Accelerating Decoupled Look-ahead to Exploit Implicit Parallelism

Raj Parihar

Advisor: Prof. Michael C. Huang

March 22, 2013
**Motivation**

- Despite the proliferation of multi-core, multi-threaded systems
  - Single-thread performance is still an important design goal
- Modern programs do not lack instruction level parallelism
- Real challenge: exploit implicit parallelism without undue costs
- One effective approach: Decoupled look-ahead
Motivation

- Despite the proliferation of multi-core, multi-threaded systems
  - Single-thread performance is still an important design goal
- Modern programs do not lack instruction level parallelism
- Real challenge: exploit implicit parallelism without undue costs
- One effective approach: Decoupled look-ahead
Motivation: Decoupled Look-ahead

- Decoupled look-ahead architecture targets
  - Performance hurdles: branch mispredictions, cache misses, etc.
  - Exploration of parallelization opportunities, dependence information
  - Microarchitectural complexity, energy inefficiency through decoupling
Motivation: Decoupled Look-ahead

- Decoupled look-ahead architecture targets
  - Performance hurdles: branch mispredictions, cache misses, etc.
  - Exploration of parallelization opportunities, dependence information
  - Microarchitectural complexity, energy inefficiency through decoupling

- The look-ahead thread can often become a *new* bottleneck
Motivation: Decoupled Look-ahead

- Decoupled look-ahead architecture targets
  - Performance hurdles: branch mispredictions, cache misses, etc.
  - Exploration of parallelization opportunities, dependence information
  - Microarchitectural complexity, energy inefficiency through decoupling

- The look-ahead thread can often become a new bottleneck

- We explore techniques to accelerate the look-ahead thread
  - Speculative parallelization: aptly suited due to increased parallelism in the look-ahead binary
  - Weak dependence: lack of correctness constraint allows weak instruction removal w/o affecting the quality of look-ahead
Outline

Motivation

Baseline decoupled look-ahead

Look-ahead thread acceleration
  Speculative parallelization in look-ahead
  Weak dependence removal in look-ahead

Experimental analysis

Summary
Baseline Decoupled Look-ahead

- Binary parser is used to generate skeleton from original program
- The skeleton runs on a separate core and
  - Maintains its memory image in local L1, no writeback to shared L2
  - Sends branch outcomes through FIFO queue; also helps prefetching

A. Garg and M. Huang, “A Performance-Correctness Explicitly Decoupled Architecture”, MICRO-08
Practical Advantages of Decoupled Look-ahead

- Look-ahead thread is a self-reliant agent, completely independent of main thread
- No need for quick spawning and register communication support
- Low management overhead on main thread
- Easier for run-time control to disable

<table>
<thead>
<tr>
<th></th>
<th>Memory references</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>90%</td>
<td>95%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DI</td>
<td>SI</td>
<td>DI</td>
</tr>
<tr>
<td>bzip2</td>
<td>1.86</td>
<td>17</td>
<td>3.15</td>
</tr>
<tr>
<td>crafty</td>
<td>0.73</td>
<td>23</td>
<td>1.04</td>
</tr>
<tr>
<td>eon</td>
<td>2.28</td>
<td>50</td>
<td>3.34</td>
</tr>
<tr>
<td>gap</td>
<td>1.35</td>
<td>15</td>
<td>1.44</td>
</tr>
<tr>
<td>gcc</td>
<td>8.49</td>
<td>153</td>
<td>8.84</td>
</tr>
<tr>
<td>gzip</td>
<td>0.1</td>
<td>6</td>
<td>0.1</td>
</tr>
<tr>
<td>mcf</td>
<td>13.1</td>
<td>13</td>
<td>14.7</td>
</tr>
<tr>
<td>parser</td>
<td>1.31</td>
<td>41</td>
<td>1.59</td>
</tr>
<tr>
<td>pbmk</td>
<td>1.87</td>
<td>35</td>
<td>2.11</td>
</tr>
<tr>
<td>twolf</td>
<td>2.69</td>
<td>23</td>
<td>3.28</td>
</tr>
<tr>
<td>vortex</td>
<td>1.96</td>
<td>42</td>
<td>2</td>
</tr>
<tr>
<td>vpr</td>
<td>7.47</td>
<td>16</td>
<td>11.6</td>
</tr>
<tr>
<td>Avg</td>
<td><strong>3.60%</strong></td>
<td><strong>36</strong></td>
<td><strong>4.44%</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Branches</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>90%</td>
<td>95%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DI</td>
<td>SI</td>
<td>DI</td>
</tr>
<tr>
<td>bzip2</td>
<td>3.9</td>
<td>52</td>
<td>4.49</td>
</tr>
<tr>
<td>crafty</td>
<td>5.33</td>
<td>235</td>
<td>6.14</td>
</tr>
<tr>
<td>eon</td>
<td>2.02</td>
<td>19</td>
<td>2.31</td>
</tr>
<tr>
<td>gap</td>
<td>2.02</td>
<td>77</td>
<td>2.64</td>
</tr>
<tr>
<td>gcc</td>
<td>8.08</td>
<td>1103</td>
<td>8.41</td>
</tr>
<tr>
<td>gzip</td>
<td>8.41</td>
<td>40</td>
<td>8.66</td>
</tr>
<tr>
<td>mcf</td>
<td>9.99</td>
<td>14</td>
<td>10.2</td>
</tr>
<tr>
<td>parser</td>
<td>6.81</td>
<td>130</td>
<td>7.3</td>
</tr>
<tr>
<td>pbmk</td>
<td>2.88</td>
<td>92</td>
<td>3.21</td>
</tr>
<tr>
<td>twolf</td>
<td>5.75</td>
<td>41</td>
<td>6.48</td>
</tr>
<tr>
<td>vortex</td>
<td>1.24</td>
<td>114</td>
<td>1.97</td>
</tr>
<tr>
<td>vpr</td>
<td>4.8</td>
<td>6</td>
<td>4.88</td>
</tr>
<tr>
<td>Avg</td>
<td><strong>5.10%</strong></td>
<td><strong>160</strong></td>
<td><strong>5.56%</strong></td>
</tr>
</tbody>
</table>
Practical Advantages of Decoupled Look-ahead

