Accelerating Decoupled Look-ahead via Weak Dependence Removal: A Metaheuristic Approach

Raj Parihar, Michael C. Huang

Department of Electrical & Computer Engineering
University of Rochester, Rochester, NY
Motivation

Despite the proliferation of multi-core, multi-threaded systems
- High 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 architecture
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 architecture targets

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

---

Raj Parihar, Michael C. Huang
Advanced Computer Architecture Lab
University of Rochester
Motivation

- 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 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

- Lack of correctness constraint allows many optimizations
  - Weak dependence: Instructions that contribute marginally to the outcome can be removed w/o affecting the quality of look-ahead
Outline

Motivation

Baseline decoupled look-ahead architecture
Look-ahead: a new bottleneck

Look-ahead thread acceleration
Weak dependences/instructions
Challenges in identifying weak instructions
Metaheuristic based approach
Experimental analysis

Summary
Baseline Decoupled Look-ahead Architecture

- Skeleton generated just for the look-ahead purposes
- The skeleton runs on a separate core and
  - Speculative state is completely contained within look-ahead context
  - Sends branch outcomes through FIFO queue; also helps prefetching

A. Garg and M. Huang, “A Performance-Correctness Explicitly Decoupled Architecture”, MICRO’08
Baseline Decoupled Look-ahead Architecture

- **Skeleton** generated just for the look-ahead purposes
- The skeleton runs on a separate core and
  - Speculative state is completely contained within look-ahead context
  - Sends branch outcomes through FIFO queue; also helps prefetching

A. Garg and M. Huang, “A Performance-Correctness Explicitly Decoupled Architecture”, *MICRO-08*
Look-ahead: A New Bottleneck

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

- Application categories:
  - Bottleneck removed or speed of look-ahead is not an issue
  - Look-ahead thread is the new bottleneck

![Graph showing IPC comparison between single-thread and look-ahead threads]
Look-ahead: A New Bottleneck

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

- Application categories:
  - Bottleneck removed; speed of look-ahead is not an issue (left half)
  - Look-ahead thread is the new bottleneck

![Graph showing IPC for different applications]
Look-ahead: A New Bottleneck

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

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

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

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

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
Weak Dependencies/Instructions

- 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 (100s of)
Weak Dependences/Instructions

- 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 (100s of)
- Single weak instruction: their impact is non-trivial
Weak Dependences/Instructions

- 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 (100s of)
- Single weak instruction: their impact is non-trivial
- Weak instruction can be experimentally defined and their impact quantified in isolation

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 (100s of)

Single weak instruction: their impact is non-trivial

Weak instruction can be experimentally defined and their impact quantified in isolation
Challenge # 1: Weak insts do not look different

- After the fact analysis: based on static attributes of insts reveals
  - Static attributes of weak and regular insts are *remarkably similar*
  - Correlation coefficient of the two distributions is very high (0.96)
Challenge # 1: Weak insts do not look different

- After the fact analysis: based on static attributes of insts reveals
  - Static attributes of weak and regular insts are **remarkably similar**
  - Correlation coefficient of the two distributions is very high (0.96)
- Weakness has very poor correlation with static attributes
- Hard to identify the weak instructions through static heuristics
Challenge # 2: False positives are extremely costly

- After the fact analysis and close inspection also reveals
  - Some instructions are more likely to be weak than others
  - Even then, a single false positive can negate all the gains
Challenge # 2: False positives are extremely costly

- After the fact analysis and close inspection also reveals
  - Some instructions are more likely to be weak than others
  - Even then, a single false positive can negate all the gains

- Case in point: `zapnot` in `gap`

  `zapnot Ra Rb Rc`

  - 84% of the `zapnot` insts are weak in isolation: **3.4% speedup**
  - Single false positive `zapnot` instruction: **6% slowdown**
  - More than 1 false positive instructions can slowdown upto **13%**
Challenge # 3: Neither absolute nor additive

- Weakness is context dependent, non-linear – much like Jenga
- All weak instructions combined together are not weak!
Challenge # 3: Neither absolute nor additive

- Weakness is context dependent, non-linear – much like Jenga
  - All weak instructions combined together are not weak!
- Example: weak instruction combining in *perlbmk*
  - About 300 weak instructions when tested in isolation
  - All combined together can result in up to 40% slowdown
Recap: **Challenges** in identifying weak instructions

