An Evaluation of Valgrind Performance

on Commercially Available Maker Devices

Vivi Chuang
Greg Cusack
Kunal Patel
Nathan Pilbrough
Zhengshuang Ren

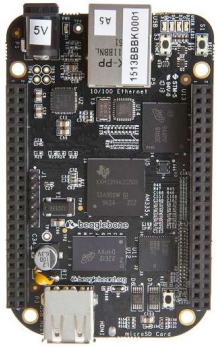
Table of Contents

Motivation

- Growth of IoT requires efficient embedded systems
- Memory leaks prevent systems from executing efficiently
- Valgrind framework used to detect memory leaks in a wide range of applications
- No study available on memory leak checking on embedded devices
- Goal: identify Valgrind's performance on common maker devices
  - Help developers effectively optimize their code for maker devices

Approach & Experiments

- Test Systems
  - BeagleBone Black
  - Intel Edison
  - Ubuntu VirtualBox VM
  - Macbook Pro (El Capitan)
- Valgrind outputs 4 types of memory leaks
  - Definitely Lost
  - Indirectly Lost
  - Possibly Lost
  - Still reachable
- Measured memory loss and execution time
- Developed a test suite of various leaks types
- Open-source case studies
  - FANN and Guetzli

[1] 

[2] 

[1]:https://www.google.com/search?q=beaglebone&source=lnms&tbm=isch&sa=X&ved=0ahUKEwiUu7Lyx7fUAhVW72MKHWThC-4Q_AUIDSgE&biw=1280&bih=703#imgrc=Vth7w1PQZ2UYrM:

[2]:https://www.google.com/search?q=intel+edison&source=lnms&tbm=isch&sa=X&ved=0ahUKEwjQ8-fqyLfUAhVN82MKHS0RDgAQ_AUIDSgE&biw=1280&bih=703#imgrc=y-wTBORfNOYJUM:

Results

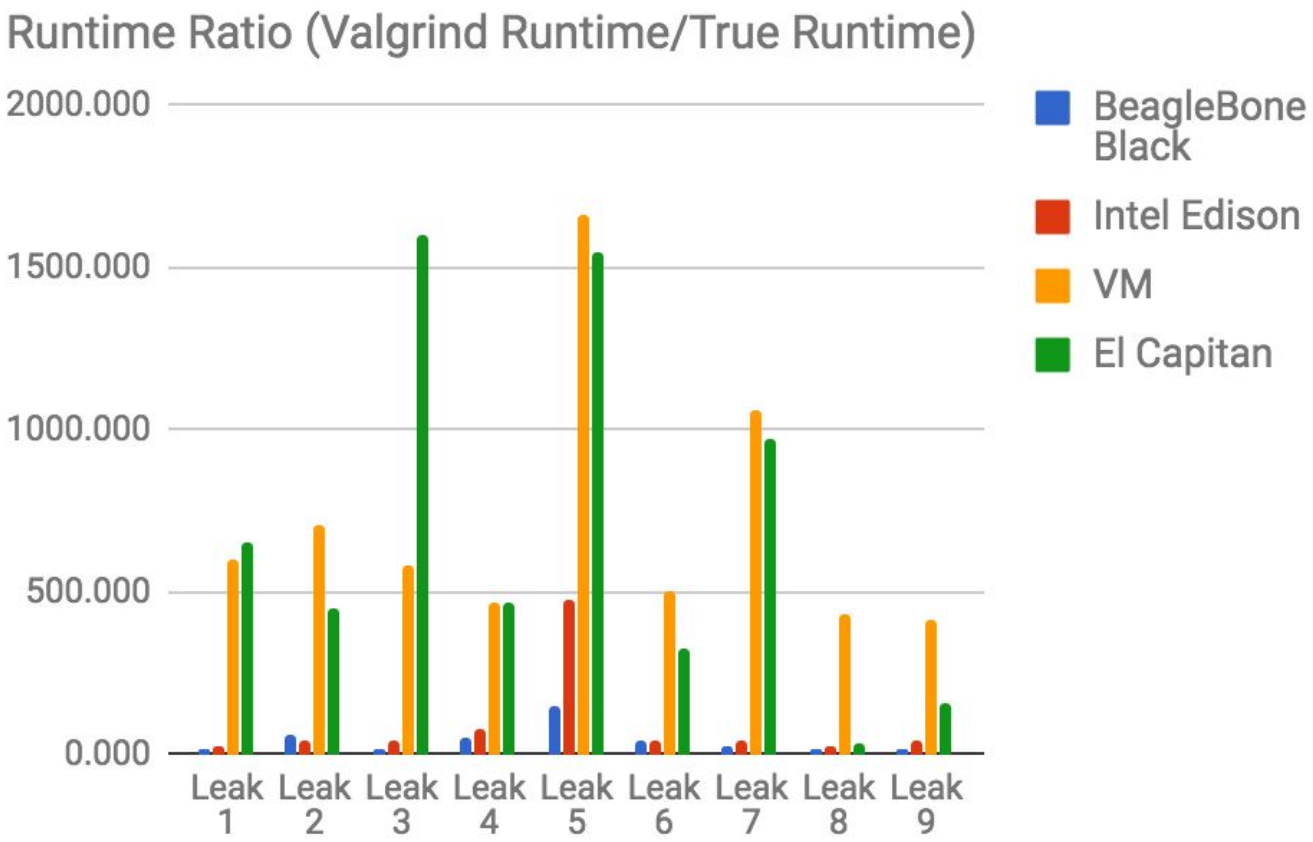
- Test Suite
  - 9 test cases tested across all 4 systems
  - All four systems report **consistent** number of memory leaks, except:
    - El Capitan reports 2064 “possibly lost” bytes for every test case
      - Other three systems report zero “possibly lost” bytes

