Dyninst Instrumentation for AMD GPU Binaries

Hsuan-Heng Wu

Computer Sciences Department

University of Wisconsin-Madison

Scalable Tools Workshop

July 2025

WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON

Dyninst Paradyn

A Classic Dyninst Instrumentation Example

Goal :

Increase a per basic block counter

Dump the result to a file in the end

Division of labor

Trace library provided by user to perform the intended operation

Increase counter

Open/write/close file

Dynisnt Mutator

Insert instrumentation into the kernel mutatee

- Prepare the call arguments (basic block ID/ filename)
- Call the functions in the trace library

BB Entry Instrumentation

```
00000000000005fa <add>:
BB_0:
     call IncreaseCounter(BB0)
5fa: push   %rbp
5fb: mov    %rsp,%rbp
5fe: mov    %edi,-0x4(%rbp)
BB_1:
     call IncreaseCounter(BB1)
601: mov    %esi,-0x8(%rbp)
60c: pop    %rbp

60d: retq
```

libtrace.so

```
XXX <IncreaseCounter>:
     …
…: // Increase
     …
…: retq
```

1. Open the binary the function you want to trace

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

2. load the trace library into the same address space as the mutatee

```
addrSpace→loadLibrary("libtrace.so");
```

3. Find the function you want instrumented

```
add = addrSpace→findFunction("add");
```

4. Find the function you want to insert at basic block entry

```
inc= addrSpace→findFunction("IncreaseCounter");
```

5. **Find the entry of all basic blocks**
6. Create the instrumentation snippet (call IncreaseCounter())
7. Insert snippets

```
add→getCFG()→getAllBasicBlocks(blocks);
BPatch_variableExpr * counters =
     app→mallocData(sizeof(int)*nBBs);
for(auto block : blocks; bbidx++) {
  bbEntry = block→findEntryPoint();
  BPatch_snippet base = &counters;
  BPatch_constExpr bbid(bbidx);
  BPatch_Vector<BPatch_snippet *> args{&base,&bbid};
  BPatch_funcCallExpr incExpr(args);
  addrSpace→insertSnippet(incExpr,bbEntry);
}
```

Function Exit Instrumentation

```
00000000000005fa <add>:
BB_0:
     call IncreaseCounter
5fa: push   %rbp
5fb: mov    %rsp,%rbp
5fe: mov    %edi,-0x4(%rbp)
BB_1:
     call IncreaseCounter
601: mov    %esi,-0x8(%rbp)
60c: pop    %rbp
     call DumpCountersToFile
60d: retq
```

libtrace.so

```
XXX <WriteCounterToFile>:
     …
…: // Open a file, write the counter
…  // and close i
…: retq
```

1. Open the binary the function you want to trace

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

2. load the trace library into the same address space as the mutatee

```
addrSpace→loadLibrary("libtrace.so");
```

3. Find the function you want instrumented

```
add = addrSpace→findFunction("add");
```

4. Find the function you want to insert at exit

```
flush= addrSpace→findFunction("DumpCountersToFile");
```

5. **Find the entry of all basic blocks**
6. Create the instrumentation snippet (call DumpCountersToFile())
7. Insert snippets

```
exit  = add→findPoint(BPatch_locExit);
BPatch_snippet base = &counters;
BPatch_constExpr nbbC(nBBs);
BPatch_Vector<BPatch_snippet *> args{&base,&nBBC};
BPatch_funcCallExpr flushExpr(flush,args);
addrSpace→insertSnippet(flushExpr,exit);
```

Demo

Kernel BB Entry Instrumentation

```
0000000000001900 <vector_add>:
BB_0:
     call IncreaseCounter(BB0)
1900: s_load_dword s0, s[4:5], 0x2c
1908: s_load_dword s1, s[4:5], 0x18
1910: s_waitcnt lgkmcnt(0)
BB_1:
     call IncreaseCounter(BB1)
1914: s_and_b32 s0, s0, 0xffff
1918: s_mul_i32 s6, s0

1994: s_endpgm
```

libtrace.so

```
XXX <IncreaseCounter>:
     …
…: // Increase
     …
…: retq
```

1. Open the binary the function you want to trace

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

2. load the trace library into the same address space as the mutatee

```
addrSpace→loadLibrary("libtrace.so");
```

