CS 538W Execution Mining
Department of Computer Science
University of British Columbia

Event Loops as Execution Signatures

Nathan Taylor

tnathan@cs.ubc.ca
Department of Computer Science
University of British Columbia
Vancouver, BC, Canada

Keywords: Dynamic instrumentation, event loop, program analysis, Valgrind

1 Introduction

At the heart of nearly every modern program is a main event loop which typically runs continuously until the termination of the program. This loop is significant for the purposes of execution analysis because it is in some sense the ”root” of the program's behaviour, be it the message pump in a GUI-driven program or an *accept*() loop for a network server.

In the absence of of having the source for a given program, it is fruitful to analyze execution traces by studying their characteristic event loops. If one were to posit that programs of a similar purpose have similar event loops, then studying the main loop of a program as a sketch of its high level behaviour may yield interesting insights.

To this end, I present Loopgrind, a tool to instrument a running process with the intent of finding and analyzing the primary loops in a program. The first goal, loop discovery, is attained by capturing a basic block-level control flow graph, optimizing out linear block chains, and applying a simple heuristic that, in most cases, found the correct loop. The second, loop analysis, is done by logging changes in memory across loop iterations, giving a concise summary of state changes as the program runs.

There are many domains that would find a use for such a tool. For instance, developers could generate a high-level sketch of how their software behaves. Programmers joining a large project could use it to quickly get up to speed with the program's behaviour. In another domain altogether, since we at no point need to query debug information or program source, Loopgrind could be used to reverse engineer unknown programs. In particular, modern malware contains unpackers, runtime interpreters, and other sophisticated machinery that are difficult to reason about using conventional techniques but might lend themselves better to analysis within their containing loops.

Section 2, 3, and 4 detail how Loopgrind detects and analyzes a program's execution. Section 5 provides implementation specifics. A case study with two nontrivial programs, as well as a digression on performance, is briefly discussed in Section 6, and ideas for future work are listed in Section 7.

2 Runtime analysis

The default behaviour for Loopgrind is to generate a control flow graph of the execution trace. In such a graph, the vertices correspond to Valgrind superblocks (a single-entry, multiple-exit segment of instructions), and the weight of the edge linking a given superblock to its jump target is increased every time the jump is taken.

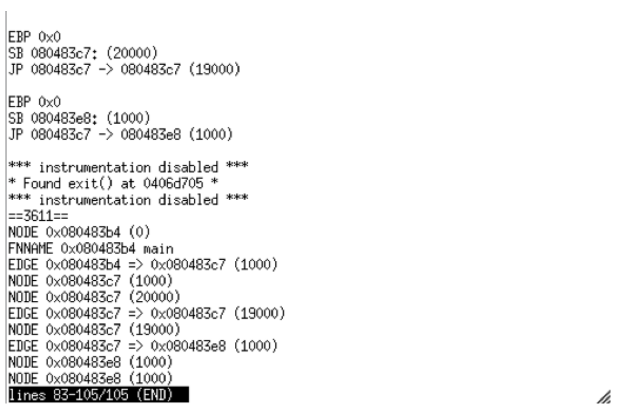
In order to keep the graph size tractable, Loopgrind only begins to build the graph after it has observed the program entering *main*(), and instrumentation ceases inside library calls (though we do log the entry into the PLT, so we log the call to the library function but not its internals). Most non-trivial programs exhibit a large control flow graph with very little a priori assumable structure. While there must be a source and sink vertex (corresponding to the entry into *main*() and the call into *exit*()), only in trivial programs will the graph be a DAG.

In our search for the primary loops' superblock headers, we wish to find the vertex in our control flow graph that best satisfies the following criteria:

2.1 Criteria for Loop Detection

1. A strong candidate for a primary loop will be entered many times through the course of the instrumentation.
2. A strong candidate for a primary loop will be first entered relatively early on in the instrumentation.
3. A strong candidate for a primary loop's stack frame will be close to *main*()'s.

