Combating Run-time Performance Bugs with Performance Claim Annotations

November 12, 2020

Zachery Casey & Michael Shah

casey.z@northeastern.edu mikeshah@northeastern.edu

Performance Bugs and Specifications

- Specifications can be small

- Specifications are necessary for code reuse

- Abstraction can make performance hard

Methods for finding Performance Bugs

1. Ad-hoc inline checks (printf debugging)

2. Ad-hoc tools (e.g. gprof, VTune)

3. Testing (e.g. Freud, RadarGun)

GNU gprof http://sourceware.org/binutils/docs/gprof/

Intel VTune https://software.intel.com/en-us/vtune

Analyzing system performance with probabilistic performance annotations <u>EuroSys '20: Proceedings of the Fifteenth</u> <u>European Conference on Computer Systems</u> <u>https://doi.org/10.1145/3342195.3387554</u>

RadarGun: Toward a Performance Testing Framework <u>8th Symposium on Software Performance 2017</u> <u>RadarGun</u>

Our Criteria

1. In-source performance specifications

2. Toggle without recompilation

3. "Accurate"

Related Work - Mobile Performance Assertions

- pa_start(id) → pa_end(id, assertion)

- Inter Process Communication (IPC) backend

- Opening the calendar application should take less than 2 seconds plus 5 ms per each appointment in current month

Performance assertions for mobile devices <u>ISSTA '06: Proceedings of the 2006 international</u> <u>symposium on Software testing and analysis</u> <u>https://doi.org/10.1145/1146238.1146264</u>

Related Work - Mobile Performance Assertions

- Implemented as library

- Closed system (software and hardware)

- Records unnecessary information (1.7ms/3ms)

Our Criteria

1. In-source performance specifications

2. Toggle without recompilation

3. "Accurate"

Our Criteria

1. In-source performance specifications

2. Toggle without recompilation

3. "Accurate"

• Simple, motivating example...

```
// Copy integers from pointer-array
 2
                      into cleared vector.
    11
 3
    void copy_into(int* ys, unsigned ys_len,
 4
                    std::vector<int>& xs) {
 5
        assert(ys != nullptr);
 6
        // Because of .reserve(),
 7
        // malloc should be called at most once.
 8
        PCA(MaxAlloc, PCA_INT 1);
 9
10
        xs.clear();
11
        xs.reserve(ys_len);
12
13
        for (unsigned i = 0; i < ys_len; ++i)</pre>
14
            xs.push_back(ys[i]);
15
```

4

5

6

8

9 10

11

12 13

14

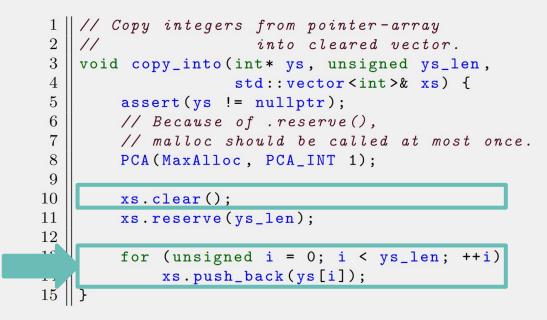
15

• Clear documentation

• Clear type signature

```
Copy integers from pointer-array
into cleared vector.
void copy_into(int* ys, unsigned ys_len,
                std::vector<int>& xs) {
    assert(ys != nullptr);
    // Because of .reserve(),
    // malloc should be called at most once.
    PCA(MaxAlloc, PCA_INT 1);
    xs.clear();
    xs.reserve(ys_len);
    for (unsigned i = 0; i < ys_len; ++i)</pre>
        xs.push_back(ys[i]);
```

• Straightforward implementation



• Implicit requirement in documentation

• Extra requirement on type

```
Copy integers from pointer-array
 \frac{2}{3}
                       into cleared vector.
    void copy_into(int* ys, unsigned ys_len,
                    std::vector<int>& xs) {
        assert(ys != nullptr);
 6
        // Because of .reserve(),
        // malloc should be called at most once.
 8
        PCA(MaxAlloc, PCA_INT 1);
 9
10
        xs.clear();
11
        xs.reserve(ys_len);
12
13
        for (unsigned i = 0; i < ys_len; ++i)</pre>
14
            xs.push_back(ys[i]);
15
```

