Useful Tools to Extend gem5 Models

The Ninja Feature of gem5

There are many useful tools inside gem5 that do not have proper documentation.
In this section, we will cover

- Probe point

OOO Action

If you have never built /gem5/build/X86/gem5.fast, please do so with the following command, as gem5 takes a long time to build.

```
cd gem5
scons build/X86/gem5.fast -j$(nproc)
```

Probe Point

Probe Point

There are three components related to probe point in gem5:

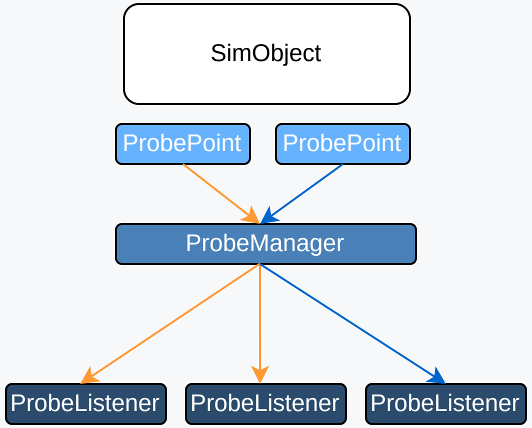
1. ProbeManger
2. ProbePoint
3. ProbeListener

Use-case of Probe Points

- Profiling a component without adding too much to the component's codebase
- Making more flexible exit events
- Tracking advance behaviors
- More

More about Probe Point

- Every SimObject has a ProbeManager
- The ProbeManager manages all registered ProbePoints and the connected ProbeListeners for the SimObject
- One ProbePoint can notify multiple ProbeListeners, and one ProbeListener can listen to multiple ProbePoints
- One ProbeListener can only attach to one SimObject



How to use Probe Point?

1. Create a ProbePoint in a SimObject
2. Register the ProbePoint with the SimObject's ProbeManager
3. Create a ProbeListener
4. Connect the ProbeListener to the SimObject and register it with the SimObject's ProbeManager

Let's try it with a simple example!

Hands-On Time!

01-local-inst-tracker

Currently, gem5 does not have a straight-forward method to raise an exit event after we execute (commit) a number of instructions for multi-core simulation. We can easily create one with Probe Point. We will start with creating a ProbeListener that listens to each core's `ppRetiredInsts` ProbePoint, then in `02-global-inst-tracker`, we will create a SimObject to manage all the ProbeListeners to raise an exit event after the simulation executes (commits) a number of instructions.

Goal

1. Create a ProbeListener called the local-instruction-tracker
2. Connect the ProbeListener to the BaseCPU and register our ProbeListener with the BaseCPU's ProbeManager
3. Run a simple simulation with the local-instruction-tracker

Hands-On Time!

01-local-inst-tracker

All completed materials can be found under `materials/03-Developing-gem5-models/09-extending-gem5-models/01-local-inst-tracker/complete`.

Let's start with creating `inst_tracker.hh` and `inst_tacker.cc` under `/src/cpu/probes`.

In the `inst_tracker.hh` file, we need to include the headers and necessary libraries:

```
#ifndef __CPU_PROBES_INST_TRACKER_HH__
#define __CPU_PROBES_INST_TRACKER_HH__

#include "sim/sim_exit.hh"
#include "sim/probe/probe.hh"
#include "params/LocalInstTracker.hh"
```

01-local-inst-tracker

Then, we can create a `ProbeListenerObject` called `LocalInstTracker`. A `ProbeListenerObject` is a minimum wrapper of the `ProbeListener` that allows us to attach it to the SimObject we want to listen to.

```
namespace gem5
{
class LocalInstTracker : public ProbeListenerObject
{
  public:
    LocalInstTracker(const LocalInstTrackerParams &params);
    virtual void regProbeListeners();
}
}
```

Now, we have a constructor for the `LocalInstTracker` and a virtual function `regProbeListeners()`. The `regProbeListeners` is called automatically when the simulation starts. We will use it to attach to the ProbePoint.

