Trace-driven simulation of multithreaded applications in gem5

Gilles SASSATELLI
sassatelli@lirmm.fr

Alejandro NOCUA, Florent BRUGUIER, Anastasiia BUTKO

Cambridge, UK – September 11, 2017
Mont-Blanc 1, 2 & 3 projects (FP7, H2020)

- Getting ARM technology ready for HPC: HW, SW & Apps
- Advances in energy efficiency towards Exascale

Initial effort: using gem5 for performance prediction (2011)

- STE Nova A9500 SoC (dual-core Cortex A9)
- Fed publicly available parameters into a gem5 FS model
- 1.5% - 18% error, due to rough DRAM model and interconnect

Background motivations
Background motivations cont’d

- **Calibration against real hardware**
  - Using gem5 for performance prediction
  - And McPAT for power estimation

- Then onto exploring fancy architecture configurations
  - Heterogeneous single-ISA multicores à la big.LITTLE
  - Assymetric, 3 levels of heterogeneity etc.
Not ready for manycores, too slow!
- 1K-1M (simulated) IPS
- Scales bad with system size
- Already much better than RTL though

Trading speed for accuracy? Any sweet spot?
Background motivations cont’d

- Not ready for manycores, too slow!
  - 1K-1M (simulated) IPS
  - Scales bad with system size
  - Already much better than RTL though

- Trading speed for accuracy?

![Diagram showing the trade-off between accuracy and speed.](image)

- High accuracy, low speed: Cycle-accurate
- Low accuracy, high speed: Cycle-approximate
- Distributed simulation
- Trace-driven simulation
- JIT simulation

1 core FS simulation

- CPU: 69% (most time)
- Cache: 15%
- Interconnect: 14%
- Memory: 2%

---

Background motivations cont’d

- Mobile/Embedded technology
- Power efficiency
- Low cost
- Performance?

- Samsung Exynos 5 Octa 9 GFLOPS
- Supercomputer node Intel Xeon E5 300 GFLOPS

Idea
- Multicore Manycore Design Space Exploration
  - Simulation
  - State-of-the-art
~ 70% simulation effort goes into CPU

- Abstracting away CPU cores sounds like a good idea
- Between 2 consecutive L1 cache misses (in-order) CPU cores perform "consistently"
SimMATE: 2-stage process

- Trace collection: tracing only L1 miss related transactions
- Trace replay: Using trace injectors that initiate transactions as previously recorded
SimMATE for faster DSE

- **Trace collection = freezing**
  - CPU parameters alongside application SW
  - Private caches sizes, speed etc.

- **Trace replay allows exploring the rest**
  - L2 size, policy etc.
  - Interconnect type & speed
  - Main memory speed

---

**Trace Collection**

- Core 0
- Core n
- Interconnect
- Memory

**Trace Simulation**

- TI 0
- TI n
- Interconnect
- Memory
- L2

*1 time

*Multiple times*
Trace replay performs event (re-) scheduling

- Simple «time shifting» approach
- Maintaining constant compute phases
**SimMATE Benchmarking**

- **Tuning DRAM latency**
  - Collection performed with 30ns
  - TD simulation from 5ns to 55ns
  - FS used as reference

---

**Execution time error [%]**

- **MJPEG**
- **FFT**

<table>
<thead>
<tr>
<th>Execution time error [%]</th>
<th>5ns</th>
<th>15ns</th>
<th>30ns</th>
<th>45ns</th>
<th>55ns</th>
</tr>
</thead>
<tbody>
<tr>
<td>5ns</td>
<td>0,6%</td>
<td>0,5%</td>
<td>0,1%</td>
<td>0,4%</td>
<td>0,4%</td>
</tr>
<tr>
<td>15ns</td>
<td>2,8%</td>
<td>4,5%</td>
<td>5,8%</td>
<td>5,1%</td>
<td>6%</td>
</tr>
<tr>
<td>30ns</td>
<td>0,6%</td>
<td>0,5%</td>
<td>0,1%</td>
<td>0,4%</td>
<td>0,4%</td>
</tr>
<tr>
<td>45ns</td>
<td>5,8%</td>
<td>5,1%</td>
<td>6%</td>
<td>5,1%</td>
<td>6%</td>
</tr>
<tr>
<td>55ns</td>
<td>5,8%</td>
<td>5,1%</td>
<td>6%</td>
<td>5,1%</td>
<td>6%</td>
</tr>
</tbody>
</table>
SimMATE Benchmarking

