;
  product = a;
  while( --b > 0 )
  {*(α + 0) |= 2;{
    product += a;
  }}*(α + 0) |= 4;
  {*(α + 0) |= 8;return product;}
}}

int main(int argc, char ** argv)
{*(α + 0) |= 16;{
  int answer, i;
  for ( i = 1; i < 10; i++ )
  {*(α + 0) |= 32;{
    answer = multiply( 10, i );
    printf("%d x %d = %d\n",
        10, i, answer);
  }}*(α + 0) |= 64;
  {*(α + 0) |= 128;return 0;}
}}
```

If we assume the common L1-D cache line size of 32 bytes, we see that 256 bits are stored in a single cache line. Since one bit represents an instrumentation point, a cache line can represent 256 sequentially assigned instrumentation points. Following the principles of space- and time-locality of execution, we can expect to find the majority of memory accesses needed to set bits in the bit string to be completely handled through L1-D cache. Therefore, few CPU cycles are needed for each instrumentation point.

Table 3 - mininst instrumentation instructions

```
;pointer into reg a
mov  0x80496c0,%eax
;add offset to byte (7) into reg d
lea  0x7(%eax),%edx
;pointer into reg a
mov  0x80496c0,%eax
;add offset to byte (7) into reg a
add  $0x7,%eax
;move zero fill value into reg a
movzbl (%eax),%eax
;set bit in byte
or  $0x1,%eax
;move result to address in reg d
mov  %al,(%edx)
```

It is also necessary to collect the bit string at the end of each time interval. In the mininst design, a new *collector thread* is created that sleeps until the end of an interval. At that point it is awakened to

1. store the value of the bit string to secondary storage,
2. clear the bit string for use in the next time interval, and
3. sleep until the next time interval end.

To avoid a race condition between the storage of the bit string and setting of bits by the target program, several additional steps were needed.

Blocking the target program while the above actions take place introduced an unacceptable overhead into the target's execution, so a second bit string was allocated. A pointer, α, refers to the bit string currently in use by the target program to record the execution of instrumentation points. Another pointer, η, refers to the bit string not currently in use. When an interval ends, the collector thread performs the following steps.

1. Exchange α and η
2. Save the values stored in the η-referenced bit string
3. Clear the bits of the η-referenced bit string for the next interval

A potential race condition still exists during the execution of step 1 by the collector thread. It is possible that a context switch could occur during the six instructions needed to exchange the α and η pointers. However, the only effect would be to include a few extra instrumentation points in the interval, and this imprecision was deemed acceptable when contrasted with the overhead needed to completely eliminate the possibility.

5 IMPLEMENTING MININST

The mininst runtime engine can be used in combination with any source code instrumentor that inserts probes at a sufficiently low level. For our trials, we used the insertion engine from the commercial CodeTEST[1] instrumentation suite to put a probe on each basic block.

The CodeTEST insertion engine runs during compile time as a pre-processor to the compiler (see Appendix 2). At each instrumentation point it inserts a call to a function called `ctTag()` which is part of CodeTEST's runtime engine. This function writes an integer tag number identifying the instrumentation point through a pipe to CodeTEST's collection engine which runs as a separate process (see Figure 2). Simultaneously the insertion engine writes the tag numbers along with their source code locations to an instrumentation data base (IDB) so that the actual path through the code can be reconstructed later from the tag numbers.

To adapt CodeTEST's insertion engine to mininst, we added an extra pre-processing step, which scans for the calls to `ctTag()` and replaces them with the bit string code described earlier. Our pre-processor also identifies the `main()` function and inserts code there to start mininst's collector thread. Finally, it also generates a map between the mininst index values and CodeTEST's tag numbers so that we can, after runtime, use the IDB to build a formatted trace.

