Performance Prediction of Memory Access Intensive Apps with Delay Insertion: A Vision

Soramichi Akiyama  Takahiro Hirofuchi  Hirotaka Ogawa
National Institute of Advanced Industrial Science and Technology (AIST), Japan

Background
Performance prediction of a given program is highly important but difficult.

Importance: environments where the program is developed and where it deployed are not the same (ex: developed in a handy laptop, then deployed to a powerful server).

Difficulty: two machines have different performance balance among components.

Processors relatively speeding up compared to memory → Newer machines are more memory latency-sensitive

Proposed approach:
- Emulate performance balance of the target machine with Dynamic Binary Instrumentation
- Run the target code as-is to retrieve more useful information than model-based techniques can provide

Real Example
8K x 8K Matrix factorization w/ Python on various machines

<table>
<thead>
<tr>
<th>Machine A</th>
<th>Machine B</th>
<th>Machine C</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU (Xeon)</td>
<td>E5-2603 v1</td>
<td>E5-2699 v3</td>
</tr>
<tr>
<td>Mem</td>
<td>DDR3 1600</td>
<td>DDR4 2133</td>
</tr>
</tbody>
</table>

Perfect Scale? Yes  No  No

→ Exactly the same program scales differently on three machines (due to different flops/memory latency ratios).

Proposal and Proof-of-Concept Implementation

Main Ideas:
1. Emulate performance balance of the target machine with Dynamic Binary Instrumentation
2. Run the target code as-is to retrieve more useful information than model-based techniques can provide

Normal Execution:
CPU stalls due to memory access 40% of time in machine A, but the stall is 50% in machine B (different perf balance).

Delayed Execution:
Memory accesses are delayed to prolong the stall to 50% in machine A.

Implementation Choices:

<table>
<thead>
<tr>
<th>Based-on</th>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware</td>
<td>Small overhead</td>
<td>Less practical</td>
</tr>
<tr>
<td>Compiler</td>
<td>Small overhead</td>
<td>New compiler for all langs</td>
</tr>
<tr>
<td>DBI</td>
<td>Applicable to any lang</td>
<td>Large overhead</td>
</tr>
</tbody>
</table>

Poc Implementation:
QEMU’s dynamic binary instrumentation mechanism (user-mode) is modified to insert delays after memory read/writes.

QEMU JIT Workflow

What we implemented

QEMU JIT Workflow

<table>
<thead>
<tr>
<th>Original Binary (e.g. x86)</th>
<th>Insert Delay</th>
<th>QEMU Intermediate Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id, Id, Id, Id</td>
<td>NOP</td>
<td>NOP</td>
</tr>
</tbody>
</table>

Nothing Special

ID, ID, ID, ID

Id Special

In the future

Technical Challenges:
1. Deciding number of NOPs to insert systematically
2. System call overhead that may break the whole balance

Preliminary Results and Future Vision

Workload: 8K x 8K matrix-vector multiplication with no tiling

Metric: Normalized throughput (inverse of the execution time)

Q: Can our method adjust perf balance among two machines?

Experiment Settings:

<table>
<thead>
<tr>
<th>Workload</th>
<th>Throughput scales better in machine A (old one), since machine A is less memory access latency sensitive.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normalized throughput</td>
<td><img src="image1" alt="Graph" /></td>
</tr>
<tr>
<td>Number of threads</td>
<td><img src="image2" alt="Graph" /></td>
</tr>
<tr>
<td>Number of threads</td>
<td><img src="image3" alt="Graph" /></td>
</tr>
<tr>
<td>Number of threads</td>
<td><img src="image4" alt="Graph" /></td>
</tr>
</tbody>
</table>

Future Vision:
1. Running profilers on our mechanism greatly helps diagnosing perf issues stemming from different perf balances.

2. Perf models of multi-threaded apps requires explicit/implicit data-dependency analysis and often inaccurate. Our mechanism automatically propagates the effect of prolonged critical sections as the whole code is executed.

Main Ideas:
1. Implementation Choices:

   - Based-on
   - Hardware
   - Compiler
   - DBI

   - Pros
   - Cons

   - Hardware
     - Small overhead
     - Less practical
   - Compiler
     - Small overhead
     - New compiler for all langs
   - DBI
     - Applicable to any lang
     - Large overhead

   - Based-on
   - Hardware
     - Small overhead
     - Less practical
   - Compiler
     - Small overhead
     - New compiler for all langs
   - DBI
     - Applicable to any lang
     - Large overhead

   - Poc Implementation:

   - QEMU JIT Workflow

   - What we implemented

   - QEMU JIT Workflow

   - | Original Binary (e.g. x86) | Insert Delay | QEMU Intermediate Code |
   - |---------------------------|--------------|------------------------|
   - | Id, Id, Id, Id | NOP | NOP |

   - Nothing Special

   - ID, ID, ID, ID

   - Id Special

Future Vision:
1. Running profilers on our mechanism greatly helps diagnosing perf issues stemming from different perf balances.

   - Prof
     - Target App
       - ![Graph](image5)
   - In the future
     - ![Graph](image6)
   - Delayed Exec

   - 2. Perf models of multi-threaded apps requires explicit/implicit data-dependency analysis and often inaccurate. Our mechanism automatically propagates the effect of prolonged critical sections as the whole code is executed.

Technical Challenges:
1. Deciding number of NOPs to insert systematically
2. System call overhead that may break the whole balance

*This paper is based on results obtained from a project commissioned by the New Energy and Industrial Technology Development Organization (NEDO).