01-local-inst-tracker

Our goal is to count the number of committed instructions for our attached core so we can listen to the `ppRetiredInsts` ProbePoint that already exists in the `BaseCPU` SimObject.

Let's look at the `ppRetiredInsts` ProbePoint a bit.
It is a `PMU probe point` that as suggested in src/cpu/base.hh that it will notify the listeners with a `uint64_t` variable.
In src/cpu/base.cc:379, we can see that it is registered to the `BaseCPU` SimObject's ProbeManager with the string `"RetiredInsts"`. All ProbePoints are registered with the ProbeManager with a unique string variable, so we can use this string later to attach our listeners to this ProbePoint. Lastly, we can find that this ProbePoint notifies its listeners with an integer `1` when there is an instruction committed in src/cpu/base.cc:393.
Now that we know what ProbePoint we are targeting, we can set it up for our LocalInstTracker.

01-local-inst-tracker

In the `inst_tracker.hh`, we need to add two things:

1. The type of argument we are going to receive from the ProbePoint. In our case here is a `uint64_t` variable

```
typedef ProbeListenerArg<LocalInstTracker, uint64_t> LocalInstTrackerListener;
```

2. We need to have a function to handle the notification from the ProbePoint. Since we are counting the number of instructions committed and wanting to exit when it reaches a certain threshold, let's also create two `uint64_t` variables for this purpose

```
void checkPc(const uint64_t& inst);
uint64_t instCount;
uint64_t instThreshold;
```

01-local-tracker

Here comes an optional part. The Probe Point tool allows dynamic attachment and detachment during the simulation. Therefore, we can create a way to start and stop listening for our LocalInstTracker.

In the `inst_tracker.hh`,

```
bool listening;
void stopListening();
void startListening() {
  listening = true;
  regProbeListeners();
}
```

01-local-tracker

In the `inst_tracker.cc`, let's define the constructor fist

```
LocalInstTracker::LocalInstTracker(const LocalInstTrackerParams &p)
    : ProbeListenerObject(p),
      instCount(0),
      instThreshold(p.inst_threshold),
      listening(p.start_listening)
{}
```

This means that we initialize the `instCount` as 0, `instThreshold` with the parameter `inst_threshold`, and listening with the parameter `start_listening`.

01-local-tracker

Then, let's define the `regProbeListeners` function, which will be called automatically when the simulation starts, also as we defined above when `startListening` is called.

```
void
LocalInstTracker::regProbeListeners()
{
    if (listening) {
        listeners.push_back(new LocalInstTrackerListener(this, "RetiredInsts",
            &LocalInstTracker::checkPc));
    }
}
```

As we can see, it uses the `LocalInstTrackerListener` type that we defined earlier. It connects our listener with the ProbePoint that is registered with the string variable `"RetiredInsts"`. When the ProbePoint notifies the Manager, it will call our function `checkPc` with the notified variable, a `uint64_t` variable in our case.

01-local-tracker

For our `checkPc` function, it should count the instruction committed, check if it reaches the threshold, then raises an exit event when it does.

```
void
LocalInstTracker::checkPc(const uint64_t& inst)
{
    instCount ++;
    if (instCount >= instThreshold) {
        exitSimLoopNow("a thread reached the max instruction count");
    }
}
```

The `exitSimLoopNow` will create an event immediately, with the string variable. It will immediately exit the simulation. This string variable is categorized as `ExitEvent.MAX_INSTS` in the standard library.

Lastly, let's defined the `stopListening` function for dynamic detachment

```
void
LocalInstTracker::stopListening()
{
    listening = false;
    for (auto l = listeners.begin(); l != listeners.end(); ++l) {
        delete (*l);
    }
    listeners.clear();
}
```

This is a really rough example of how it can be done. It does not check what ProbePoint the listeners are attaching to, so if our ProbeListener listens to multiple ProbePoints, we will need to check the registered string variables for detaching the correct ProbeListeners.
For our simple case here, this rough method will serve the purpose.
For more detailed information about how the dynamic detachment can be done, please refer to src/sim/probe/probe.hh