- Look-ahead thread is a self-reliant agent, completely independent of main thread
  - No need for quick spawning and register communication support
  - Low management overhead on main thread
  - Easier for run-time control to disable
- Natural throttling mechanism to prevent
  - Run-away prefetching, cache pollution
Practical Advantages of Decoupled Look-ahead

- Look-ahead thread is a self-reliant agent, completely independent of main thread
  - No need for quick spawning and register communication support
  - Low management overhead on main thread
  - Easier for run-time control to disable
- Natural throttling mechanism to prevent
  - Run-away prefetching, cache pollution
- Look-ahead thread size comparable to aggregation of short helper threads

<table>
<thead>
<tr>
<th>Library</th>
<th>Memory references 90%</th>
<th>Memory references 95%</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DI</td>
<td>SI</td>
</tr>
<tr>
<td>bzip2</td>
<td>1.86</td>
<td>17</td>
</tr>
<tr>
<td>crafty</td>
<td>0.73</td>
<td>23</td>
</tr>
<tr>
<td>eon</td>
<td>2.28</td>
<td>50</td>
</tr>
<tr>
<td>gap</td>
<td>1.35</td>
<td>15</td>
</tr>
<tr>
<td>gec</td>
<td>8.49</td>
<td>153</td>
</tr>
<tr>
<td>gzip</td>
<td>0.1</td>
<td>6</td>
</tr>
<tr>
<td>mcf</td>
<td>13.1</td>
<td>13</td>
</tr>
<tr>
<td>parser</td>
<td>1.31</td>
<td>41</td>
</tr>
<tr>
<td>pbmk</td>
<td>1.87</td>
<td>35</td>
</tr>
<tr>
<td>twolf</td>
<td>2.69</td>
<td>23</td>
</tr>
<tr>
<td>vortex</td>
<td>1.96</td>
<td>42</td>
</tr>
<tr>
<td>vpr</td>
<td>7.47</td>
<td>16</td>
</tr>
<tr>
<td>Avg</td>
<td>3.60%</td>
<td>36</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Library</th>
<th>Branches 90%</th>
<th>Branches 95%</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DI</td>
<td>SI</td>
</tr>
<tr>
<td>bzip2</td>
<td>3.9</td>
<td>52</td>
</tr>
<tr>
<td>crafty</td>
<td>5.33</td>
<td>235</td>
</tr>
<tr>
<td>eon</td>
<td>2.02</td>
<td>19</td>
</tr>
<tr>
<td>gap</td>
<td>2.02</td>
<td>77</td>
</tr>
<tr>
<td>gec</td>
<td>8.08</td>
<td>1103</td>
</tr>
<tr>
<td>gzip</td>
<td>8.41</td>
<td>40</td>
</tr>
<tr>
<td>mcf</td>
<td>9.99</td>
<td>14</td>
</tr>
<tr>
<td>parser</td>
<td>6.81</td>
<td>130</td>
</tr>
<tr>
<td>pbmk</td>
<td>2.88</td>
<td>92</td>
</tr>
<tr>
<td>twolf</td>
<td>5.75</td>
<td>41</td>
</tr>
<tr>
<td>vortex</td>
<td>1.24</td>
<td>114</td>
</tr>
<tr>
<td>vpr</td>
<td>4.8</td>
<td>6</td>
</tr>
<tr>
<td>Avg</td>
<td>5.10%</td>
<td>160</td>
</tr>
</tbody>
</table>
Look-ahead: A New Bottleneck

- Comparing four systems to discover new bottlenecks
  - Single-thread, decoupled look-ahead, ideal, and look-ahead alone
- Application categories:
  - Bottleneck removed or speed of look-ahead is not an issue
  - **Look-ahead thread is the new bottleneck** (right half)

![Graph showing IPC for different systems and configurations]

<table>
<thead>
<tr>
<th>Single-thread</th>
<th>Decoupled look-ahead</th>
<th>Ideal(Cache,Br)</th>
<th>Look-ahead only</th>
</tr>
</thead>
<tbody>
<tr>
<td>apl msa wup mgr six swim fac gal gcc gap eon fm3d gzip crft vor apsi vpr bp2 eqk amp luc art pbmkmcf twlf</td>
<td>0 1 2 3 4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Raj Parihar