During the runtime phase (Figure 2), the mininst collector thread simply sleeps until it is awakened by an operating system signal `SIGUSR1` sent from a test driver to start tracing at the beginning of a test. Up to that point, the only overhead has been the execution of the seven machine instructions of Table 3 on each basic block, so the performance impact during startup is low. From then on, bit setting continues as described in the previous section until, at the end of the test, the test driver sends the `SIGUSR2` signal. The collector thread responds by exchanging the two bit string pointers and writing the full one (η) to the raw trace file.

6 INSTRUMENTATION EXPERIMENTS

For an initial evaluation of the mininst design, we conducted a series of experiments to evaluate its performance using several different target software systems and to compare it to the performance of other instrumentors that we had readily available.

First, we used artificial programs of different sizes generated by a program we wrote (see Appendix 1). The generated programs, ranging from 150,000 to 600,000 lines of code, have a balanced distribution of functions, loops, variables and other control structures. These tests were intended to show how the different instrumentation tools scale.

Second we used the well know Apache `httpd` server, version 2.2.9. This is a large (300 KLOC), long-running, event-driven system and thus bears some resemblance to the SERC affiliates' industrial high-performance systems. Its code structure will be more typical of industrial software than the generated programs.

As well as measuring the performance impact of mininst on different software systems, we also compared it to the impact of other instrumentation tools: gcov, a version of CodeTEST, and Recon3.

6.1 Gnu compiler gcov options

The Gnu gcc compiler provides options for collecting source line coverage data [FREESOFT]. The software engineer simply sets the `-fprofile-arcs` and `-ftest-coverage` flags on

---

[1] CodeTEST is a trademark of Freescale Semiconductor [FREESCALE].

the compile command line; the insertion engine is built into the compiler (Figure 2). When these flags are used, the gcc runtime engine counts the number of times each line is executed. As the program shuts down, these counts are used to update coverage files, one for each instrumented source file. Post runtime, the software engineer can use the gcov tool to get a formatted report showing how often each line was executed.

This facility is poorly adapted to feature location in long-running programs since it is not possible to get coverage for each test when a run contains multiple tests (Figure 1). Gnu's coverage data is only written as the target program shuts down. However, since this is a well known and accepted tool, we measured the gcc runtime performance to provide a reference point for comparison to other tools.

6.2 Modified CodeTEST

As described elsewhere [SIMMONS2005], the standard version of CodeTEST cannot be used for interval traces because the normal collection engine is difficult to control manually to delimit the start and end of the test intervals. As part of an earlier study, we developed a modified collection engine which allowed more processing to be moved to the post-runtime phase. This modified CodeTEST was used successfully at Motorola for a demonstration case study of feature location [JIANG2007].

As previously described, the CodeTEST instrumentor runs during compile time and inserts calls to `ctTag()` at each instrumentation point. As for the normal CodeTEST, this `ctTag()` writes the tag numbers through a pipe to our collection engine. In these experiments, the test driver programs send signals to the collection engine through a socket to mark the start and stop of each test. The collection engine writes these marks into the stream of tag numbers which it is storing as the raw trace. All further processing to prepare a formatted trace is performed in the post-runtime phase and so has no performance impact on the target program.

6.3 Recon3 instrumentation toolset

Recon3 is a student-written toolset designed for flexible tracing and feature location [RECON3]. However it was not designed with runtime efficiency as a priority so there is much overhead at each instrumentation point. Recon3 includes a C/C++ insertion engine, a runtime engine called the *trace manager interface* and a collection engine called the *trace manager client* which runs as a separate process communicating with the runtime engine via shared memory. Several different modes of tracing are provided; in these experiments we used *count mode*, meaning that a sequential trace was not written to the trace file. Instead only a final count was written indicating the number of times each instrumentation point was executed during each time interval. (See section 2.2.3 of [WILDE2000]) However using count mode only improves the performance of the collection engine, not of the trace code inserted at each instrumentation point.

