Benchmarking and improving performance in uftrace

Progress Report Meeting

Clément Guidi

Polytechnique Montréal

May 16, 2022

Table of contents

Table of Contents

Introduction

About uftrace: a userspace function tracer for C/C++ applications

Development efforts:

- increase instrumentation coverage (new probe insertion methods)
- minimize the overhead of probes
- integrate with other tools (e.g. LTTng support)

Need for benchmarking tools:

- to quantify performance
- to identify and target performance issues
- to compare the efficiency of new methods
- to provide scientific measurements

Table of Contents

Previous work

Main contributions

- LTTng integration: emit events through **LTTng-UST** channels
- Libpatch: lightweight dynamic patching with extensive features (external library) *[Olivier Dion]*
- indirect jump resolution: improve patching success rate by identifying **indirect jump locations** (external library) *[Gabriel Pollo-Guilbert]*
- x86 runtime instrumentation: add and remove tracepoints at execution using a **locking mechanism** and **out of line execution** *[Christian Harper-Cyr, Anas Balboul, Ahmad Shahnejat and Gabriel Pollo-Guilbert]*
- client command: send commands to a **libmcount daemon** running inside a uftrace target *[Clément Guidi]*

Previous work

Side work

- enhance conditional compilation
  - build configuration flags `--without-libresolver`, `--without-libpatch`, `--without-lttng` and `--without-daemon`
- make uftrace suitable for benchmarking
  - add architecture dependent statistics
  - add `--dry-run` option
- follow upstream changes (rebase)

Table of Contents

The uftrace benchmark: a tool to evaluate the performance of two domains

- instrumenting: efficiency of probe insertion
- tracing: efficiency of probe execution

All in one tool for efficient deployment and reproducibility. Features:

- application building (build farm with multiple versions of binaries)
- instrumentation benchmarking
- probe execution benchmarking
- results display and archiving (work in progress)

Technical details:

- build around a set of python scripts and C programs using `perf` events

Benchmark

Structure of the benchmark

Benchmarking uftrace on a list of ≈30 applications with mixed characteristics:

- bigger or smaller binary size
- higher or lower function count
- single- or multi-threaded
- C or C++ code

Uftrace versions to compare:

- baseline (upstream)
- fully dynamic instrumentation
- LTTng integration

Sample output of raw instrumentation data on AMD64:

```
dynamic patch stats for 'ls'
   total:       478
 patched:       464 (97.07%)
  failed:        14 ( 2.92%)
        total:        14
    bad symbol:        0 (  0.00%)
      capstone:        0 (  0.00%)
     no detail:        0 (  0.00%)
 relative jump:        0 (  0.00%)
 relative call:        0 (  0.00%)
           pic:        3 ( 21.42%)
 jump prologue:        0 (  0.00%)
 jump function:       11 ( 78.57%)
 skipped:          0 ( 0.00%)
```

Benchmark

Results– instrumenting

app	python			baseline		full dynamic	
gcc flag				-O0	-O3	-O0	-O3
coverage	total			9079	9079	9079	9079
	patched			99.94%	97.31%	98.86%	90.18%
	failed			0.05%	1.05%	1.13%	8.19%
		no detail			9.15%		
		relative jump			1.85%	4.85%	88.55%
		pic		100.00%	89.00%	95.15%	11.45%
		jump prologue					
	skipped				1.62%		1.62%
		cold			22.90%		22.90%
		min size			77.10%		77.10%
time	latency (us	mean		34	50	31	46
		median		16	23	16	24
		std		148	215	101	128
		min		2	2	3	3
		max		9368	12142	5674	4891
	total time **(ms)**			320	261	352	276

Benchmark

Results– instrumenting

Benchmarking on `python` binary (has REPL, useful for runtime testing)

Comments about coverage:

- total of 9070 functions
- patching failures due to position-independent code
- patching coverage goes down with optimization
  - possible relative jumps
  - symbols missing details
  - function too small (need tracing?)
  - code optimization
- fully dynamic implementation: indirect jump resolution disabled so less coverage