- Weak instructions look very similar to regular instructions
- False positives are extremely costly and can negate all the gain
- Weakness is context dependent: neither absolute nor additive
Recap: **Challenges** in identifying weak instructions
- Weak instructions look very similar to regular instructions
- False positives are extremely costly and can negate all the gain
- Weakness is context dependent: neither absolute nor additive

Our approach: **Metaheuristic based self-tuning**
- Experimentally identify/verify weakness
- Search for profitable combination via metaheuristic
Metaheuristic Based Trail-and-Error Approach

- Recap: **Challenges** in identifying weak instructions
  - Weak instructions look very similar to regular instructions
  - False positives are extremely costly and can negate all the gain
  - Weakness is context dependent: neither absolute nor additive

- Our approach: **Metaheuristic based self-tuning**
  - Experimentally identify/verify weakness
  - Search for profitable combination via metaheuristic

- Metaheuristic: Completely agnostic of meaning of solution
  - Derive new solutions from current solutions through modifications
  - Example: genetic algorithm, simulated annealing, etc.
Genetic Algorithm based Framework

- The problem naturally maps to genetic algorithm

Diagram:

1. Program Binary
2. Look-ahead Binary
3. Multi-Instruction Genes
4. Single-Instruction Genes
5. Orthogonal Chromosome
6. Superposition Chromosome
7. Single-Gene Chromosome
8. Initial Chromosome Population

GA evolution:
- Roulette Wheel
- Fitness test, Elitism
- Parent selection
- De-duplication
- Xover & Mutation
- Reproduction

Raj Parihar, Michael C. Huang
Genetic Algorithm based Framework

- The problem naturally maps to genetic algorithm
  - Skeleton is represented by a bit vector
  - Natural mapping: weak inst → gene, collection → chromosome
  - Objective: find optimal combination (chromosome)

Program
Binary
Look-ahead
Binary

1. Look-ahead construction
2. Program Binary
3. Initial Chromosome Population
4. Genetic evolution: Procreation, mutation, fitness-based selection
5. Parent selection
6. Reproduction
7. De-duplication
8. Fitness test, Elitism

Genetic Algorithm based Framework

- Raj Parihar, Michael C. Huang
- Advanced Computer Architecture Lab
- University of Rochester
Genetic Algorithm based Framework

- The problem naturally maps to genetic algorithm
  - Skeleton is represented by a bit vector
  - Natural mapping: weak inst → gene, collection → chromosome
  - Objective: find optimal combination (chromosome)

- Genetic evolution: Procreation, mutation, fitness-based selection

Program
Binary
Look-ahead
Binary

1. Look-ahead construction
2. (Binary Parser)
3. Initial Chromosome Population
4. Multi-Instruction Genes
5. Orthogonal Chromosome
6. Superposition Chromosome
7. Single-Gene Chromosome
8. Roulette Wheel

Parents Pool
Children Pool
Fitness test, Elitism
De-duplication
Parent selection
Xover & Mutation
Reproduction
Hybridization: Heuristically Designed Initial Solutions

- Genetic evolution could be a slow and lengthy process
- Heuristic based solutions are helpful to **jump start** the evolution

![Diagram showing hybridization of chromosomes with single-instruction genes and multi-instruction genes.](image-url)
Hybridization: Heuristically Designed Initial Solutions

- Genetic evolution could be a slow and lengthy process
- Heuristic based solutions are helpful to **jump start** the evolution
- Heuristically designed solutions in our system:
  - Superposition chromosome; Orthogonal subroutine chromosome

![Hybridization Diagram]

- Single-Instruction Genes
- Multi-Instruction Genes

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

Raj Parihar, Michael C. Huang
Advanced Computer Architecture Lab
University of Rochester
Motivation
Baseline decoupled look-ahead architecture
Look-ahead thread acceleration
Summary

Weak dependences/instructions
Challenges in identifying weak instructions
Metaheuristic based approach
Experimental analysis

Experimental Setup

- Program/binary analysis tool: ALTO
- Simulator: detailed out-of-order, cycle-level in-house
  - 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

Experimental Setup

- Program/binary analysis tool: ALTO
- Simulator: detailed out-of-order, cycle-level in-house
  - 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

Microarchitectural configurations:

