Combating Run-time Performance Bugs with Performance Claim Annotations

Zachery Casey, Michael D. Shah
{casey.z, mikeshah}@northeastern.edu
Northeastern University, Boston, MA

Abstract

Bugs in software are classified by a failure to meet some aspect of a specification. A piece of code which does not match the performance given by a specification contains a performance bug. We believe there is a need for better in-source language support and tools to assist a developer in mitigating and documenting performance bugs during the software development life cycle.

In this paper, we present our *performance claim annotation* framework for specifying and monitoring the performance of a program. A performance claim annotation (PCA) is written by a programmer to assert a section of code's run-time execution coincides with a specific metric (e.g. time elapsed) and they want to perform some action, typically logging, if the code fails to match the metric during execution. Our implementation uses a combination of the DWARF debugging format [4] and the Pin dynamic binary instrumentation tool [2] to provide an interface for building, using, and checking performance claims in order to reduce performance bugs during the development life cycle.

1 Introduction

When writing software, there are often minimum requirements for its performance. For example, a game must consistently render 30 frames per second or a server needs to reliably handle some number of requests per second. If an application fails to meet the respective benchmarks, a programmer may rely on a profiler such as perf or VTune to identify *hotspots* in the code [6, 7]. The programmer can then tune code until the application runs fast enough to meet the given specification. This ad-hoc style of performance tuning is often not documented, and the measurements may have to be performed again by multiple developers during a project's life cycle.

To combat certain classes of program logic bugs, programmers will often rely on assertions to ensure parts of their program *function* a certain way. Likewise, we want to be able to assert whether certain parts of the program *perform* a certain way. If the code fails to keep these assertions about performance, we want all available information about it to be logged for future review. In previous work on this concept, performance assertions are often defined external to the source program [1, 5]. Rather than externally applying constraints at some point in the future, we envision a workflow where performance claims are integrated into the design of functions in a fashion similar to regular assertions about program logic: the programmer will determine the contract of a function and then write the appropriate documentation, type signature, assertions, and annotations. Listing 1 provides an example of the result of this process. The PCA on line 8 of Listing 1 serves a similar purpose to the assertion: it assists the programmer in debugging if they forget to add a call to `reserve` and provides another refinement to the function's documentation.

```
 1 // Copy integers from pointer-array
 2 //                 into cleared vector.
 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)
14         xs.push_back(ys[i]);
15 }
```

Listing 1: A performance claim annotation about unnecessary allocation.

In this paper we present our implementation for performance claim annotations (PCAs), a tool for software developers to precisely define the performance characteristics of the software they write in C/C++.

2 Implementation

In order to create performance claim annotations, we established four criteria which should be met:

1. Declare claims in-source.
2. Easily write or extend complex claims about execution.
3. Disable monitoring without need to recompile.
4. Be unaffected by changes in optimization level.

Criteria 2, 3, and 4 eliminate the possibility of implementing PCAs as a library in C/C++. Most notably, if the programmer needs to place one or more calls to an external library into functions, there is the possibility of affecting the process of function inlining; furthermore, it would be non-trivial to avoid recompilation to entirely disable monitoring. However, without non-portable extensions to the compiler, it seems difficult to inject the necessary data collection and checking.

2.1 Overview

To implement PCAs, we rely on DWARF debugging symbols which can be enabled when invoking any mainstream C/C++ compiler. Regardless of optimization level, debugging symbols record information about both the source and the resulting binary. Although they are typically used by debuggers, we are able to store data about annotations in them. We then rely on a dynamic binary instrumentation tool, in our case Pin, to do the work of monitoring and checking claims instead of injecting instrumentation code earlier in the compilation pipeline.

By utilizing both of these tools, we achieve our four implementation criteria. In-source declarations are created using pre-processor macros in C/C++, and the process is portable to any compiler with support for identifier mangling and outputting DWARF symbols (As an example we wrote a proof of concept for Rust programming language). Since the implementation of claims is written using Pin, claims have full access to instruction level instrumentation and the various other tooling Pin provides. And, finally, the only effect on the binary is the addition of debugging symbols: to disable monitoring, the executable is not run through Pin and, likewise, optimization during compilation remains entirely unaffected.

2.2 Working Example

Returning to the code in Listing 1, we can imagine a version of the function which does not call `reserve` on the vector. Depending on the size of the pointer-array `ys` and the initial capacity of `xs`, we may see any number of allocations. We expect to catch this failure with our `MaxAlloc` PCA.

When compiled, the unhygienic `PCA` macro expands into a mangled, unused identifier representing the type of claim and the arguments passed to it. Currently, the encoding we use supports integers, as seen on line 8 of Listing 1, along with strings and floating-point arguments. In the resulting binary, the unused identifier generated by the macro is stored as debugging information, but has no effect on execution as the compiler can safely remove it.

After compilation, the binary can be used as usual, since the only change is the addition of debugging symbols. However, before we are able to run the program and check the PCAs, it must be passed to a small external utility we have written which scans all DWARF symbols looking for the encoded PCA identifiers. Once one is found, it is decoded and written to a plain-text file along with relevant contextual information which will be needed in Pin in order to instrument the executable. Currently, the contextual information only contains the start and end of the scope surrounding the PCA, this is a detail we will return to in Section 3.1.

```
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});
```

Listing 2: A claim to check the number of calls to `malloc`.