- Runtime Overhead Analysis
  - Slowdown ratio is defined as:

$$slowdown = \frac{runtime\ with\ Valgrind\ (ms)}{true\ runtime\ (ms)}$$

  - Slowdown factor of BeagleBone and Edison are much smaller compared to that of VM and El Capitan

Results


Runtime Ratio (Valgrind Runtime/True Runtime)
2000.000
1500.000
1000.000
500.000
0.000
Leak 1
Leak 2
Leak 3
Leak 4
Leak 5
Leak 6
Leak 7
Leak 8
Leak 9
BeagleBone Black
Intel Edison
VM
El Capitan

Case Study - FANN

- Fast Artificial Neural Network (FANN)
  - Multilayer artificial neural network in C
- All testing platforms did not detect memory leak in source code
  - Active leaks: added detrimental code into FANN library
  - Passive leaks: removed deallocation in FANN libraries
- All platforms found similar memory lost values
  - BeagleBone and Edison detected identical total memory lost
  - Active bugs placed in specific locations not identified by Valgrind

Case Study - Guetzli

- Open-source image compressor
- Does not typically run on maker devices
- Very memory intensive application
  - Used to test the the maker devices at their limit
- Injected memory leaks
  - Active: added malicious memory allocations over time
  - Passive: removed deallocation methods from source code
- Results differed across devices, unlike the FANN case
  - Require deeper understanding of the Libpng function: `png_create_info_struct()`

Case Study - SmartBed

- Design
  - IoT application developed by UCLA students to help monitor patient activity
  - Array of Edisons connected to 9DOF sensors communicate with a 5th server Edison
  - Implements FANN library to learn patient behavior
- Application ported to BeagleBone for IoT application
- Results
  - Memory leaks found in server source code
  - BeagleBone and Edison reported same number of leaks
  - SmartBed team able to fix some reported leaks

[3]

[3] http://www.hill-rom.com/contentassets/00942b52a7f54d669b53e4592f07ea2b/ai0450_1550.png

Conclusion

- Maker devices accurately identify memory leaks in test suite cases
- Valgrind is not an exhaustive memory leak detector
- Case Studies
  - FANN
    - Similar performance on maker devices
    - False negatives
  - Guetzli
    - Passive leak tests yield varying results
  - SmartBed
    - Memory leaks found in team's software
    - Allowed them to make quick performance improvements to source code

Future Work

- Test more large scale IoT applications
  - Tensorflow
  - OpenCV
  - Tiny-DNN
- Investigate false negatives in FANN
  - Why leak injection location affects reporting of errors
- Raspberry Pi failure identification
  - Port Valgrind to Raspberry Pi successfully
  - Study performance metrics on this system