• Maximum number of allocations in a scope

 Implementation should have <= 1 allocations

```
// Copy integers from pointer-array
 2
                      into cleared vector.
 3
    void copy_into(int* ys, unsigned ys_len,
                    std::vector<int>& xs) {
 4
 5
        assert(ys != nullptr);
 6
        // Because of .reserve(),
        // malloc should be called at most once.
        PCA(MaxAlloc, PCA_INT 1);
 9
10
        xs.clear();
11
        xs.reserve(ys_len);
12
13
        for (unsigned i = 0; i < ys_len; ++i)</pre>
14
            xs.push_back(ys[i]);
```

Figure 1: PCA on unnecessary allocation

15

• Maximum number of allocations in a scope

 Implementation should have <= 1 allocations

```
// Copy integers from pointer-array
 2
                      into cleared vector.
 3
    void copy_into(int* ys, unsigned ys_len,
                    std::vector<int>& xs) {
 4
 5
        assert(ys != nullptr);
 6
        // Because of .reserve(),
        // malloc should be called at most once.
 8
        PCA(MaxAlloc, PCA_INT 1);
 9
10
        xs.clear();
12
13
        for (unsigned i = 0; i < ys_len; ++i)</pre>
14
            xs.push_back(ys[i]);
15
```

• Maximum number of allocations in a scope

• Implementation should have <= 1 allocations

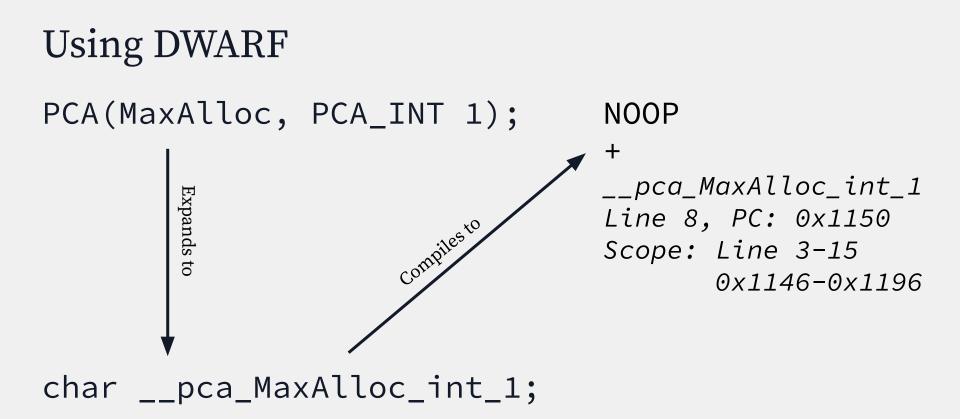
```
// Copy integers from pointer-array
 \frac{2}{3}
                       into cleared vector.
    void copy_into(int* ys, unsigned ys_len,
                    std::vector<int>& xs) {
 4
 5
        assert(ys != nullptr);
 6
        // Because of .reserve(),
        // malloc should be called at most once.
 8
        PCA(MaxAlloc, PCA_INT 1);
 9
10
        xs.clear();
11
        xs.reserve(ys_len);
12
13
        for (unsigned i = 0; i < ys_len; ++i)</pre>
14
             xs.push_back(ys[i]);
15
```

PCA Overview

1. Write performance claims in assert style or start-end

2. Compile with debug information

3. Check PCAs using dynamic binary instrumentation (Pin)



DWARF Debugging Standard <u>http://dwarfstd.org/</u>

Using DWARF

• Annotations are stored in the binary

• No runtime overhead

• Can freely access annotations as required

Using DWARF

\$ read_pcas ./exec ./pcas.txt

\$ cat ./pcas.txt

MaxAlloc INT 1 [1146 1196]

Pin - Dynamic Binary Instrumentation (DBI)