Accelerating Decoupled Look-ahead to Exploit Implicit Parallelism
Look-ahead: A New Bottleneck

- Comparing four systems to discover new bottlenecks
  - Single-thread, decoupled look-ahead, ideal, and look-ahead alone
- Application categories:
  - Bottleneck removed or speed of look-ahead is not an issue
  - Look-ahead thread is the new bottleneck (right half)

![Bar chart showing IPC for different application categories]
Look-ahead: A New Bottleneck

- Comparing four systems to discover new bottlenecks
  - Single-thread, decoupled look-ahead, ideal, and look-ahead alone
- Application categories:
  - Bottleneck removed or speed of look-ahead is not an issue
  - Look-ahead thread is the new bottleneck (right half)

![IPC Comparison Chart]

Raj Parihar
Accelerating Decoupled Look-ahead to Exploit Implicit Parallelism
Look-ahead: A New Bottleneck

- Comparing four systems to discover new bottlenecks
  - Single-thread, decoupled look-ahead, ideal, and look-ahead alone

Application categories:
- Bottleneck removed or speed of look-ahead is not an issue
- **Look-ahead thread is the new bottleneck** (right half)
Unique Opportunities for Speculative Parallelization

- **Skeleton code offers more parallelism**
- Certain dependencies removed during slicing for skeleton
- Short-distance dependence chains become long-distance chains, suitable for TLP exploitation

```
0x12000da84  lda a5, 744(sp)
0x12000dac0  ld1 t7, 0(a5)
0x12000daec  lda a5, 4(a5)
0x120011984  ldq a0, 80(sp)
0x120011490  ldt $f0, 0(a0)
0x1200114bc  stt $f0, 0(a2)
0x1200119a0  ldt $f12, 32(sp)
0x1200119ac  lda t8, 168(sp)
0x1200119f8  bis 0, t8, t11
0x120011b04  lda a0, 8(a0)
```
Unique Opportunities for Speculative Parallelization

- Skeleton code offers more parallelism
  - Certain dependencies removed during slicing for skeleton
  - Short-distance dependence chains become long-distance chains, suitable for TLP exploitation

- Look-ahead is inherently error-tolerant
  - Can ignore dependence violations
  - Little to no support needed, unlike in conventional TLS
Software Support

- Dependence analysis
  - Profile guided, coarse-grain at basic block level

Parallelism at basic block level

Available parallelism for 2 core/context system

Raj Parihar
Accelerating Decoupled Look-ahead to Exploit Implicit Parallelism
Software Support

- Dependence analysis
  - Profile guided, coarse-grain at basic block level
- Spawn and Target points
  - Basic blocks with **consistent** dependence distance of more than threshold of $D_{MIN}$
  - Spawned thread executes from target point

Available parallelism for 2 core/contexts system
Software Support

- Dependence analysis
  - Profile guided, coarse-grain at basic block level
- Spawn and Target points
  - Basic blocks with *consistent* dependence distance of more than threshold of $D_{MIN}$
  - Spawned thread executes from target point
- Loop level parallelism is also exploited

Available parallelism for 2 core/contexts system
Parallelism Potential in Look-ahead Binary

- Available parallelism for 2 core/contexts system; $D_{MIN} = 15BB$
Parallelism Potential in Look-ahead Binary

- Available parallelism for 2 core/contexts system; $D_{MIN} = 15BB$
  - Skeleton exhibits significant more BB level parallelism (17%)
Parallelism Potential in Look-ahead Binary

- Available parallelism for 2 core/contexts system; $D_{MIN} = 15BB$

- Skeleton exhibits significant more BB level parallelism (17%)

- Loop based FP applications exhibit more BB level parallelism

![Graph showing parallelism potential comparison between original binary and skeleton](image-url)
Hardware and Runtime Support

- Thread spawning and merging are very similar to regular thread spawning except
  - Spawned thread shares the same register and memory state
  - Spawning thread terminates at the target PC
Hardware and Runtime Support

- Thread spawning and merging are very similar to regular thread spawning except:
  - Spawned thread shares the same register and memory state
  - Spawning thread terminates at the target PC
- Value communication:
  - Register-based naturally through shared registers in SMT
  - Memory-based communication can be supported at different levels
    - Partial versioning in cache at line level
14 applications in which the look-ahead thread is bottleneck
14 applications in which the look-ahead thread is bottleneck

Speedup of look-ahead systems over single-thread

- Decoupled look-ahead over single-thread baseline: 1.53x
Speedup of Speculative Parallelization

- 14 applications in which the look-ahead thread is bottleneck
- Speedup of look-ahead systems over single-thread
  - Decoupled look-ahead over single-thread baseline: \(1.53\times\)
  - Speculative parallel look-ahead over single-thread: \(1.73\times\)
Speedup of Speculative Parallelization

- 14 applications in which the look-ahead thread is bottleneck
- Speedup of look-ahead systems over single-thread
  - Decoupled look-ahead over single-thread baseline: **1.53x**
  - Speculative parallel look-ahead over single-thread: **1.73x**
- Speculative look-ahead over decoupled look-ahead: **1.13x**
Speculative Look-ahead vs Conventional TLS