7 EXPERIMENTAL RESULTS

The performance evaluation experiments were run on a computer with an Intel 780 chip, running at 2.26GHz, under Linux version 2.6.25. The operating system was running normally and the computer was connected to the network, but it had no other user applications executing while the tests were conducted.

We compared the time required for execution of each uninstrumented program with results when using mininst and the other instrumentation tools. The main performance measure was the "wall clock" execution time in nanoseconds as recorded by the `clock_gettime()` function.

Timing data was collected by driver code to measure just the impact of the instrumentation as the target program runs. Each driver would

1. initialize the target program
2. do one or more cycles of:
   a. turn on tracing
   b. call `clock_gettime()` for a start time
   c. run a test of the target program
   d. call `clock_gettime()` for a stop time
   e. turn off tracing and write out the trace
   f. log the resulting times
3. terminate the target program

The recorded time intervals (between steps (b) and (d)) thus do not include input/output during program start up or shut down, nor do they include the time required to record the trace to the file system.

Table 4 - Test results with generated programs

Tool	Execution Time in ms % increase			
	150 Kloc	**300 Kloc**	**450 Kloc**	**600 Kloc**
Unmodified	106	209	320	427
mininst	126 19%	245 17%	351 10%	476 11%
GNU gcov	129 22%	277 32%	399 25%	564 32%
Modified CodeTEST	350 230%	670 221%	1025 220%	1364 219%
Recon3	156900 147919%	290618 138952%	482341 150632%	604167 141391%

Table 4 summarizes the timing results for four generated programs. The first row is the timing without any instrumentation or tracing. Each successive row corresponds to the indicated instrumentation tool. Each entry has a percentage time increase from the original, unmodified execution. Mininst is consistently lower than any other tool with gcov being the only tool that is close to mininst's performance. The timing of mininst and gcov is close for the smallest program, but when the program size increases, the overhead of mininst is amortized over the entire execution resulting in better performance.

Table 5 summarizes the execution timing of the Apache web server, `httpd`. With multiple threads, processes and more complex code, the timing results clearly demonstrate that mininst and gcov have the least impact on runtime performance. The mininst introduced overhead exemplifies the expected performance for large systems.

8 PLANNED WORK

We are currently working with SERC affiliate Northrop Grumman to evaluate mininst on a large, time-sensitive software system. A port of mininst will be created and evaluated in their operating environment. Experiments will be performed to provide additional data for the evaluation of the system's overhead and usability.

Table 5 - Test results with Apache web server

Tool	Execution Time in ms % increase Apache
Unmodified	924
mininst	929 1%
GNU gcov	987 7%
Modified CodeTEST	7469 709%
Recon3	21515 2228%

We would also like to further evaluate the possibility of using object-code instrumentation for feature location. Object-code instrumentation, as performed by tools such as Pin [PIN], analyses and instruments executable code as it is loaded and therefore does not require the creation of special builds for instrumentation. Given the time pressures that industrial Software Engineers face, such a simplification of the feature location process would be most desirable.

Another simplifying alternative would be to simply leave the instrumentation permanently in the code. As described in section 5, mininst trace collection need not start until a test driver sends a signal marking the beginning of a test's execution. Up to that point the only overhead is the additional seven machine instructions per basic block.

Since it may be effective to leave the instrumentation dormant in the shipped code, we plan to extend the current study to measure the performance impact under such conditions.

9 CONCLUSIONS

