### Dynamic Optimization

"Applications spend 90% of their time in 10% of a static binary" - Pareto principle

**Dynamic optimization:**

1. Profiles applications for hot paths (consecutive basic blocks that are evaluated together with high probability)
2. Optimizes those hot paths into contiguous traces at runtime
3. Enables applications to execute instructions from traces

**Code fragment:**

<table>
<thead>
<tr>
<th>Function Call</th>
<th>Message length</th>
<th>Hits in L1 Trace Cache</th>
<th>Hot path message (8 bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SW</td>
<td>16/128</td>
<td>8</td>
<td>128 Bytes</td>
</tr>
</tbody>
</table>

**Instruction cache fetch pattern**

- Original code: A = B = C = D = E = F = H = M
- Optimized code: A = B = C = D = E = F = H = M

**Optimized Trace (512 Bytes)**

- Network latency: 16-256 cycles
- Network buffering: 16, 256, 4 bytes fills
- Application working set

**Benefits:**

- Reduced verification costs, ease of design, greater code compatibility
- Additionally, Dynamic optimization can be scheduled to any core

Any dedicated hardware cost is replicated per-core.

### Partner Core Motivation

**High level:**

![Diagram showing a system with annotations](image)

**Research question:** Is it possible for the Partner core to keep up (rate match) with the App core, given minimal dedicated hardware support?

**Benefits:**

- Reduced verification costs
- Ease of design
- Greater code compatibility
- Additionally, dynamic optimization can be scheduled to any core

### Results

**System Configurations**

<table>
<thead>
<tr>
<th>SW</th>
<th>SW-L</th>
<th>SW-L-C</th>
<th>SW-F</th>
<th>HW</th>
<th>HW-U</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max Trace Length (kB)</td>
<td>Trace Occupance Threshold (in Bytes)</td>
<td>L1 Cache Capacity (in Bytes)</td>
<td>Network Latency (cycles)</td>
<td>Network Buffer Depth (in Flits)</td>
<td></td>
</tr>
<tr>
<td>16/128</td>
<td>16/128</td>
<td>16/128</td>
<td>16/128</td>
<td>16/128</td>
<td>16/128</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>16</td>
<td>256</td>
<td>256</td>
<td>256</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Branch Prediction Case Study**

- App/Partner core system + static branch predictor has same branch prediction accuracy as single core baseline + hybrid branch predictor
- Our system’s performance breaks even with a baseline using the same branch predictor

**Examples:**

- Nested Loops (trace length = up to 2 basic blocks)

**Color guide:**

- Blue: Branch
- Green: Early exit
- Red: Regular exit

**Trace Coverage**

<table>
<thead>
<tr>
<th>SW</th>
<th>SW-L</th>
<th>SW-L-C</th>
<th>SW-F</th>
<th>HW</th>
<th>HW-U</th>
</tr>
</thead>
<tbody>
<tr>
<td>0%</td>
<td>20%</td>
<td>40%</td>
<td>60%</td>
<td>80%</td>
<td>100%</td>
</tr>
</tbody>
</table>

**Network**

- Messages Started Due to:
  - Backwards Branches / Function Calls: 55%
  - Trace Early Exits: 30%
  - Trace Regular Exits: 15%

- Other Network Statistics:
  - Mappable Dropped due to Network Contention: 36%
  - Messages Shorter than Branch Limit: 50%
  - Average Network Injection Rate: > 100 cycles
  - Average Network Utilization: 12% - 25%

**Example:**

- Nested Loops (trace length = up to 2 basic blocks)