- Skeleton provides **more opportunities** for parallelization

![Graph showing speculative parallelization effects](image-url)
Speculative Look-ahead vs Conventional TLS

- Skeleton provides **more opportunities** for parallelization
- Speculative look-ahead over decoupled LA baseline: \(1.13x\)
Speculative Look-ahead vs Conventional TLS

- Skeleton provides **more opportunities** for parallelization
- Speculative look-ahead over decoupled LA baseline: **1.13x**
- Speculative main thread over single thread baseline: **1.07x**
Motivation for Exploiting Weak Dependences

- Not all instructions are equally important and critical
- Example of weak instructions:
  - Inconsequential adjustments
  - Load and store instructions that are (mostly) silent
  - Dynamic NOP instructions

```
PC:  Instructions  Inputs  Frequent Value  Output
...  ...  beq t2, 0x120036498
0x12003698:  ...  ...
0x120036e0:  ...  ...
0x120036e0:  ...  ...
0x1200367c:  bne t5, 0x1200367a8
0x12003678:  ...  ...
0x1200367b:  bne t0, 0x120036780
0x1200a1f4:  cmple t4, 0x064, v0
0x1200a1f8:  ...  ...
0x1200a1fc:  and v0, ra, v0
0x1200a200:  bne v0, 0x1200367a8
```

Plenty of weak instructions are present in programs

Challenges involved:
- Context-based, hard to identify and combine – much like Jenga
Motivation for Exploiting Weak Dependences

- Not all instructions are equally important and critical.
- Example of weak instructions:
  - Inconsequential adjustments
  - Load and store instructions that are (mostly) silent
  - Dynamic NOP instructions
- Plenty of weak instructions are present in programs
- Challenges involved:
  - Context-based, hard to identify and combine – much like Jenga
Comparison of Weak and Strong Instructions

- Static attributes of weak and strong insts are remarkably same
- Static attributes: opcode, number of inputs
- The correlation coefficient of the two distributions is 0.96
Comparison of Weak and Strong Instructions

- Static attributes of weak and strong insts are **remarkably** same
  - Static attributes: opcode, number of inputs
  - The correlation coefficient of the two distributions is **0.96**
- Weakness has very poor correlation with static attributes
  - Hard to identify the weak insts through static heuristics
Genetic Algorithm based Framework

- Genetic algorithm based framework to identify and eliminate weak instructions from the look-ahead skeleton
Genetic Algorithm based Framework

- Genetic algorithm based framework to identify and eliminate weak instructions from the look-ahead skeleton
- Genetic evolution: procreation and natural selection
Genetic Algorithm based Framework

- Genetic algorithm based framework to identify and eliminate weak instructions from the look-ahead skeleton
- Genetic evolution: procreation and natural selection
- Chromosomes creation and hybridization
- Baseline look-ahead skeleton construction
Heuristic Based Solutions

- Heuristic based solutions are helpful to *jump start* the evolution
  - Superposition based chromosomes
  - Orthogonal subroutine based chromosomes

---

Heuristic based solutions are helpful to *jump start* the evolution

- Superposition based chromosomes
- Orthogonal subroutine based chromosomes

---

Initial Chromosomes

- Single-Instruction Genes
- Multi-Instruction Genes

---

(a) Single-gene Chromosomes
(b) Superposition Chromosomes
(c) Orthogonal Chromosomes

---

Subroutines A, B, C
Progress of Genetic Evolution Process

- Per generation progress compared to the final best solution
  - After 2 generations, more than half of the benefits are achieved
  - After 5 generations, at least 90% of benefits are achieved
Experimental Setup

- Program/binary analysis tool: based on ALTO
- Simulator: based on heavily modified SimpleScalar
  - SMT, look-ahead and speculative parallelization support
  - True execution-driven simulation (faithfully value modeling)
- Genetic algorithm framework
- Modeled as offline and online extension to the simulator

<table>
<thead>
<tr>
<th>Microarchitectural config:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Baseline core</strong></td>
</tr>
<tr>
<td>Fetch/Decode/Commit ROB</td>
</tr>
<tr>
<td>Functional units</td>
</tr>
<tr>
<td>Issue Q / Reg. (int,fp)</td>
</tr>
<tr>
<td>LSQ(LQ,SQ)</td>
</tr>
<tr>
<td>Branch predictor</td>
</tr>
<tr>
<td>- Gshare</td>
</tr>
<tr>
<td>- Bimodal/Meta/ BTB</td>
</tr>
<tr>
<td>Br. mispred. penalty</td>
</tr>
<tr>
<td>L1 data cache</td>
</tr>
<tr>
<td>L1 I cache (not shared)</td>
</tr>
<tr>
<td>L2 cache (uni. shared)</td>
</tr>
<tr>
<td>Memory access latency</td>
</tr>
</tbody>
</table>

| Look-ahead core:           |
| Baseline core with L0 cache: (16KB, 4-way, 32B line, 2 cycle, 2 ports). Round trip latency to L1 is 6 cycles |

| Communication:             |
| BOQ: 512 entries; PAB: 256 entries; register copy latency (during recovery): 32 cycles |
Motivation
Baseline decoupled look-ahead
Look-ahead thread acceleration
Experimental analysis
Summary