Reference

[1] “MSDN: Microsoft Development, MSDN Subscriptions, Resources.” URL: https://msdn.microsoft.com/en-us/library/ms859408.aspx
[2] J. Clause and A. Orso, “LEAKPOINT: Pinpointing the Causes of Memory Leaks,” *ICSE*, pp. 515–524, 2010.
[3] “The nine types of memory leaks,” *Software Verify*, 2011. URL: http://blog.softwareverify.com/the-nine-types-of-memory-leak/
[4] J. Zhao, Y. Jin, and R. M. Jr, “Injecting Memory Leaks to Accelerate Software Failures,” *IEEE International Symposium on Software Reliability Engineering*, pp. 260–269, 2011.
[5] T. Tsai, K. Vaidyanathan, and K. Gross, “Low-Overhead Run-Time Memory Leak Detection and Recovery,” *Pacific Rim International Symposium on Dependable Computing*, 2006.
[6] R. Matias, A. Andrzejak, F. Machida, D. Elias, and K. Trivedi, “A Systematic Differential Analysis for Fast and Robust Detection of Software Aging,” *IEEE 33rd Int. Symp. Reliab. Distrib. Syst.*, 2014.
[7] A. Bovenzi, “On the Aging Effects due to Concurrency Bugs : a Case Study on MySQL,” *IEEE 23rd Int. Symp. Softw. Reliab. Eng.*, pp. 211–220, 2012.
[8] F. Qin, S. Lu, and Y. Zhou, “SafeMem: Exploiting ECC-Memory for Detecting Memory Leaks and Memory Corruption During Production Runs,” International Symposium on High-Performance Computer Architecture, 2005.
[9] J. Seward. Valgrind. “Question #4: Other Tools?” URL: http://www.Valgrind.org
[10] S. Nissen, “Implementation of a Fast Artificial Neural Network Library,” *Department of Computer Science University of Copenhagen*, 2003.
[11] Alakuijala, J., Obryk, R., Stoliarchuk, O., Szabadka, Z., Vandevenne, L., and Wassenberg, J. “Guetzli: Perceptually Guided JPEG Encoder.” *CoRR, abs/1703.04421*, 2017.
[12] S. Lu and Z. Li, “*BugBench*: Benchmarks for Evaluating Bug Detection Tools”, *UIUC*,2005.
[13] G. C. Necula, S. McPeak, and W. Weimer. “CCured: Type-safe retrofitting of legacy code.” *POPL*, 2002.
[14] R. Hastings and B. Joyce. “Purify: Fast detection of memory leaks and access errors.” *Usenix Winter Technical Conference*, 1992.
[15] N. Nethercote, J. Seward, “Valgrind: a framework for heavyweight dynamic binary instrumentation,” *ACM SIGPLAN*, v.42 n.6, 2007.
[16] F. Qin, C. Wang, Z. Li, H. Kim, Y. Zhou, and Y. Wu. “Lift:A low-overhead practical information flow tracking system for detecting security attacks.” *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture*, 2006.
[17] D. Bond and K. McKinley, “Bell: Bit-Encoding Online Memory Leak Detection,” *ASPLOS*, pp. 61–72, 2006.
[18] Y. Sui, D. Ye, and J. Xue, “Static Memory Leak Detection Using Full-Sparse Value-Flow Analysis,” *ISSTA*, pp. 254–264, 2012.
[19] Itseez, Open Source Computer Vision Library, “OpenCV DevZone,” 2014. URL: http://code.opencv.org/projects/opencv/issues

Thanks! for listening

Questions?

Appendix Follows

Related Work

- Valgrind Performance Evaluation in BugBench
  - BugBench evaluates Valgrind compared to other bug detecting tools CCured and Purify
  - We focused on Valgrind to narrow scope
- Valgrind memory following properties
  - Netercote et al. found that, compared to Purify, memcheck is more capable at tracking undefined value errors without altering program functionality
  - Clause et al. developed a tool *Leakpoint* that allows users to find the source of memory leaks based on Valgrind messages
  - Sleigh is another tool developed and though it is less accurate than Valgrind, it only adds 11-29% overhead execution time
- Valgrind vs. Saber, a static memory leak detector
  - Saber uses sparse value flow graphs to detect memory leaks
  - Saber detected leaks with a false positive rate of 18.7%, but runtime tests were not conducted

