Trace-driven simulation of multithreaded applications in gem5
Gilles Sassatelli, Alejandro Nocua, Florent Bruguier, Anastasiia Butko

To cite this version:

HAL Id: lirmm-01723755
https://hal-lirmm.ccsd.cnrs.fr/lirmm-01723755
Submitted on 18 Jun 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Trace-driven simulation of multithreaded applications in gem5

Gilles SASSATELLI
sassatelli@lirmm.fr

Alejandro NOCUA, Florent BRUGUIER, Anastasiia BUTKO

Cambridge, UK – September 11, 2017
Background motivations

- 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

Sources of error
- Specification
  - Memory & Interconnect

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.

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.
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? Any sweet spot?

![](chart.png)
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](image)

1 core FS simulation

- CPU: 69%
- Cache: 14%
- Interconnect: 15%
- Memory: 2%

Accuracy vs Speed

- RTL: Cycle-accurate
- Distributed: Trace-driven
- JIT

Ideal

10s MIPS
Trace-driven simulation principle

- ~ 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 »

![Simulation Diagram](image-url)
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

**Diagram Notes:**
- Hits & Misses
- Memory traces
- 1 time
- Multiple times
SimMATE for faster DSE cont’d

- Trace replay performs event (re-) scheduling
  - Simple « time shifting » approach
  - Maintaining constant compute phases

![Diagram showing memory and interconnect timing with delays and communication](image)
SimMATE Benchmarking

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

![Graph showing execution time error in percentages for TD Simulation and FS Simulation across different latency values. The graph compares MJPEG and FFT execution times.](image)

**Execution time error [%]**
- MJPEG: 0.6%, 0.5%, 0.1%, 0.4%, 0.4%
- FFT: 2.8%, 4.5%, 5.8%, 5.1%, 6%

---

**Trace Collection**
- TD Simulation: 30 ns
- FS Simulation: 5 ns, 15 ns, 45 ns, 55 ns

**Interconnect**
- Memory

---

**Figure Legend:**
- TI0, TI1
- ES, FS
- Interconnect
- Memory
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>0,1%</td>
<td>8%</td>
<td>7,5%</td>
<td>1%</td>
<td>1%</td>
<td>1%</td>
</tr>
<tr>
<td>15%</td>
<td>14%</td>
<td>10%</td>
<td>8%</td>
<td>8%</td>
<td>7%</td>
</tr>
<tr>
<td>18%</td>
<td>15%</td>
<td>10%</td>
<td>8%</td>
<td>8%</td>
<td>7%</td>
</tr>
</tbody>
</table>

SimMATE Benchmarking
Multithreaded applications

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

![Trace Collection and Simulation Diagram]

Number of Cache Misses

Core 0/Thread 0

Core 1/Thread 1
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
Elastic Traces: Trace-driven simulation for OoO

- Modeling micro-architecture timing & dependencies
  - Tracing with O3 model + probes, without L2 cache
  - Replay done in a smart « elastic » fashion

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.
ESW wrapup

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

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

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

Unmodified sources → gem5 Trace Collection → Generated Traces → gem5 Trace Replay

- App Binaries
  - #pragma omp parallel
  - for(i=0;i<n;i++) {
    /* do_something */
  }

- Nanos++ runtime
  - #pragma omp parallel{
    for(i=0;i<n;i++){
      nanos_create_team();
      /* do_something */
      nanos_end_team();
    }
  }

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

Speedup & accuracy?
- Experiments on Rodinia application kernels

---

Benchmarking

- Execution Time [ms]
- Error Percentage [%]
- L2 Cache Size
- 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>512kB</td>
</tr>
<tr>
<td>2MB</td>
<td>1MB</td>
<td>1MB</td>
</tr>
<tr>
<td>2MB</td>
<td>2MB</td>
<td>2MB</td>
</tr>
</tbody>
</table>

---

![Graph showing execution time and error percentage across different L2 cache sizes.](image-url)
Two main use cases:

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

Speedup & accuracy?

- Experiments on Rodinia application kernels
**Benchmarking**

- **Two main use cases:**
  - Fast parameter exploration
  - Scalability study: « trace replication »

- **Speedup & accuracy?**
  - Experiments on Rodinia application kernels

---

**L2 Cache Miss Rate [%]**

<table>
<thead>
<tr>
<th>L2 Cache Size</th>
<th>FS</th>
<th>ESM</th>
</tr>
</thead>
<tbody>
<tr>
<td>16kB</td>
<td>70</td>
<td>60</td>
</tr>
<tr>
<td>128kB</td>
<td>60</td>
<td>50</td>
</tr>
<tr>
<td>512kB</td>
<td>50</td>
<td>40</td>
</tr>
<tr>
<td>1MB</td>
<td>40</td>
<td>30</td>
</tr>
<tr>
<td>2MB</td>
<td>30</td>
<td>20</td>
</tr>
</tbody>
</table>

---

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

<table>
<thead>
<tr>
<th>L2 Cache Size</th>
<th>FS</th>
<th>ESM</th>
</tr>
</thead>
<tbody>
<tr>
<td>16kB</td>
<td>1.6E+10</td>
<td>1.4E+10</td>
</tr>
<tr>
<td>128kB</td>
<td>1.4E+10</td>
<td>1.2E+10</td>
</tr>
<tr>
<td>512kB</td>
<td>1.2E+10</td>
<td>1.0E+10</td>
</tr>
<tr>
<td>1MB</td>
<td>1.0E+10</td>
<td>8.0E+9</td>
</tr>
<tr>
<td>2MB</td>
<td>8.0E+9</td>
<td>6.0E+9</td>
</tr>
</tbody>
</table>
Two main use cases:
- Fast parameter exploration
- Scalability study: « trace replication »

Speedup & accuracy?
- Experiments on Rodinia application kernels

128 cores  KMEANS

### Benchmarking

**Execution Time [s]**

<table>
<thead>
<tr>
<th>Number of Cores</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blackscholes</td>
<td>200</td>
<td>400</td>
<td>600</td>
<td>800</td>
<td>1000</td>
<td>1200</td>
<td>1400</td>
<td>1600</td>
</tr>
<tr>
<td>Hotspot</td>
<td>100</td>
<td>200</td>
<td>300</td>
<td>400</td>
<td>600</td>
<td>800</td>
<td>1000</td>
<td>1200</td>
</tr>
</tbody>
</table>

**Simulation Time [min]**

<table>
<thead>
<tr>
<th>Number of Cores</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blackscholes</td>
<td>10</td>
<td>20</td>
<td>40</td>
<td>80</td>
<td>160</td>
<td>320</td>
<td>640</td>
<td>1280</td>
</tr>
<tr>
<td>Hotspot</td>
<td>10</td>
<td>20</td>
<td>40</td>
<td>80</td>
<td>160</td>
<td>320</td>
<td>640</td>
<td>1280</td>
</tr>
</tbody>
</table>
Perspectives

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**

Scheduler / runtime (emulated)

Ready Task queue

Scheduler
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…)

http://montblanc-project.eu