To handle software that grows ever larger and more complex, dynamic analysis tools need to address difficult issues of scale and performance. Much current software is multithread and/or multiprocess as well as being time sensitive. While increased overhead at compile-time is acceptable, runtime costs must be minimized to avoid the adverse effects of system probing.

For feature location, many of the published analysis methods do not require coverage information that is temporally ordered. This provides an opportunity to reduce considerably the runtime performance impact of tracing.

This paper has described preliminary results from a method of instrumenting source code that reduces runtime overhead. This approach, mininst, uses single bits to identify the execution of each instrumentation point. At each one, a single source code statement is inserted which is compiled into seven fast machine language instructions. Collection relies on a separate thread that stores the bit set to secondary storage.

Preliminary experimental results were presented that showed that the run-time impact of mininst was less than that of three other available instrumentation systems. Mininst's performance penalty was 10 - 20% for several large artificially generated programs, and only 1% for the well-known Apache `httpd` web server. Further trials on time-sensitive software are planned.

We hope that tool designs based on mininst can enable the use of dynamic feature location techniques on the growing class of large, time-sensitive software systems.

Acknowledgement

This research was partially supported by Northrop Grumman Corporation through the Software Engineering Research Center (SERC), an NSF-funded Industry University Cooperative Research Center which brings together several universities with companies in software intensive industries. http://www.serc.net.

10 REFERENCES

[BRYANT2003] Randal E. Bryant and David R. O'Hallaron, Computer Systems: A Programmer's Perspective. Prentice Hall, 2003, ISBN 0-13-034074-X, Section 6.2.

[CHEN2000] Chen, K. and Rajlich, V. 2000. Case Study of Feature Location Using Dependence Graph. In *Proceedings of the 8th international Workshop on Program Comprehension* (June 10 - 11, 2000). IWPC. IEEE Computer Society, Washington, DC, 241.

[EADDY2008] Eaddy, M., Aho, A. V., Antoniol, G., and Guéhéneuc, Y. 2008. CERBERUS: Tracing Requirements to Source Code Using Information Retrieval, Dynamic Analysis, and Program Analysis. In Proceedings of the 2008 the 16th IEEE international Conference on Program Comprehension, (June 10 - 13, 2008). ICPC. IEEE Computer Society, Washington, DC, 53-62. DOI= http://dx.doi.org/10.1109/ICPC.2008.39

[EDWARDS2004] Dennis Edwards, Sharon Simmons, Norman Wilde, "An Approach to Feature Location in Distributed Systems", Journal of Systems and Software, Vol. 79 (2006), No. 1 pp 57-68. DOI: 10.1016/j.jss.2004.12.018.

[EISENBARTH2002] Eisenbarth, T. Koschke, R. Simon, D. Incremental location of combined features for large-scale programs. Proc. International Conference on Software Maintenance, 2002, 3-6 Oct. 2002, pp. 273 - 282

[EISENBARTH2003] Eisenbarth, T., Koschke, R., and Simon, D. 2003. Locating Features in Source Code. IEEE Trans. Softw. Eng. 29, 3 (Mar. 2003), 210-224. DOI= http://dx.doi.org/10.1109/TSE.2003.1183929

[FREESOFT] Free Software Foundation, Using the GNU Compiler Collection (GCC) - Section 9 - gcov—a Test Coverage Program, http://gcc.gnu.org/onlinedocs/gcc-4.3.2/gcc/, URL current November, 2008.

[FREESCALE] Freescale Semiconductor, *CodeTEST Software Analysis Tools*, http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=CODETEST. URL current October, 2008

[JIANG2006] Jiang, Michael, Groble, Michael, Simmons, Sharon, Edwards, Dennis and Wilde, Norman, Software Feature Understanding in an Industrial Setting, in: ICSM '06: Proceedings of the 22nd IEEE International Conference on Software Maintenance, pages 66-67, IEEE Computer Society, 2006. DOI= http://dx.doi.org/10.1109/icsm.2006.59

[JIANG2007] Michael Jiang, Jing Zhang, Sharon Simmons, Dennis Edwards, Norman Wilde, *TraceGraph 4: A Demonstration Case Study*, SERC-TR-290, Software Engineering Research Center, http://www.serc.net/, July, 2007.

[LUK2005] Luk, Chi-Keung, Cohn, Robert, Muth, Robert, Patil, Harish, Klauser, Artur, Lowney, Geoff, Wallace, Steven, Reddi, Vijay Janapa and Hazelwood, Kim, Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation, in: PLDI'05, pages 190-200, ACM, Chicago, 2005. DOI=http://doi.acm.org/10.1145/178243.178260.

[PIN] Pin, http://rogue.colorado.edu/Wikipin/index.php/Main_Page, URL current December, 2008.

