A hybrid approach to application instrumentation

Ashay Rane, Leo Fialho and James Browne

4th August, 2014
Petascale Tools Workshop

Program Instrumentation

What is instrumentation?

Addition of statements to user code for measuring execution behavior.

Used for measuring runtime code behavior, e.g.:

- Performance monitoring (e.g. function invocation count).
- Logging / recording events (e.g. recording memory access trace).
- Enforcing specific behaviors (e.g. preventing out-of-bounds accesses).

Our goals

- Make program instrumentation easy.
- Not sacrifice accuracy in making instrumentation easy.
- Not force the instrumentation-developer to reinvent the wheel.

Democratize accurate instrumentation

Compiler-based instrumentation

Source code → Intermediate rep. → Assembly → Executable (binary).

- Applies instrumentation at the IR level.
- Allows using sophisticated compiler APIs (e.g. alias analysis).
- However, compiler optimizations are impacted.
- Instrumentation may produce incorrect performance measurements.

Binary instrumentation

Source code → Intermediate rep. → Assembly → Executable (binary).

- Applies instrumentation to machine (x86) instructions.
- Is accurate because measurements are based on executing instructions.

  Unlike compliation, code is not translated any further.
- However, requires knowledge of machine instruction semantics.
- Code analysis is difficult, sometimes even impossible.

Motivation

- Accurate measurements possible only using binary instrumentation.
- But binary instrumentation is not easy.
- Binary instrumentation clients have to reconstruct information that was available (but thrown away) during compilation.

Can we have our cake and eat it too?

- Specify instrumentation at a higher level during compilation.
- Add instrumentation instructions after all optimizations are applied.

Motivating example

Wrong vectorization diagnosis

- Instrumentation code measured branch outcomes (true v/s false count).
- Without instrumentation, the branch was optimized away by the compiler.
- With instrumentation, the branch was retained in the loop.
- Instrumentation (incorrectly) concluded that loop was not vectorizable.

Motivating example

Wrong vectorization diagnosis

```
for (i = 0; i < 1024; i++) {
    if (a[i] < b[i]) {
        x = a[i];
    }
}
```

Without instrumentation, compiler can vectorize this branch with `VPCMOV` or `BLEND` instructions.

Motivating example

Wrong vectorization diagnosis

Solution:

1. Let compiler-level analysis find the branch to be instrumented.
2. Compiler nodes are “tagged” with instrumentation information.
3. Instrumentation added to the binary depending on associated tags.

What about the average Joe?

- Both compile-time instrumentation and binary instrumentation require non-trivial effort (writing compiler passes, DynInst client, PINTool, etc.).
- Certain instrumentations (e.g. generating address trace) are required for many different measurements.
- Can we make program instrumentation easier?

Instrumentation for the average Joe

Configuration file defines what and how to instrument.

```
instrumentation:
    - type: address-traces
      location: loop at foo.cc:565, function bar(int, int)
      output: trace-output.txt

    - type: invocation-counter
      location: function kernel(void)
      output: call-counts.txt
```

Workflow of hybrid instrumentation

1. Configuration file defines type, location of source-level instrumentation.
2. Compiler-level static analysis identifies what to instrument.
3. Instrumentation (as meta-level info) is associated with IR instructions.
4. Meta-level info is carried across compiler optimizations.
5. Meta-level info added to executable binary using assembly labels.
6. Flesh out these special assembly labels into instrumentation code.

Proof-of-concept implementation uses LLVM and DynInst.

Demo: Detecting false sharing

Step #1: Sample configuration file.

```
instrumentation:
    - type: false-sharing
      location: function kernel(int)
      output: thrashing-candidates.txt
```

Demo: Detecting false sharing

Step #2: Compiler-level static analysis.

- Find `store` instructions to arrays in multi-threaded function.
- These instructions represent potential false-sharing accesses and need to be instrumented.

Demo: Detecting false sharing

Step #3: Associate instrumentation information with IR instructions.

```
uint32_t addr_expr = mdfactory->set_effective_addr_md(store_inst);
uint32_t var_name = mdfactory->set_constant_md(store_inst, name_string);
/* Construct list of arguments for function call. */
std::vector params;
params.push_back(addr_expr);
params.push_back(var_name);
mdfactory->set_function_call_md(store_inst, "record_addr", params);
```

Demo: Detecting false sharing

Step #4: Propagate instrumentation information across optimizations.

Handled transparently by our modified LLVM compiler backend.

Demo: Detecting false sharing

Step #5: Insert specially-encoded labels that represent instrumentation.

`.GSYM.5.0`, `.GSYM.9.1.counts`, `.GSYM.10.2.record_addr.0.1`

- `.GSYM` : Special prefix.
- `10` : Instrumentation type (call).
- `2` : ID of label.
- `record_addr` : Function name.
- `0` , `1` : Function argument list (#0 ⇒ address, #1 ⇒ name).

Demo: Detecting false sharing

Step #6: Flesh out instrumentation code from label definitions.

- Implemented as a binary-rewriting tool in DynInst.
- Loops over labels, inserts `BPatch_snippet` objects at the label locations.
- Also performs basic type checking of label components.

Supported snippets

- `actual address`
- `original address`
- `breakpoint`
- `function call`
- `return`
- `thread index`
- `bytes accessed`
- `effective address`
- `parameter`
- `dynamic target`
- `arithmetic`
- `boolean`
- `constant`
- `sequence`
- `if statement`

Caveats

- Instrumentation is secondary to code optimizations. Hence optimization-induced errors in instrumentation should be handled by callee.
- Passing arbitrary pointer constants to instrumentation functions not supported because of separation of address spaces.
- Currently implemented for the “fast” instruction selection path. Changing generalized instruction selection code requires much more effort.

Summary

- Described an approach for accurate source-based instrumentation.
- Allows leveraging compiler APIs for static analysis, while not perturbing optimizations. Thus improves accuracy of instrumentation.
- Obviates knowledge of compiler APIs or machine instructions for common instrumentation kinds.
- Proof-of-concept implementation built using LLVM and DynInst.

Open questions and future work

- Can the workflow be implemented in a production compiler?
- Seeking suggestions and collaborations for extending this approach.
- Is this approach a logical next generation of Dyninst?