01-local-tracker

In addition to the above functionality, we can also add some getter and setter functions, such as

```
void changeThreshold(uint64_t newThreshold) {
  instThreshold = newThreshold;
}
void resetCounter() {
  instCount = 0;
}
bool ifListening() const {
  return listening;
}
uint64_t getThreshold() const {
  return instThreshold;
}
```

01-local-inst-tracker

Now, let's set up the Python object of the LocalInstTracker.
Let's create a file called `InstTracker.py` under the same directory `src/cpu/probes`.

```
from m5.objects.Probe import ProbeListenerObject
from m5.params import *
from m5.util.pybind import *

class LocalInstTracker(ProbeListenerObject):
    type = "LocalInstTracker"
    cxx_header = "cpu/probes/inst_tracker.hh"
    cxx_class = "gem5::LocalInstTracker"

    cxx_exports = [
        PyBindMethod("stopListening"),
        PyBindMethod("startListening"),
        PyBindMethod("changeThreshold"),
        PyBindMethod("resetCounter"),
        PyBindMethod("ifListening"),
        PyBindMethod("getThreshold")
    ]

    inst_threshold = Param.Counter("The instruction threshold to trigger an"
                                                            " exit event")
    start_listening = Param.Counter(True, "Start listening for instructions")
```

01-local-inst-tracker

Like all new objects, we need to register it in Scons, so let's modify src/cpu/probes/SConscript and add

```
SimObject(
    "InstTracker.py",
    sim_objects=["LocalInstTracker"],
)
Source("inst_tracker.cc")
```

Now we have everything setup for our `LocalInstTracker`!

Let's build gem5 again

```
cd gem5
scons build/X86/gem5.fast -j$(nproc)
```

01-local-inst-tracker

After it is built, we can test our `LocalInstTracker` with the materials/03-Developing-gem5-models/09-extending-gem5-models/01-local-inst-tracker/simple-sim.py

```
cd /workspaces/2024/materials/03-Developing-gem5-models/09-extending-gem5-models/01-local-inst-tracker
/workspaces/2024/gem5/build/X86/gem5.fast -re --outdir=simple-sim-m5out simple-sim.py
```

This SE script runs a simple openmp workload that sums up an array of numbers. The source code of this workload can be found in materials/03-Developing-gem5-models/09-extending-gem5-models/simple-omp-workload/simple_workload.c.

```
m5_work_begin(0, 0);
for (j = 0; j < ARRAY_SIZE; j++) {
    #pragma omp parallel for reduction(+:sum)
    for (i = 0; i < NUM_ITERATIONS; i++) {
        sum += array[j];
    }
}
m5_work_end(0, 0);
```

01-local-inst-tracker

For our SE script, we first attach a LocalInstTracker to each core object with a threshold of 100,000 instructions. We will not start listening to the core's committed instructions from the start of the simulation.

```
from m5.objects import LocalInstTracker
for core in processor.get_cores():
    tracker = LocalInstTracker(
        start_listening = False,
        inst_threshold = 100000
    )
    core.core.probeListener = tracker
    all_trackers.append(tracker)
```

01-local-inst-tracker

We will start listening when the simulation raises an workbegin exit event, so we need a workbegin handler to do that

```
def workbegin_handler():
    print("Reached workbegin, now start listening for instructions")
    for tracker in all_trackers:
        tracker.startListening()
    yield False
```

Let's make a workend exit event handler for fun:

```
def workend_handler():
    print("Reached workend")
    yield False
```

01-local-inst-tracker

We know that after reaching the threshold, our LocalInstTracker will raise an `ExitEvent.MAX_INSTS` exit event, so we need a handler for it too

```
def max_inst_handler():
    counter = 1
    while counter < len(processor.get_cores()):
        print("Max Inst exit event triggered")
        print(f"Reached {counter}")
        counter += 1
        print("Fall back to simulation")
        yield False
    print(f"All {counter} cores have reached the max instruction threshold")
    print("Now stop listening for instructions")
    for tracker in all_trackers:
        tracker.stopListening()
    yield False
```

01-local-inst-tracker

After setting these handlers with `simulator`

```
simulator = Simulator(
    board=board,
    on_exit_event={
        ExitEvent.MAX_INSTS: max_inst_handler(),
        ExitEvent.WORKBEGIN: workbegin_handler(),
        ExitEvent.WORKEND: workend_handler(),
    }
)
```

We should expect 8 `MAX_INSTS` events after the `WORKBEGIN` event.