[POSH2007] Poshyvanyk, Denys, Guéhéneuc, Yann-Gaël, Marcus, Andrian, Antoniol, Giuliano and Rajlich, Vaclav. Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval. IEEE Trans. Softw. Eng. 33, 6 (Jun. 2007), 420-432. DOI= http://dx.doi.org/10.1109/TSE.2007.1016

[RECON3] Recon3, University of West Florida, http://www.cs.uwf.edu/~recon/recon3/. Url current October, 2008.

[ROBI2007] Robillard, Martin P. and Murphy, Gail C., Representing concerns in source code, ACM Trans. Softw. Eng. Methodol., volume 16, number 1, ISSN 1049-331X, 2007 DOI= http://doi.acm.org/10.1145/1189748.1189751.

[SIMMONS2005] Simmons, S., Edwards, D., Wilde, N., Homan, J., Groble, M., Using Industrial Tools for Software Feature Location and Understanding, SERC-TR-275, August 1, 2005, http://www.serc.net/report/tr275.pdf.

[SIMMONS2006] Sharon Simmons, Dennis Edwards, Norman Wilde, Josh Homan, Michael Groble, "Industrial tools for the feature location problem: an exploratory study", Journal of Software Maintenance and Evolution: Research and Practice, Volume 18, No. 6, pp. 457-474, 2006, DOI: 10.1002/smr.338.

[SRIVASTAVA1994] Srivastava, A. and Eustace, A. 1994. ATOM: a system for building customized program analysis tools. In Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (Orlando, Florida, United States, June 20 - 24, 1994). PLDI '94. ACM, New York, NY, 196-205. DOI= http://doi.acm.org/10.1145/178243.178260.

[WILDE1992] Wilde, N., Gomez, J., Gust, T., Strasburg, D., "Locating User Functionality in Old Code" Proc. IEEE Conf. on Software Maintenance - 1992, Orlando, Florida, November 1992, pp. 200 - 205.

[WILDE1995] Wilde, N. and Scully, M. C. 1995. Software reconnaissance: mapping program features to code. Journal of Software Maintenance 7, 1 (Jan. 1995), 49-62. DOI= http://dx.doi.org/10.1002/smr.4360070105

[WILDE1996] Wilde, Norman and Casey, Christopher, Early field experience with the Software Reconnaissance technique for program comprehension, in: ICSM '96: Proceedings of the 1996 International Conference on Software Maintenance, pages 312-318, IEEE Computer Society, 1996.

[WILDE1998] Wilde N., Casey C., Vandeville J., Trio G., Hotz D. Reverse engineering of software threads: A design recovery technique for large multi-process systems (1998) Journal of Systems and Software, 43 (1), pp. 11-17.

[WILDE2000] Norman Wilde , Laura White , Aaron Tarnosky, The Architecture of Recon3, http://www.cs.uwf.edu/~recon/recon3/r3uarch.htm. Url current October, 2008.

[WONG1999] Wong WE, Gokhale SS, Horgan JR. Metrics for quantifying the disparity, concentration, and dedication between program components and features. Sixth IEEE International Software Metrics Symposium, IEEE Computer Society: Los Alamitos CA, 1999; 189–198.

[ZHAO2006] Zhao, W., Zhang, L., Liu, Y., Sun, J., and Yang, F. 2006. SNIAFL: Towards a static noninteractive approach to feature location. ACM Trans. Softw. Eng. Methodol. 15, 2 (Apr. 2006), 195-226. DOI= http://doi.acm.org.ezproxy.lib.uwf.edu/10.1145/1131421.1131424

APPENDIX 1 - GENERATED TEST PROGRAMS

We generated a series of large C++ test programs, each consisting of a single large source file. The test programs were structured to have many basic blocks so that we could measure how well the instrumentation tools handled programs of different scale. Most of the code is straight line, but there is looping to provide some of the locality that is typical of most real-world software. Below is an example of one of the generated test programs showing the basic program structure. Each test program consisted of:

- N global integer variables.
- 9/10 N simple functions. Each of these consisted of an if statement, thus giving rise to at least two basic blocks, one of which contained some simple integer and floating point computations as well as conversions between the two data types. Each function manipulated two of the global variables.
- 1/10 N loop functions. Each of these loops through a large array of doubles, setting element values and accumulating a total.
- A main function that called each of the N functions in sequence, passing a data value from one call to the next, thus preventing elimination of code by compiler optimization. The values passed ensured that only the basic block containing the computations would actually be executed in each function.

We generated programs using N values of 2, 15000, 30000, 45000 and 60000. Since each value of N gave 10 lines of code, this gave us programs of approximately 60 LOC, 150 KLOC, 300 KLOC, 450 KLOC and 600 KLOC.

