Dyninst

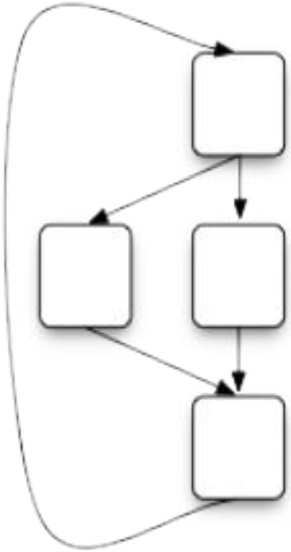
Scalable Tools Workshop

Granlibakken Resort
Lake Tahoe, California

WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON

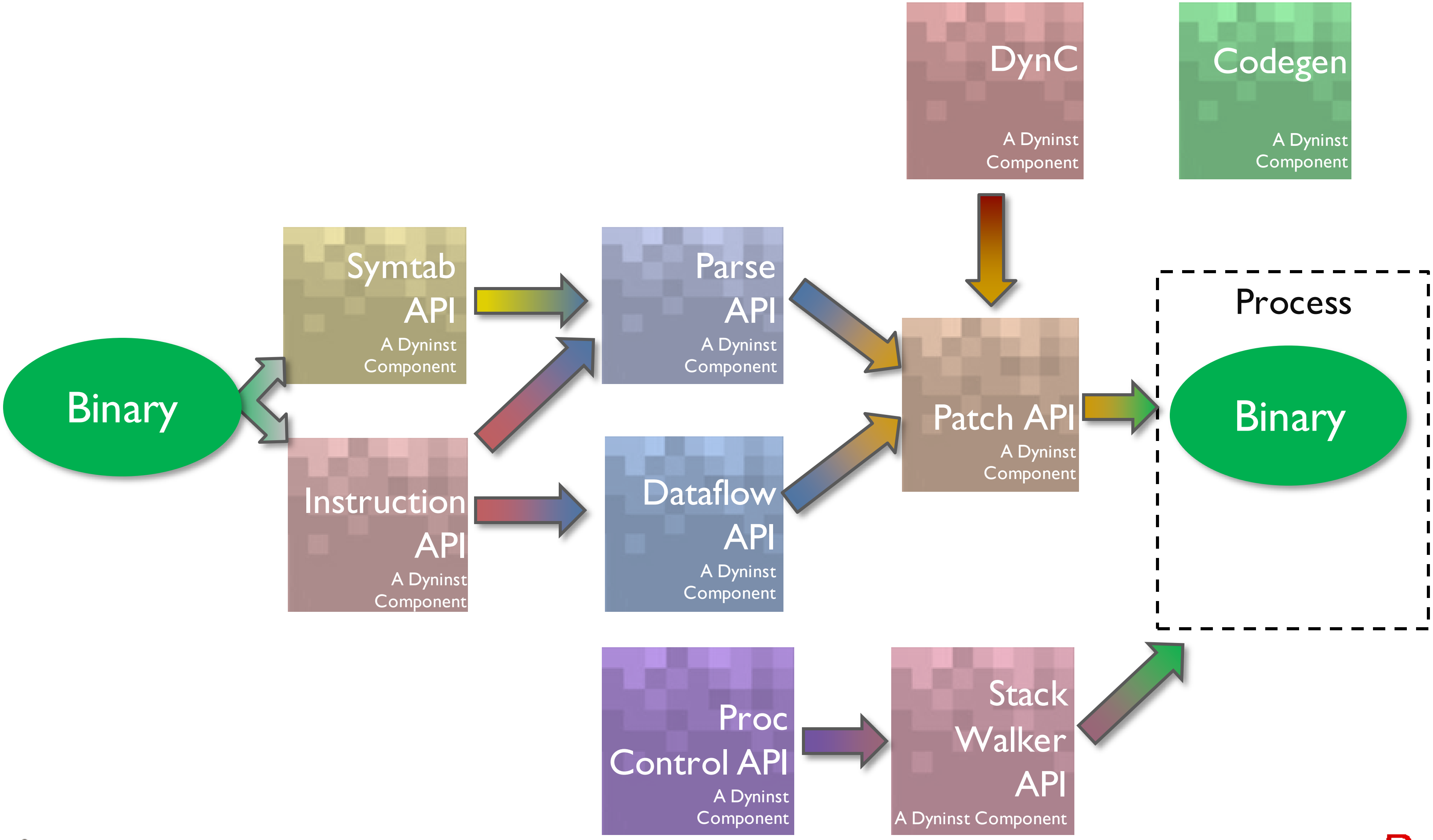
Paradyn™

A Brief Introduction to Dyninst



Dyninst: a tool for static and dynamic binary instrumentation and modification

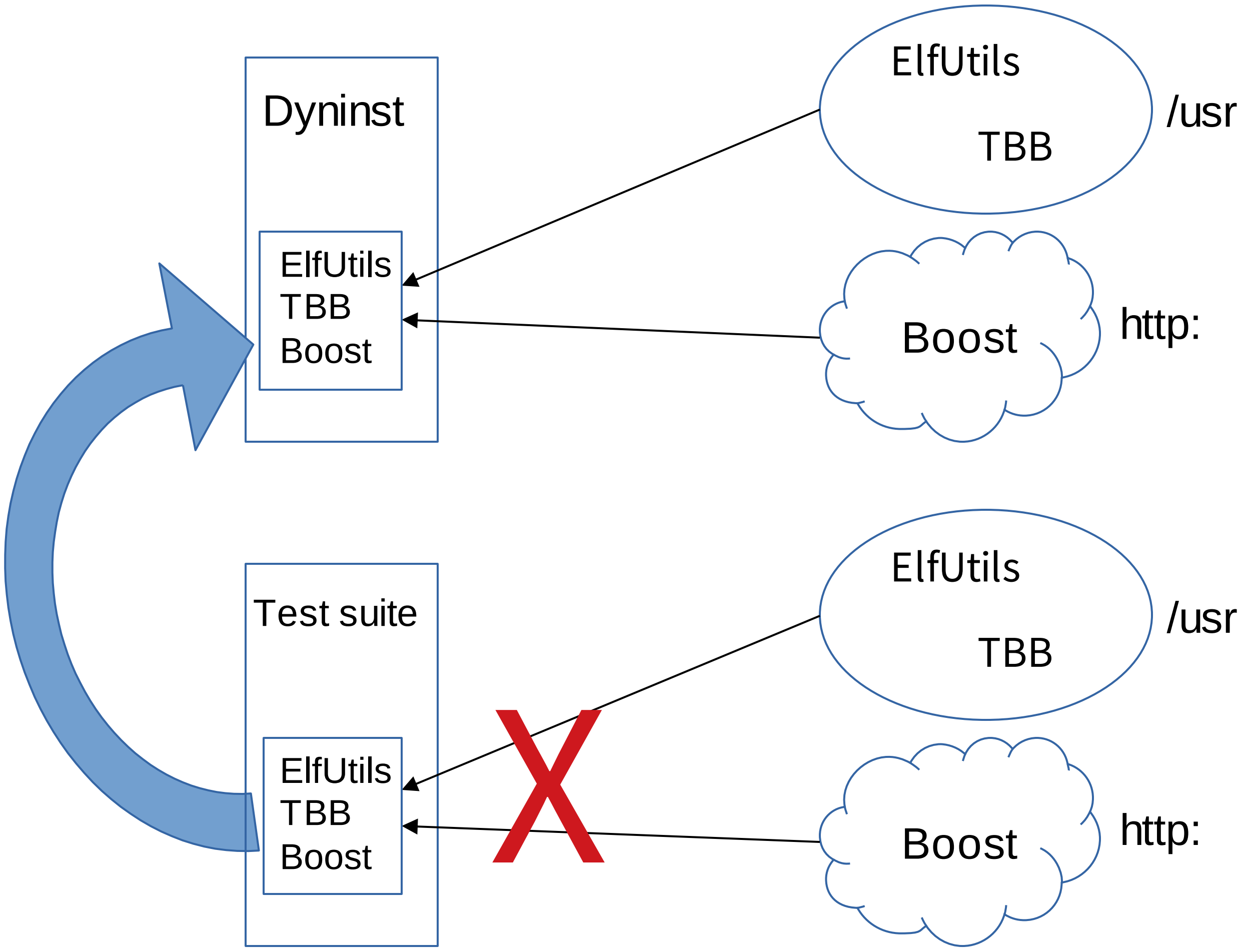
How is Dyninst organized?