01-local-inst-tracker

We should expect to see below log in `simout.txt`

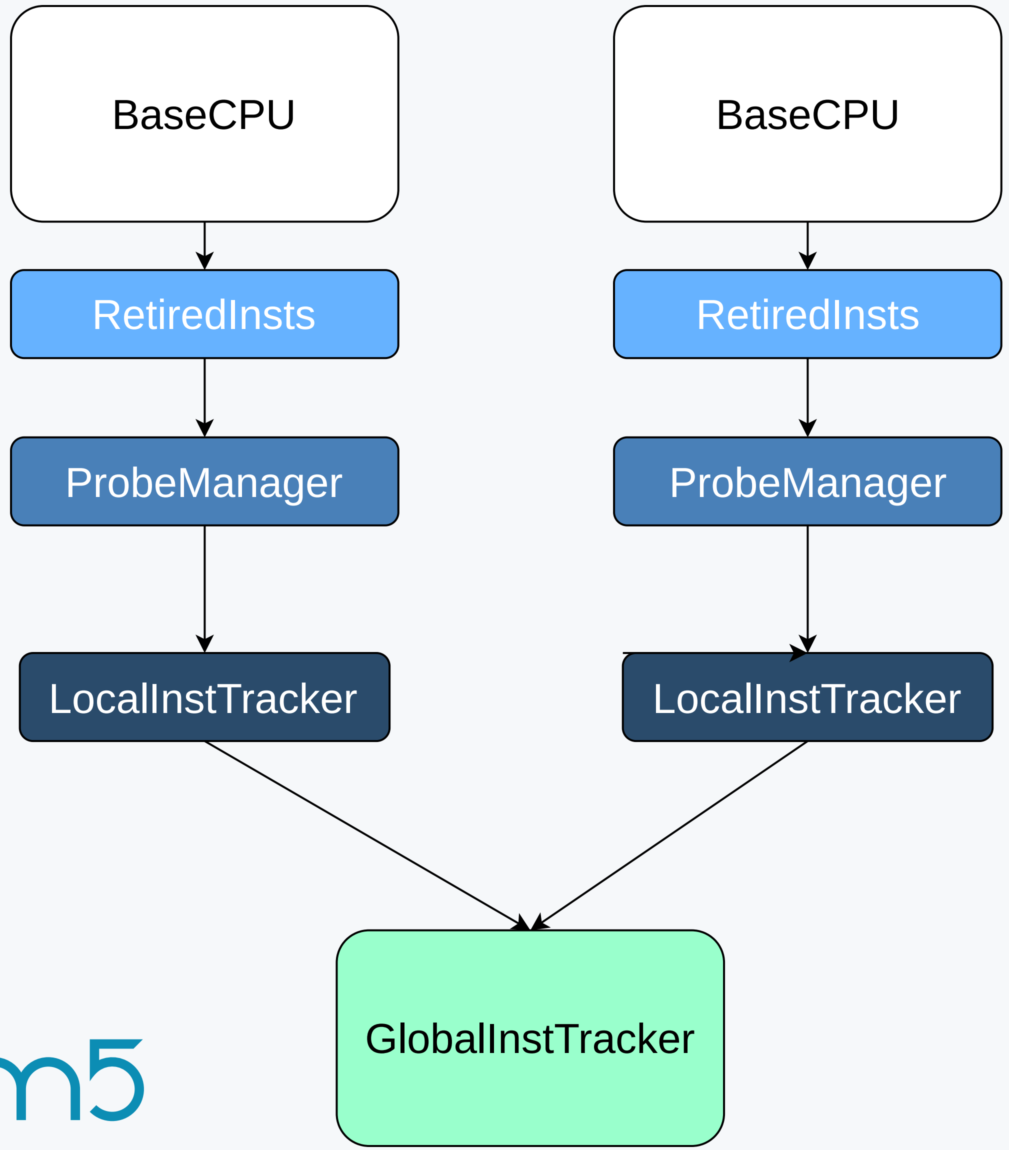
```
Global frequency set at 1000000000000 ticks per second
Running with 8 threads
Reached workbegin, now start listening for instructions
Max Inst exit event triggered
Reached 1
Fall back to simulation
Max Inst exit event triggered
Reached 2
Fall back to simulation
Max Inst exit event triggered
Reached 3
Fall back to simulation
Max Inst exit event triggered
Reached 4
Fall back to simulation
Max Inst exit event triggered
Reached 5
Fall back to simulation
Max Inst exit event triggered
Reached 6
Fall back to simulation
Max Inst exit event triggered
Reached 7
Fall back to simulation
All 8 cores have reached the max instruction threshold
Now stop listening for instructions
Reached workend
Sum: 332833500000
Simulation Done
```