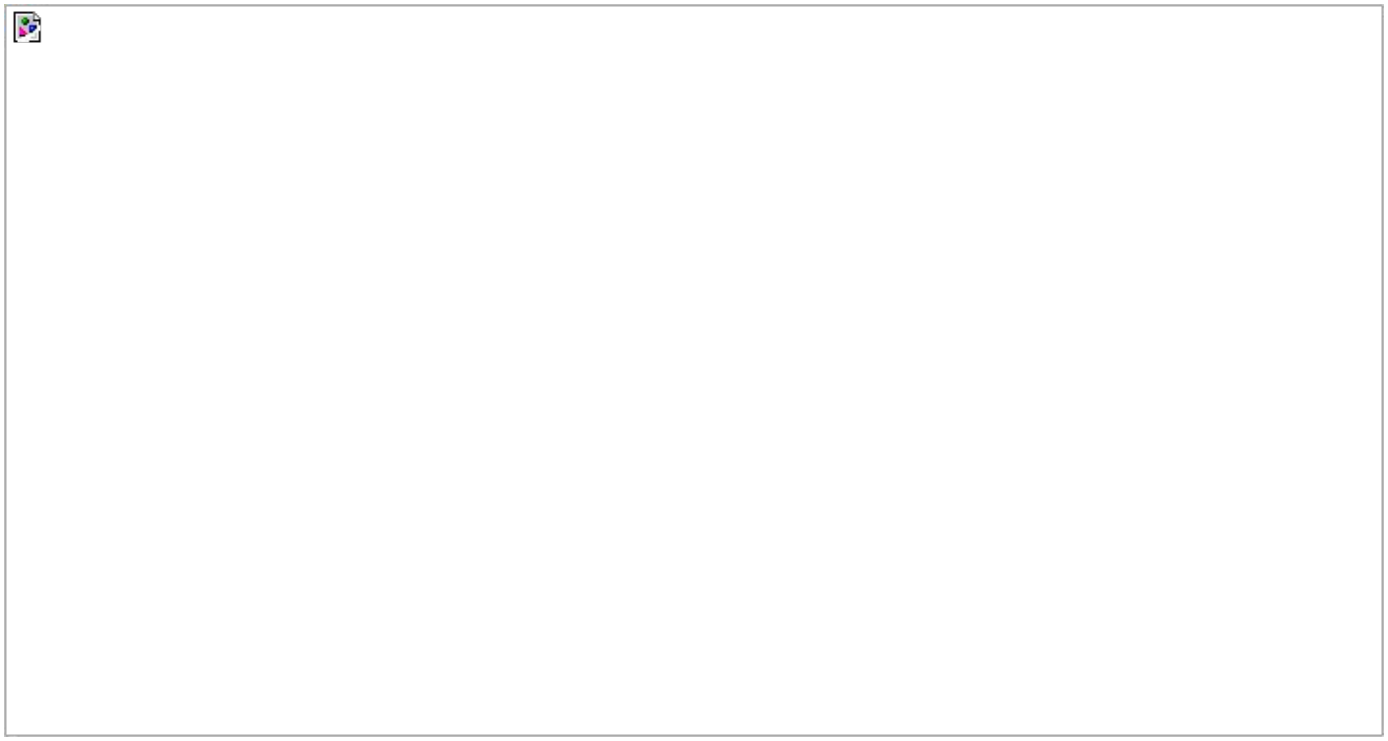
Reason for Dynamic Analysis Tool

```
1   void createLeak()
2   {
3       int *memptr;
4       memptr = new int[2];
5       try {
6           if(winnerScore <= loserScore) {
7               throw 1;
8           }
9       }
10      catch(int x) {
11          cout << "Winner score <= Loser score.  ERROR: " << x << endl;
12          return;
13      }
14      memptr[0] = winnerScore;
15      memptr[1] = loserScore;
16      //do whatever you want here
17      delete [] memptr;
18  }
```

Guetzli Active Leak Injection

```
1   case 3: {
2           // RGB
3           int *leak_ptr;
4           fprintf(stderr, "Reached RGB area\n");
5           for (int y = 0; y < *ysize; ++y) {
6             leak_ptr = new int[10];
7             const uint8_t* row_in = row_pointers[y];
8             uint8_t* row_out = &(*rgb)[3 * y * (*xsize)];
9             memcpy(row_out, row_in, 3 * (*xsize));
10          }
11          break;
12        }
```