What is new since July 2018?

- Changes to the build system
- Test suite enhancements to support Continuous Integration testing
- Parallel parsing
- ARMv8 advances

Changes to the Build System



Changes to the Build System

- All CMake variable names are now uniform across all dependencies
- Many new variables have been exposed for finer control of the build
- Export all CMake variables into cache
    - applications use them through CMake's `load_cache`

Caveat: rpath doesn't work fully (yet)

github.com/dyninst/dyninst/wiki

Getting the Test Suite Ready for CI

Dyninst is one of three applications testing the CI workflow for ECP

EXASCALE COMPUTING PROJECT

New build script for turnkey building and testing of Dyninst

```
git clone dyninst
git clone testsuite
perl build.pl
```

github.com/dyninst/testsuite

1. Build Dyninst
2. Build Test suite
3. Run Test suite
4. Upload results to our dashboard

Getting the Test Suite Ready for CI

What effect does this have on you?

Users: None

Developers: Github pull requests will be manually run through the new build script before acceptance

Future:

- All PRs will be automatically run through CI
  - this is 6+ months away
- Automatic nightly and weekly builds of the head of Master

Parallel Parsing

- Dyninst 10.0 added parallel code parsing
  - Uses function level parallelism in ParseAPI
  - Speedup relative to 1 thread is **~2-4x**
- Dyninst 10.1 fixed a few bugs
- Performance is still limited by serial code
  - GNU memory allocator
  - symbol table construction
  - parse frame initialization
  - parse finalization

Parallel Parsing

- Post 10.1 features under testing:
  - Optimized parallel ParseAPI
  - New parallel SymtabAPI
- Changes include:
  - Parallel symbol table construction
  - Parallel parsing frame initialization
  - Parallel finalization
  - Remove redundant calculation
  - Scalable allocator from TBB

Parallel Parsing

Performance compared to 10.1 version

Speedup

	min	max
10.1	baseline	3.4x
10.1 optimized	1.1x	6.1x

ARMv8 Advances

Code analysis: **complete** ✓

Dynamic instrumentation: **complete** ✓

Binary rewriting:

- Dynamically-linked code: **complete** ✓
- Statically-linked code: **in progress**

What can I do with Dyninst?

Function entry/exit tracing

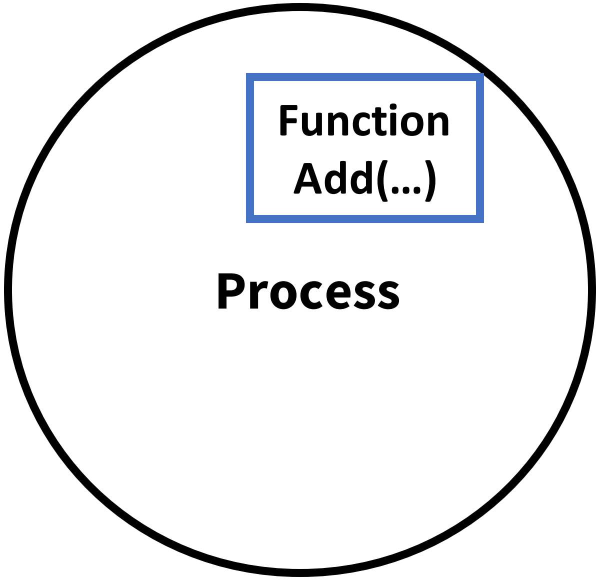
Stack walking

Code coverage

Function Entry/Exit Instrumentation

You have a process (or a binary) that contains a function and you want to collect information such as:

- How often it was called
- Time the function
- Get parameters supplied to the function



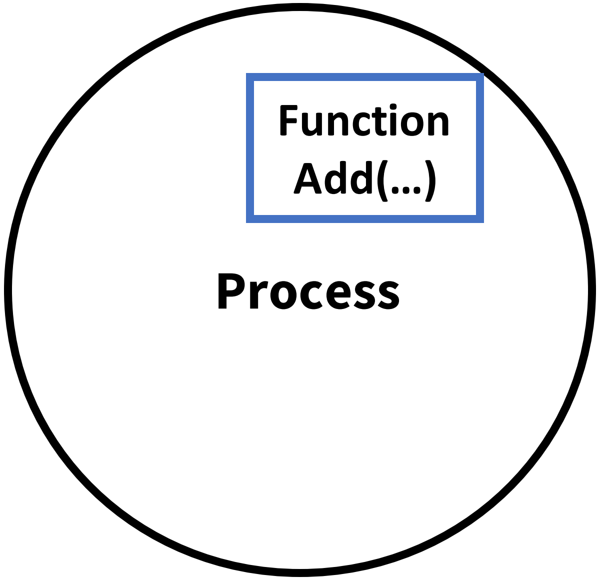
How would you do this?

You could modify the source code to include your instrumentation, recompile, and rerun

Function Entry/Exit Instrumentation

You have a process (or a binary) that contains a function and you want to collect information such as:

- How often it was called
- Time the function
- Get parameters supplied to the function



How would you do this?

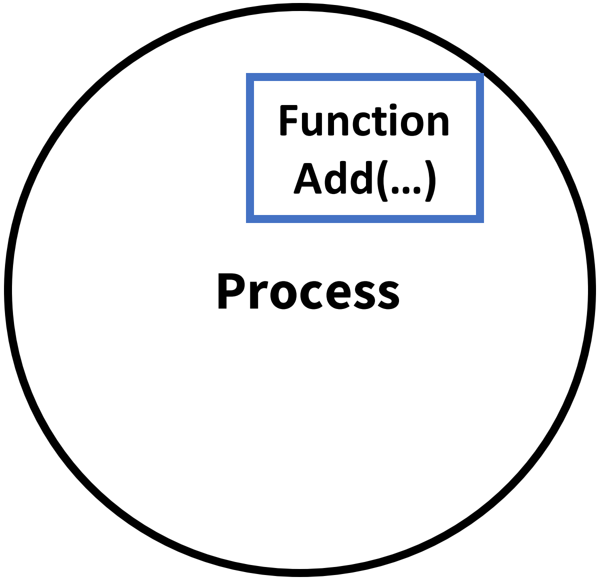
You could modify the source code to include your instrumentation, recompile, and rerun

Drawbacks: Maintain instrumentation w/ application source, source code may not be available (i.e. closed source), etc

Function Entry/Exit Instrumentation

You have a process (or a binary) that contains a function and you want to collect information such as:

- How often it was called
- Time the function
- Get parameters supplied to the function



How would you do this?

You could modify the source code to include your instrumentation, recompile, and rerun

> **Drawbacks**: Maintain instrumentation w/ application source, source code may not be available (i.e. closed source), etc.

You could avoid these problems by using binary instrumentation.

Function Entry/Exit Instrumentation

```
0000000000005fa <add>:

5fa: push    %rbp
5fb: mov     %rsp,%rbp
5fe: mov     %edi,-0x4(%rbp)
601: mov     %esi,-0x8(%rbp)
604: mov     -0x4(%rbp),%edx
607: mov     -0x8(%rbp),%eax
60a: add     %edx,%eax
60c: pop     %rbp

60d: retq
```

Show an example of how to use Dyninst to insert entry/exit instrumentation into a function.

- Function call to a tracing library at entry/exit.