Comment about performance:

- fully dynamic implementation has serial synchronization step
  - individual patching faster
  - overall patching slower

Benchmark

Results– instrumenting

app	git			baseline		full dynamic	
gcc flag				-O0	-O3	-O0	-O3
coverage	total			5212	5212	5212	5212
	patched			99,92%	94.43%	99.38%	92.49%
	failed			0,07%	3.95%	0.61%	5.89%
		no detail			66.51%		
		relative jump				87.50%	77.53%
		pic		100.00%	32.52%	12.50%	21.82%
		jump prologue			0.97%		0.65%
	skipped				1.61%		1.61%
		cold			19.05%		19.05%
		min size			80.95%		80.95%
time	latency (us	mean		32	66	32	64
		median		16	28	16	28
		std		73	174	71	165
		min		2	2	3	3
		max		2758	2628	2534	2533
	total time (**ms**)			389	342	425	356

Benchmark

Results– instrumenting

Benchmarking on `git` binary

Comments about coverage:

- total of 5212 functions
- patching failures due to position-independent code
- patching coverage goes down with optimization
  - possible relative jumps
  - symbols missing details
  - jumps in function prologues
  - function too small (need tracing?)
  - code optimization

Comment about performance:

- same observations as before
- patching measured on patching success (function count varies)

Benchmark

Results– instrumenting

app	make			baseline		full dynamic	
gcc flag				-O0	-O3	-O0	-O3
coverage	total			344	344	344	344
	patched			99.48%	90.69%	96.03%	87.79%
	failed			0.51%	2.03%	3.06%	4.94%
		no detail					
		relative jump			71.43%	83.34%	88.24%
		pic		100.00%	28.57%	16.66%	11.76%
		jump prologue					
	skipped				7.26%		7.26%
		cold					
		min size			100.00%		100.00%
time	latency (us	mean		82	77	70	74
		median		30	31	29	30
		std		198	192	159	187
		min		3	3	3	4
		max		2152	2194	1756	2251
	total time (**ms**)			33	25	29	26

Benchmarking on `make` binary

Comments about coverage:

- total of 344 functions
- patching failures due to position-independent code
- patching coverage goes down with optimization
  - possible relative jumps
  - function too small (need tracing?)

Comment about performance:

- same observations as before
- fully dynamic implementation overall faster, due to patch failures

Benchmark

Results– tracing

- dynamic: fully dynamic instrumentation
- pg: compiled with -pg flag (mcount call)
- fentry: compiled with `-finstrument-functions` (`cyg_prof` calls)

	baseline			full dynamic			lttng		
	dynamic	pg	cygprof	dynamic	pg	cygprof	dynamic	pg	cygprof
overhead (ns)	2389	2400	4768	2405	2395	4787	4834	4847	9655
branch misses	5	4	7	5	4	7	7	6	11
instruction count	1439	1413	2819	1583	1465	3015	4799	4673	9419

Benchmark

Results– tracing

- fully dynamic on par with baseline, adds a small overhead (data lookup in hashmaps)
- fully dynamic as efficient as compiler-assisted pg builds
- LTTng brings a consistent overhead (no buffering in libmcount)

Table of Contents

Table of Contents

Work in progress

Further benchmarking

Future work on the benchmark includes:

- benchmarking memory footprint of probes
- testing batch patching strategies (optimize threshold)
- stress testing runtime instrumentation
- benchmarking tracepoint removal
- benchmarking libpatch in uftrace

Work in progress
Upstreaming

Slow progress on upstreaming: objective of the summer

- fully dynamic patching
- LTTng integration
- indirect jump resolution (bugs to fix)

Table of Contents

Conclusion

- benchmarks useful to identify weaknesses and prevent regressions
  - solutions are under development
- room for improvement in current methods
- more comprehensive benchmark to come

Conclusion

Find prototypes at `https://github.com/dorsal-lab/uftrace`

Thank you!