- Tuning L2 size
  - Collection performed without L2
  - TD simulation from 0 to 16MB L2
  - Errors originate from Cold-start bias / cache warmup

**Execution time error [%]**

Traces: ET, ET+init, ET+OS boot

<table>
<thead>
<tr>
<th>L2 cache size</th>
<th>Original</th>
<th>256kB</th>
<th>1MB</th>
<th>8MB</th>
<th>16MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution time error [%]</td>
<td>0.1%</td>
<td>15%</td>
<td>8%</td>
<td>7.5%</td>
<td>18%</td>
</tr>
</tbody>
</table>
Having these traces collected makes it easy to:

- Perform « Trace replication » i.e. emulate more CPU cores for scalability study
- This corresponds to weak scaling experiments, i.e. per-core workload remains same
Yet synchronizations must be accounted for!

- Using whatever API: POSIX threads, OpenMP 3.0 …
- Approach: embed synchronizations into traces
- Have an arbiter that takes care of locking (when barrier reached) and unlocking TIs
Limitation: in-order only!

- And most ARM AP are OoO (Out-of-Order)
  - Meaning multiple outstanding memory transactions
  - The assumption of constant time btw. 2 misses does not hold

- big-LITTLE & other heterogeneous friends everywhere
  - And there microarchitecture details cannot be overlooked
Modeling micro-architecture timing & dependencies

- Tracing with O3 model + probes, without L2 cache
- Replay done in a smart « elastic » fashion

Elastic Traces: Trace-driven simulation for OoO

http://gem5.org/TraceCPU
Smart TraceCPU

- Updating a dependency graph pushing ready instructions into a queue for issue

Elastic Traces: Trace-driven simulation for OoO

http://gem5.org/TraceCPU
SimMATE + Elastic Traces = ElasticSimMATE

- Enabling both OoO + multithreaded applications
- Key: embed synchronization information @ tracing time.
Proper tracing of synchronizations

- API-dependant: OpenMP 3.x
- Tracing whenever entering or leaving parallel region, barrier etc.

```cpp
#pragma omp parallel
for(i=0;i<n;i++) { /* do_some_work */ }
```

```cpp
for(i=0;i<n;i++) {
    OMP_runtime_call()
    /* do_some_work */
}
```

```
<table>
<thead>
<tr>
<th>Tick</th>
<th>PC</th>
<th>th_id</th>
<th>type</th>
<th>IC</th>
<th>DC</th>
</tr>
</thead>
<tbody>
<tr>
<td>389178</td>
<td>177216</td>
<td>0</td>
<td>1</td>
<td>1081157</td>
<td>165738</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
ESM flow wrapup

- Using BSC *Mercurium compiler* / *Nanos++* runtime
- Tweaked runtime such that custom m5 pseudo instructions produce *trace records*

