Trace-driven Simulation of Multithreaded Applications

Alejandro Rico, Alejandro Duran, Felipe Cabarcas
Yoav Etsion, Alex Ramirez and Mateo Valero
Multithreaded applications and trace-driven simulation

- Most computer architecture research employ execution-driven simulation tools.
- Trace-driven simulation cannot capture the dynamic behavior of multithreaded applications.

Scenario 1

Core 0
- acquire_lock
- check
- acquired

Critical section
- release lock

Core 1
- acquire_lock
- check
- acquired

Scenario 2

Core 0
- acquire_lock
- check
- acquired

Critical section
- wait!

Core 1
- acquire_lock
- check
- acquired

Critical section
- release lock
Trace-driven simulation has advantages

- Avoid computational requirements of simulated applications.
  - Memory footprint.
  - Disk space for input sets.
- Simulate applications with non-accessible sources, but accessible traces.
  - Confidential/restricted applications.
- Lower modeling complexity.
  - Different host\(^1\) and target\(^2\) ISAs / endianness.

- **Problem:** How to appropriately simulate multithreaded applications using traces?

\(^1\) *Host*: system where the simulator executes.
\(^2\) *Target*: system modeled in the simulator.
Targeting applications with decoupled execution

- Distinguish the user code (sequential code sections) from parallelism-management operations (parops).

<table>
<thead>
<tr>
<th>Seq. code section</th>
<th>parop call</th>
<th>parop execution</th>
<th>Idle</th>
<th>Switch</th>
</tr>
</thead>
</table>

**Task-based parallel applications**

Core 0
- create task 1
- sync

Core 1
- exec task 1
- completion task 1

**Loop-based parallel applications**

Core 0
- parallel loop
- sync

Core 1
- sync

Core 2
- sync

Core 3
- sync
How traces are collected (I)
How traces are collected (II)

- Capture traces for sequential code sections.
- Execution is independent of the environment.

20: sub r15, r12, r13
24: store r35, r15 (0x7e6a0)
28: sub r3, r31, r4
2c: load r21, r7 (0x80a88)
30: addi r3, r3
34: beq r3 (next_i: 7C)
7c: mul r32, r8, r9
80: mul r33, r10, r11
84: mul r34, r12, r13
88: store r32, r17 (0x7f280)
8c: store r33, r18 (0x7f284)
How traces are collected (III)

- Capture traces for sequential code sections.
  - Execution is independent of the environment.
- Capture calls to \textit{parops}.
  - Specific \textit{parop call} events are included in the trace.

![Diagram showing the call to parallel loop and calls to sync between cores.](image-url)
How traces are collected (IV)

- Capture traces for sequential code sections. Execution is independent of the environment.
- Capture calls to parops. Specific parop call events are included in the trace.
- Do not capture the execution of parops. Execution depends on the environment.

Call to parallel loop

Calls to sync
Simulation framework

- Trace-driven simulator simulates *sequential code sections*.
- The dynamic component executes parops at simulation time.
  - Includes the implementation of parops.
- Parops are exposed to the simulator through the parop interface.
- The architecture state is exposed to the dynamic component through the target architecture interface.
Sample implementation: TaskSim – NANOS++

- Parops are exposed to the simulator through the parop interface
  - It includes operations for task management and synchronization.
- The architecture state and associated actions are exposed to NANOS++ through the architecture-dependent module.
  - NANOS++ can alter the simulator state and manage the simulated thread according to the decisions based on the target architecture.
OmpSs application example

- Cholesky factorization.
- Tasks are spawned on `pragma task` annotations.
- Inputs and outputs are specified for automatic dependence resolution.

```c
float A[N][N][M][M]; // NxN blocked matrix,  
// with MxM blocks
for (int j = 0; j<N; j++) {
    for (int k = 0; k<j; k++)
        for (int i = j+1; i<N; i++)
            #pragma task input(a, b) inout(c)
                sgemm_t(A[i][k], A[j][k], A[i][j]);

    for (int i = 0; i<j; i++)
        #pragma task input(a) inout(b)
            ssyrk_t(A[j][i], A[j][j]);

    #pragma task inout(a)
    spotrf_t(A[j][j]);

    for (int i = j+1; i<N; i++)
        #pragma task input(a) inout(b)
            strsm_t(A[j][j], A[i][j]);
}
Traces for OmpSs applications

- Sequential code sections correspond to tasks.
- One trace for the main task
  - The thread starting the program execution at the `main` function
- One trace for each task
- Information for each function call
  - E.g., for task creation it needs the task id and the input and output data addresses and sizes
1. Simulation starts the *main* task.

![Diagram showing simulation example]

- TaskSim
- Parop interface
- Architecture dependent operations
- NANOS++

Core 0

initialization

Core 1
2. On a create task event, it calls the interface in the Parop interface.
3. That triggers the creation of the task in Nanos++. 

![Diagram showing TaskSim, Parop interface, and NANOS++ with Core 0 and Core 1 with initialization and create task 1]
4. Returns control to TaskSim. Core 1 takes task 1 for simulation.
Simulation example (V)

5. TaskSim resumes simulation, and Core 1 starts simulating task 1.
6. On create task 2 event, TaskSim calls the runtime again.
7. NANOS++ creates task 2, and returns control to TaskSim.
8. When Core 1 finishes the execution of task 1, starts task 2.

\begin{itemize}
    \item Initialization
    \item Create task 1
    \item Create task 2
    \item Task wait
    \item Exec task 1
    \item Exec task 2
\end{itemize}
10. All tasks are finished, and TaskSim continues the main task simulation.
Task generation scheme scalability

- Task generation (green) on the main task limits scalability (on the left).
- Parallelization of task generation (on the right) is crucial to avoid this bottleneck.
Coverage and opportunities

- Appropriate for high-level programming models.
  - OpenMP, OmpSs, Cilk,…
  - Mixing scheduling/synchronization and application code is limited.
  - Runtime system can be used as the *dynamic component*.

- Not suitable for:
  - Scheduling dependent on user code (user-guided scheduling).
  - Computation based on *random* values (e.g., Monte Carlo algorithms).

- Runtime system development:
  - Scheduling policies.
  - Overall efficiency optimizations.
  - For future machines before the actual hardware is available.

- Runtime software/hardware co-design.
  - Hardware support for runtime system.
Conclusions

- We propose a novel trace-driven simulation methodology for multithreaded applications.

- The methodology is based on distinguishing:
  - Application intrinsic behavior (user code).
  - Parallelism-management operations (parops).

- It allows to properly simulate different architecture configurations:
  - With different numbers of cores.
  - Using a single trace per application.

- It provides a framework not only for architecture exploration but also for runtime system development.