3. Find the kernel you want instrumented

```
add = addrSpace→findFunction("vector_add");
```

4. Find the function you want to insert at basic block entry

```
inc= addrSpace→findFunction("IncreaseCounter");
```

5. **Find the entry of all basic blocks**
6. Create the instrumentation snippet (call IncreaseCounter())
7. Insert snippets

```
add→getCFG()→getAllBasicBlocks(blocks);
BPatch_perWaveVar * counters =
     app→allocatePerWave(sizeof(int)*nBBs);
for(auto block : blocks; bbidx++) {
  bbEntry = block→findEntryPoint();
  BPatch_snippet base = counters.address();
  BPatch_constExpr bbid(bbidx);
  BPatch_Vector<BPatch_snippet *> args{&base,&bbid};
  BPatch_funcCallExpr incExpr(args);
  addrSpace→insertSnippet(incExpr,bbEntry);
}
```

Kernel Exit Instrumentation

```
0000000000001900 <vector_add>:
BB_0:
     call IncreaseCounter(0)
1900: s_load_dword s0, s[4:5], 0x2c
1908: s_load_dword s1, s[4:5], 0x18
1910: s_waitcnt lgkmcnt(0)
BB_1:
     call IncreaseCounter(1)
1914: s_and_b32 s0, s0, 0xffff
1918: s_mul_i32 s6, s0
     call DumpCountersToFile
1994: s_endpgm
```

libtrace.so

```
XXX <WriteCounterToFile>:
     …
…: // Open a file, write the counter
…  // and close i
…: retq
```

1. Open the binary the function you want to trace

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

2. load the trace library into the same address space as the mutatee

```
addrSpace→loadLibrary("libtrace.so");
```

3. Find the kernel you want instrumented

```
add = addrSpace→findFunction("vector_add");
```

4. Find the function you want to insert at basic block entry

```
flush= addrSpace→findFunction("DumpCountersToFile");
```

5. **Find the entry of all basic blocks**
6. Create the instrumentation snippet (call DumpCountersToFile())
7. Insert snippets

```
exit  = add→findPoint(BPatch_locExit);
BPatch_snippet base = counters.address();
BPatch_constExpr nbbC(nBBs);
BPatch_Vector<BPatch_snippet *> args{&base,&nBBC};
BPatch_funcCallExpr flushExpr(flush,args);
addrSpace→insertSnippet(flushExpr,exit);
```

Design and Implementation

Analogy to instrumenting Multi-threaded CPU Binary

A invoker invokes one or more invokees (threads/ waves)

- Each invokee executing the same instrumented code
- Each invokee carry their own architecture state

Instrument invoker and the invokee

Questions:

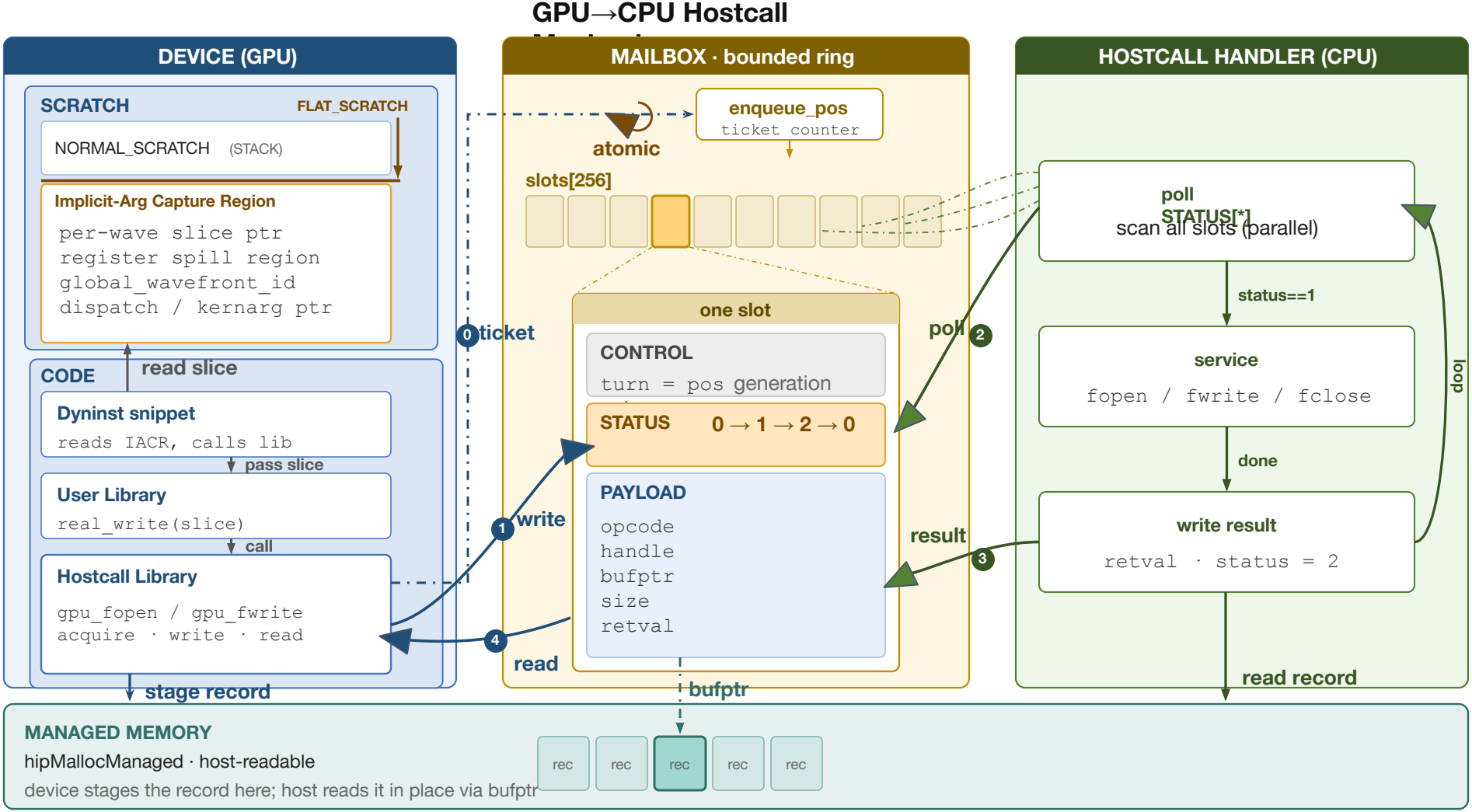
- How do we guarantee correctness concurrent invokee executions
- How do we enable communication between the invoker and invokee