```
------------------------------------
/* EXAMPLE OF A GENERATED PROGRAM
 * This would be called selfTimer_000002.c since it contains two
 * functions. The program makes ten passes through its functions,
 * timing each one. A comma separated values (CSV)
 * record is appended to file statistics.txt for each timed pass.
 * See the Makefile for how to compile. See comments in file
 * Common/programTimer.c for the format of statistics.txt.
 */

/* Program with 2 functions */
/* Uncomment lines marked RECON3 if using recon3 tracing*/
```

```
#include <stdio.h>
#include <stdlib.h>
#include "programTimer.h"
#define ARR_SIZE (10000)
double gArray[ARR_SIZE];

int g_000000 = 1;
int g_000001 = 1;
int g_000002 = 1;

int f_000001 (int a) {
  int x; double tmp = 0;
  for (x = 0; x < ARR_SIZE; x++) {
    gArray[x] = (double)x; tmp += gArray[x]; }
  g_000001 = (int)((2.0001 * tmp) /(ARR_SIZE * (ARR_SIZE + 1.0) ) );
  return a + g_000001 ;
}

int f_000002 (int a) {
  if( (a+1) > 0 )
    g_000002 = ((int)((1.011*(float)g_000002)))*((int)((1.02*g_000001)/(1.01*g_000001)));
  else
    g_000002 -- ;
  return a + g_000002 ;
}

int main(int argc, char * argv[] ) {
  int iteration;
  /* RECON3 system("r3ctmc start &");
  char newtraceCommand[128]; */
  printf("Functions: 2\n"); /* debug */
  for(iteration = 1; iteration <= 10; iteration++) {
    printf(" Starting iteration: %d\n", iteration);
    /* RECON3 sprintf(newtraceCommand, "r3ctmc newtrace trace%d.r3t", iteration);
    system(newtraceCommand); */

    startTimers();

    int x = 1;

    x = f_000001(x);
    x = f_000002(x);

    stopTimers();
    writeTimes(argv[0], iteration);
    /* RECON3 system("r3ctmc newframe");*/

  }//iteration loop
  /* RECON3 system("r3ctmc stop");*/
  return 0;
}//main
```

APPENDIX 2 - USING MININST WITH CODETEST

The current version of the mininst instrumentor is designed to be used in conjunction with the CodeTEST instrumentation engine from Freescale Semiconductor, Inc. The most recent version of the CodeTEST C Instrumentor, 2.8.01, has the ability to call external programs either before or after the instrumentation process. This facility allowed a proof-of-concept prototype of mininst to be written without developing a complete C parser.

From the user's point of view, creating an executable that is instrumented with mininst requires two steps.

1. Create a configuration file identifying the instrumentation parameters to use
2. Call `make` with arguments to use CodeTEST