01-local-inst-tracker

Congratulations! We now have our LocalInstTracker!

However, this local instruction exit event can be done with the [scheduleInstStop]() function in `BaseCPU`. Our goal is to have an instruction exit event that tracks the global committed instructions, which does not have an interface to do so easily in gem5 yet.

Since each ProbeListener can only attach to one SimObject, we can modify our LocalInstTracker to notify a global object to keep tracking all committed instructions in all ProbeListeners.



02-global-inst-tracker

All materials about this section can be found under `materials/03-Developing-gem5-models/09-extending-gem5-models/02-global-inst-tracker`.

We can create a new SimObject to help us to keep track of all ProbeListeners.
Let's start to modify the `inst_tracker.hh` by adding a new SimObject class called `GlobalInstTracker`.

```
#include "params/GlobalInstTracker.hh"
class GlobalInstTracker : public SimObject
{
  public:
    GlobalInstTracker(const GlobalInstTrackerParams &params);
}
```

02-global-inst-tracker

Since all the counting and threshold checking will be done by the `GlobalInstTracker`, let's move all the related variables and functions to the `GlobalInstTracker`.

```
  private:
  uint64_t instCount;
  uint64_t instThreshold;

public:
  void changeThreshold(uint64_t newThreshold) {
    instThreshold = newThreshold;
  }
  void resetCounter() {
    instCount = 0;
  }
  uint64_t getThreshold() const {
    return instThreshold;
  }
```

02-global-inst-tracker

So our `LocalInstTracker` now should only be like the following. Note that it has an pointer to a `GlobalInstTracker`. This is how we can notify the `GlobalInstTracker` from the `LocalInstTracker`.

```
class LocalInstTracker : public ProbeListenerObject
{
  public:
    LocalInstTracker(const LocalInstTrackerParams &params);
    virtual void regProbeListeners();
    void checkPc(const uint64_t& inst);
  private:
    typedef ProbeListenerArg<LocalInstTracker, uint64_t>
      LocalInstTrackerListener;
    bool listening;
    GlobalInstTracker *globalInstTracker;

  public:
    void stopListening();
    void startListening() {
      listening = true;
      regProbeListeners();
    }
};
```

02-global-inst-tracker

Now, we need to decide how the `GlobalInstTracker` handles the notification from the `LocalInstTracker`.
We want it to count the number of global committed instruction, check if it reaches the threshold, and raise an exit event if it does.
Therefore, in `inst_tracker.hh`, let's add a `checkPc` function to the `GlobalInstTracker` too.

```
void checkPc(const uint64_t& inst);
```

In `inst_tracker.cc`, let's define it as

```
void
GlobalInstTracker::checkPc(const uint64_t& inst)
{
    instCount ++;
    if (instCount >= instThreshold) {
        exitSimLoopNow("a thread reached the max instruction count");
    }
}
```

gem5

02-global-inst-tracker

Now, we need to modify the original `checkPc` function for the `LocalInstTracker` to notify the `GlobalInstTracker`

```
void
LocalInstTracker::checkPc(const uint64_t& inst)
{
    globalInstTracker->checkPc(inst);
}
```