• Dynamically insert instrumentation at any location

• Instrumentation is performed at run-time, can be toggled

• Inspect, at instruction-level, program execution

A dynamic binary instrumentation engine for the ARM architecture <u>CASES '06: Proceedings of the 2006 international</u> <u>conference on Compilers, architecture and</u> <u>synthesis for embedded systems</u> <u>https://doi.org/10.1145/1176760.1176793</u> <u>Pin Site</u>

Our Criteria

1. In-source performance specifications (DWARF)

2. Toggle without recompilation (Dynamic BI)

3. "Accurate" (NOOP, Look-ahead)

• Maximum number of allocations in a scope

• Plugin-style API

```
unsigned* MaxAlloc_start(const PCA* pca) {
2
        unsigned* total_calls = new(0);
3
        pca->on_function("malloc",
                           [](unsigned* i) {
4
5
                               *i += 1:
6
                          }.
                          total_calls);
8
        return total_calls;
9
10
11
    void MaxAlloc_end(const PCA* pca,
12
                       unsigned* total_calls) {
13
        unsigned max_calls = pca->args()[0];
14
        if (!(*total_calls <= max_calls))</pre>
15
            pca->log_failure(*total_calls,
16
                               max calls):
17
        pca->clear_on_function("malloc");
18
        delete total_calls:
19
20
21
    void MaxAlloc_inject(const PCA* pca) {
22
        pca->at_start(MaxAlloc_start);
23
        pca->at_end(MaxAlloc_end);
24
25
26
   PCA_CLAIM({"MaxAlloc", MaxAlloc_inject});
```

• Maximum number of allocations in a scope

• Plugin-style API

```
unsigned* MaxAlloc_start(const PCA* pca) {
2
        unsigned* total_calls = new(0);
3
        pca->on_function("malloc",
                           [](unsigned* i) {
4
5
                               *i += 1:
6
                          }.
                          total_calls);
8
        return total calls:
9
10
11
    void MaxAlloc_end(const PCA* pca,
12
                       unsigned* total_calls) {
13
        unsigned max_calls = pca->args()[0];
14
        if (!(*total_calls <= max_calls))</pre>
15
            pca->log_failure(*total_calls,
16
                               max calls):
17
        pca->clear_on_function("malloc");
18
        delete total_calls:
19
20
21
    void MaxAlloc_inject(const PCA* pca) {
22
        pca->at_start(MaxAlloc_start);
23
        pca->at_end(MaxAlloc_end);
24
25
```

• PCA data and Pin accessed through PCA

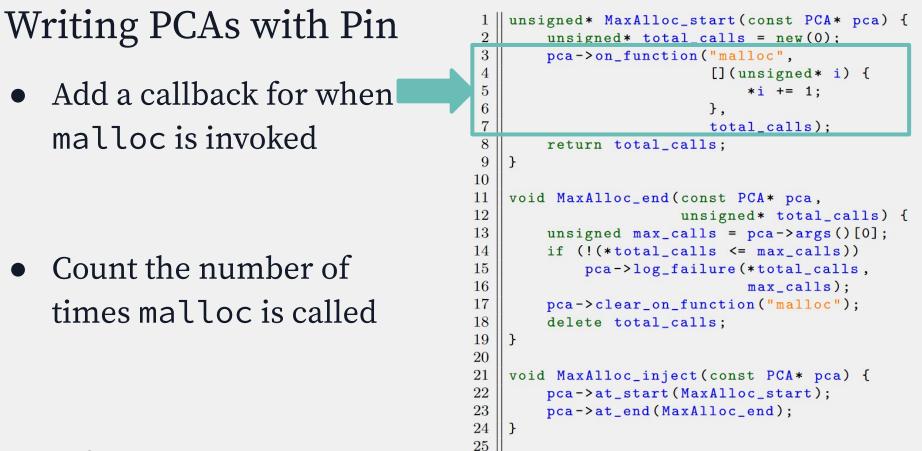
• Register hooks at start and end of PCA block

```
unsigned* MaxAlloc_start(const PCA* pca) {
 2
        unsigned* total_calls = new(0);
 3
        pca->on_function("malloc",
                           [](unsigned* i) {
 4
 5
                               *i += 1:
 6
                          }.
                          total_calls);
 8
        return total_calls;
9
10
11
    void MaxAlloc_end(const PCA* pca,
12
                       unsigned* total_calls) {
13
        unsigned max_calls = pca->args()[0];
14
        if (!(*total_calls <= max_calls))</pre>
15
            pca->log_failure(*total_calls,
16
                               max calls):
17
        pca->clear_on_function("malloc");
18
        delete total_calls:
19
20
21
    void MaxAlloc_inject(const PCA* pca) {
22
        pca->at_start(MaxAlloc_start);
23
        pca->at_end(MaxAlloc_end);
24
25
   PCA_CLAIM({"MaxAlloc", MaxAlloc_inject});
```