In terms of the control flow graph, criterion 1 implies that the in-edges to the vertex we wish to find will have high weights. Criterion 2 implies that there exists a short path to the vertex from the *main*() source node. Criterion 3 exists because so far we have effectively flattened out the scoping component of the execution. It is in some sense the dual of the previous criterion in that it also insists that the loop is close to the entry point. We wish to place emphasis on superblocks whose stack frame is close to *main*()'s. To this end, every time we jump to a new superblock, the value that is added to the corresponding edge in the control flow graph decays exponentially, the farther away the current stack frame is from *main*()'s. Currently, we use the function $y = e^{-\frac{1}{512}x}$, where $x = EBP_{main} - EBP_{curr}$, though the decay coefficient was chosen through trial and error and should not be taken as gospel. It was initially felt that weighting the offset from *main*() would have the desirable result that we would place even less emphasis on functions with large parameters (ie. string functions). However, in hindsight, it's not necessarily clear that this is the case. Removing the decay coefficient unknown by simply incrementing a counter on *call*s and decrementing on *ret*s may prove to be a simpler and no less effective a solution.



```
EBP 0x0
SB 080483c7: (20000)
JP 080483c7 -> 080483c7 (19000)

EBP 0x0
SB 080483e8: (1000)
JP 080483c7 -> 080483e8 (1000)

*** instrumentation disabled ***
* Found exit() at 0406d705 *
*** instrumentation disabled ***
==3611==
NODE 0x080483b4 (0)
FNNAME 0x080483b4 main
EDGE 0x080483b4 => 0x080483c7 (1000)
NODE 0x080483c7 (1000)
NODE 0x080483c7 (20000)
EDGE 0x080483c7 => 0x080483c7 (19000)
NODE 0x080483c7 (19000)
EDGE 0x080483c7 => 0x080483e8 (1000)
NODE 0x080483e8 (1000)
NODE 0x080483e8 (1000)
lines 83-105/105 (END)
```

Figure 1: Generating control flow graph

3 Loop Detection

Loopgrind's analysis component is performed offline, in a helper Perl script that the Valgrind tool's output is piped to. There is no fundamental reason why any of the operations we perform in this script could not have been done during instrumentation, but making use of Perl's graph algorithms module allowed for rapid prototyping and experimentation.

As the script reads the output from the Valgrind tool, it builds up a representation of the control flow graph using the *Graph* :: *Directed* and *Graphviz* modules. Also, debug information is scraped from running *objdump* on the original binary, though this is purely for user presentation purposes and is in no way required; the analysis would be no different if the executable had not been compiled with debug information enabled.

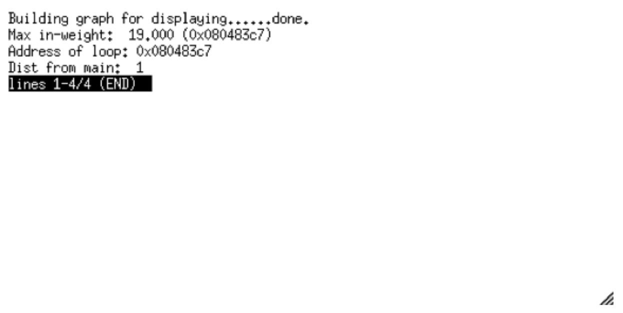
The initial plan was to simply find the largest cycle in the graph. However, in the presence of library calls, logging functions, or other "cross-cutting concerns", this scheme fails. To see why, consider a program that calls *malloc*() both inside the primary event loop, as well as in some outside initialization code. If we merely considered cycles in the graph, then we would have found a path from the initialization code *into the main loop*. Indeed, during experimentation it was not uncommon to generate a graph that was strongly connected for this reason.

After EOF is reached, the script contracts vertices linear chains of vertices, which in many cases, cuts down the number of vertices in the graph by an order of magnitude. To reduce the number of nodes in the graph, we compress superblock chains (ie. a series of vertices that only feature one in-edge and one out-edge). Removing self-loops to increase the number of chains found was attempted but it was discovered that this introduced incorrect results for our tests whose loop was only one superblock in length.

The vertices are then sorted by weights of all incoming vertices, and the top quartile of "distinct" vertices (that is, superblocks whose addresses differ by at least $0x100$) are presented along with their distance to *main*(). Multiple possible vertices are presented because in a sufficiently complicated program, there are likely many possible loops of interest. The top-ranked vertex is marked in the Graphviz representation before being printed as a .png. In instances where the binary's source is available, each vertex will be annotated with the superblock's starting line of code.