Speedup of Self-tuned Look-ahead

Applications in which the look-ahead thread is a bottleneck

![Chart showing speedup comparison between baseline look-ahead and GA based look-ahead among various applications. The chart indicates significant speedup for certain applications, such as eqk and art, with a mean speedup of approximately 2.5 over the single-thread baseline.](chart.png)
Speedup of Self-tuned Look-ahead

- Applications in which the look-ahead thread is a bottleneck
- Self-tuned, genetic algorithm based decoupled look-ahead
  - Speedup over baseline decoupled look-ahead: **1.16x**

![Graph showing speedup comparison between baseline and GA-based look-ahead](image-url)
Speedup of Self-tuned Look-ahead

- Applications in which the look-ahead thread is a bottleneck
- Self-tuned, genetic algorithm based decoupled look-ahead
  - Speedup over baseline decoupled look-ahead: 1.16x
  - Speedup over single-thread baseline: 1.78x
Comparison with Speculative Parallel Look-ahead

- Self-tuned skeleton is used in the speculative parallel look-ahead.
- In some cases, self-tuned and speculative parallel look-ahead techniques are synergistic (*ammp, art*)
Ongoing and Future Explorations

- **Load balancing through skipping non-critical branches**
- Weak instruction classification and identification
- Single-core version of decoupled look-ahead
- Static heuristics to construct adaptive skeletons
- Role of look-ahead to promote parallelization and accelerate the execution of interpreted programs
Summary

- Decoupled look-ahead can uncover significant implicit parallelism
  - However, look-ahead thread often becomes a new bottleneck

- Fortunately, look-ahead lends itself to various optimizations:
  - Speculative parallelization is more beneficial in look-ahead thread
  - Weak instructions can be removed w/o affecting look-ahead quality

- **Intelligent** look-ahead technique is a promising solution in the era of flat frequency and modest microarchitecture scaling

- Idle cores in multicore environment will further strengthen the case of decoupled look-ahead adoption in mainstream systems
Accelerating Decoupled Look-ahead to Exploit Implicit Parallelism

Raj Parihar

Advisor: Prof. Michael C. Huang

March 22, 2013
Microthreads vs Decoupled Look-ahead

**Lightweight Microthreads:**
- Process
- Thread #1
- Thread #2
- Microthreads

**Decoupled Look-ahead:**
- Register state synchronization
- Look-ahead Core
  - Branch Queue
  - Branch predictions
- Main Core
  - Prefetching hints
  - L1
  - L2
Look-ahead Skeleton Construction

(A) Illustration of a biased conditional branch (loop) turned into unconditional branch in the skeleton

(B) Illustration of a long communication store and its consumer load computation optimization in skeleton
Performance Benefits of Decoupled Look-ahead

<table>
<thead>
<tr>
<th></th>
<th>INT (11 apps)</th>
<th></th>
<th>FP (14 apps)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Max</td>
<td>Avg.</td>
<td>Min</td>
<td>Max</td>
</tr>
<tr>
<td>Speedup over single-thread baseline</td>
<td>2.16</td>
<td>1.35</td>
<td>1.05</td>
<td>4.44</td>
</tr>
<tr>
<td>Energy consumption w.r.t. baseline (%)</td>
<td>150</td>
<td>110</td>
<td>55</td>
<td>155</td>
</tr>
<tr>
<td>Look-ahead dynamic skeleton size (% prog)</td>
<td>92.6</td>
<td>74.7</td>
<td>9.66</td>
<td>92.9</td>
</tr>
<tr>
<td>Recoveries per 10K instructions</td>
<td>7.98</td>
<td>2.12</td>
<td>0.05</td>
<td>3.17</td>
</tr>
<tr>
<td>L2 miss reductions from baseline (%)</td>
<td>99.5</td>
<td>91.9</td>
<td>75</td>
<td>99.7</td>
</tr>
</tbody>
</table>
### IPC of Speculative Parallelization

<table>
<thead>
<tr>
<th></th>
<th>bzip2</th>
<th>crafty</th>
<th>eon</th>
<th>gap</th>
<th>gcc</th>
<th>gzip</th>
<th>mcf</th>
<th>pbnk</th>
<th>twolf</th>
<th>vortex</th>
<th>vpr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline (BL)</td>
<td>1.32</td>
<td>2.30</td>
<td>2.63</td>
<td>1.92</td>
<td>2.20</td>
<td>2.14</td>
<td>0.51</td>
<td>0.89</td>
<td>0.57</td>
<td>1.93</td>
<td>1.31</td>
</tr>
<tr>
<td>Decoupled (DLA)</td>
<td>1.75</td>
<td>2.47</td>
<td>2.90</td>
<td>3.35</td>
<td>4.60</td>
<td>2.34</td>
<td>0.83</td>
<td>1.14</td>
<td>0.74</td>
<td>2.24</td>
<td>1.77</td>
</tr>
<tr>
<td>Speculative (SPA)</td>
<td>1.75</td>
<td>2.48</td>
<td>2.91</td>
<td>3.36</td>
<td>4.81</td>
<td>2.36</td>
<td>0.84</td>
<td>1.35</td>
<td>1.01</td>
<td>2.27</td>
<td>2.43</td>
</tr>
</tbody>
</table>

(a) Integer applications.