<table>
<thead>
<tr>
<th>Baseline core (Similar to POWER5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch/Decode/Issue/Commit 8 / 4 / 6 / 6</td>
</tr>
<tr>
<td>ROB 128</td>
</tr>
<tr>
<td>Functional units INT 2+1 mul +1 div, FP 2+1 mul +1 div (32, 32) / (32, 32) / (80, 80)</td>
</tr>
<tr>
<td>Fetch Q/ Issue Q / Reg. (int,fp) 64 (32,32) 2 search ports</td>
</tr>
<tr>
<td>LSQ(LQ,SQ) Gshare – 8K entries, 13 bit history at least 7 cycles</td>
</tr>
<tr>
<td>Branch predictor 32KB, 4-way, 64B line, 2 cycles, 2 ports</td>
</tr>
<tr>
<td>Br. mispred. penalty 64KB, 2-way, 128B, 2 cycles</td>
</tr>
<tr>
<td>L1 data cache (private) 1MB, 8-way, 128B, 15 cycles</td>
</tr>
<tr>
<td>L1 inst cache (private) 200 cycles</td>
</tr>
<tr>
<td>L2 cache (shared)</td>
</tr>
<tr>
<td>Memory access latency</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Look-ahead core: Baseline core with only LQ, no SQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0 cache: 32KB, 4-way, 64B line, 2 cycles Round trip latency to L1: 6 cycles</td>
</tr>
</tbody>
</table>

| Communication: Branch Output Queue: 512 entries Reg copy latency (recovery): 64 cycles |
Speedup of Self-tuned Look-ahead

- Applications in which the look-ahead thread is a bottleneck

![Speedup Chart](chart.png)

- Speedup over single-thread
  - Baseline look-ahead
  - GA based look-ahead

Raj Parihar, Michael C. Huang
Advanced Computer Architecture Lab
University of Rochester
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: **1.14x** (geomean)
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: **1.14x** (geomean)
  - Overall speedup over single-thread baseline: **1.58x**

![Speedup Graph](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: 1.14x (geomean)
  - Overall speedup over single-thread baseline: 1.58x
- 2-core speedup of 1.58x: a compelling solution for turbo boosting
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, significant performance benefits are achieved

![Progress relative to best GA solution](chart.png)

- eon
- mcf
- pbmk
- twolf
- vpr
- art
- eqk
- fma
- amp
- lucas
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, significant performance benefits are achieved
- GA evolution, helped by hybridization shows good progress

![Progress relative to best GA solution graph](image-url)
Evolution can be Online or Offline

- Offline evolution: one time tuning (e.g. install time)
  - Fitness tests need not take long (2-20s on target machine)
  - Different input and configuration do not invalidate result
Evolution can be Online or Offline

- **Offline evolution**: one time tuning (e.g. install time)
  - Fitness tests need not take long (2-20s on target machine)
  - Different input and configuration do not invalidate result

- **Online evolution** takes longer but has little overhead
  - Additional work minimum: book keeping, bit vector manipulation
  - Main source of slowdown: testing bad configurations
Other Details in the Paper

- The evolution process is remarkably robust
  - Different inputs and configuration do not invalidate results
  - Can use sampling to accelerate fitness test w/o appreciable impact on quality of solution found

- Energy reduction → due to less activity and stalling
  - About 10% dynamic instructions removed from skeleton
  - About 11% energy saving compared to baseline decoupled look-ahead

- Impact of weak insts removal on look-ahead quality is very small
  - Similar prefetch and branch hint accuracy
Summary

- Decoupled look-ahead can uncover significant implicit parallelism
  - However, look-ahead thread often becomes a new bottleneck
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:
  - **Weak instructions can be removed** w/o affecting look-ahead quality
  - Plenty of weak instructions present across all programs (100s of)
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:
  - **Weak instructions can be removed** w/o affecting look-ahead quality
  - Plenty of weak instructions present across all programs (100s of)

- Removing weak instructions through simple heuristics is hard
  - Expensive false positives; weakness interdependence
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:
  - Weak instructions can be removed w/o affecting look-ahead quality
  - Plenty of weak instructions present across all programs (100s of)
- Removing weak instructions through simple heuristics is hard
  - Expensive false positives; weakness interdependence
- Metaheuristic based self-tuning approach is simple and robust
  - Improves decoupled look-ahead performance by 1.14x while saving energy by 10%
Accelerating Decoupled Look-ahead via Weak Dependence Removal: A Metaheuristic Approach

Raj Parihar, Michael C. Huang

Department of Electrical & Computer Engineering

University of Rochester, Rochester, NY
Self-tuned Look-ahead: SPEC 2006

Self-tuned look-ahead achieves 1.10x speedup over baseline look-ahead for SPEC CPU 2006 applications.
Self-tuned Look-ahead: Speedup Analysis

- A larger code (with more genes) takes slightly more time to evolve

Relative Performance Gain = \( \frac{GA - DLA}{Ideal - DLA} \)

