Optimizing Your Dyninst Program

Matthew LeGendre
legendre@cs.wisc.edu

Optimizing Dyninst

- Dyninst is being used to insert more instrumentation into bigger programs. For example:
  - Instrumenting every memory instruction
  - Working with binaries 200MB in size

- Performance is a big consideration

- What can we do, and what can you do to help?

Performance Problem Areas

- **Parsing**: Analyzing the binary and reading debug information.
- **Instrumenting**: Rewriting the binary to insert instrumentation.
- **Runtime**: Instrumentation slows down a mutatee at runtime.

Optimizing Dyninst

- Programmer Optimizations
  - Telling Dyninst not to output tramp guards.

- Dyninst Optimizations
  - Reducing the number of registers saved around instrumentation.

Parsing Overview

- Control Flow
  - Identifies executable regions
- Data Flow
  - Analyzes code prior to instrumentation
- Debug
  - Reads debugging information, e.g. line info
- Symbol
  - Reads from the symbol table

- Lazy Parsing: Not parsed until it is needed

Control Flow

- Dyninst needs to analyze a binary before it can instrument it.
  - Identifies functions and basic blocks

- Granularity
  - Parses all of a module at once.

- Triggers
  - Operating on a BPatch_function
  - Requesting BPatch_instPoint objects
  - Performing Stackwalks (on x86)

Data Flow

- Dyninst analyzes a function before instrumenting it.
  - Live register analysis
  - Reaching allocs on IA-64
- Granularity
  - Analyzes a function at a time.
- Triggers
  - The first time a function is instrumented

Debug Information

- Reads debug information from a binary.
- Granularity
  - Parses all of a module at once.
- Triggers
  - Line number information
  - Type information
  - Local variable information

Symbol Table

- Extracts function and data information from the symbol
- Granularity
  - Parses all of a module at once.
- Triggers
  - Not done lazily. At module load.

Lazy Parsing Overview

	Granularity	Triggered By
Control Flow	Module	BPatch_function Queries
Data Flow	Function	Instrumentation
Debug	Module	Debug Info Queries
Symbol	Module	Automatically

Lazy parsing allows you to avoid or defer costs.

Inserting Instrumentation

- What happens when we re-instrument a function?

```
foo:
0x1000: push ebp
0x1001: movl esp,ebp
0x1002: push $1
0x1004: call bar
0x1005: leave
0x1006: ret
```

```
foo:
0x4000: jmp entry_instr
0x4005: push ebp
0x4006: movl esp,ebp
0x4007: push $1
0x4009: jmp call_instr
0x400F: call bar
0x4014: leave
0x4015: jmp exit_instr
0x401A: ret
```

Inserting Instrumentation

- Bracket instrumentation requests with:

```
beginInsertionSet()
  .
  .
  .
endInsertionSet()
```

- Batches instrumentation
  - Allows for transactional instrumentation
  - Improves efficiency (rewrite)

Runtime Overhead

- Two factors determine how instrumentation slows down a program.
  - What does the instrumentation cost?
    - Increment a variable
    - Call a cache simulator
  - How often does instrumentation run?
    - Every time read/write are called
    - Every memory instruction
- Additional Dyninst overhead on each instrumentation point.

Runtime Overhead - Basetramps

A Basetramp

```
save all GPR
save all FPR
t = DYNINSTthreadIndex()
if (!guards[t]) {
  guards[t] = true
  jump to minitramps
  guards[t] = false
}
restore all FPR
restore all GPR
```

Save Registers

Calculate Thread Index

Check Tramp Guards

Run All Minitramps

Restore Registers

Runtime Overhead - Registers

A Basetramp

```
save all GPR
save all FPR
t = DYNINSTthreadIndex()
if (!guards[t]) {
  guards[t] = true
  jump to minitramps
  guards[t] = false
}
restore all FPR
restore all GPR
```