<table>
<thead>
<tr>
<th></th>
<th>ammp</th>
<th>applu</th>
<th>apsi</th>
<th>art</th>
<th>equke</th>
<th>facres</th>
<th>fma3d</th>
<th>galgel</th>
<th>lucas</th>
<th>mesa</th>
<th>mgrid</th>
<th>sixtrk</th>
<th>swim</th>
<th>wup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline (BL)</td>
<td>0.79</td>
<td>1.81</td>
<td>1.75</td>
<td>0.27</td>
<td>1.09</td>
<td>3.03</td>
<td>2.72</td>
<td>2.35</td>
<td>0.58</td>
<td>2.99</td>
<td>3.03</td>
<td>2.84</td>
<td>1.26</td>
<td>3.77</td>
</tr>
<tr>
<td>Decoupled (DLA)</td>
<td>1.92</td>
<td>2.76</td>
<td>2.34</td>
<td>1.13</td>
<td>2.73</td>
<td>3.65</td>
<td>3.12</td>
<td>3.79</td>
<td>1.75</td>
<td>3.36</td>
<td>3.83</td>
<td>3.12</td>
<td>3.78</td>
<td>4.11</td>
</tr>
<tr>
<td>Speculative (SPA)</td>
<td>1.93</td>
<td>2.75</td>
<td>2.49</td>
<td>1.57</td>
<td>2.85</td>
<td>3.68</td>
<td>3.12</td>
<td>4.17</td>
<td>2.44</td>
<td>3.37</td>
<td>3.83</td>
<td>3.12</td>
<td>3.78</td>
<td>4.11</td>
</tr>
</tbody>
</table>

(b) Floating-point applications.
Speculative Parallelization: Cortex-A9 vs POWER5

<table>
<thead>
<tr>
<th>Specification</th>
<th>Cortex-A9 (3-way OoO)</th>
<th>POWER5 (4-way OoO)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pipeline width: fetch / decode / issue / commit</td>
<td>4 / 2 / 3 / 3</td>
<td>8 / 4 / 6 / 6</td>
</tr>
<tr>
<td>Queue size: fetch / issue:int / issue:fp</td>
<td>16 / 8 / 8</td>
<td>32 / 32 / 32</td>
</tr>
<tr>
<td>Registers: Regs:int / Regs:fp / ROB / LSQ</td>
<td>56 / 56 / 64 / 32</td>
<td>80 / 80 / 128 / 64</td>
</tr>
<tr>
<td>ALU resources: INT / FP</td>
<td>2 / 2</td>
<td>4 / 4</td>
</tr>
<tr>
<td>Caches (kB): DL0 / DL1 / DL2 / IL1</td>
<td>16 / 32 / 512 / 32</td>
<td>32 / 64 / 1024 / 64</td>
</tr>
<tr>
<td>TLB entries: DTLB / ITLB</td>
<td>1k / 512</td>
<td>8k / 2k</td>
</tr>
<tr>
<td>Br. Predictor: Bimod / 2-lev / RAS / BTB</td>
<td>2k / 4k / 32 / 1k</td>
<td>4k / 8k / 32 / 4k</td>
</tr>
</tbody>
</table>

![Graph showing speedup over single thread for various benchmarks](image)

- **Cortex-A9 (3-way OoO core)**
- **POWER5 (4-way OoO core)**
Flexibility in Look-ahead Hardware Design

- Comparison of regular (partial versioning) cache support with two other alternatives
  - No cache versioning support
  - Dependence violation detection and squash

![Graph showing comparison of cache support alternatives](image)
Partial Recoveries and Spawns

Partial recoveries:

<table>
<thead>
<tr>
<th></th>
<th>gap</th>
<th>mcf</th>
<th>pbm</th>
<th>twf</th>
<th>vor</th>
<th>vpr</th>
<th>amp</th>
<th>eqk</th>
<th>fac</th>
<th>fma</th>
<th>gal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recv-Merge</td>
<td>1116</td>
<td>1309</td>
<td>6408</td>
<td>57910</td>
<td>4862</td>
<td>12314</td>
<td>4252</td>
<td>9174</td>
<td>925</td>
<td>4062</td>
<td>291</td>
</tr>
<tr>
<td>Live 200</td>
<td>1031</td>
<td>1308</td>
<td>3642</td>
<td>39305</td>
<td>4801</td>
<td>11284</td>
<td>2970</td>
<td>4347</td>
<td>526</td>
<td>4055</td>
<td>290</td>
</tr>
<tr>
<td>Live 1000</td>
<td>917</td>
<td>1306</td>
<td>3355</td>
<td>17601</td>
<td>3774</td>
<td>10178</td>
<td>2626</td>
<td>3495</td>
<td>522</td>
<td>4041</td>
<td>290</td>
</tr>
</tbody>
</table>

Breakdown of all the spawns:

