Combating Run-time Performance Bugs with Performance Claim Annotations

November 12, 2020

Zachery Casey & Michael Shah

casey.z@northeastern.edu mikes Shah@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)
Our Criteria

1. In-source performance specifications

2. Toggle without recompilation

3. “Accurate”
Related Work - Mobile Performance Assertions

- \( \text{pa}_\text{start}(id) \rightarrow \text{pa}_\text{end}(id, \text{assertion}) \)

- Inter Process Communication (IPC) backend

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

- Simple, motivating example...

```c++
// 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)
        xs.push_back(ys[i]);
}
```

Figure 1: PCA on unnecessary allocation
Performance Claim Annotation

- Clear documentation
- Clear type signature

Figure 1: PCA on unnecessary allocation

```c
// 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)
        xs.push_back(ys[i]);
}
```
Performance Claim Annotation

- Straightforward implementation

```cpp
// 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)
        xs.push_back(ys[i]);
}
```

Figure 1: PCA on unnecessary allocation
Performance Claim Annotation

- Implicit requirement in documentation

- Extra requirement on type

```c
// 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)
        xs.push_back(ys[i]);
}
```

Figure 1: PCA on unnecessary allocation
Performance Claim Annotation

- Maximum number of allocations in a scope
- Implementation should have $\leq 1$ allocations

```cpp
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)
    xs.push_back(ys[i]);
}
```

Figure 1: PCA on unnecessary allocation
Performance Claim Annotation

- Maximum number of allocations in a scope
- Implementation should have $\leq 1$ allocations

```cpp
// 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();

    for (unsigned i = 0; i < ys_len; ++i)
        xs.push_back(ys[i]);
```

Figure 1: PCA on unnecessary allocation
Performance Claim Annotation

- Maximum number of allocations in a scope
  - Implementation should have $\leq 1$ allocations

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

Figure 1: PCA on unnecessary allocation
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)
Using DWARF

PCA(MaxAlloc, PCA_INT 1); NOOP
+ __pca_MaxAlloc_int_1
Line 8, PC: 0x1150
Scope: Line 3-15
0x1146-0x1196

char __pca_MaxAlloc_int_1;
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
Our Criteria

1. In-source performance specifications (DWARF)

2. Toggle without recompilation (Dynamic BI)

3. “Accurate” (NOOP, Look-ahead)
Writing PCAs with Pin

- Maximum number of allocations in a scope

- Plugin-style API

```c
unsigned* MaxAlloc_start(const PCA* pca) {
    unsigned* total_calls = new(0);
    pca->on_function("malloc",
        [](unsigned* i) {
            *i += 1;
            total_calls);
    return total_calls;
}

void MaxAlloc_end(const PCA* pca,
        unsigned* total_calls) {
    unsigned max_calls = pca->args()[0];
    if (!(total_calls <= max_calls))
        pca->log_failure(*total_calls,
            max_calls);
    pca->clear_on_function("malloc");
    delete total_calls;
}

void MaxAlloc_inject(const PCA* pca) {
    pca->at_start(MaxAlloc_start);
    pca->at_end(MaxAlloc_end);
}

PCA_CLAIM("MaxAlloc", MaxAlloc_inject);
```
Writing PCAs with Pin

- Maximum number of allocations in a scope
- Plugin-style API

```c
unsigned* MaxAlloc_start(const PCA* pca) {
    unsigned* total_calls = new(0);
    pca->on_function("malloc",
        [](unsigned* i) {
            *i += 1;
            total_calls);
    return total_calls;
}

void MaxAlloc_end(const PCA* pca, 
    unsigned* total_calls) {
    unsigned max_calls = pca->args()[0];
    if (!(*total_calls <= max_calls))
        pca->log_failure(*total_calls, 
            max_calls);
    pca->clear_on_function("malloc");
delete total_calls;
}

void MaxAlloc_inject(const PCA* pca) {
    pca->at_start(MaxAlloc_start);
    pca->at_end(MaxAlloc_end);
}

PCA_CLAIM({"MaxAlloc", MaxAlloc_inject});
```
Writing PCAs with Pin

- PCA data and Pin accessed through PCA
- Register hooks at start and end of PCA block

```c
unsigned* MaxAlloc_start(const PCA* pca) {
    unsigned* total_calls = new(0);
    pca->on_function("malloc",
        [](unsigned* i) {
            *i += 1;
            total_calls);
    return total_calls;
}

void MaxAlloc_end(const PCA* pca,
    unsigned* total_calls) {
    unsigned max_calls = pca->args()[0];
    if (!(*total_calls <= max_calls))
        pca->log_failure(*total_calls,
            max_calls);
    pca->clear_on_function("malloc");
delete total_calls;
}

void MaxAlloc_inject(const PCA* pca) {
    pca->at_start(MaxAlloc_start);
    pca->at_end(MaxAlloc_end);
}
```

