Smoothing on Dynamic Concurrency Throttling

Janaína Schwarzrock\textsuperscript{1}, Hiago Mayk G. de A. Rocha\textsuperscript{1},
Arthur F. Lorenzon\textsuperscript{2}, Antonio Carlos S. Beck\textsuperscript{1}

\textsuperscript{1} Federal University of Rio Grande do Sul (UFRGS), Brazil
\textsuperscript{2} Federal University of Pampa, Brazil

Seventeenth International Workshop on Automatic Performance Tuning (iWAPT2022)
Outline

• Introduction
• Motivation
• Smoothing on DCT
• Experimental Setup
• Evaluation
• Final consideration
Outline

• Introduction
• Motivation
• Smoothing on DCT
• Experimental Setup
• Evaluation
• Final consideration
Parallel applications scalability

Some applications do not scale as the number of threads increase.

Energy-Delay Product (EDP)

\[ EDP = \text{Energy consumption} \times \text{Execution Time} \]
Different parallel regions of an application

- Usually, a parallel application has more than one parallel region
- Each parallel region may exhibit different behavior

They may have a different optimal number of threads
It lacks adaptability

Offline

Search phase: Before execution

Static

Execution of parallel regions

\[ t^* = 12 \]

Dynamic

Search phase: Before execution

\[ B = (12, 6, 8) \]

Online

Dynamic

Search phase: During execution

Stable phase

\[ B = (t_1^*, ..., t_k^*) \]
Tuning thread count approaches

**Offline**

- **Search phase:** Before execution

**Static**

- Execution timeline
- $t^* = 12$

**Dynamic**

- Execution timeline
- $B = (12, 6, 8)$

**Online**

- **Dynamic**
- **Search phase:** Before execution

- Execution timeline
- $B = (t_1^*, ..., t_k^*)$

- It lacks adaptability
- It can adapt to any changes at run-time

- Pusukuri et al. (2011); De Sensi (2016)
- Wang et al. (2016); Popov et al. (2019)
- Lee et al. (2010); Chadha et al. (2012); Suleman et al. (2008); Curtis-Maury et al. (2006, 2008); Li et al. (2010); Sridharan et al. (2014); De Sensi et al. (2016); Li and Martinez et al. (2006); Alessi et al. (2015); Lorenzon et al. (2018); Schwarzrock et al. (2020)
Tuning thread count approaches

**Static**
- Offline: Search phase: Before execution
- **Upper-bound**: $O(n)$
- **Solution timeline**: $t^* = 12$

**Dynamic**
- Offline: Search phase: Before execution
- **Upper-bound**: $O(n^k) \rightarrow$ unfeasible
- **Execution timeline**: $B = (12, 6, 8)$

**Online**
- **Dynamic**
- Search phase: During execution
- **Best-effort dynamic solution**: $B = (t^*_1, \ldots, t^*_k)$

**Analysis**
- **Before execution**
  - Static: $O(n)$
  - Dynamic: $O(n^k) \rightarrow$ unfeasible
- **During execution**
  - Static: $O(n)$ + $O(n)$ + $O(n)$
  - Dynamic: $B = (t^*_1, \ldots, t^*_k)$

**Discussion**
- A single thread count that optimizes the whole application
- A set that optimizes each parallel region individually
- An optimized thread count for the whole application

**B = Best-effort dynamic solution**

**Execution timeline**

**Institution**: Instituto de Informática UFRO
Outline

• Introduction
• Experimental Setup
• **Motivation**
• Smoothing on DCT
• Evaluation
• Final consideration
Motivation

Offline learning to get results with no learning overhead

Baseline (the default execution): Execution with the maximum number of threads

 Optimization upper-bound
Best-effort dynamic solution

Parallel applications

Platform A

Relative EDP

0.2 0.4 0.6 0.8 1.0 1.2

BT.B  CG.B  FFT  FT.C  JA  LU.B  MG.B  PO  SP.B  GMEAN

Parallel applications

Platform B

Relative EDP

0.2 0.4 0.6 0.8 1.0 1.2

BT.B  CG.B  FFT  FT.C  JA  LU.B  MG.B  PO  SP.B  GMEAN

Parallel applications
Dynamic solution is the best one.