Function Entry/Exit Instrumentation

```
00000000000005fa <add>:

5fa: push   %rbp
5fb: mov    %rsp,%rbp
5fe: mov    %edi,-0x4(%rbp)
601: mov    %esi,-0x8(%rbp)
604: mov    -0x4(%rbp),%edx
607: mov    -0x8(%rbp),%eax
60a: add    %edx,%eax
60c: pop    %rbp

60d: retq
```

1. Open the binary/attach to or create the process with the function you want to trace

```
addrSpace = bpatch.processCreate(…);
```

Function Entry/Exit Instrumentation

```
00000000000005fa <add>:

5fa: push   %rbp
5fb: mov    %rsp,%rbp
5fe: mov    %edi,-0x4(%rbp)
601: mov    %esi,-0x8(%rbp)
604: mov    -0x4(%rbp),%edx
607: mov    -0x8(%rbp),%eax
60a: add    %edx,%eax
60c: pop    %rbp

60d: retq
```

1. Open the binary/attach to or create the process with the function you want to trace

```
addrSpace = bpatch.processCreate(…);
```

2. Insert the tracing library (contains the function you want to call at entry/exit)

```
addrSpace->loadLibrary("libtrace.so");
```

Function Entry/Exit Instrumentation

```
00000000000005fa <add>:

5fa: push   %rbp
5fb: mov    %rsp,%rbp
5fe: mov    %edi,-0x4(%rbp)
601: mov    %esi,-0x8(%rbp)
604: mov    -0x4(%rbp),%edx
607: mov    -0x8(%rbp),%eax
60a: add    %edx,%eax
60c: pop    %rbp

60d: retq
```

libtrace.so

```
XXX <Trace>:

…: push   %rbp
     …
…: retq
```

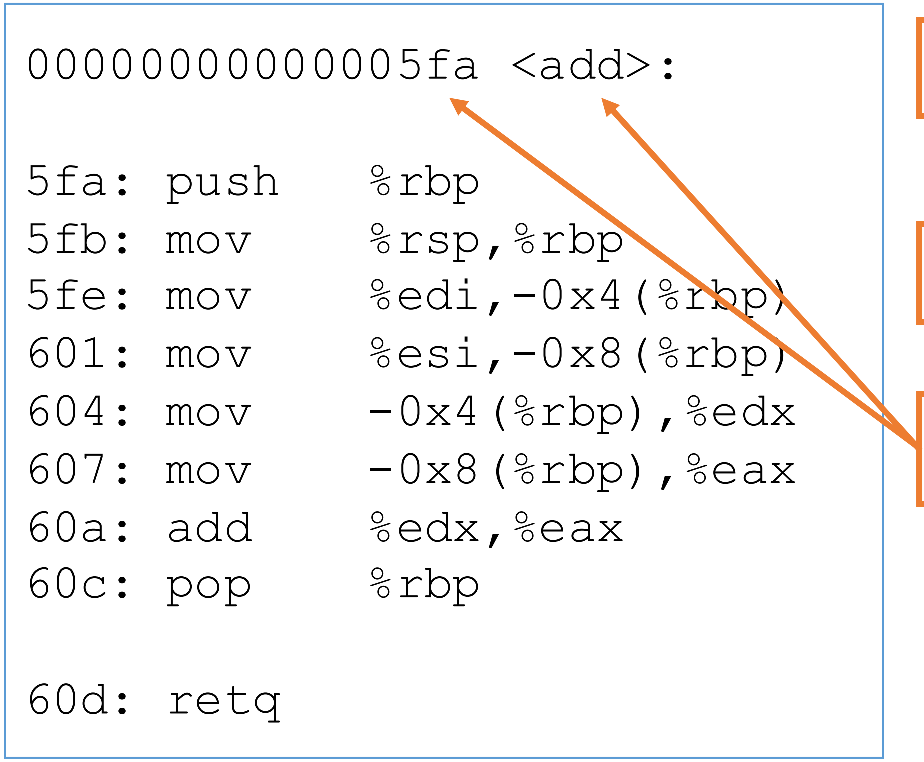
1. Open the binary/attach to or create the process with the function you want to trace

```
addrSpace = bpatch.processCreate(…);
```

2. Insert the tracing library (contains the function you want to call at entry/exit)

```
addrSpace->loadLibrary(“libtrace.so”);
```

Function Entry/Exit Instrumentation



```
00000000000005fa <add>:

5fa: push   %rbp
5fb: mov    %rsp,%rbp
5fe: mov    %edi,-0x4(%rbp)
601: mov    %esi,-0x8(%rbp)
604: mov    -0x4(%rbp),%edx
607: mov    -0x8(%rbp),%eax
60a: add    %edx,%eax
60c: pop    %rbp

60d: retq
```

libtrace.so

```
XXX <Trace>:

…: push   %rbp
     …
…: retq
```

1. Open the binary/attach to or create the process with the function you want to trace

```
addrSpace = bpatch.processCreate(…);
```

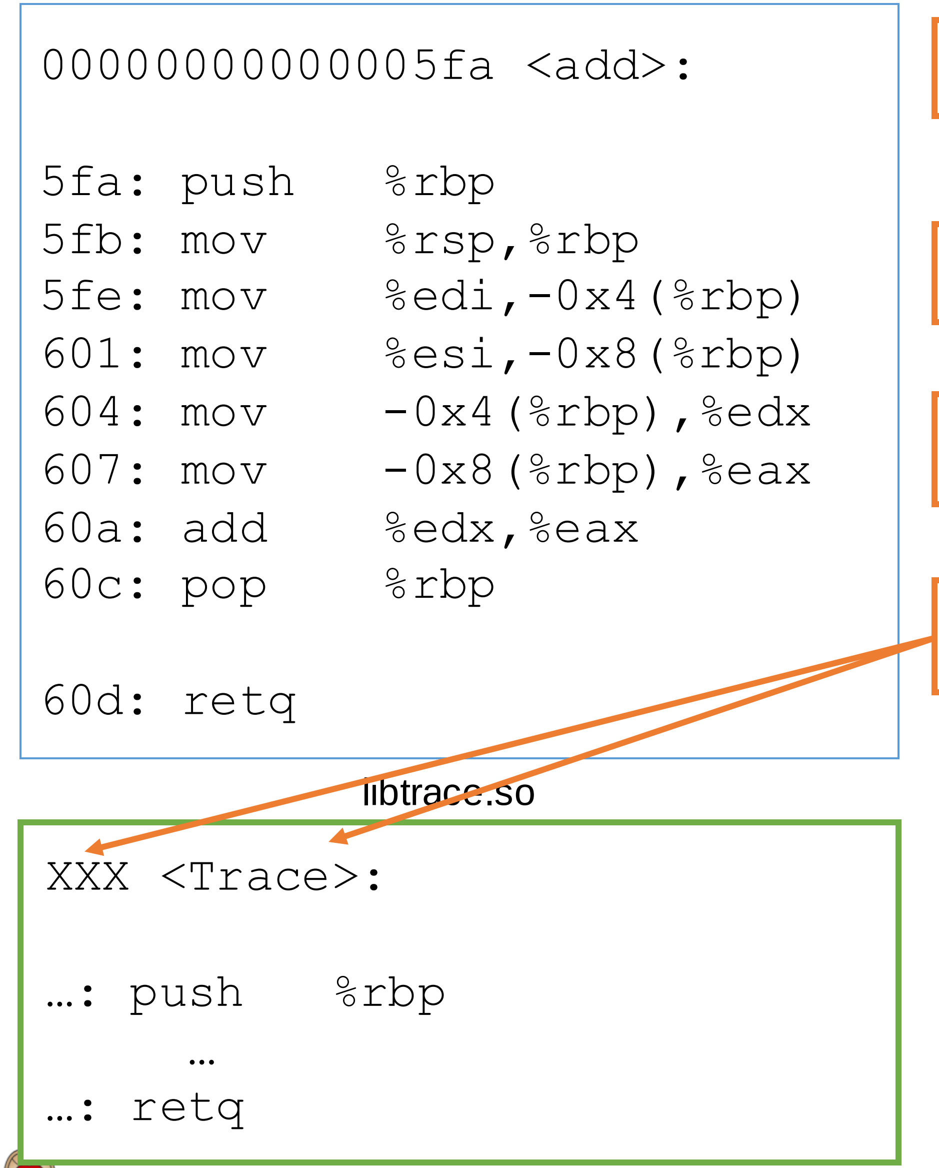
2. Insert the tracing library (contains the function you want to call at entry/exit)

```
addrSpace->loadLibrary(“libtrace.so”);
```