Don't forget to change the constructor of the `LocalInstTracker`

```
LocalInstTracker::LocalInstTracker(const LocalInstTrackerParams &p)
    : ProbeListenerObject(p),
      globalInstTracker(p.global_inst_tracker),
      listening(p.start_listening)
{}
```

02-global-inst-tracker

We are almost done with C++ part. Let's don't forget about the `GlobalInstTracker`'s constructor in the `inst_tracker.cc`

```
GlobalInstTracker::GlobalInstTracker(const GlobalInstTrackerParams &p)
    : SimObject(p),
      instCount(0),
      instThreshold(p.inst_threshold)
{}
```

After this, we need to modify the `InstTracker.py` for the new `GlobalInstTracker` and the modified `LocalInstTracker`

```python
from m5.objects import SimObject
from m5.objects.Probe import ProbeListenerObject
from m5.params import *
from m5.util.pybind import *


class GlobalInstTracker(SimObject):
    type = "GlobalInstTracker"
    cxx_header = "cpu/probes/inst_tracker.hh"
    cxx_class = "gem5::GlobalInstTracker"

    cxx_exports = [
        PyBindMethod("changeThreshold"),
        PyBindMethod("resetCounter"),
        PyBindMethod("getThreshold")
    ]

    inst_threshold = Param.Counter("The instruction threshold to trigger an"
                                                                " exit event")
```

```python
class LocalInstTracker(ProbeListenerObject):
    type = "LocalInstTracker"
    cxx_header = "cpu/probes/inst_tracker.hh"
    cxx_class = "gem5::LocalInstTracker"

    cxx_exports = [
        PyBindMethod("stopListening"),
        PyBindMethod("startListening")
    ]

    global_inst_tracker = Param.GlobalInstTracker("Global instruction tracker")
    start_listening = Param.Counter(True, "Start listening for instructions")
```

02-global-inst-tracker

Finally, the gem5/src/cpu/probes/SConscript

```
SimObject(
    "InstTracker.py",
    sim_objects=["GlobalInstTracker", "LocalInstTracker"],
)
Source("inst_tracker.cc")
```

Let's build gem5 with our new `GlobalInstTracker`!

```
cd gem5
scons build/X86/gem5.fast -j$(nproc)
```

02-global-inst-tracker

There is a simple SE script in materials/03-Developing-gem5-models/09-extending-gem5-models/02-global-inst-tracker/simple-sim.py.

We can test our `GlobalInstTracker` with it using the command

```
cd /workspaces/2024/materials/03-Developing-gem5-models/09-extending-gem5-models/02-global-inst-tracker
/workspaces/2024/gem5/build/X86/gem5.fast -re --outdir=simple-sim-m5out simple-sim.py
```

This script runs the same workload we did in 01-local-inst-tracker, but with the `GlobalInstTracker` setup.

02-global-inst-tracker

It creates a `GlobalInstTracker` and when each `LocalInstTracker` attaches to the core, it passes itself as a reference to the `global_inst_tracker` parameter

```
from m5.objects import LocalInstTracker, GlobalInstTracker

global_inst_tracker = GlobalInstTracker(
    inst_threshold = 100000
)
all_trackers = []
for core in processor.get_cores():
    tracker = LocalInstTracker(
        global_inst_tracker = global_inst_tracker,
        start_listening = False,
    )
    core.core.probeListener = tracker
    all_trackers.append(tracker)
```

02-global-inst-tracker

We start to listen when workbegin is raised, then exit the simulation after 100,000 instructions are committed accumulatively by all cores.
Also, we reset the stats at workbegin, so we can verify if the `GlobalInstTracker` actually did its job.

If the simulation finished, we can count the stats.
There is a helper python file [materials/03-Developing-gem5-models/09-extending-gem5-models/02-global-inst-tracker/count_commited_inst.py]() for us to easily calculate the total committed instructions by all 8 cores.

Let's run it with

```
python3 count_commited_inst.py
```

We should see the following if the `GlobalInstTracker` works.

```
Total committed instructions: 100000
```

Summary

The ProbePoint is a useful tool to profile or add helper features for our simulation without adding too much to the components' codebase.