3S: Program Instrumentation and Characterisation Framework

Simon A. Spacey

ABSTRACT

3S is an efficient program instrumentation and profiling framework. 3S is only 288 lines of framework code, yet it can produce the same reports as Valgrind [1] and is up to an order of magnitude faster.

I. INTRODUCTION

3S stands for "Spacey Stream Splitter". 3S is a framework used to instrument an x86 program. You use the framework together with 3S analysis tools to analyse a program's control flow. The 3S framework provides the 3S tools with a stream of control and data flow information as the instrumented target program runs. The 3S tools split the control and data flow information stream to create their reports.

This document introduces the 3S framework and some of the 3S tools. I provide a summary of the 3S framework methodology in section 2, examine two of the 3S tools in section 3, evaluate the performance of 3S in section 4 and consider some possible enhancements in section 5.

This document is not a survey of different instrumentation frameworks and it is not a proposal for future research. If you are interested in information on any of those topics you should start with a review of the references at the end of this document.

II. 3S FRAMEWORK METHODOLOGY

The 3S framework is only 288 lines of code, yet it can produce the same reports as Valgrind [1] and is up to an order of magnitude faster. The 3S framework works by inserting instrumentation stubs in to the assembly of a program and then linking the modified assembly with a 3S analysis tool specified at instrumentation time. The instrumentation stubs call the 3S analysis tool with a stream of program control and data flow information as the program runs. The 3S tool analyses the stream and creates reports from it.

By working at the assembly level, 3S does not have to be concerned with its stubs over-writing program instructions or jump targets [2] and also benefits from the basic block identification algorithms already implemented in the source code to assembler stage of the compiler. These simplifications make instrumenting at the assembly level a straightforward process of text file parsing. 3S simply takes a compiled program's assembly and adds 3S stubs after each compiler generated function or jump target label. The one complication is call instructions which are easy to spot in the assembly using regular expressions.

In stark contrast to object code instrumentation frameworks like Valgrind, the 3S instrumented output is fully readable assembly text. This drastically reduced the development and debugging time of the 3S framework itself and also benefits users who can see exactly what 3S does to their programs to assure themselves that the 3S instrumentation can not interfere with their program's validity.

A. Instruction and Block Level Instrumentation

The first release of 3S was designed to instrument a program at the block level only. By version 2, instruction level instrumentation features had also been added. Working at the block level, 3S can create control flow and hotspot reports. At the instruction level, it is possible to create data flow and memory access reports like the Valgrind cachegrind report.

The difference between the block and instruction modes of operation in 3S is seen in the type and placement of 3S stubs inserted into the target program's assembly file. In the block mode, stubs are only inserted at the start of functions, basic blocks and around call instructions. In instruction mode, stubs are inserted before each instruction as well as at the start of functions and blocks.

Obviously the overhead of 3S instruction level instrumentation is much greater than that of 3S block level instrumentation, however it can still be less than that of alternative frameworks like Valgrind. The reason for this lies in the way 3S and Valgrind analyse Intel's complex instructions.

Valgrind expands the x86 CISC instructions to an internal RISC form and then instruments the RISC instructions before mapping them back (one-to-one) to x86 instructions to be run on the processor [1]. This results in not only a tool call overhead per instruction as with 3S, but also an instruction expansion overhead as a single x86 CISC instruction can be translated into multiple x86 RISC equivalents.

With 3S the x86 instructions are analysed statically at instrumentation time. The static analysis creates a description that contains all the aspects of the CISC instruction for consideration by the 3S tool at run-time. There is then a single 3S tool call per instrumented x86 instruction and the 3S tool is presented with a full description of the CISC instruction and it's parameters at run-time.

B. Clock Ticks vs Instruction Requests

The 3S framework stubs by default pass CPU clock tick information on to the 3S tool. The clock tick value is measured around the instrumented assembly code using the x86 RDTSC instruction. The tick information is passed to the 3S

tool in the 64 bit globals _3S_previous_block_start_ticks and _3S_previous_block_end_ticks. The tool can use these figures to calculate the time a block took to run.