Design Pillars

Per wave variable backed by managed memory allocated at launch
- Allows access on both the host and the device
- User can access them on the device after kernel launch if they wish
- Ports better to unified memory architectures (Maybe?)

Mutator rewrites device mutatees only to
- Prepare arguments and insert calls to user-prepared trace library
- Follow real world use cases (omnitrace/TAU)
- Avoid bringing the complexity of code generation into Dyninst

Provide hostcall helpers to deliver instrumentation data
- Currently support fopen/fread/fwrite


GPU→CPU Hostcall
DEVICE (GPU)
SCRATCH
FLAT_SCRATCH
NORMAL_SCRATCH (STACK)
Implicit-Arg Capture Region
per-wave slice ptr
register spill region
global_wavefront_id
dispatch / kernarg ptr
CODE
read slice
Dyninst snippet
reads IACR, calls lib
pass slice
User Library
real_write(slice)
call
Hostcall Library
gpu_fopen / gpu_fwrite
acquire · write · read
stage record
MAILBOX · bounded ring
enqueue_pos
ticket counter
atomic
slots[256]
one slot
CONTROL
turn = pos generation
STATUS 0 → 1 → 2 → 0
PAYLOAD
opcode
handle
bufptr
size
retval
0 ticket
1 write
2 poll
3 result
4 read
bufptr
HOSTCALL HANDLER (CPU)
poll STATUS[*]
scan all slots (parallel)
status==1
service
fopen / fwrite / fclose
done
write result
retval · status = 2
loop
read record
MANAGED MEMORY
hipMallocManaged · host-readable
device stages the record here; host reads it in place via bufptr
rec
rec
rec
rec
rec

The crux: Implement cross-code-object device linking

Instrumentation lives in a **separately compiled** device library → inserted call is **cross-code-object**