Figure 2: Checking calls to malloc
Writing PCAs with Pin

- Add a callback for when `malloc` is invoked
- Count the number of times `malloc` is called

```c
unsigned* MaxAlloc_start(const PCA* pca) {
    unsigned* total_calls = new(0);
    pca->on_function("malloc",
        [](unsigned* i) {
            *i += 1;
        },
        total_calls);
    return total_calls;
}

void MaxAlloc_end(const PCA* pca, unsigned* total_calls) {
    unsigned max_calls = pca->args()[0];
    if (!(*total_calls <= max_calls))
        pca->log_failure(*total_calls,
            max_calls);
    pca->clear_on_function("malloc");
    delete total_calls;
}

void MaxAlloc_inject(const PCA* pca) {
    pca->at_start(MaxAlloc_start);
    pca->at_end(MaxAlloc_end);
}

PCA_CLAIM({"MaxAlloc", MaxAlloc_inject});
```

Figure 2: Checking calls to `malloc`
Writing PCAs with Pin

- Add a callback for when malloc is invoked
- Count the number of times malloc is called

```c
unsigned* MaxAlloc_start(const PCA* pca) {
    unsigned* total_calls = new(0);
    pca->on_function("malloc",
        []((unsigned* i) {
            *i += 1;
        },
        total_calls);
    return total_calls;
}

void MaxAlloc_end(const PCA* pca,
    unsigned* total_calls) {
    unsigned max_calls = pca->args()[0];
    if (*total_calls <= max_calls)
        pca->log_failure(*total_calls,
            max_calls);
    pca->clear_on_function("malloc");
    delete total_calls;
}

void MaxAlloc_inject(const PCA* pca) {
    pca->at_start(MaxAlloc_start);
    pca->at_end(MaxAlloc_end);
}

PCACLAIM({"MaxAlloc", MaxAlloc_inject});

Figure 2: Checking calls to malloc
Writing PCAs with Pin

- Add a callback for when malloc is invoked
- Count the number of times malloc is called

```c
unsigned* MaxAlloc_start(const PCA* pca) {
    unsigned* total_calls = new(0);
    pca->on_function("malloc",
                    [](unsigned* i) {
                        *i += 1;
                    },
                    total_calls);
    return total_calls;
}

void MaxAlloc_end(const PCA* pca,
                   unsigned* total_calls) {
    unsigned max_calls = pca->args()[0];
    if (!(*total_calls <= max_calls))
        pca->log_failure(*total_calls, max_calls);
    pca->clear_on_function("malloc");
    delete total_calls;
}

void MaxAlloc_inject(const PCA* pca) {
    pca->at_start(MaxAlloc_start);
    pca->at_end(MaxAlloc_end);
}
```

Figure 2: Checking calls to malloc
Writing PCAs with Pin

- Retrieve argument(s)
- Check the PCA
- Cleanup

Figure 2: Checking calls to malloc

```c
unsigned* MaxAlloc_start(const PCA* pca) {
    unsigned* total_calls = new(0);
    pca->on_function("malloc",
        [](unsigned* i) {
            *i += 1;
        },
        total_calls);
    return total_calls;
}

void MaxAlloc_end(const PCA* pca,
    unsigned* total_calls) {
    unsigned max_calls = pca->args()[0];
    if (!(total_calls <= max_calls))
        pca->log_failure(*total_calls, max_calls);
    pca->clear_on_function("malloc");
    delete total_calls;
}

void MaxAlloc_inject(const PCA* pca) {
    pca->at_start(MaxAlloc_start);
    pca->at_end(MaxAlloc_end);
}

PCA_CLAIM({"MaxAlloc", MaxAlloc_inject});
```
Writing PCAs with Pin

- Retrieve argument(s)
- Check the PCA
- Cleanup

Figure 2: Checking calls to malloc
Writing PCAs with Pin

- Retrieve argument(s)
- Check the PCA
- Cleanup

Figure 2: Checking calls to malloc

```c
unsigned* MaxAlloc_start(const PCA* pca) {
    unsigned* total_calls = new(0);
    pca->on_function("malloc",
        [](unsigned* i) {
            *i += 1;
        },
    total_calls);
    return total_calls;
}

void MaxAlloc_end(const PCA* pca,
    unsigned* total_calls) {
    unsigned max_calls = pca->args()[0];
    if (!(total_calls <= max_calls))
        pca->log_failure(*total_calls,
            max_calls);
    pca->clear_on_function("malloc");
    delete total_calls;
}

void MaxAlloc_inject(const PCA* pca) {
    pca->at_start(MaxAlloc_start);
    pca->at_end(MaxAlloc_end);
}

PCA_CLAIM({"MaxAlloc", MaxAlloc_inject});
```
Writing PCAs with Pin

- Retrieve argument(s)
- Check the PCA
- Cleanup

Figure 2: Checking calls to malloc

```c
unsigned* MaxAlloc_start(const PCA* pca) {
    unsigned* total_calls = new(0);
    pca->on_function("malloc",
        [](unsigned* i) {
            *i += 1;
        },
        total_calls);
    return total_calls;
}

void MaxAlloc_end(const PCA* pca,
    unsigned* total_calls) {
    unsigned max_calls = pca->args()[0];
    if (!(*total_calls <= max_calls))
        pca->log_failure(*total_calls,
            max_calls);

    pca->clear_on_function("malloc");
    delete_total_calls;
}

void MaxAlloc_inject(const PCA* pca) {
    pca->at_start(MaxAlloc_start);
    pca->at_end(MaxAlloc_end);
}

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?

2. 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 mikes Shah@northeastern.edu