```c
#pragma omp parallel
for (i=0; i<n; i++) {
    /* do_something */
}
```

---

**Diagram:**
- **Unmodified sources**
- **gem5 Trace Collection**
- **Generated Traces**
- **gem5 Trace Replay**

**Diagram Notes:**
- **App Binaries**
  ```c
  #pragma omp parallel
  for (i=0; i<n; i++) {
      nanos_create_team();
      /* do_something */
      nanos_end_team();
  }
  ```

**Legend:**
- Instruction
- Dependency
- Synchronization

**Thread creation & such**
- `m5_trace(TYPE, th_id)`
Two main use cases:
- Fast parameter exploration
- Scalability study: « trace replication »

Speedup & accuracy?
- Experiments on Rodinia application kernels

---

Benchmarking

<table>
<thead>
<tr>
<th>L2 Cache Size</th>
<th>FS</th>
<th>ET</th>
<th>ESM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0MB</td>
<td>2.5</td>
<td>2</td>
<td>2.2</td>
</tr>
<tr>
<td>16kB</td>
<td>2</td>
<td>2</td>
<td>2.2</td>
</tr>
<tr>
<td>128kB</td>
<td>2</td>
<td>2</td>
<td>2.2</td>
</tr>
<tr>
<td>512kB</td>
<td>2</td>
<td>2</td>
<td>2.2</td>
</tr>
<tr>
<td>1MB</td>
<td>2</td>
<td>2</td>
<td>2.2</td>
</tr>
<tr>
<td>2MB</td>
<td>2</td>
<td>2</td>
<td>2.2</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>L2 Cache Size</th>
<th>Error Percentage [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0MB</td>
<td>-4</td>
</tr>
<tr>
<td>16kB</td>
<td>0</td>
</tr>
<tr>
<td>128kB</td>
<td>2</td>
</tr>
<tr>
<td>512kB</td>
<td>4</td>
</tr>
<tr>
<td>1MB</td>
<td>6</td>
</tr>
<tr>
<td>2MB</td>
<td>8</td>
</tr>
</tbody>
</table>

---

1 core HOTSPOT
Two main use cases:
- Fast parameter exploration
- Scalability study: « trace replication »

Speedup & accuracy?
- Experiments on Rodinia application kernels

BENCHMARKING

1 core KMEANS

<table>
<thead>
<tr>
<th>L2 Cache Size</th>
<th>Execution Time [ms]</th>
<th>Error Percentage [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0MB</td>
<td>16kB</td>
<td>128kB</td>
</tr>
<tr>
<td>1MB</td>
<td>2MB</td>
<td>1MB</td>
</tr>
</tbody>
</table>

FS vs ET
FS vs ESM
Two main use cases:

- Fast parameter exploration
- Scalability study: « trace replication »

Speedup & accuracy?

- Experiments on Rodinia application kernels
Two main use cases:

- Fast parameter exploration
- Scalability study: « trace replication »

Speedup & accuracy?

- Experiments on Rodinia application kernels

---

**Benchmarking**

- **L2 Cache Miss Rate [%]**
  - **L2 Cache Size**
    - 16kB, 128kB, 512kB, 1MB, 2MB

**L2 Cache Overall Miss Latency [Cycles]**

- **L2 Cache Size**
  - 16kB, 128kB, 512kB, 1MB, 2MB
Two main use cases:
- Fast parameter exploration
- Scalability study: « trace replication »

Speedup & accuracy?
- Experiments on Rodinia application kernels

128 cores  KMEANS
Scalability analysis has limits

- Requires additional features s.a. **address offsetting**
- **Weak scaling** only (replicated per-core workloads)

Programming models moving from loops to tasks

- OpenMP 4.0, OmpSs
- Still pragma-based
- More parallelisms available at run-time … more opportunities for smart job scheduling

BSC OmpSs: Cholesky decomposition
**Unbinding traces from cores**

- One **trace** per **Task**, not per core!
- Assign traces to cores by emulating runtime behaviour in trace replay
- This is real strong scaling

**Trace collection**

---

**Trace simulation**

- Ready Task queue
- Task dependencies
- Task 1 trace
- Task 2 trace
- Task N trace
- Interconnect
- Memory

---

Scheduler / runtime (emulated)
Current ESM prototype
- ~5x - 10x speedup for low core count, probably more for tens / thousands
- Nice solution for fast DSE
- Remaining accuracy issues for some applications
  - Common to ESM & Elastic Traces
  - Under investigation with ARM

Use cases
- Exploration of memory subsystem
- Some microarchitecture parameters (Elastic Traces)
- ...

Future directions
- Ruby compatibility
- Could be combined with other initiatives (dist-gem5)
- Can be extended to other PM / APIs (Tasking, MPI…)

Conclusion
http://montblanc-project.eu