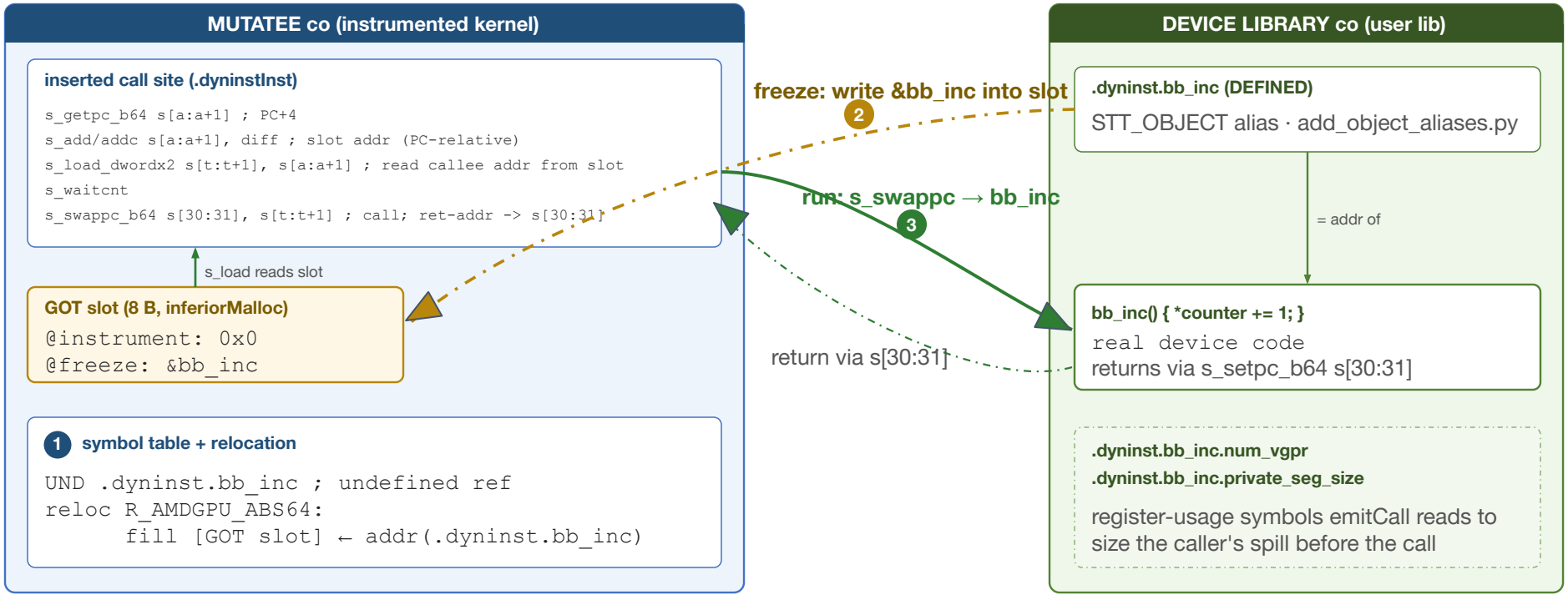
No official support of dynamic linking for AMDGPU code objects
Loader binds the call target as an **OBJECT** symbol, **not FUNC**

Cross-object link requires **identical ABI**
arch + code-object version + features

hipModuleLoad can't link two code objects for the same arch
HSA executable API can load several + `freeze` resolves relocs

Cross-object call: mutatee kernel → device-library function

callee lives in another code object — address unknown until load; bridged by a GOT slot the loader fills (like PLT/GOT)



1 INSTRUMENT (Dyninst)

emitCall sees callee in another object :
Emit UND `.dyninst.bb_inc`, malloc an. 8-byte GOT slot;
register a dynamic reloc `R_AMDGPU_ABS64`
Code = getpc+add → s_load_dwordx2 → s_swappc.

2 FREEZE (HSA loader)

Load both co into one `hsa_executable` and frozen. The loader resolves `.dyninst.bb_inc` (defined in the lib) and writes `&bb_inc` into the slot — "cross-object relocs resolved".

3 RUN

The call site computes the slot addr (PC-relative), `s_load_dwordx2` the resolved `&bb_inc`, and `s_swappc` into it. bb_inc runs and returns via the ABI reg `s[30:31]`.

Dyninst Helper Library

Mutator only rewrites device mutatee
- Rely on host call to relay data back to host
- Inspired by the libC for GPU project
  - A dedicated thread on the host polling shared memory
  - Device side implement a thin wrapper to perform RPC

Each wave atomically acquires a ticket to bounded ring buffer (mailbox)
- Allow waves on different slots to invoke rpc in parallel

User Trace Library

Similar design to NVBit

Implement as hip functions

Enable builtin variables access

Compile along dyninst helpers

```
void bb_flush_pw(void* base, int nbb) {
…
unsigned wid = (blockIdx.x * blockDim.x + threadIdx.x) / 64u;
char* b = (char*)base;
volatile unsigned* counts = (volatile unsigned*)b;
fi = put_str(nm, fi, "bbcount_");
fi = put_uint(nm, fi, wid);
fi = put_str(nm, fi, ".txt");
nm[fi] = '\0';
long long h = gpu_fopen(nm, "w");
int i = 0;
for (int k = 0; k < nbb; k++) {
    i = put_str(t, i, "bb ");  i = put_uint(t, i, (unsigned)k);
    i = put_str(t, i, ": ");   i = put_uint(t, i, counts[k]);
    t[i++] = '\n';
}
gpu_fwrite(h, t, i);
}
```