3. Find the function you want instrument

```
add = addrSpace->findFunction(“add”);
```

Function Entry/Exit Instrumentation



```
00000000000005fa <add>:

5fa: push   %rbp
5fb: mov    %rsp,%rbp
5fe: mov    %edi,-0x4(%rbp)
601: mov    %esi,-0x8(%rbp)
604: mov    -0x4(%rbp),%edx
607: mov    -0x8(%rbp),%eax
60a: add    %edx,%eax
60c: pop    %rbp

60d: retq
```

libtrace.so

```
XXX <Trace>:

…: push   %rbp
      …
…: retq
```

1. Open the binary/attach to or create the process with the function you want to trace

```
addrSpace = bpatch.processCreate(…);
```

2. Insert the tracing library (contains the function you want to call at entry/exit)

```
addrSpace->loadLibrary(“libtrace.so”);
```

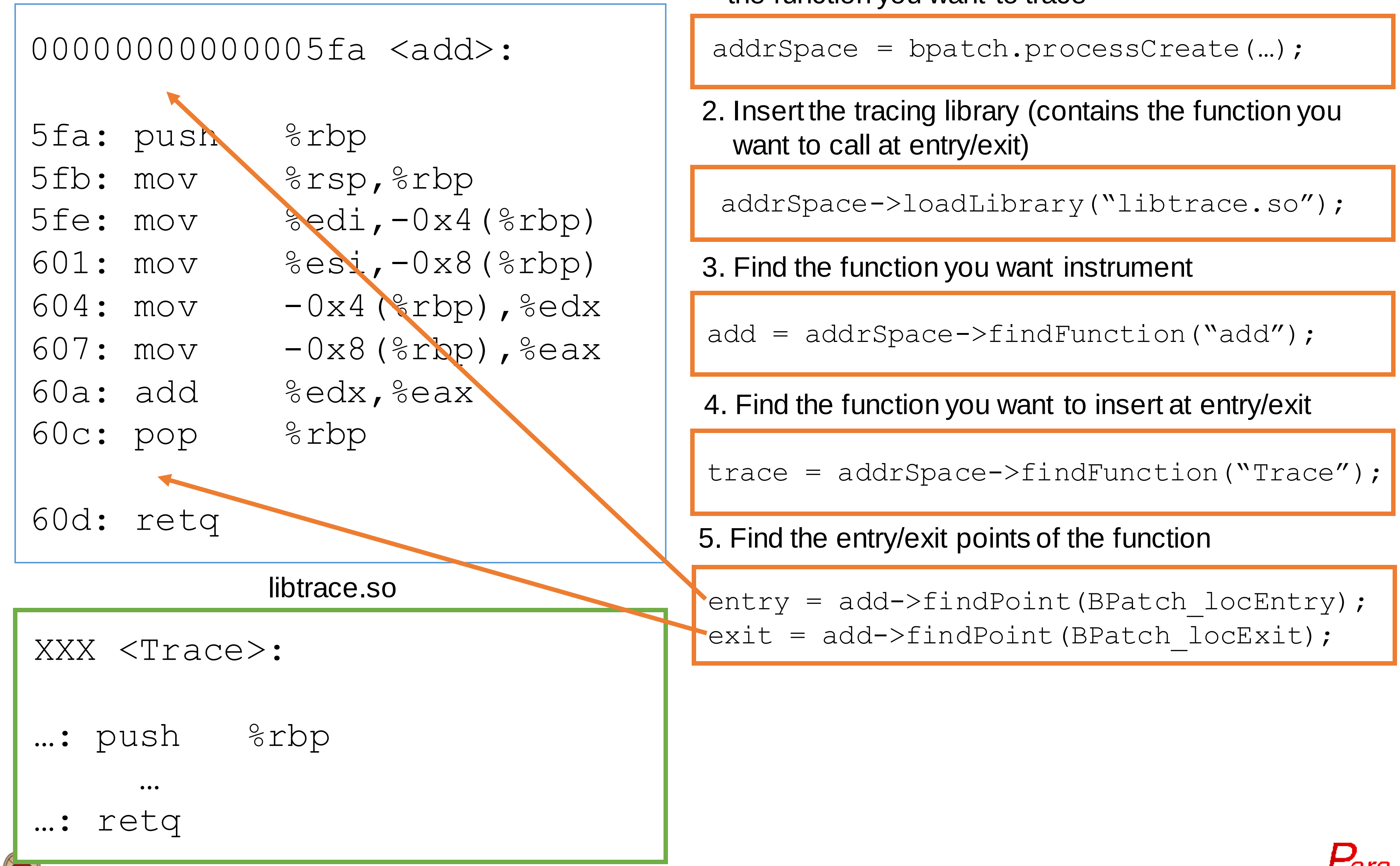
3. Find the function you want instrument

```
add = addrSpace->findFunction(“add”);
```

4. Find the function you want to insert at entry/exit

```
trace = addrSpace->findFunction(“Trace”);
```

Function Entry/Exit Instrumentation



```
0000000000005fa <add>:

5fa: push   %rbp
5fb: mov    %rsp,%rbp
5fe: mov    %edi,-0x4(%rbp)
601: mov    %esi,-0x8(%rbp)
604: mov    -0x4(%rbp),%edx
607: mov    -0x8(%rbp),%eax
60a: add    %edx,%eax
60c: pop    %rbp

60d: retq
```

libtrace.so

```
XXX <Trace>:

…: push   %rbp
       …
…: retq
```

1. Open the binary/attach to or create the process with the function you want to trace

```
addrSpace = bpatch.processCreate(…);
```

2. Insert the tracing library (contains the function you want to call at entry/exit)

```
addrSpace->loadLibrary(“libtrace.so”);
```

3. Find the function you want instrument

```
add = addrSpace->findFunction(“add”);
```

4. Find the function you want to insert at entry/exit

```
trace = addrSpace->findFunction(“Trace”);
```

5. Find the entry/exit points of the function

```
entry = add->findPoint(BPatch_locEntry);
exit = add->findPoint(BPatch_locExit);
```

Function Entry/Exit Instrumentation

```
00000000000005fa <add>:

5fa: push   %rbp
5fb: mov    %rsp,%rbp
5fe: mov    %edi,-0x4(%rbp)
601: mov    %esi,-0x8(%rbp)
604: mov    -0x4(%rbp),%edx
607: mov    -0x8(%rbp),%eax
60a: add    %edx,%eax
60c: pop    %rbp

60d: retq
```

libtrace.so

```
XXX <Trace>:

…: push   %rbp
      …
…: retq
```

1. Open the binary/attach to or create the process with the function you want to trace

```
addrSpace = bpatch.processCreate(…);
```

2. Insert the tracing library (contains the function you want to call at entry/exit)

```
addrSpace->loadLibrary(“libtrace.so”);
```

3. Find the function you want instrument

```
add = addrSpace->findFunction(“add”);
```

4. Find the function you want to insert at entry/exit

```
trace = addrSpace->findFunction(“Trace”);
```

5. Find the entry/exit points of the function

```
entry = add->findPoint(BPatch_locEntry);
exit = add->findPoint(BPatch_locExit);
```

6. Create the instrumentation snippet (call Trace())

```
BPatch_funcCallExpr traceExpr(trace,…)
```

Function Entry/Exit Instrumentation

```
0000000000005fa <add>:

5fa: push   %rbp
5fb: mov    %rsp,%rbp
5fe: mov    %edi,-0x4(%rbp)
601: mov    %esi,-0x8(%rbp)
604: mov    -0x4(%rbp),%edx
607: mov    -0x8(%rbp),%eax
60a: add    %edx,%eax
60c: pop    %rbp

60d: retq
```

libtrace.so

```
XXX <Trace>:

…: push   %rbp
      …
…: retq
```

1. Open the binary/attach to or create the process with the function you want to trace

```
addrSpace = bpatch.processCreate(…);
```