<table>
<thead>
<tr>
<th></th>
<th>mcf</th>
<th>pbmk</th>
<th>twolf</th>
<th>vortex</th>
<th>vpr</th>
<th>ammp</th>
<th>art</th>
<th>equake</th>
<th>fma3d</th>
<th>galgel</th>
<th>lucas</th>
</tr>
</thead>
</table>
| Spawns invoked under correct path
| Successful    | 2297 | 26873| 21067 | 1273  | 42082| 6328 | 29598| 16676  | 9687  | 20997  | 24022 |
| Runaway       | 257  | 245  | 1738  | 37    | 409  | 3542 | 363  | 0      | 3965  | 0      | 1     |
| Spawns invoked under incorrect path
| No dispatch   | 11   | 707  | 2837  | 96    | 1633 | 26   | 29   | 245    | 363   | 1      | 0     |
| Few dispatch  | 28   | 69   | 1803  | 6     | 273  | 45   | 116  | 10     | 1     | 0      | 0     |
| Wrong Path    | 11   | 184  | 2997  | 152   | 111  | 339  | 6    | 62     | 4     | 17     | 0     |
Genetic Algorithm Evolution

- Initial Population
- Evaluate Objective Function
- Select and Reproduce (Create new designs)
- Replace old population with new designs
- Desired state achieved?
  - No
  - Yes: Stop
### Multi-instruction Gene Examples

<table>
<thead>
<tr>
<th>PC:</th>
<th>Instructions</th>
<th>Inputs</th>
<th>Frequent Value</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0x120036758:</td>
<td><code>s4addq t10, t10, t10</code></td>
<td>t10 -&gt; ~0</td>
<td>t10 -&gt; ~0</td>
<td>...</td>
</tr>
<tr>
<td>0x12003675c:</td>
<td><code>addq t11, t10, t10</code></td>
<td>t11 -&gt; 0x140012200, t10 -&gt; 0</td>
<td>t10 -&gt; 0x140012200</td>
<td>...</td>
</tr>
<tr>
<td>0x120036760:</td>
<td><code>ld1 t10, 0(t10)</code></td>
<td>t10 -&gt; 0x140012200</td>
<td>t10 -&gt; ~0</td>
<td>...</td>
</tr>
<tr>
<td>0x120036764:</td>
<td><code>bne t10, 0x120036808</code></td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

(A) Vpr

<table>
<thead>
<tr>
<th>PC:</th>
<th>Instructions</th>
<th>Inputs</th>
<th>Frequent Value</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0x120011d90:</td>
<td><code>lda s3,-25724(gp)</code></td>
<td>gp -&gt; 0x140011940</td>
<td>s3 -&gt; 0x14000b4c4</td>
<td>...</td>
</tr>
<tr>
<td>0x120011d94:</td>
<td><code>ld1 s3, 0(s3)</code></td>
<td>s3 -&gt; 0x14000b4c4</td>
<td>s3 -&gt; 32767</td>
<td>...</td>
</tr>
<tr>
<td>0x120011d98:</td>
<td><code>cmpult s2, s3, s3</code></td>
<td>s2 -&gt; 1, s3 -&gt; 32767</td>
<td>s3 -&gt; 0</td>
<td>...</td>
</tr>
<tr>
<td>0x120011d9c:</td>
<td><code>bne s3, 0x120011c50</code></td>
<td>s3 -&gt; 0</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

(B) Gzip
Optimizations to Implementation

- Fitness test optimizations
  - Sampling based fitness
  - Multi-instruction genes
  - Early termination of tests

- GA framework optimizations
  - Hybridization of solutions
  - Adaptive mutation rate
  - Unique chromosomes
  - Fusion crossover operator
  - Elitism policy
Superposition based Chromosomes

Graphs showing fitness relative to baseline look-ahead for different superposition chromosome ids for peribmk and Vpr superposition chromosomes.
Recovery based Early Termination of Fitness Test

(a) Vpr

(b) Pbmk
Weak Dependence: IPC and Instructions Removed

### IPC comparison:

<table>
<thead>
<tr>
<th></th>
<th>crafty</th>
<th>eon</th>
<th>gzip</th>
<th>mcf</th>
<th>pbmk</th>
<th>twolf</th>
<th>vortex</th>
<th>vpr</th>
<th>ammp</th>
<th>art</th>
<th>equake</th>
<th>fma3d</th>
<th>galgel</th>
<th>lucas</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST-BL</td>
<td>2.30</td>
<td>2.63</td>
<td>2.14</td>
<td>0.51</td>
<td>0.77</td>
<td>1.92</td>
<td>1.31</td>
<td>0.79</td>
<td>0.27</td>
<td>1.09</td>
<td>2.71</td>
<td>2.35</td>
<td>0.58</td>
<td></td>
</tr>
<tr>
<td>BL-DLA</td>
<td>2.42</td>
<td>2.76</td>
<td>2.34</td>
<td>0.86</td>
<td>1.07</td>
<td>2.16</td>
<td>1.78</td>
<td>1.53</td>
<td>1.20</td>
<td>1.69</td>
<td>3.00</td>
<td>3.82</td>
<td>1.71</td>
<td></td>
</tr>
<tr>
<td>GA-DLA</td>
<td>2.47</td>
<td>3.10</td>
<td>2.44</td>
<td>1.16</td>
<td>1.29</td>
<td>2.20</td>
<td>2.06</td>
<td>1.56</td>
<td>1.78</td>
<td>2.52</td>
<td>3.21</td>
<td>4.16</td>
<td>2.04</td>
<td></td>
</tr>
</tbody>
</table>