SmartBed High-Level System Overview



Guetzli Performance

Guetzli		Beaglebone Black	Intel Edison	VM	El Capitan
Passive	Definitely Lost (bytes)	1272	1168	1680	3256
	Indirectly Lost (bytes)	395525	382236	396593	396968
	Possibly Lost (bytes)	0	13320	0	2064
Active	Definitely Lost (bytes)	10320	10320	10320	0
	Indirectly Lost (bytes)	0	0	0	0
	Possibly Lost (bytes)	0	0	0	2064

FANN Passive Leak Performance

Test Case, Lost Memory		Beaglebone Black	Intel Edison	VM	El Capitan
Momentums	Definitely Lost (bytes)	1996	1996	2856	2856
	Indirectly Lost (bytes)	654868	540820	802640	802640
	Possibly Lost (bytes)	0	114048	0	2064
Mushroom	Definitely Lost (bytes)	340	340	504	504
	Indirectly Lost (bytes)	2247572	2247572	2328980	4359980
	Possibly Lost (bytes)	2031000	2031000	2031000	2064
Robot	Definitely Lost (bytes)	340	340	504	504
	Indirectly Lost (bytes)	269452	155404	297200	297200
	Possibly Lost (bytes)	0	114048	0	2064
	Total Lost (bytes)	5205568	5205568	5463684	5469876

FANN Active Leak Performance

Test Case, Lost Memory		Beaglebone Black	Intel Edison	VM	El Capitan
Momentums	Definitely Lost (bytes)	1996	1996	2856	2856
	Indirectly Lost (bytes)	654868	540820	802640	802640
	Possibly Lost (bytes)	0	114048	0	2064
Mushroom	Definitely Lost (bytes)	340	340	504	504
	Indirectly Lost (bytes)	2247572	2247572	2328980	4359980
	Possibly Lost (bytes)	2031000	2031000	2031000	2064
Robot	Definitely Lost (bytes)	340	340	504	504
	Indirectly Lost (bytes)	269452	155404	297200	297200
	Possibly Lost (bytes)	0	114048	0	2064
	Total Lost (bytes)	5205568	5205568	5463684	5469876

Threats to Validity

External Validity

- While we acknowledge that we only tested two maker devices (BeagleBone Black and Intel Edison) and only evaluated memory leaks on two open source C/C++ projects (FANN and Guetzli), this paper itself serves as an external validity to test Valgrind's performance as we compared it among different platforms. We weren't able to incorporate as many maker devices as possibles due to limited resources available to the team. But we tried to evaluate memory leaks on some of the more popular open source libraries. FANN and Guetzli are both memory intensive and FANN works well with IoT devices. And we believe that understanding Valgrind performance on large scale open source applications can be valuable to other programs.

Internal Validity

- When testing the number of memory leaks in FANN, Guetzli, and SmartBed, the leaks recorded are assuming that Valgrind identified all memory leaks. However, as proven in the FANN case study, Valgrind was not completely accurate in reporting memory leaks. As a result, there are factors within Valgrind that we did not examine that prevent us from guaranteeing all memory leaks were found. Furthermore, the accuracy of Valgrind on the maker devices cannot be fully identified in this report due to this found Valgrind issue.

Construction Validity

- Our goal was to measure the performance of Valgrind's memcheck tool on common maker devices. In order to do so, we identified different types of memory leaks that occur in code. We then took those common leaks and implemented them into the nine test cases mentioned in Section IV. However, while we were able to identify "possibly lost" bytes in the two case studies, we were unable to replicate "possibly lost" bytes in our test cases in Section IV. Furthermore, there is also a chance that other types of memory leaks exist that we did not cover. In the end, we were able to develop strong and extensive test cases; however, we cannot guarantee the test cases exhaust all types of memory leaks.

Runtime ratio	Average	Standard Deviation
BBB	45.5	44
Edison	91.9	144.9
VM	715.5	407.2
El Capitan	689.9	568.7

Motivation

- Growth of IoT resulting in more embedded devices running critical applications
- Restricting memory leaks on embedded devices is vital
- Memory leaks are likely to cause issues on low-resource devices
  - Leaked memory exhausts low-memory systems quickly
- Study on memory leak checking is not currently available for maker devices
- Memory leaks are essentially improper static or dynamic deallocation of memory in source code
- Neglecting to handle memory leaks appropriately can result in the system crashing once all memory has been used.
- Valgrind
  - Popular open-source framework to analyze code performance issues
  - memcheck tool: for finding memory leaks within code
- This study is useful in aiding developers effectively optimize their code for maker devices