It should be noted that by using tick deltas, we remove most of the tool instrumentation overhead from tick measurements. However, the tick delta figures can still be inaccurate for small blocks because of a residual caused by 6 instructions that are outside the RDTSC instructions in the 3S stub and because of instruction scheduling in the processor. The stub residual has been measured to be around 9 ticks on an Opteron processor.

Because of the inaccuracy in tick measurements for small blocks, Valgrind [1] does not use tick figures. Instead it uses Instruction Requests (IR). 3S tools also have IR information available to them by default. IR can be readily calculated in the tool as the instructions per block multiplied by the entries per block.

As measuring clock tick information adds overhead to the instrumented program, it is possible to flag a 3S tool as not requiring ticks by defining the variable _3S_INSTRUMENT_NO_TICKS in the tool's C code. This flag is defined in memory, regex, callgrind and some other 3S tools.

III. 3S Tools

The 3S framework makes creating program analysis tools easy. For example, the 3S hotspot tool generates an execution profile for a target program with only 2 lines of active C code.

Several example 3S tools can be found in the /tools directory of the 3S(ex) distribution. All tools use the header tool.h which describes the global variables that the 3S framework provides for the 3S tools.

To use the 3S tools you need to place your source files in a subdirectory of /source and run the 3SInstrument.sh script. This script is a wrapper that calls the 3S framework parser program (3SInstrument.py) to compile each of your source files to assembly and instrument them in turn. When all your files have been instrumented, the 3SInstrument.sh script links the instrumented assembly files together with the 3S tool you specified. The final 3S instrumented executable is placed in the /build directory.

Most tools create an output called <toolname>.3s in the working directory after the instrumented program has been executed. Some tools also create a pictorial report as a postscript file. The following sub-sections describe two of the 3S tools in more detail.

A. 3S Tool: loopgraph_d

The 3S loopgraph_d tool works at the block level and creates a deterministic regular expression describing a whole program's execution trace. The regular expression is annotated with hotspot information and saved in the loopgraph_d.3s report file. The tool also creates a dotty file loopgraph_d.dot with groups of acyclic-paths connected by repetition markers. The dotty file is compiled automatically into a postscript picture using neato [7] if the number of group nodes is less than 100.