Implicit Argument Capture Region

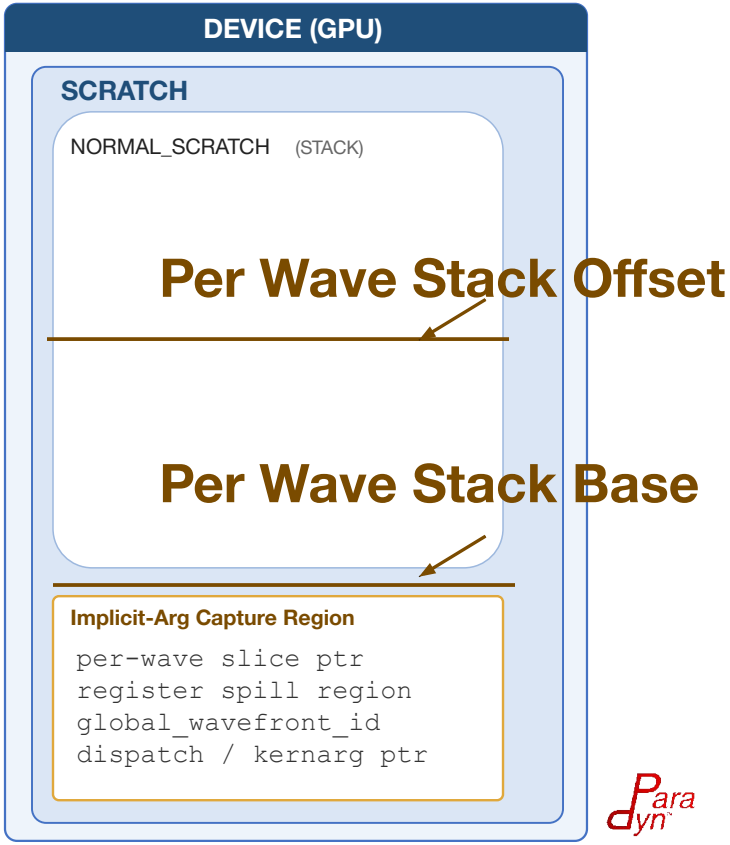
A region below per wave stack base

Stores

- Implicit arguments: threadIdx
- Address of per wave variable
- Dyninst spilled register

How

- Move the base address by a fixed offset
- Avoid messing up user code



Successes and Limits

What is Working

- The ability to perform host call from kernel
- Enables both tracing and dumping model of instrumentation data

Limitations

- Currently we do not support instrumenting kernels that contains indirect control flow
- Prototyped for MI100(gfx908)
  - But we expect migrating to unified memory make things easier