2. Insert the tracing library (contains the function you want to call at entry/exit)

```
addrSpace->loadLibrary(“libtrace.so”);
```

3. Find the function you want instrument

```
add = addrSpace->findFunction(“add”);
```

4. Find the function you want to insert at entry/exit

```
trace = addrSpace->findFunction(“Trace”);
```

5. Find the entry/exit points of the function

```
entry = add->findPoint(BPatch_locEntry);
exit = add->findPoint(BPatch_locExit);
```

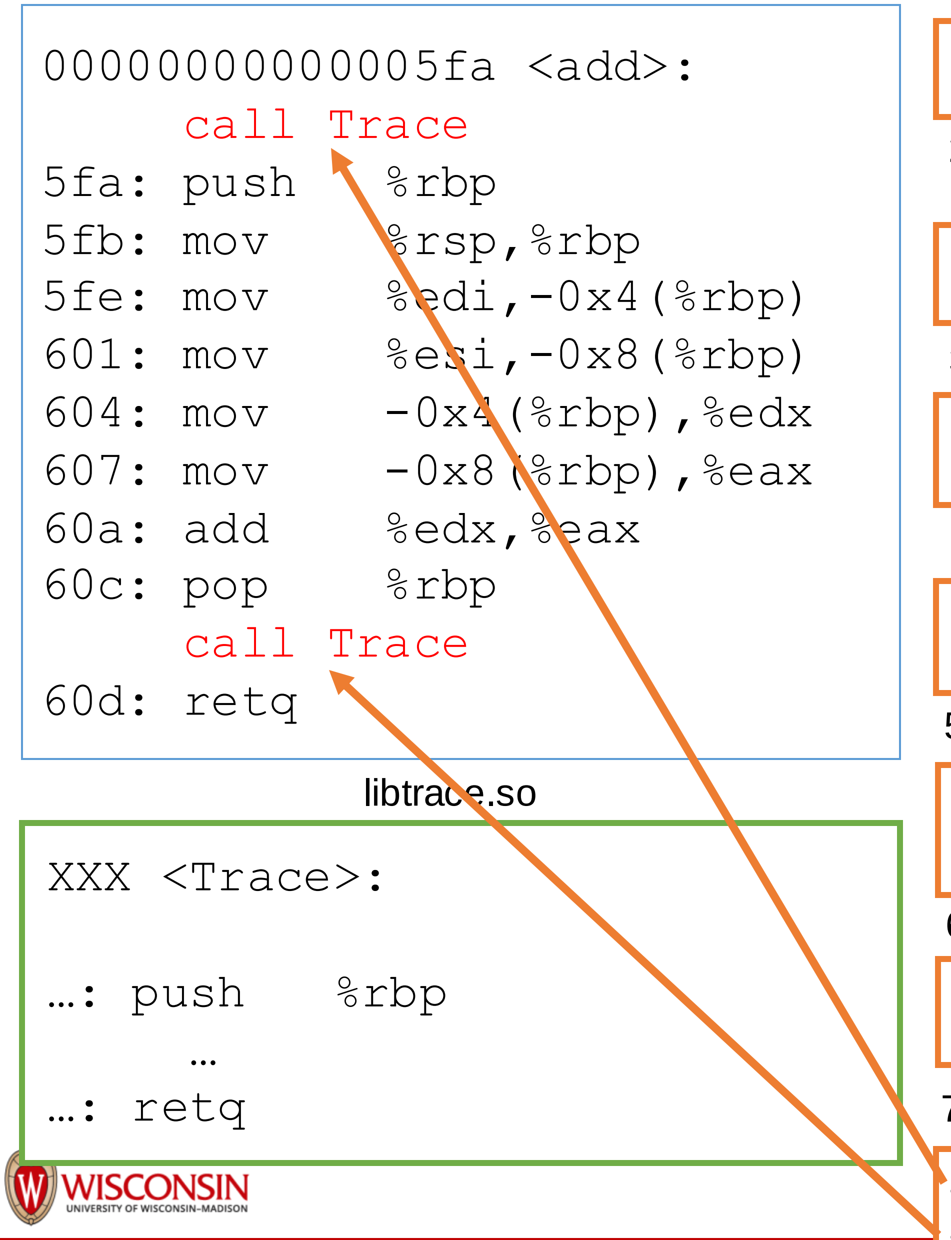
6. Create the instrumentation snippet (call Trace())

```
BPatch_funcCallExpr traceExpr(trace,…)
```

7. Insert Snippet

```
addrSpace->insertSnippet(traceExpr,entry)
```

Function Entry/Exit Instrumentation



```
0000000000005fa <add>:
     call Trace
5fa: push   %rbp
5fb: mov    %rsp,%rbp
5fe: mov    %edi,-0x4(%rbp)
601: mov    %esi,-0x8(%rbp)
604: mov    -0x4(%rbp),%edx
607: mov    -0x8(%rbp),%eax
60a: add    %edx,%eax
60c: pop    %rbp
     call Trace
60d: retq
```

libtrace.so

```
XXX <Trace>:

…: push   %rbp
       …
…: retq
```

1. Open the binary/attach to or create the process with the function you want to trace

```
addrSpace = bpatch.processCreate(…);
```

2. Insert the tracing library (contains the function you want to call at entry/exit)

```
addrSpace->loadLibrary(“libtrace.so”);
```

3. Find the function you want instrument

```
add = addrSpace->findFunction(“add”);
```

4. Find the function you want to insert at entry/exit

```
trace = addrSpace->findFunction(“Trace”);
```

5. Find the entry/exit points of the function

```
entry = add->findPoint(BPatch_locEntry);
exit = add->findPoint(BPatch_locExit);
```

6. Create the instrumentation snippet (call Trace())

```
BPatch_funcCallExpr traceExpr(trace,…)
```

7. Insert Snippet

```
addrSpace->insertSnippet(traceExpr,entry)
addrSpace->insertSnippet(traceExpr,exit)
```

Function Entry/Exit Instrumentation

```
0000000000005fa <add>:
     call Trace
5fa: push   %rbp
5fb: mov    %rsp,%rbp
5fe: mov    %edi,-0x4(%rbp)
601: mov    %esi,-0x8(%rbp)
604: mov    -0x4(%rbp),%edx
607: mov    -0x8(%rbp),%eax
60a: add    %edx,%eax
60c: pop    %rbp
     call Trace
60d: retq
```

libtrace.so

```
XXX <Trace>:

…: push   %rbp
     …
…: retq
```

Only minor modifications are needed to extend this example to:

- Basic Block Instrumentation
- Memory Tracing

Function Entry/Exit Instrumentation

```
00000000000005fa <add>:
     call Trace
5fa: push   %rbp
5fb: mov    %rsp,%rbp
5fe: mov    %edi,-0x4(%rbp)
601: mov    %esi,-0x8(%rbp)
604: mov    -0x4(%rbp),%edx
607: mov    -0x8(%rbp),%eax
60a: add    %edx,%eax
60c: pop    %rbp
     call Trace
60d: retq
```

libtrace.so

```
XXX <Trace>:

…: push   %rbp
     …
…: retq
```

1. Open the binary/attach to or create the process with the function you want to trace

```
addrSpace = bpatch.processCreate(…);
```

2. Insert the tracing library (contains the function you want to call at entry/exit)

```
addrSpace->loadLibrary(“libtrace.so”);
```

3. Find the function you want instrument

```
add = addrSpace->findFunction(“add”);
```

4. Find the function you want to insert at entry/exit

```
trace = addrSpace->findFunction(“Trace”);
```

5. Find the entry/exit points of the function

```
entry = add->findPoint(BPatch_locEntry);
exit = add->findPoint(BPatch_locExit);
```