Finally, in order to run the program and check the annotations, the binary must be passed to Pin at the command-line along with the PCA Pintool and the plain-text file containing the relevant annotations. The PCA Pintool is a shared object containing basic startup code as well as the interface for claim definitions. A new claim can be created by implementing the interface and registering it under a name which will be used by programmers to reference it. Listing 2 provides an example of using this interface; but, as the implementation relies on details specific to Pin, it is partially abridged with helper methods for the purpose of exposition.

2.3 Checking Claims

When the program is run using Pin and a scope containing a `MaxAlloc` PCA is executed for the first time, the `MaxAlloc_inject` function will be run. The `PCA` object passed to it contains the details of the annotation and provides access to Pin features. Here, `MaxAlloc_inject` uses Pin to add the hooks `MaxAlloc_start` and `MaxAlloc_end` which will be run whenever execution enters and exits the scope surrounding the annotation, respectively. In our particular example, MaxAlloc_start will be run at the beginning of copy_into and MaxAlloc_end is run before the final return instruction of copy_into.

MaxAlloc_start allocates an integer counter and requests to be notified every time malloc is called so the total_calls can be incremented. As the function finishes, total_calls is passed to MaxAlloc_end. Since we want the number of calls to malloc to be less than the first argument in the annotation, we first retrieve this argument from our PCA object. We can then check if we have called malloc more times than we expected. For copy_into, if total_calls is greater than 1, we want to log the failure and actual number of allocations done. Furthermore, we must cleanup both the hook on malloc and the total_calls pointer.

3 Discussion

Initial feedback we received to the notion of PCAs suggested unit testing as an alternative and effective way to capture performance characteristics. However, unit tests may not capture a true execution of a program. Real-time applications, such as databases or games, will receive many varied inputs which cannot be fully captured in unit testing. Furthermore, unit tests serve as poor documentation when trying to localize where performance matters in a real time system. PCAs provide a simple, well-defined method for programmers to convey and verify the performance characteristics of their code without a lot of testing work.

3.1 Limitations and Related Work

As hinted previously, a current limitation of our tool is relying on the containing scope of a PCA to determine its beginning and end. Although in most cases this is a reasonable assumption, the previous system most closely resembling ours did not have this restriction [3]. The performance assertions infrastructure described by Lancevicius and Metz contains a number of features we believe are specific to the mobile devices they were developing, such as assertions which begin in one processes and end in an entirely different process.

However, much of the general functionality can be replicated or usurped with minor extensions to our tool. Specifically, to support non-lexical PCAs, we can extend our system to have similar PCA_START and PCA_END calls; when we scan the binary's DWARF symbols, rather than output the start and end of the surrounding scope, we are able to simply output the specific location of those calls. Furthermore, because we rely on Pin for dynamic binary instrumentation, our tool has access to fine grain control of exactly what we wish to monitor and can naturally inspect every part of the run-time. This contrasts with relying on a distinct process to monitor only the more general aspects of system and only be queried as needed.

3.2 Conclusion and Future Work

In the future, we will continue adding support for various extensions, such as non-lexical PCAs. As expected, implementing more advanced claims using Pin can become troublesome, as its generality and power becomes a hindrance; so, as we continue developing claims, we hope to slowly build a small utility library in order to have Listing 2 closely align with the reality of building or extending claims.

Since our tool is active, it is meant to be used during development, rather than reactive, being applied once a problem is found, we hope to find places to utilize it during real-world development. Unlike most instrumentation, benchmarks are not particularly interesting, since they match exactly to using Pin instrumentation to monitor various metrics outside of our tool. By taking the workflow we have described, we want to continue to explore how useful it is to programmers, specifically outside of embedded and mobile devices where previous works show performance assertions are effective.

References

[1] S. E. Perl and W. E. Weihl. "Performance Assertion Checking". In: *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*. SOSP '93. Asheville, North Carolina, USA: ACM, 1993, pp. 134–145.

[2] K. Hazelwood and A. Klauser. "A Dynamic Binary Instrumentation Engine for the ARM Architecture". In: *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*. CASES '06. Seoul, Korea: Association for Computing Machinery, 2006, pp. 261–270.

[3] R. Lencevicius and E. Metz. "Performance Assertions for Mobile Devices". In: *Proceedings of the 2006 International Symposium on Software Testing and Analysis*. ISSTA '06. Portland, Maine, USA: ACM, 2006, pp. 225–232.

[4] *DWARF Debugging Standard*. DWARF Debugging Information Format Committee. Feb. 2017.

[5] D. Rogora et al. "Analyzing System Performance with Probabilistic Performance Annotations". In: *Proceedings of the Fifteenth European Conference on Computer Systems*. EuroSys '20. Heraklion, Greece: Association for Computing Machinery, 2020.

[6] *Intel vTune Amplifier*. Online. https://software.intel.com/en-us/vtune. Intel Software.

[7] *Perf wiki*. Online. https://perf.wiki.kernel.org/index.php/Main_Page.