Criticality Stacks: Identifying Critical Threads in Parallel Programs using Synchronization Behavior

Kristof Du Bois, Stijn Eyerman, Jennifer B. Sartor, Lieven Eeckhout
ELIS Department, Ghent University, Belgium

ISCA – 2013
Motivation

- **Critical threads**: Threads whose performance is determinative of the program performance as a whole

- Due to synchronization, certain threads
  - Make other thread wait because they hold a lock
  - Or they have yet to reach to barrier (due to their slowness)

- How to identify a critical thread in a multi-threaded execution
  - Think of some solution and it could be a valuable contribution
  - Their solution is based on *synchronization behavior*
Thread Criticality Metric

• Thread criticality metric combines two things
  • How much time a thread is performing useful work, and
  • How many co-running threads are waiting

• They show that thread criticality can be calculated online
  • With modest hardware addition and low overhead

• Use the criticality metric to create criticality stacks

• Break total execution time into each thread’s criticality component
  • Allowing for easy visual analysis of parallel imbalance
Criticality Metric’s Application

• Scale the frequency of the most critical thread and show it achieves the largest performance improvement

• Applicability of criticality stacks by using them to perform three types of optimizations:
  • Program analysis to remove parallel bottlenecks
  • Dynamically identifying the most critical thread and accelerating it using frequency scaling to improve performance
  • Accelerating only the most critical thread allows for targeted energy reduction
Critical & Non-critical Threads

- Threads identified as critical can be targeted for performance optimization
  - Through software re-design or through hardware techniques.

- Speeding up critical threads can speed up the whole program.

- Or inversely, slowing down non-critical threads has almost no impact on performance,
  - Enables a more energy-efficient execution.
Speeding up a thread

• Speeding up a thread can be done
  • By migrating it to a faster core in a heterogeneous multicore
  • By temporarily boosting the frequency of the core it executes on
  • By raising the fetch priority of that thread in an SMT context
  • By allowing more task stealing from this thread in a task stealing context, etc.

• Speeding up can also be done by using the decoupled look-ahead for the critical thread only

• We were trying to devise a CMP version of DLA
  • If the insight of this paper is believed then we only need one extra core to speedup the critical thread and determine it online
Example: Criticality Stack

Figure 1: BFS’s criticality stack and total program speedups from accelerating the identified critical and non-critical threads.
Definition: Thread Criticality

Criticality stack calculation

\[ C_j = \sum_{i=0}^{N-1} \left\{ \begin{array}{ll}
\frac{t_i}{r_i}, & \text{if } j \in R_i \\
0, & \text{if } j \notin R_i
\end{array} \right. \]

Thread 0: \( \frac{t_0}{4} + \frac{t_6}{4} + t_7 = 6.5 \)
Thread 1: \( \ldots = 5 \)
Thread 2: \( \ldots = 5 \)
Thread 3: \( \ldots = 5.5 \)

Thread 0 is determined to be most critical thread

\[ \sum_{j=0}^{n-1} C_j = T. \]

Figure 2: Criticality calculation example.
Identifying Running Thread

- Two main causes why a thread is not performing useful work
  - Either it is scheduled out by the operating system, or
  - It is spinning (using a waiting loop, constantly checking the synchronization variable).

- The operating system can easily communicate when it schedules threads in and out.

- Spinning is more difficult to detect, since the thread is executing instructions, albeit useless ones

- Either software or hardware can detect spinning
  - Software solutions involve adding extra instructions that denote spinning threads.
Hardware for Online Criticality

A represents active thread

Figure 3: Hardware device for online criticality calculation ('A' is the active bit per thread).
Experimental Setup

8-core and 16-core running 8 and 16 threads
Each core is 4-way, Out-of-order superscalar
Private L1 and L2 caches, and last level shared L3 cache

<table>
<thead>
<tr>
<th>Suite</th>
<th>Benchmark</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPLASH-2</td>
<td>Cholesky</td>
<td>tk29.O</td>
</tr>
<tr>
<td></td>
<td>FFT</td>
<td>4,194,304 points</td>
</tr>
<tr>
<td></td>
<td>FMM</td>
<td>32,768 particles</td>
</tr>
<tr>
<td></td>
<td>Lu cont.</td>
<td>1024×1024 matrix</td>
</tr>
<tr>
<td></td>
<td>Lu non-cont.</td>
<td>1024×1024 matrix</td>
</tr>
<tr>
<td></td>
<td>Ocean cont.</td>
<td>1026×1026 ocean</td>
</tr>
<tr>
<td></td>
<td>Ocean non-cont.</td>
<td>1026×1026 ocean</td>
</tr>
<tr>
<td>PARSEC</td>
<td>Canneal</td>
<td>Simmedium</td>
</tr>
<tr>
<td></td>
<td>Facesim</td>
<td>Simmedium</td>
</tr>
<tr>
<td></td>
<td>Fluidanimate</td>
<td>Simmedium</td>
</tr>
<tr>
<td></td>
<td>Streamcluster</td>
<td>Simmedium</td>
</tr>
<tr>
<td>Rodinia</td>
<td>BFS</td>
<td>1,000,000 nodes</td>
</tr>
<tr>
<td></td>
<td>Srad</td>
<td>2048×2048 matrix</td>
</tr>
<tr>
<td></td>
<td>Lud_omp</td>
<td>512×512 matrix</td>
</tr>
<tr>
<td></td>
<td>Needle</td>
<td>4096×4096 matrix</td>
</tr>
</tbody>
</table>

Table 2: Evaluated benchmarks.
Evaluation: Criticality Stack

(a) Criticality stacks
Speedups

(b) Speedups

![Bar chart showing speedups for various applications and components.](chart.png)

- Cholesky
- FFT
- FMM
- Lu cont.
- Lu non-cont.
- Ocean cont.
- Ocean non-cont.
- Canneal
- Facesim
- Fluidanimate
- Streamcluster
- BFS
- Srad
- Lud_omp
- Needle

Legend:
- Largest component
- Second largest component
- Third largest component
Comparison with others

(a) 8 threads

- Maximum
- Criticality metric
- Cache misses metric
Frequency Scaling Speedup

(a) Fluidanimate

(b) BFS

(c) Lud_omp
Use 1: Software Optimization

Figure 8: Example of using criticality stacks as a guide for software optimization (BFS benchmark).
Use 2: Critical Thread Acceleration

Dynamically measure thread criticality
Over a time slice (10ms) and scale up
The frequency.

Frequency is changed from 2 GHz to 2.5 GHz

Figure 9: Results for the dynamic frequency scaling policy.
Use 3: Energy Optimization

Figure 9: Results for the dynamic frequency scaling policy.
Summary

- Identifying a critical thread in a multi-threaded execution can allow us to improve performance and optimize power
  - By speeding up the critical thread (improve performance)
  - By slowing down the non-critical thread (minimize power)

- Paper proposes a criticality metric that takes into account
  - How much useful work a thread is doing?
  - How many threads are waiting for a particular thread?

- Also proposes a criticality stack based on criticality metric to facilitate the analysis of parallel imbalance
  - Shown to perform better than previously proposed criticality metrics that are based on cache misses