```
make -CC=ctcc
```

Figure A-1 shows the processing that occurs in instrumenting, compiling and executing the target program, followed by post-runtime processing to produce the trace. The steps involved are as follows:

Pre-compile and compile phases:

1) Source file source_1.c goes through the c preprocessor to create source_1.i
2) Source_1.i goes through CodeTEST's instrumentor, ctci, to create source_1._i containing calls to ctTag(), each identified by a distinct integer tag number. The codetest.idb "instrumentation data base" maps these tag numbers to line numbers in the source file.
3) Source_1._i goes through the mininst program which, as described in Section 5, replaces the calls to ctTag() with mininst statements to set bits in the bit string. At the same time it creates file mininst.map which maps the mininist indices to the CodeTEST tag numbers.
4) The resulting mininst instrumented file goes through the compiler to create an object file source_1.o.
5) The linker combines that object file with other similarly instrumented object files and with the libmininst.a library archive containing mininst_init.c, which is the code to allocate shared memory and start tracing.
6) The result is the instrumented target executable, denoted as "target".

Runtime phase:

1) When the target program starts, it forks the mininst_tracer collector process.
2) As the target program executes it is traced as described in Section 5; the instrumentation in the target program sets bits in the bit strings, which are collected and written at the end of each interval by the mininst_tracer process. The result is the binary trace file trace.bin.

Post-runtime phase:

1) A joiner program reads trace.bin (the binary trace) and uses CodeTEST's instrumentation data base (codetest.idb) and the mininst.map file to create formatted trace records in trace.r3t.

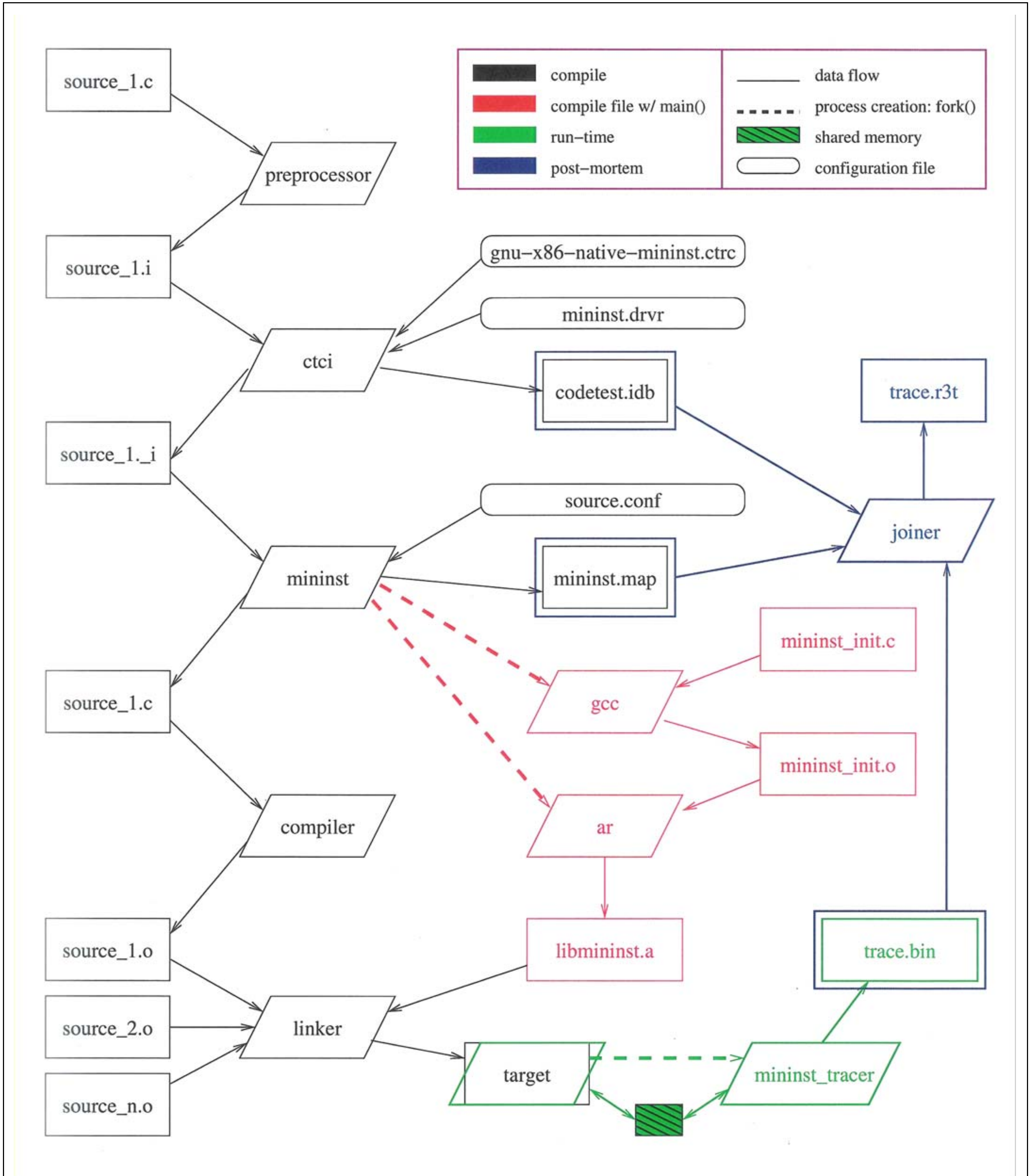

Figure A-1 - Overview of processing involved in using mininst with CodeTEST