Platform A

The dynamic solution is far from the best one.

Platform B
Motivation

**BT on machine A**

Dynamic solution is the best one

**MG on machine B**

The dynamic solution is far from the best one

When the thread count changes very often, the benefit of using the best configuration for each parallel region may not compensate for the switching cost

*Creating/destroying/migrating threads; data warm-up (memory caches warm-up, TLB misses)*
Our proposal: smoothing thread count changes

- It alleviates the switching overheads.
- Our proposal is generic and aims further to improve the optimization results of any DCT technique (offline and online).

We propose a smoothing-based strategy to minimize the thread count changes.

MG on machine B

![Graph showing thread count changes with and without smoothing]
Outline

- Introduction
- Experimental Setup
- Motivation
- **Smoothing on DCT**
- Evaluation
- Final consideration
Parallel region | Best #threads
--- | ---
08 | 08
24 | 24
12 | 12

\[ \mathcal{B} = (08, 24, 12) \]

**Weighted Moving Average (WMA)**

\[
\mathcal{Y} = (y_1, y_2, y_3, \ldots, y_m)
\]

\[
\mathcal{E} = (w_1, w_2, w_3, \ldots, w_m)
\]

\[
\bar{y}_i = \frac{(y_i w_i) + (y_{i-1} w_{i-1}) + \cdots + (y_{i-n+1} w_{i-n+1})}{w_i + w_{i-1} + \cdots + w_{i-n+1}}
\]

\[
\bar{\mathcal{Y}} = (\bar{y}_1, \bar{y}_2, \bar{y}_3, \ldots, \bar{y}_m)
\]

- A lightweight and powerful smoothing technique
The time series (thread count):
\[ \mathcal{Y} = (08, 24, 12, 08, 24, 12) \]

The weights (exec. time):
\[ \mathcal{E} = (25, 25, 5, 25, 25, 5) \]

**Weighted Moving Average (WMA)**

\[ \overline{y}_i = \frac{(y_i w_i) + (y_{i-1} w_{i-1}) + \cdots + (y_{i-n-1} w_{i-n-1})}{w_i + w_{i-1} + \cdots + w_{i-n-1}} \]

A lightweight and powerful smoothing technique.
points: \( Y = (04, 07, 06, 04, 07, 06) \)

Index best #threads

\[ \bar{Y} = (8, 12, \bar{y}_3, \ldots, \bar{y}_m) \]

Round to 6
Index 6 = 12 threads

\[ \bar{y}_2 = \frac{(7 \times 25) + (4 \times 25)}{50} = 5.5 \]

\[ \bar{y}_i = \frac{(y_i w_i) + (y_{i-1} w_{i-1}) + \cdots + (y_{i-n-1} w_{i-n-1})}{w_i + w_{i-1} + \cdots + w_{i-n-1}} \]
Outline

• Introduction
• Motivation
• Smoothing on DCT
• **Experimental Setup**
• Evaluation
• Final consideration
### Execution Environment

<table>
<thead>
<tr>
<th>Machine</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>Intel Xeon E5-2630 (Sandy Bridge) 2.3GHz</td>
<td>Intel Xeon E5-2699v4 (Broadwell) 2.2 GHz</td>
</tr>
<tr>
<td>#Sockets (#nodes)</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>#Cores per socket</td>
<td>6 (2-way SMT)</td>
<td>22 (2-way SMT)</td>
</tr>
<tr>
<td>#Threads total</td>
<td>24</td>
<td>88</td>
</tr>
<tr>
<td>L1 cache (private)</td>
<td>12 x 32KB</td>
<td>44 x 32KB</td>
</tr>
<tr>
<td>L2 cache (private)</td>
<td>12 x 256KB</td>
<td>44 x 256KB</td>
</tr>
<tr>
<td>L3 cache (shared)</td>
<td>2 x 15MB</td>
<td>2 x 55MB</td>
</tr>
<tr>
<td>RAM Memory</td>
<td>2 x 16GB</td>
<td>2 x 128GB</td>
</tr>
</tbody>
</table>

- OS Linux kernel v. 4.19.0.