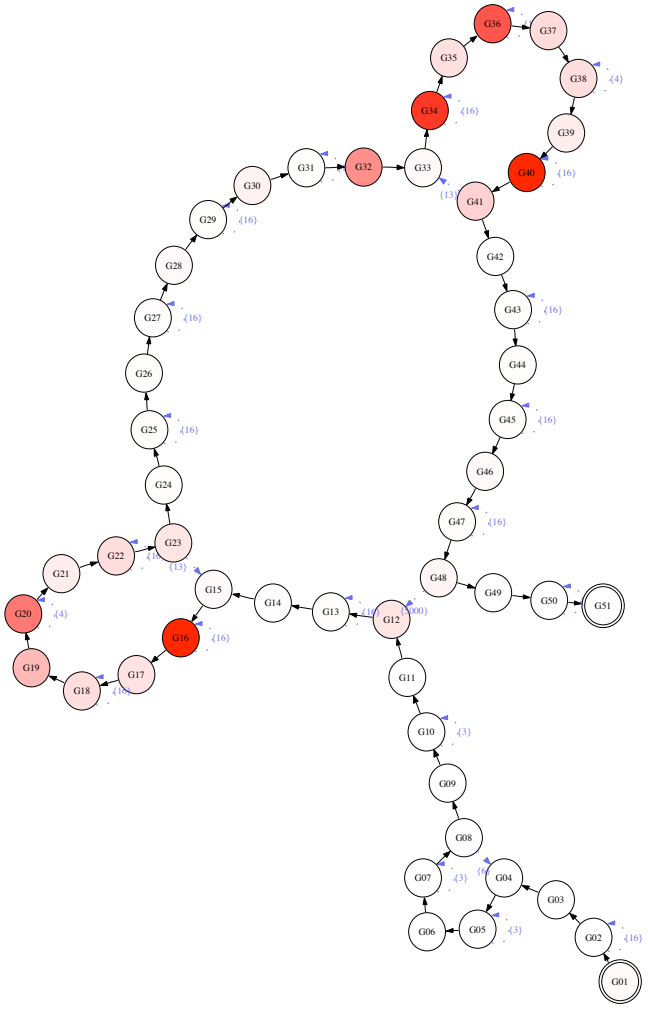

Fig. 1. 3S loopgraph_d tool for PRIV8®AES256 with gcc -O0

An example of the loopgraph_d postscript output is shown in Figure 1 for the PRIV8®AES256 8.2 [8] implementation. This picture was generated for 10,000 OFB AES256 iterations using source compiled to assembly with gcc -O0. Figure 2 shows the same program compiled with g++ -O0.

B. 3S Tool: memory

The 3S memory tool works at the instruction level. The 3S tool creates a report of all memory addresses read and written by every assembly instruction. The report is saved to the file memory.3s and can be easily plotted using Excel or a similar application.

Figure 3 shows the stack memory accesses by original source assembly line for 10,000 OFB iterations of the PRIV8®AES256 8.2 implementation compiled with g++ -O0.

IV. Performance

3S has been measured to be between 2 and 15 times as fast as Valgrind for a comparative tool implementation. The performance improvement is dependant on the x86 instructions

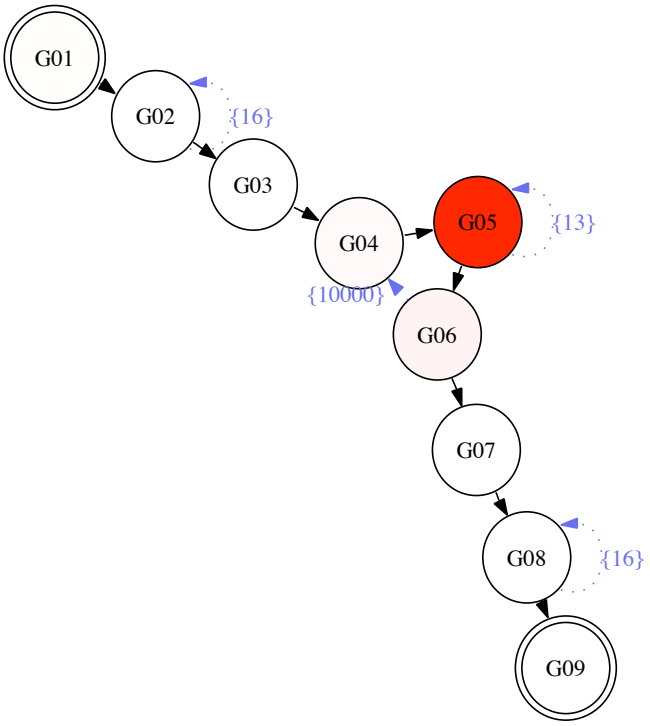

Fig. 2. 3S loopgraph_d tool for PRIV8®AES256 with g++ -O0

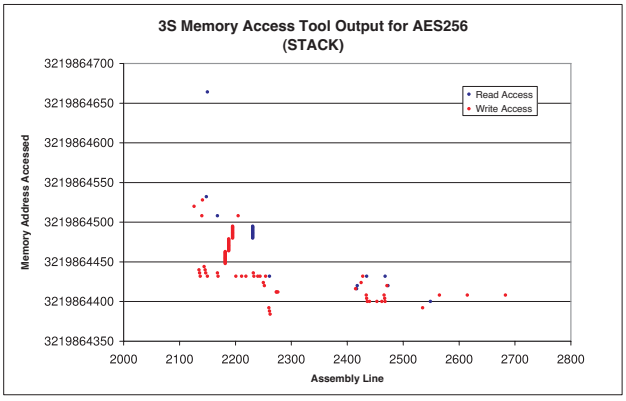

Fig. 3. 3S memory tool for PRIV8®AES256 with g++ -O0

used and the size of the basic blocks which are in turn governed by the source code compiler.