4 Memory diffs over loop iterations

Once we have identified a primary loop, we would next like to analyze how the program's state changes across iterations. To this end, we hash the address and value of each memory write, and each time the running executable jumps back to the loop header, the table is reset and its contents are printed out. As a result, across iterations, it becomes simple to identify commonalities between writes, such as

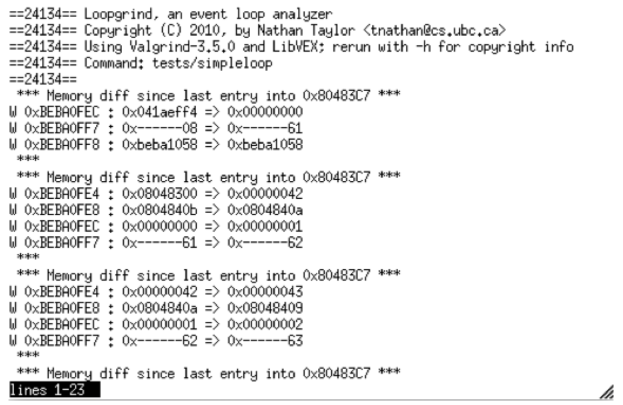


```
Building graph for displaying......done.
Max in-weight:  19.000 (0x080483c7)
Address of loop: 0x080483c7
Dist from main:  1
lines 1-4/4 (END)
```

Figure 2: Calculating loop header vertex

correlating a value that increments by a fixed amount over each iteration as a counter.

Obviously, x86 is an untyped assembly language. However, Valgrind's VEX intermediary representation has the nice property of being typed. As a result, it becomes far easier to infer the type of a memory location by its writes than if one was simply looking at the stream of x86 instructions. For instance, if an address is constantly being written to with values in the range of valid ASCII characters, it is far more likely that it is actually a char if the length of that write is that of $sizeof(char)$ as opposed to a full word.



```
==24134== Loopgrind, an event loop analyzer
==24134== Copyright (C) 2010, by Nathan Taylor <tnathan@cs.ubc.ca>
==24134== Using Valgrind-3.5.0 and LibVEX; rerun with -h for copyright info
==24134== Command: tests/simpleloop
==24134==
 *** Memory diff since last entry into 0x80483C7 ***
W 0xBEBA0FEC : 0x041aeff4 => 0x00000000
W 0xBEBA0FF7 : 0x------08 => 0x------61
W 0xBEBA0FF8 : 0xbeba1058 => 0xbeba1058
 ***
 *** Memory diff since last entry into 0x80483C7 ***
W 0xBEBA0FE4 : 0x08048300 => 0x00000042
W 0xBEBA0FE8 : 0x0804840b => 0x0804840a
W 0xBEBA0FEC : 0x00000000 => 0x00000001
W 0xBEBA0FF7 : 0x------61 => 0x------62
 ***
 *** Memory diff since last entry into 0x80483C7 ***
W 0xBEBA0FE4 : 0x00000042 => 0x00000043
W 0xBEBA0FE8 : 0x0804840a => 0x08048409
W 0xBEBA0FEC : 0x00000001 => 0x00000002
W 0xBEBA0FF7 : 0x------62 => 0x------63
 ***
 *** Memory diff since last entry into 0x80483C7 ***
lines 1-23
```

Figure 3: Listing memory diffs

Valgrind has trouble with floating point variables in general. For instance, it has no support for printing floats in their implementation of $printf()$ (better floating support has been on the Valgrind Google Summer of Code page for several years now), and in our tests, Loopgrind would die inside the Valgrind internals while instrumenting programs that call $atof()$. Therefore, the memory diff functionality is currently only reliable with primarily integer-based programs.