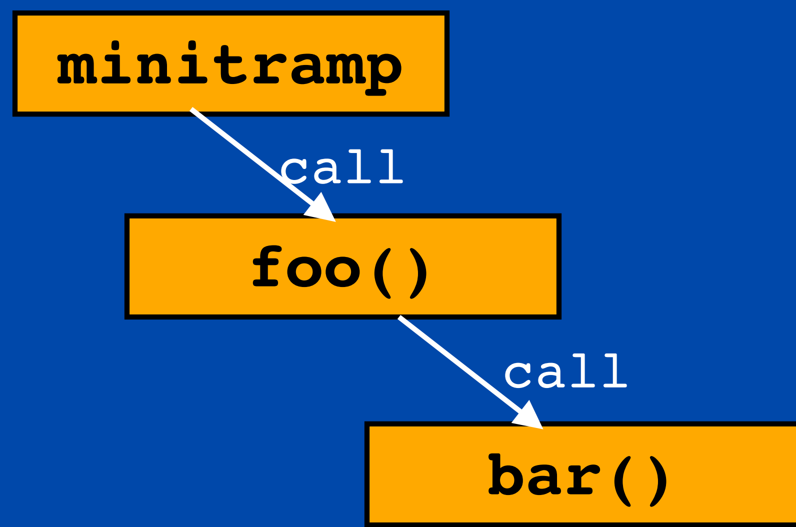
- Analyzes minitramps for register usage.
- Analyzes functions for register liveness.
- Only saves what is live and used.

Runtime Overhead - Registers

A Basetramp

```
save all GPR
save all FPR
t = DYNINSTthreadIndex()
if (!guards[t]) {
  guards[t] = true
  jump to minitramps
  guards[t] = false
}
restore all FPR
restore all GPR
```

- Called functions are recursively analyzed to a max call depth of 2.



Runtime Overhead - Registers

A Basetramp

```
save live GPR
t = DYNINSTthreadIndex()
if (!guards[t]) {
  guards[t] = true
  jump to minitramps
  guards[t] = false
}
restore live GPR
```

- Use shallow function call chains under instrumentation, so Dyninst can analyze all reachable code.
- Use `BPatch::setSaveFPR()` to disable all floating point saves.

Runtime Overhead - Tramp Guards

A Basetramp

```
save live GPR
t = DYNINSTthreadIndex()
if (!guards[t]) {
  guards[t] = true
  jump to minitramps
  guards[t] = false
}
Restore live GPR
```

- Prevents recursive instrumentation.
- Needs to be thread aware.

Runtime Overhead - Tramp Guards

A Basetramp

```
save live GPR
t = DYNINSTthreadIndex()

  jump to minitramps

restore live GPR
```

- Build instrumentation that doesn't make function calls (no `BPatch_funcCallExpr` snippets)
- Use `setTrampRecursive()` if you're sure instrumentation won't recurse.

Runtime Overhead - Threads

A Basetramp

```
save live GPR
t = DYNINSTthreadIndex()

  jump to minitramps

restore live GPR
```

- Returns an index value (0..N) unique to the current thread.
- Used by tramp guards and for thread local storage by instrumentation
- Expensive

Runtime Overhead - Threads

A Basetramp

```
save live GPR

  jump to minitramps

restore live GPR
```

- Not needed if there are no tramp guards.
- Only used on mutatees linked with a threading library (e.g. libpthread)

Runtime Overhead - Minitramps

A Basetramp

```
save live GPR

   jump to minitramps

restore live GPR
```

- Minitramps contain the actual instrumentation.
- What can we do with minitramps?

Runtime Overhead - Minitramps

Minitramp A

```
//Increment var by 1
load var -> reg
reg = reg + 1
store reg -> var
jmp Minitramp B
```

Minitramp B

```
//Call foo(arg1)
push arg1
call foo
jmp BaseTramp
```

- Created by our code generator, which assumes a RISC like architecture.
- Instrumentation linked by jumps.

Runtime Overhead - Minitramps

Minitramp A

```
//Increment var by 1

inc var

jmp Minitramp B
```

Minitramp B

```
//Call foo(arg1)
push arg1
call foo
jmp BaseTramp
```

- New code generator recognizes common instrumentation snippets and outputs CISC instructions.
- Works on simple arithmetic, and stores.

Runtime Overhead - Minitramps

Minitramp A / Mergetramp

```
//Increment var by 1

inc var

jmp Minitramp B
```

Minitramp B

```
//Call foo(arg1)
push arg1
call foo
jmp BaseTramp
```

- New merge tramps combine minitramps together with basetramp.
- Faster execution, slower re-instrumentation.
- Change behavior with `BPatch::setMergeTramp`

Runtime Overhead

- Where does the Dyninst's runtime overhead go?
  - 87% Calculating thread indexes
  - 12% Saving registers
  - 1% Trampoline Guards

- Dyninst allows inexpensive instrumentation to be inexpensive.

Summary

- Make use of lazy parsing
- Use insertion sets when inserting instrumentation.
- Small, easy to understand snippets are easier for Dyninst to optimize.
  - Try to avoid function calls in instrumentation.

Questions?