• Add a callback for when malloc is invoked

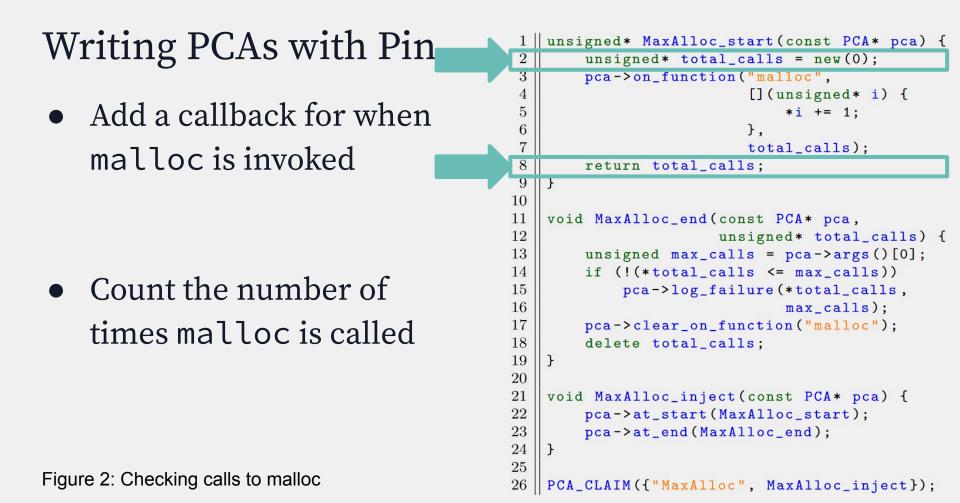
• Count the number of times malloc is called

```
unsigned* MaxAlloc_start(const PCA* pca) {
2
        unsigned* total_calls = new(0);
3
        pca->on_function("malloc",
4
                           [](unsigned* i) {
5
                               *i += 1:
6
                          }.
 7
                          total_calls);
8
        return total calls:
9
10
11
    void MaxAlloc_end(const PCA* pca,
12
                       unsigned* total_calls) {
13
        unsigned max_calls = pca->args()[0];
14
        if (!(*total_calls <= max_calls))</pre>
15
            pca->log_failure(*total_calls,
16
                               max calls):
17
        pca->clear_on_function("malloc");
18
        delete total_calls:
19
20
21
    void MaxAlloc_inject(const PCA* pca) {
22
        pca->at_start(MaxAlloc_start);
23
        pca->at_end(MaxAlloc_end);
24
25
26
   PCA_CLAIM({"MaxAlloc", MaxAlloc_inject});
```



26

PCA_CLAIM({"MaxAlloc", MaxAlloc_inject});



• Retrieve argument(s)

• Check the PCA

• Cleanup

```
unsigned* MaxAlloc_start(const PCA* pca) {
2
        unsigned* total_calls = new(0);
3
        pca->on_function("malloc",
                           [](unsigned* i) {
4
5
                               *i += 1:
6
                          }.
 7
                          total_calls);
8
        return total calls:
9
10
    void MaxAlloc_end(const PCA* pca,
11
12
                       unsigned* total_calls) {
13
        unsigned max_calls = pca->args()[0];
14
        if (!(*total_calls <= max_calls))</pre>
15
            pca->log_failure(*total_calls,
                               max calls):
16
17
        pca->clear_on_function("malloc");
18
        delete total_calls:
19
20
21
    void MaxAlloc_inject(const PCA* pca) {
22
        pca->at_start(MaxAlloc_start);
23
        pca->at_end(MaxAlloc_end);
24
25
26
   PCA_CLAIM({"MaxAlloc", MaxAlloc_inject});
```