### Instructions removed from baseline skeleton:

<table>
<thead>
<tr>
<th></th>
<th>cra</th>
<th>eon</th>
<th>gzip</th>
<th>mcf</th>
<th>pbmk</th>
<th>twolf</th>
<th>vor</th>
<th>vpr</th>
<th>amp</th>
<th>art</th>
<th>eqk</th>
<th>fma</th>
<th>gal</th>
<th>luc</th>
<th>Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>BL LA(%dyn)</td>
<td>87.14</td>
<td>72.29</td>
<td>64.72</td>
<td>59.95</td>
<td>78.98</td>
<td>81.05</td>
<td>79.25</td>
<td>67.06</td>
<td>66.60</td>
<td>54.33</td>
<td>32.86</td>
<td>41.13</td>
<td>32.21</td>
<td>62.57</td>
<td></td>
</tr>
<tr>
<td>GA LA(%dyn)</td>
<td>84.66</td>
<td>65.83</td>
<td>57.35</td>
<td>53.61</td>
<td>70.22</td>
<td>58.10</td>
<td>57.01</td>
<td>64.21</td>
<td>41.11</td>
<td>30.26</td>
<td>79.50</td>
<td>23.62</td>
<td>28.53</td>
<td>56.44</td>
<td></td>
</tr>
<tr>
<td>Prog Insts(s)</td>
<td>57568</td>
<td>79730</td>
<td>23205</td>
<td>2076</td>
<td>474</td>
<td>120529</td>
<td>53936</td>
<td>95121</td>
<td>42089</td>
<td>43154</td>
<td>25588</td>
<td>25639</td>
<td>249464</td>
<td>139032</td>
<td></td>
</tr>
<tr>
<td>Instrs 100m(s)</td>
<td>15773</td>
<td>8294</td>
<td>46</td>
<td>43</td>
<td>40</td>
<td>2983</td>
<td>35417</td>
<td>1443</td>
<td>1181</td>
<td>1768</td>
<td>1550</td>
<td>34</td>
<td>37</td>
<td>514</td>
<td></td>
</tr>
<tr>
<td>Weak Instrs(s)</td>
<td>58</td>
<td>110</td>
<td>46</td>
<td>43</td>
<td>15</td>
<td>29</td>
<td>1443</td>
<td>1181</td>
<td>1768</td>
<td>1550</td>
<td>34</td>
<td>37</td>
<td>514</td>
<td>6483</td>
<td></td>
</tr>
</tbody>
</table>

Raj Parihar

Accelerating Decoupled Look-ahead to Exploit Implicit Parallelism
Sampling based Fitness Test

![Graphs showing progress over time for different benchmarks](image)
L2 Cache Sensitivity Study

- Speedup for various L2 caches is quite stable
  - 1.161x (1 MB), 1.154x (2 MB), and 1.152x (4 MB) L2 caches
- Avg. speedups, shown in the figure, are relative to single-threaded execution with a 1 MB L2 cache
Single-Gene vs Heuristic based Chromosomes
Genetic Algorithm based Heuristics

![Graph showing performance comparison of different load ZVS methods](image-url)
Other Details

- Energy reduction: 11% over baseline decoupled look-ahead
  - Reduced cache accesses, less stalling of main thread
- On an average, 10% of the dynamic instructions are removed from the baseline skeleton
- Offline profiling and control software overhead
  - Offline profiling time: 2 to 20 seconds on the target machine
  - Online control software: 17 million instructions for whole evolution
- Average extra recoveries: 3-4 per 100,000 instructions

<table>
<thead>
<tr>
<th></th>
<th>INT (8 apps)</th>
<th></th>
<th></th>
<th>FP (6 apps)</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Max</td>
<td>Avg.</td>
<td>Min</td>
<td>Max</td>
<td>Avg.</td>
<td>Min</td>
</tr>
<tr>
<td>Baseline look-ahead</td>
<td>7.98</td>
<td>2.68</td>
<td>0.05</td>
<td>3.17</td>
<td>0.67</td>
<td>0.00</td>
</tr>
<tr>
<td>GA tuned look-ahead</td>
<td>11.20</td>
<td>3.14</td>
<td>0.39</td>
<td>4.35</td>
<td>1.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>
Load Balancing in Look-ahead

- Non-critical branches can be transformed to accelerate the look-ahead thread and achieve better load balancing.
- Non-critical branches are skipped in the look-ahead thread.
- Main thread executes such branches by itself w/o any help.
- We call these branches *Do-It-Yourself* or *DIY* branches.
Preliminary Speedup of Load Balancing

- Load balancing through skipping of the non-critical branches
- Max. speedup over decoupled look-ahead: $1.76x$ (art)
- Avg. speedup over decoupled look-ahead: $1.12x$ (Gmean)