**Thread count search space:**
- Machine A: 2, 4, 6, 8, 10, 12 and 24
- Machine B: 2, 4, 6, 8, ..., 44 and 88

- Physical cores (only even numbers)
- The maximum number of threads
Benchmarks

• 9 OpenMP Parallel Applications written in C/C++:

Six kernels from the NAS Parallel Benchmark:
• BT, CG, FT, LU, MG, and SP

Three applications from different domains:
• Fast Fourier Transform (FFT);
• Jacobi (JA);
• Poisson (PO).

• GCC version 8.3 (OpenMP 4.5) with –O3

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>BT</td>
<td>Class B</td>
</tr>
<tr>
<td>CG</td>
<td>Class B</td>
</tr>
<tr>
<td>FT</td>
<td>Class C</td>
</tr>
<tr>
<td>LU</td>
<td>Class B</td>
</tr>
<tr>
<td>MG</td>
<td>Class B</td>
</tr>
<tr>
<td>SP</td>
<td>Class B</td>
</tr>
<tr>
<td>FFT</td>
<td>Array of 10000 elements</td>
</tr>
<tr>
<td>JA</td>
<td>Square matrix of 8192</td>
</tr>
<tr>
<td>PO</td>
<td>Square matrix of 768</td>
</tr>
</tbody>
</table>
Outline

• Introduction
• Experimental Setup
• Motivation
• Smoothing on DCT
• Evaluation
• Final consideration
a) Evaluate the effectiveness of the smoothing technique (without online cost):

- Offline learning a dynamic solution $\mathcal{B}$
- Run the application with the Smoothed time series

b) Evaluate the online smoothing overhead:

- Offline learning a dynamic solution $\mathcal{B}$
- Run the application and smooth $\mathcal{B}$ during execution

b) Evaluate the online smoothing into a DCT online learning technique:

- Online learning phase
- Smoothing
- $\mathcal{B} = (t_1^*, ..., t_k^*)$
a) the effectiveness of the smoothing technique

It improves B’ results

EDP optimization near the upper-bound

Platform A (24 hw threads)

Platform B (88 hw threads)
b) the online smoothing overhead

Our online smoothing technique has low overhead

Platform A
(24 hw threads)

Platform B
(88 hw threads)
c) Smoothing into a DCT online learning technique

- Online learning DCT technique
- Hoder [1]

**Search phase**

24 24 24 6 6 6 10

**Fibonacci based algorithm**

**Execution timeline**

\[ B = (t_1^*, ..., t_k^*) \]

c) Smoothing into a DCT online learning technique

Platform A
(24 hw threads)

Platform B
(88 hw threads)

large thread count
search space

50%
Outline

• Introduction
• Motivation
• Smoothing on DCT
• Experimental Setup
• Evaluation
• Final consideration
Final consideration

- A smoothing-based strategy to further improve the optimization results of any DCT technique
- Our strategy smooths the thread count changes alleviating the switching overheads, which is generated by DCT when changing the number of threads during application execution
- Experiments on two multicore systems with nine well-known benchmarks show that our smoothing technique improves EDP results of offline and online state-of-the-art DCT techniques by up to 93% and 89% (overall mean of 22%), respectively.
Thanks for your attention!

Questions?

Smoothing on Dynamic Concurrency Throttling

Janaína Schwarzrock¹, Hiago Mayk G. de A. Rocha¹,
Arthur F. Lorenzon², Antonio Carlos S. Beck¹

¹ Federal University of Rio Grande do Sul (UFRGS), Brazil
² Federal University of Pampa, Brazil

jschwarzrock@inf.ufrgs.br