• Retrieve argument(s)

• Check the PCA

• Cleanup

```
unsigned* MaxAlloc_start(const PCA* pca) {
2
        unsigned* total_calls = new(0);
3
        pca->on_function("malloc",
4
                           [](unsigned* i) {
5
                               *i += 1:
6
                          }.
7
                          total_calls);
8
        return total calls:
9
10
11
    void MaxAlloc_end(const PCA* pca,
12
                       unsigned* total_calls) {
13
        unsigned max_calls = pca->args()[0];
        if (!(*total_calls <= max_calls))</pre>
14
15
            pca->log_failure(*total_calls,
16
                               max calls):
17
        pca->clear_on_function("malloc");
18
        delete total_calls:
19
20
21
    void MaxAlloc_inject(const PCA* pca) {
22
        pca->at_start(MaxAlloc_start);
23
        pca->at_end(MaxAlloc_end);
24
25
26
   PCA_CLAIM({"MaxAlloc", MaxAlloc_inject});
```

• Retrieve argument(s)

• Check the PCA

• Cleanup

```
unsigned* MaxAlloc_start(const PCA* pca) {
2
        unsigned* total_calls = new(0);
3
        pca->on_function("malloc",
4
                           [](unsigned* i) {
5
                               *i += 1:
6
                          }.
7
                          total_calls);
8
        return total calls:
9
10
11
    void MaxAlloc_end(const PCA* pca,
12
                       unsigned* total_calls) {
        unsigned max_calls = pca->args()[0];
13
14
        if (!(*total_calls <= max_calls))</pre>
15
            pca->log_failure(*total_calls,
16
                               max calls):
17
        pca->clear_on_function("malloc");
18
        delete total_calls:
19
20
21
    void MaxAlloc_inject(const PCA* pca) {
22
        pca->at_start(MaxAlloc_start);
23
        pca->at_end(MaxAlloc_end);
24
25
26
   PCA_CLAIM({"MaxAlloc", MaxAlloc_inject});
```

• Retrieve argument(s)

• Check the PCA

• Cleanup

```
unsigned* MaxAlloc_start(const PCA* pca) {
2
        unsigned* total_calls = new(0);
3
        pca->on_function("malloc",
4
                           [](unsigned* i) {
5
                               *i += 1:
6
                          }.
 7
                          total_calls);
8
        return total calls:
9
10
11
    void MaxAlloc_end(const PCA* pca,
12
                       unsigned* total_calls) {
        unsigned max_calls = pca->args()[0];
13
14
        if (!(*total_calls <= max_calls))</pre>
15
            pca->log_failure(*total_calls,
                               max_calls);
16
        pca->clear_on_function("malloc");
17
18
        delete total calls:
19
20
21
    void MaxAlloc_inject(const PCA* pca) {
22
        pca->at_start(MaxAlloc_start);
23
        pca->at_end(MaxAlloc_end);
24
25
26
   PCA_CLAIM({"MaxAlloc", MaxAlloc_inject});
```

All Together

\$ gcc -g -O3 main.c -o exec

\$ read_pcas ./exec ./pcas.txt

\$./pin -t pca.so -i ./pcas.txt -- ./exec

Summary - A simple mechanism to:

1. Specify performance requirements for functions which

may be difficult when testing

2. Assist in document assumptions callers can make about a

function's execution

3. Check annotations easily and dynamically

Questions and Future Work

1. Can programmers easily integrate it into their workflow?

 Where is this more general system applicable?
 Everywhere? Server software? Or is it only a minor upgrade for embedded devices?

Thank You

Zachery Casey & Michael Shah

casey.z@northeastern.edu mikeshah@northeastern.edu