5 Implementation

Loopgrind is implemented in two parts. As stated above, loop detection and memory analysis is done through a Valgrind plugin, and the loop analysis and miscellaneous utilities are implemented with a Perl script. We perform no instrumentation that a more lightweight tool such as Pin or DynamoRIO couldn't also have done. Valgrind was used mainly because at the start of the project it was not clear whether a heavyweight solution would be needed. A secondary reason involved analyzing primary loop superblocks' instructions; conceivably, a smaller RISC-like instruction set may be easy to reason about than a larger ISA. Lastly, an arguably self-aggrandizing reason was that I was simply more keen to experiment with Valgrind than any of the alternatives.

While the Valgrind component is entirely self-contained and only requires a relatively modern *gcc*, the Perl script requires several modules to be installed via CPAN. For details, please see the README.

6 Case Study and Performance

During implementation, Loopgrind was tested primarily with trivial C programs with a known primary loop, to ensure the loop detection algorithm was working as required. For a real case study, Loopgrind was tested against two nontrivial programs, one of which was partially written by me, and another that was not.

The pieces of software in question were a Pascal compiler, *pal*, which was written by a team of four undergraduates for a senior-year compiler class at the University of Alberta, and *asc*, a JVM-like stack machine which the compiler generates code for. The rationale for choosing these programs were threefold: they are relatively complex pieces of software that do something useful for the number of lines of code that they are (as contrast, */bin/ls* is roughly 5000 lines long!) Also, the process of code generation and runtime execution closely mimics elements of modern malware which exhibit similar behaviour. Lastly, since I was familiar with the source already, it was easier to determine how accurate a job the tool was doing than a completely unknown piece of software.

pal and *asc* were instrumented while compiling a 140 line program and executing it in the runtime environment.

Loopgrind found two loops of interest in *pal*. The first was the token match loop inside the lexer (ceiling/src/lex.yy.c:946). Since the token is the smallest lexeme in the Pascal grammar, it makes sense that this loop is entered most frequently. The second is more interesting: it matched entry points into a function called $emitCode(char*)$, which writes translated instructions out to an assembly file (ceiling/src/codegen.c:172). This was surprising for two reasons: first, the function itself contains no loops, but is called within various loops across the compiler. While those loops themselves were not weighted highly, by virtue of repeatedly calling this one function, $emitCode()$ ended up being ranked highly. Also, because the function takes a string as argument, its frame offset will be quite large, so each jump into this superblock would be given a very small weight. It is surprising that despite this, the weights were sufficiently high that it ended up being the second-largest weight in the whole run.

In *asc*, Loopgrind found one loop, a $while(1)$... loop located in the $execute()$ function. This is the loop that iterates over a sequence of instructions and decodes each in a giant switch statement. This loop is prefaced with a comment indicating that this is the "main loop", which is about as strong a favourable metric for success as any.

No attempt to formally analyze the slowdown of Loopgrind was made. However, there is no reason to think that the performance of the Valgrind component would be any worse than the slowdown documented elsewhere. On the other hand, the Perl analysis component was far slower, especially in the non-trivial case studies; for instance, the vertex compression for *asc* took upwards of ten minutes to complete. It's not clear whether this was due to inefficiencies in the graph algorithms used in *Graph* :: *Directed*, or simply by virtue of the Perl implementation.

7 Future work

There are any number of directions that one could take this work. Perhaps the most critical one is to improve stability with floating point-heavy binaries. It should be noted that this is an issue with Valgrind in general and not related to Loopgrind in general. Also, there is very little material in the Perl script that could not be performed online as the Valgrind tool runs; moving the graph algorithm analysis into the latter component would be an excellent avenue of future work.

Currently we are generating output but make no attempt to study it. In particular, applying techniques from machine learning to both the main loop analysis as well as the memory diff component could yield interesting results. In particular, correlating the instructions (either the raw x86 assembly or the VEX IR) against a training set of representative program samples would be helpful to pinpoint what equivalence class a particular program might lie in. On the flip side, by virtue of the memory diff recording changes of length one, applying a hidden Markov model-like approach to studying what changes between iterations might also yield interesting results.

Lastly, we are unfortunately limited to only being able to study x86 Linux binaries. While arguably interesting in its own right, there's unfortunately a substantial amount of interesting executables – namely, Windows malware – which would be interesting to study. Porting this tool to run in QEMU would allow analysis of arbitrary binaries.