Instrumenting the PRIV8®AES256 8.2 implementation with a 3S callgrind tool using assembly generated by gcc -O2 produced a 5.5x performance improvement over Valgrind. With the g++ -O2 compiler, the performance improvement was 15.5x. The SPEC2000 GZIP benchmark could only be compiled with gcc. Despite this, the performance increase was consistently over 3x when compared with the current Valgrind distribution (3.2.0).

The 3S performance improvement can be verified using the 3SPerformance.sh script that comes with the 3S(ex) distribution.

V. Possible Enhancements

A. Optimisations

There are several optimisations that could be added to the 3S framework. Some obvious possibilities are:

1) only instrumenting a sub-set of blocks or instructions
2) in-lining the tool assembly
3) using register re-mapping to make the stub code more efficient

However, perhaps the single most useful characteristic of 3S is the ease with which a new user can pick-up the 288 line framework and start writing new tools. By adding performance optimisations, I believe this characteristic would be lost. I therefore strongly recommend that changes and additional features be kept to a minimum in 3S. If you must have a feature, it should be implemented in a specialist branch of the 3S code so that the current simple framework is not lost.

B. New 3S Tools

There are several new tools that would be of benefit to the 3S community. They include:

1) a non-deterministic (statistical) loopgraph variant
2) a block level static memory prediction function evaluated at run-time
3) a Valgrind style cachgrind tool

Creating 3S tools as separate modules that integrate with the framework does not complicate the 3S code and I recommend that someone set about creating these new 3S tools. Creating these new 3S tools would make a good Masters project. Please feel free to e-mail me if you would like to help.

VI. Conclusion

This document presented a brief overview of the 3S instrumentation framework and 3S analysis tools. One of the main advantages of 3S over other instrumentation frameworks is that it is extremely simple to understand being only 288 lines of code. This simplicity makes creating new 3S analysis tools easy and brings previously unimagined program analysis possibilities within the researcher's grasp.

Because of the 3S framework's simplicity, several unique analysis tools have already been created in record time. These include the loopgraph_d tool which creates a regular expression from a whole program execution trace and the memory tool which displays instruction level memory accesses. With loopgraph_d we have a way to automatically identify loops and control dependancies and with memory we can identify data dependancies.

The existing 3S tools are already casting new light on important commercial programs [8]. With the proposed new 3S tools, the rapidly growing 3S community will have a unique ability to shape the future of hardware and software engineering for years to come.

REFERENCES

[1] NETHERCOTE, N., SEWARD, J. Valgrind: A program supervision framework. *Proceedings of the 3rd Workshop on Runtime Verification* (http://valgrind.kde.org/, 2003).

[2] PEARCE, D.J., KELLY, P.H.J., FIELD, T., HARDER, U. GILK: A dynamic instrumentation tool for the linux kernel. *Proceedings of the 12th International Conference on Computer Performance Evaluation, Modelling Techniques and Tools 37*, pp 220–226, (2002).

[3] LARUS, J.R., SCHNARR, E. EEL: machine-independent executable editing. *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pp 291–300, (1995).

[4] SRIVASTAVA, A., EUSTACE, A. ATOM: a system for building customized program analysis tools. *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pp 196–205, (1994).

[5] LARUS, J.R. Whole program paths. *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pp 259–269, (1999).

[6] LUK, C. ET AL Pin: building customized program analysis tools with dynamic instrumentation. *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation* (2005).

[7] GANSNER, E.R., NORTH, S.C. An Open Graph Visualization System and its Applications to Software Engineering. *Software Practice And Experience,* 1-5, (http://www.graphviz.org/Documentation/GN99.pdf, 1999).

[8] PRIV8 LTD. http://www.priv8.com/.