AMD Wishlist

Proper dynamic linking for AMD GPU code objects

Working semantics library

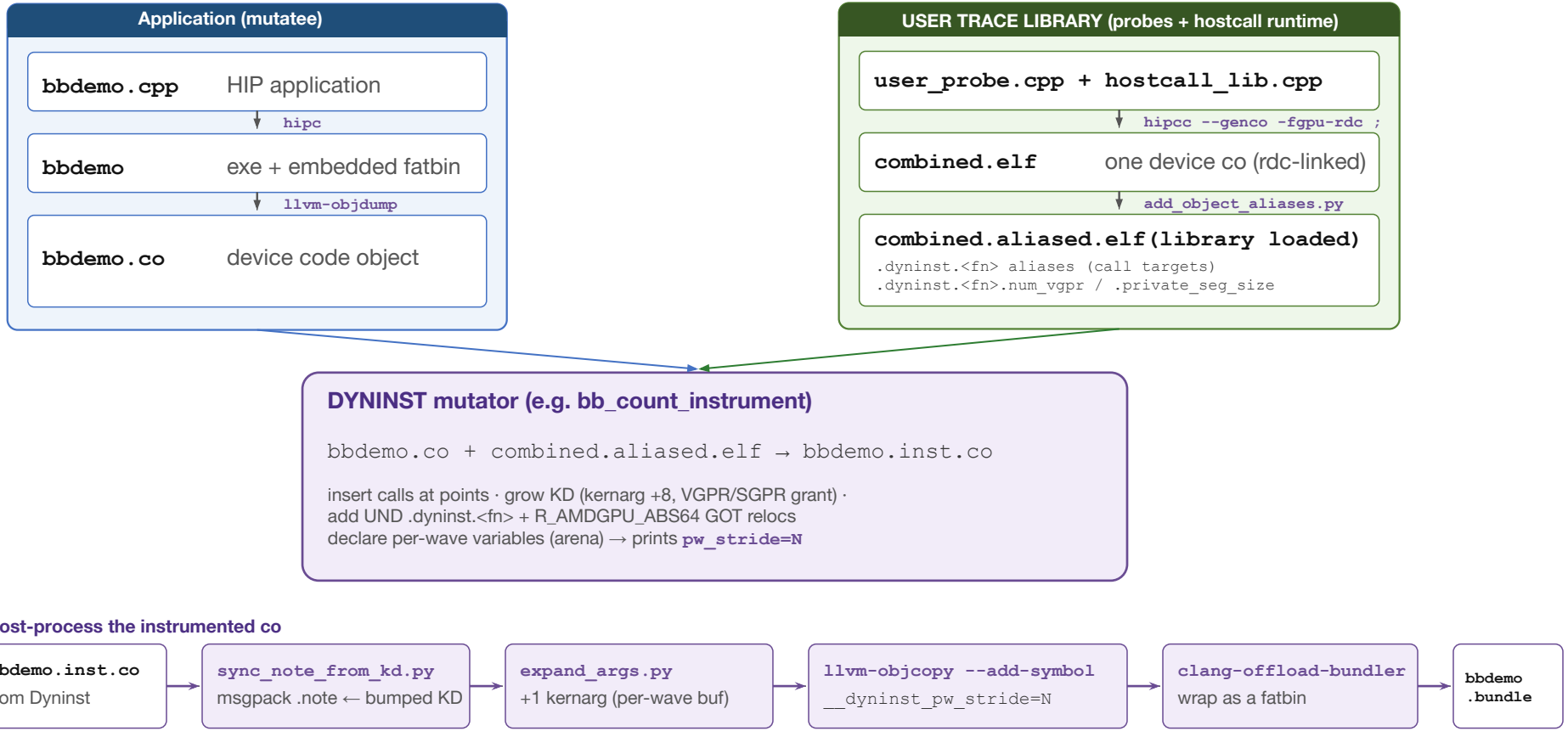
Clean, general interface for C library calls from device to host.

Questions ?

https://github.com/dyninst/dyninst-amdgpu-poc

hwu337@wisc.edu

Building the instrumented mutatee (offline pipeline)



Launching the instrumented mutatee (LD_PRELOAD interposition)

the preload hooks HIP/HSA so the unmodified app transparently loads the instrumented co, links the trace lib, and gets a managed per-wave buffer

APP process + LD_PRELOAD=preload.so (calls intercepted, in order)

1 `__hipRegisterFatBinary(app fatbin)`

→ **SUBSTITUTE with the instrumented bundle**

2 `hsa_code_object_reader_* / _load_agent_code_object`

→ **AUGMENT the executable: define `mailbox`, load the trace-lib co into it**

3 `hsa_executable_freeze`

loader resolves .dyninst.<fn> GOT relocs + the
mailbox extern → status = 0 (cross-object linked)

→ **start the CPU service thread (once)**

4 `hipLaunchKernel(func, grid, block, args)`

```
STRIDE = co_read(__dyninst_pw_stride)
nwaves = grid*block / 64
```

→ **hipMallocManaged(nwaves * STRIDE)**
→ **append &buf as args[N], then call the real launch**

5 `kernel executes (instrumented)`

probes write their per-wave slices;

→ **GPU→CPU hostcalls via the mailbox ring**

(fopen / fwrite / fclose serviced by the CPU thread)

Result: per-wave files written; kernel result unchanged — all transparent to the app.

A standalone launcher (raw HSA, rocgdb-attachable) performs the same load+link+service; only the HIP hooks differ.

runtime state (created by the hooks)

MAILBOX (fine-grained, host-coherent)

bounded ring of slots; defined as the `mailbox` symbol; the trace-lib co links against it

created @ step 2

CPU SERVICE THREAD

polls all slots; does the real fopen/fwrite/fclose; writes bbcount_<wid>.txt

started @ step 3

MANAGED PER-WAVE BUFFER

hipMallocManaged(nwaves * STRIDE)
passed to the kernel as the extra kernarg
one STRIDE-byte slice per wave; host-readable

created @ step 4

OUTPUT

bbcount_<wid>.txt (via hostcalls)
+ CPU reads the managed buffer directly