Liner Regression Line (\( r = -0.46 \))
Self-tuned Look-ahead: Speedup Analysis

- Performance gain has strong correlation with # of generations

![Graph showing the relationship between saturation generation and the number of static instructions](image)
Sampling based Fitness Test
L2 Cache Sensitivity Study

- Speedup for various L2 caches is quite stable
  - 1.139x (1 MB), 1.133x (2 MB), and 1.131x (4 MB) L2 caches
- Avg. speedups, shown in the figure, are relative to single-threaded execution with a 1 MB L2 cache
## Look-ahead Skeleton: Size and Other Stats

<table>
<thead>
<tr>
<th></th>
<th>crafty</th>
<th>eon</th>
<th>gap</th>
<th>gzip</th>
<th>mcf</th>
<th>perl</th>
<th>twolf</th>
<th>vortex</th>
<th>vpr</th>
<th>ammp</th>
<th>art</th>
<th>equake</th>
<th>fma3d</th>
<th>lucas</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline look-ahead skeleton (%dyn)</td>
<td>87.14</td>
<td>72.29</td>
<td>76.22</td>
<td>64.72</td>
<td>59.95</td>
<td>78.98</td>
<td>81.05</td>
<td>58.10</td>
<td>67.06</td>
<td>66.60</td>
<td>54.33</td>
<td>32.86</td>
<td>79.50</td>
<td>32.21</td>
<td>65.07</td>
</tr>
<tr>
<td>GA tuned look-ahead skeleton (%dyn)</td>
<td>82.66</td>
<td>65.83</td>
<td>67.79</td>
<td>57.35</td>
<td>52.61</td>
<td>70.22</td>
<td>78.25</td>
<td>56.31</td>
<td>59.01</td>
<td>63.22</td>
<td>41.11</td>
<td>30.06</td>
<td>73.50</td>
<td>28.53</td>
<td>59.03</td>
</tr>
<tr>
<td>Total program instructions (static)</td>
<td>57568</td>
<td>79730</td>
<td>74650</td>
<td>23205</td>
<td>18286</td>
<td>120529</td>
<td>350936</td>
<td>95121</td>
<td>42089</td>
<td>43154</td>
<td>25588</td>
<td>25639</td>
<td>249464</td>
<td>103235</td>
<td>72299</td>
</tr>
<tr>
<td>Instructions in 100m window (static)</td>
<td>12543</td>
<td>6562</td>
<td>4130</td>
<td>1424</td>
<td>381</td>
<td>9692</td>
<td>2456</td>
<td>12230</td>
<td>1061</td>
<td>958</td>
<td>582</td>
<td>1041</td>
<td>3098</td>
<td>319</td>
<td>4034</td>
</tr>
<tr>
<td>Individual weak instructions (static)</td>
<td>172</td>
<td>57</td>
<td>211</td>
<td>207</td>
<td>117</td>
<td>417</td>
<td>110</td>
<td>398</td>
<td>261</td>
<td>223</td>
<td>173</td>
<td>628</td>
<td>104</td>
<td>55</td>
<td>224</td>
</tr>
<tr>
<td>Instructions removed using GA (static)</td>
<td>51</td>
<td>15</td>
<td>37</td>
<td>56</td>
<td>20</td>
<td>30</td>
<td>24</td>
<td>37</td>
<td>33</td>
<td>24</td>
<td>36</td>
<td>40</td>
<td>35</td>
<td>12</td>
<td>32</td>
</tr>
</tbody>
</table>
Practical Advantages of Decoupled Look-ahead

- Micro helper thread based approach:
  - Targets top cache misses and branch mispredictions (low coverage)
  - Support for quick spawning and register communication (not trivial)
Practical Advantages of Decoupled Look-ahead

- Micro helper thread based approach:
  - Targets top cache misses and branch mispredictions (low coverage)
  - Support for quick spawning and register communication (not trivial)

- Decoupled look-ahead approach:
  - Easy to disable, low management overhead on main thread
  - Natural throttling to prevent run-away prefetching, cache pollution

---

Raj Parihar, Michael C. Huang
Advanced Computer Architecture Lab  University of Rochester
Practical Advantages of Decoupled Look-ahead

- **Micro helper thread based approach:**
  - Targets top cache misses and branch mispredictions (low coverage)
  - Support for quick spawning and register communication (not trivial)

- **Decoupled look-ahead approach:**
  - Easy to disable, low management overhead on main thread
  - Natural throttling to prevent run-away prefetching, cache pollution