6. Create the instrumentation snippet (call Trace())

```
BPatch_funcCallExpr traceExpr(trace,…)
```

7. Insert Snippet

```
addrSpace->insertSnippet(traceExpr,entry)
addrSpace->insertSnippet(traceExpr,exit)
```

Basic Block Instrumentation

```
0000000000005fa <add>:
     call Trace
5fa: push   %rbp
5fb: mov    %rsp,%rbp
5fe: mov    %edi,-0x4(%rbp)
601: mov    %esi,-0x8(%rbp)
604: mov    -0x4(%rbp),%edx
607: mov    -0x8(%rbp),%eax
60a: add    %edx,%eax
60c: pop    %rbp
     call Trace
60d: retq
```

libtrace.so

```
XXX <Trace>:

…: push   %rbp
     …
…: retq
```

1. Open the binary/attach to or create the process with the function you want to trace

```
addrSpace = bpatch.processCreate(…);
```

2. Insert the tracing library (contains the function you want to call at entry/exit)

```
addrSpace->loadLibrary(“libtrace.so”);
```

3. Find the function you want instrument

```
add = addrSpace->findFunction(“add”);
```

4. Find the function you want to insert at entry/exit

```
trace = addrSpace->findFunction(“Trace”);
```

5. Find the entry/exit points of the blocks in the function

```
add->getCFG()->getAllBasicBlocks(blocks)
for(auto block : blocks)
   entry.push_back(block->findEntryPoint())
   exit.push_back(block->findExitPoint())
```

6. Create the instrumentation snippet (call Trace())

```
BPatch_funcCallExpr traceExpr(trace,…)
```

7. Insert Snippet

```
addrSpace->insertSnippet(traceExpr,entry)
addrSpace->insertSnippet(traceExpr,exit)
```

Load/Store Operation Instrumentation

```
0000000000005fa <add>:
     call Trace
5fa: push   %rbp
5fb: mov    %rsp,%rbp
5fe: mov    %edi,-0x4(%rbp)
601: mov    %esi,-0x8(%rbp)
604: mov    -0x4(%rbp),%edx
607: mov    -0x8(%rbp),%eax
60a: add    %edx,%eax
60c: pop    %rbp
     call Trace
60d: retq
```

libtrace.so

```
XXX <Trace>:

…: push   %rbp
     …
…: retq
```

1. Open the binary/attach to or create the process with the function you want to trace

```
addrSpace = bpatch.processCreate(…);
```

2. Insert the tracing library (contains the function you want to call at entry/exit)

```
addrSpace->loadLibrary(“libtrace.so”);
```

3. Find the function you want instrument

```
add = addrSpace->findFunction(“add”);
```

4. Find the function you want to insert at entry/exit

```
trace = addrSpace->findFunction(“Trace”);
```

5. Find Load/Store operations in the function

```
std::set<BPatch_opCode> axs;
axs.insert(BPatch_opLoad);
axs.insert(BPatch_opStore);
lsp = add->findPoint(axs)
```

6. Create the instrumentation snippet (call Trace())

```
BPatch_funcCallExpr traceExpr(trace,…)
```

7. Insert Snippet

```
addrSpace->insertSnippet(traceExpr,lsp)
```

Stack Tracing

```
void main() {
  int a;
  foo(0);
  …
}

void foo(int b) {
  int c;
  bar();
  …
}

void bar() {
  int d;
  while(1);
}
```

Walk through the stack frames that lead to the current program counter address

Example Use Cases:

- Examining the context of crashes
- Attribute performance measurement

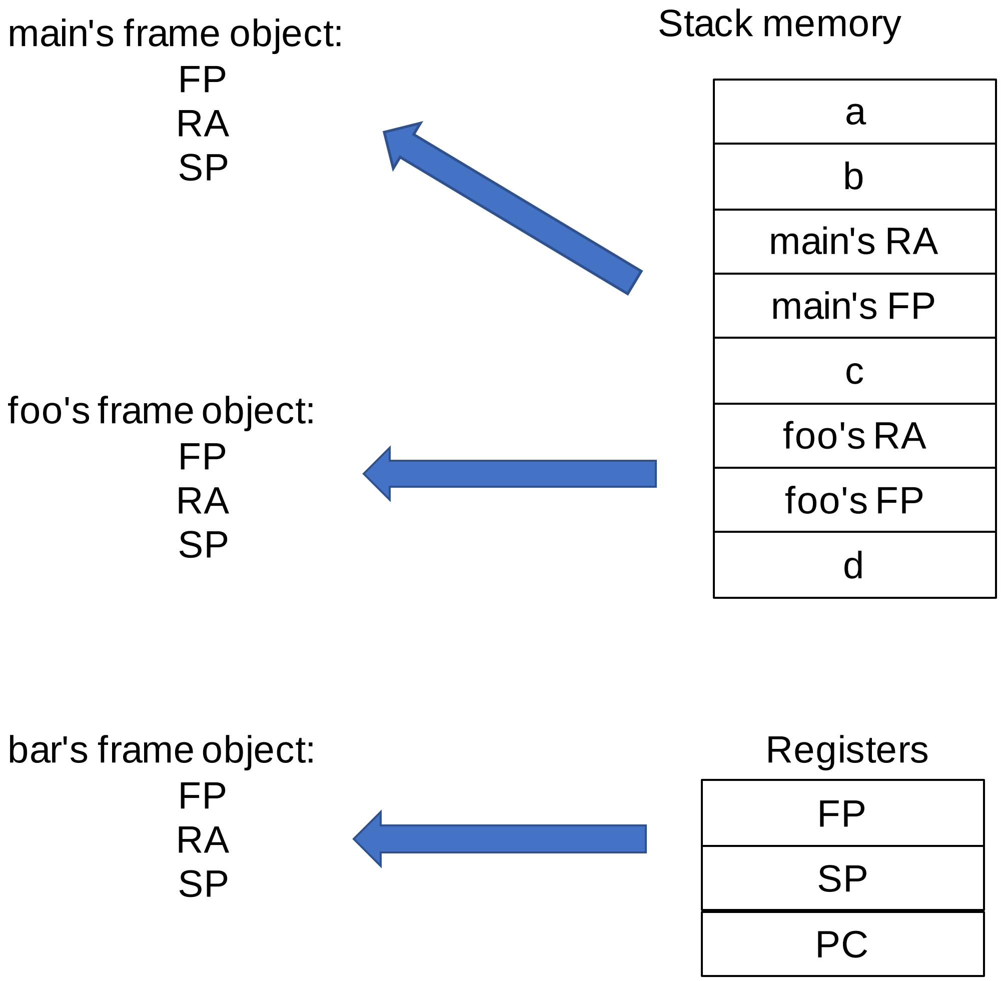
We will be doing third-party stack walk (attach to the process of the code example)

Stack Tracing

```
void main() {
   int a;
   foo(0);
   …
}

void foo(int b) {
   int c;
   bar();
   …
}

void bar() {
   int d;
   while(1);
}
```



main's frame object:
FP
RA
SP

foo's frame object:
FP
RA
SP

bar's frame object:
FP
RA
SP

Stack memory

a
b
main's RA
main's FP
c
foo's RA
foo's FP
d

Registers

FP
SP
PC

We use binary analysis, DWARF frame information, signal context information, and analyze Dyninst instrumentation to extract frames

Stack Tracing

```
void main() {
   int a;
   foo(0);
   …
}

void foo(int b) {
   int c;
   bar();
   …
}

void bar() {
   int d;
   while(1);
}
```

main's frame object:
FP
RA
SP

foo's frame object:
FP
RA
SP

bar's frame object:
FP
RA
SP

1. Attach to or create the process to perform stack walk

```
walker = Walker::newWalker(pid);
```

2. Perform the stack walk

```
vector<Frame> frames;
Walker->walkStack(frames);
```

3. Examine each frame

```
// function name of frame i
frame[i].getName(s);
// stack pointer value of frame i
frame[i].getSP()
// frame pointer value of frame i
frame[i].getFP()
// return address of frame i
frame[i].getRA()
```

Code Coverage

Determine which code in a binary have been executed through a test run.

Code Coverage

Determine which code in a binary have been executed through a test run.

What for?

To know how well their tests actually test their code.

To know whether they have enough testing in place.

Code Coverage

Determine which code in a binary have been executed through a test run.

What for?

To know how well their tests actually test their code.

To know whether they have enough testing in place.

Dyninst can be used to perform code coverage at function level or basic block level.

Code coverage

```
XXX <add>:


XXX: push    %rbp
...
XXX: retq
```

```
YYY <sub>:


YYY: push    %rbp
...
YYY: retq
```

```
ZZZ <printf>:


ZZZ: push    %rbp
...
ZZZ: retq
```

1. Get all functions

```
functions = addrSpace->getProcedures()
```

Code coverage

```
XXX <add>:


XXX: push    %rbp
...
XXX: retq
```

```
YYY <sub>:


YYY: push    %rbp
...
YYY: retq
```

```
ZZZ <printf>:


ZZZ: push    %rbp
...
ZZZ: retq
```

1. Get all functions

```
functions = addrSpace->getProcedures()
```

2. Allocate memory for flags

```
vectorFlag = addrSpace()->malloc(n)
```

Code coverage

1. Get all functions

```
functions = addrSpace->getProcedures()
```

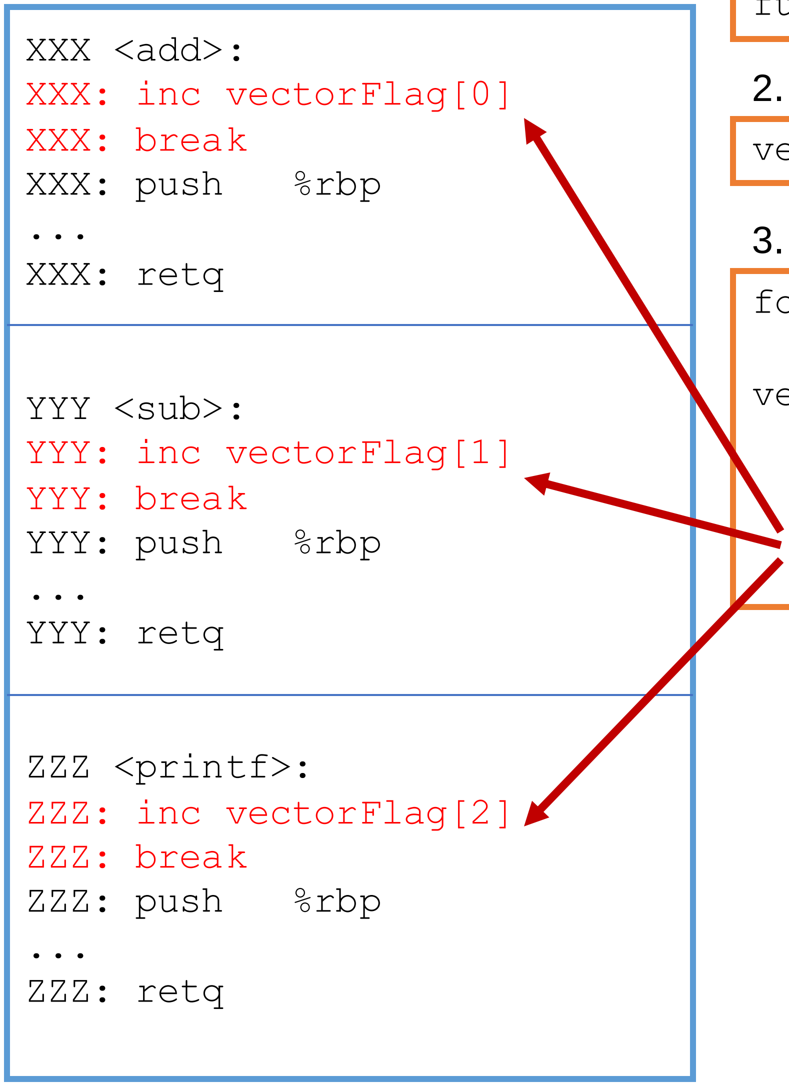
2. Allocate memory for flags

```
vectorFlag = addrSpace()->malloc(n)
```

3. Instrument entry point of every function

```
for (auto i : functions)
  BPatch_arithExpr assign(BPatch_assign,
vectorFlag[i],  BPatch_constExpr(1));
  BPatch_breakPointExpr bp;

  addrSpace->insertSnippet(assign,point)
  addrSpace->insertSnippet(bp,point)
```



```
XXX <add>:
XXX: inc vectorFlag[0]
XXX: break
XXX: push   %rbp
...
XXX: retq
```

```
YYY <sub>:
YYY: inc vectorFlag[1]
YYY: break
YYY: push   %rbp
...
YYY: retq
```

```
ZZZ <printf>:
ZZZ: inc vectorFlag[2]
ZZZ: break
ZZZ: push   %rbp
...
ZZZ: retq
```

Code coverage

```
XXX <add>:
XXX: inc vectorFlag[0]
XXX: break
XXX: push   %rbp
...
XXX: retq
```

```
YYY <sub>:
YYY: inc vectorFlag[1]
YYY: break
YYY: push   %rbp
...
YYY: retq
```

```
ZZZ <printf>:


ZZZ: push   %rbp
...
ZZZ: retq
```

1. Get all functions

```
functions = addrSpace->getProcedures()
```

2. Allocate memory for flags

```
vectorFlag = addrSpace()->malloc(n)
```

3. Instrument entry point of every function

```
for (auto i : functions)
  BPatch_arithExpr assign(BPatch_assign,
vectorFlag[i],  BPatch_constExpr(1));
  BPatch_breakPointExpr bp;

  addrSpace->insertSnippet(assign,point)
  addrSpace->insertSnippet(bp,point)
```

4. Clean up

```
While(process != terminated){
   // continueExecution()
   // WaitForStatusChange()
   // Program at breakpoint
   // Copy from mutatee to mutator
       vectorFlag
   For(int i = 0; i < vectorFlag.size(); i++)
       If (vectorFlag[i] == 1)
          addrSpace->deleteSnippet(assign)
          addrSpace->deleteSnippet(bp)
}
```

Code coverage (basic block)

1. Count basic blocks

```
m = for each function -> number of basic blocks
```

2. Allocate memory for flags

```
vectorFlag = addrSpace()->malloc(m)
```

3. Instrument entry point of every function

```
for (auto i : functions)
  add->getCFG()->getAllBasicBlocks(blocks)
  for(auto block : blocks)
    BPatch_arithExpr assign(BPatch_assign,
vectorFlag[i],  BPatch_constExpr(1));
    BPatch_breakPointExpr bp;

    addrSpace->insertSnippet(assign,point)
    addrSpace->insertSnippet(bp,point)
```

4. Clean up

```
While(process != terminated){
   // continueExecution()
   // WaitForStatusChange()
   // Program at breakpoint
   // Copy from mutatee to mutator
       vectorFlag
   For(int i = 0; i < vectorFlag.size(); i++)
       If (vectorFlag[i] == 1)
          addrSpace->deleteSnippet(assign)
          addrSpace->deleteSnippet(bp)
}
```

The Road Ahead: Features for 2020

- Statically-linked code for ARMv8
- Continue to improve build system
- Automate CI testing
- Parallel DWARF parsing

Who collaborates on Dyninst?

Thanks to contributions from:

- John Mellor-Crummey and Mark Krentel (Rice)
- Matt Legendre (LLNL)
- Ben Woodard (Red Hat)
- Stan Cox (Red Hat)

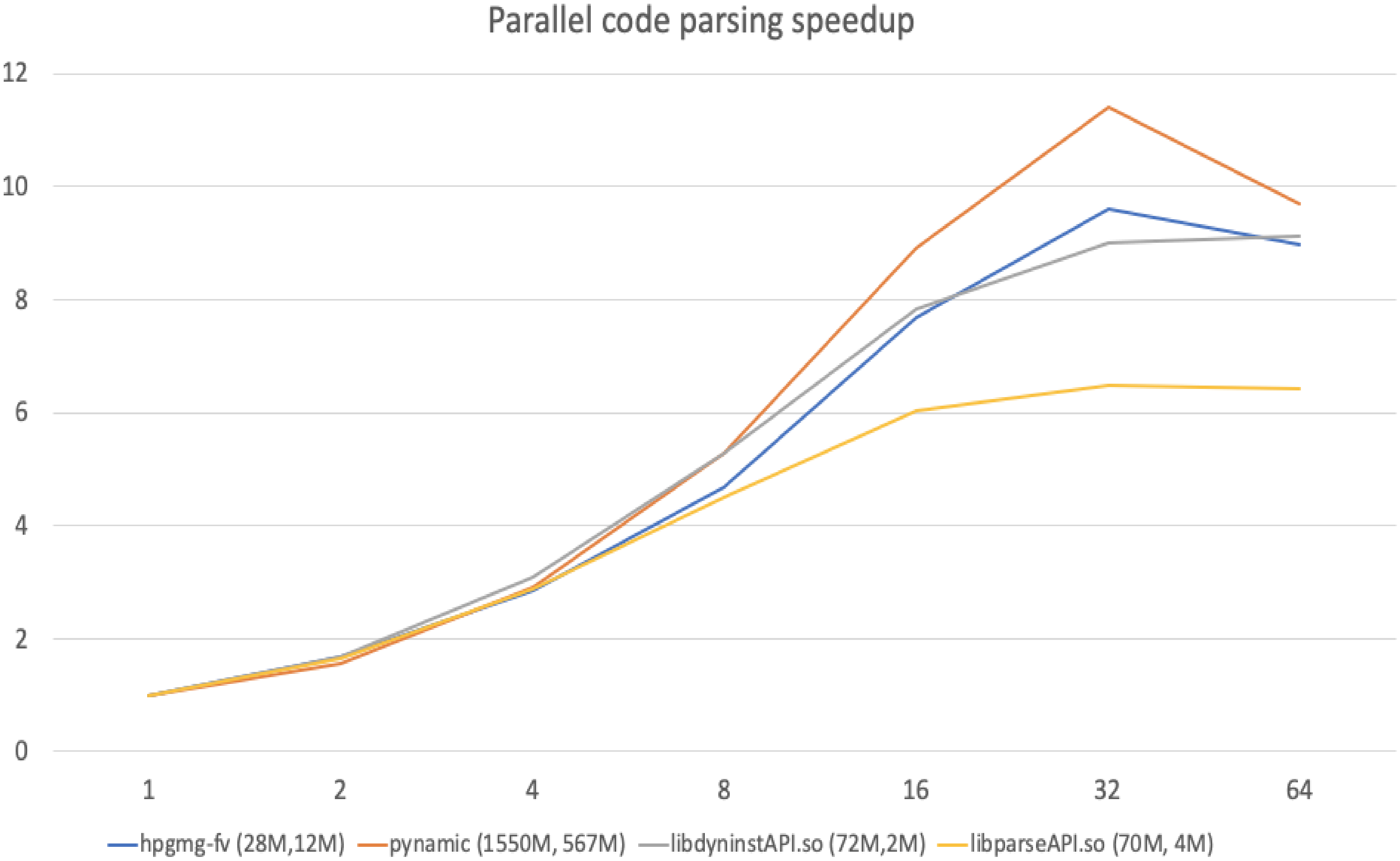
Who uses Dyninst?

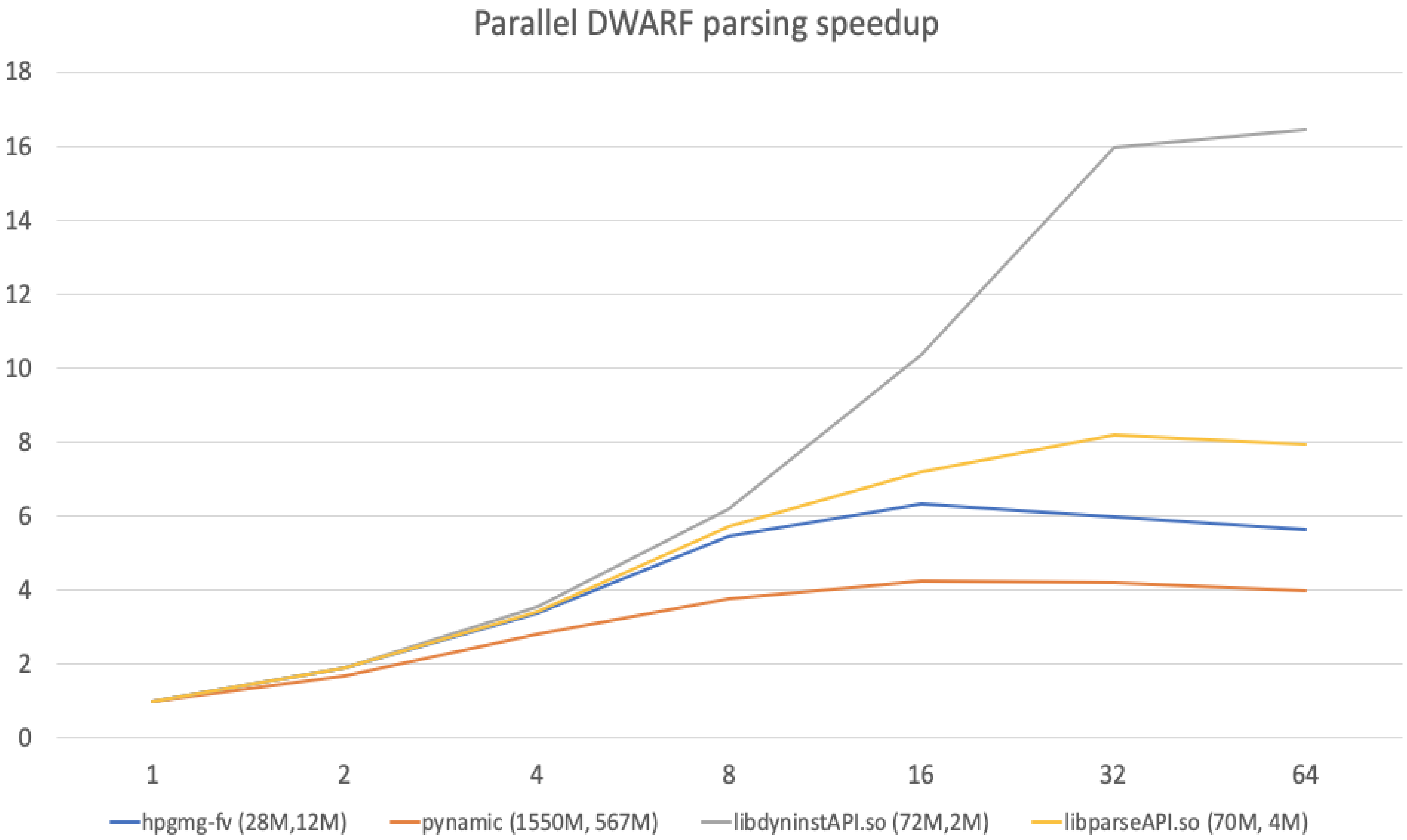
Dyninst is used by:

- HPCToolkit (Rice)
- SystemTap (Red Hat)
- Open|SpeedShop
- stat (LLNL)
- ATP (Cray)
- …

Not on this list? Let us know!

Questions?


Parallel code parsing speedup
12
10
8
6
4
2
0
1
2
4
8
16
32
64
hpgmg-fv (28M,12M)
pynamic (1550M, 567M)
libdyninstAPI.so (72M,2M)
libparseAPI.so (70M, 4M)


Parallel DWARF parsing speedup
18
16
14
12
10
8
6
4
2
0
1
2
4
8
16
32
64
hpgmg-fv (28M,12M)
pynamic (1550M, 567M)
libdyninstAPI.so (72M,2M)
libparseAPI.so (70M, 4M)