Accelerating Decoupled Look-ahead to Exploit Implicit Parallelism

by

Raj Parihar

Submitted in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy

Supervised by

Professor Michael C. Huang

Department of Electrical and Computer Engineering Arts, Sciences and Engineering Edmund A. Hajim School of Engineering and Applied Sciences

University of Rochester
Rochester, New York

2016
Dedication

To my parents, wife, siblings, friends

and

To all those who are never afraid of asking “Why?”
Biographical Sketch

Raj Parihar was born in 1984 in Satna, a small town in the state of Madhya Pradesh in central India. He graduated from Birla Institute of Technology and Science - Pilani, India in 2006, with Bachelor of Engineering degree in Electrical and Electronics Engineering. Upon graduation in 2006, Raj joined Microchip Technology and worked there for two years. While working on MIPS based advanced microcontroller designs, Raj developed interest in computer architecture. He entered graduate studies at the University of Rochester in the Fall of 2008, pursuing research in computer architecture, under the direction of Professor Michael C. Huang. He received a Master of Science degree in Electrical and Computer Engineering from the University of Rochester in 2010. During his PhD studies, Raj has contributed to seven original articles in international peer-reviewed conferences and workshops. During the summer of 2012, Raj was an intern at IBM Research - Almaden where he worked on content protection and secure hardware architectures. Since January 2014, Raj has been working at Imagination Technologies as a CPU Performance Microarchitect in MIPS microarchitecture group. Besides work, Raj enjoys adventure sports, traveling and Indian classical music.

List of articles and publications:


Acknowledgments

It is a pleasure to thank many people who made this thesis possible.

First and foremost, my special gratitude to my PhD advisor, Prof. Michael C. Huang. Throughout my graduate studies, he provided sound advice, constructive criticism, good feedback and plenty of novel research ideas. I would like to extend my gratitude to my thesis committee members, Prof. Chen Ding, Dr. Jose Moreira, Prof. Engin Ipek and Prof. Sandhya Dwarkadas to agree to serve as chair of my defense.

I am indebted to my student colleagues for providing a stimulating and fun environment. I am especially grateful to Alok Garg, Arrvindh Shriraman, Xin Li, Aaron Carpenter, Jing Xue, Ananya Pareek, Ram Kumar Sabesan, Shantanu Agrawal, Balaraman Rajan, Rohit Nayak, Omid Rajabi, Renzo Bustamante and Sushant Kondguli.

I am thankful to my mentor at Imagination, Dr. Wayne Yamamoto; and my colleague Harit Modi who helped me to develop a good understanding of performance debug and state-of-the-art simulation framework. I would also like to thank my undergraduate advisors, Prof. Anu Gupta, Dr. Chandra Shekhar; and my mentor at Microsoft Research, Dr. Kentaro Toyama. My special thanks to high school teacher, Mr. Puspraj Singh; middle school teacher, Mr. Ramesh Bhatt; and primary school teacher, Mr. Sharda Prasad Dwivedi.

I would like to thank Sheila Hank and Joan David (my host families at Rochester) for treating me like their own family member and inviting me to numerous occasions – a wide range of events that included barn dances and Christmas dinners.

I wish to thank my friend from high school, Vikash Patel; my friend from undergraduate, Satyam Srivastava; and my roommate and friends during graduate school, Himanshu Shekhar and Karla Mercado; for helping me get through the difficult times, camaraderie and the caring they provided. My special thanks to my wife, Richa, for her continuous support and interest in what I do. I also thank my sister, Khusboo and my brothers, Ravinandan and Tushar.

Lastly, and most importantly, I wish to thank my parents, Neelam Singh and Krishna Bali Singh. They raised me, supported my every decision, nurtured me, inspired me and loved me unconditionally.
Abstract

Despite the proliferation of multi-core and multi-threaded architectures, exploiting *implicit* parallelism for a single semantic thread is still a crucial component in achieving high performance. While a canonical out-of-order engine can effectively uncover implicit parallelism in sequential programs, its effectiveness is often hindered by instruction and data supply imperfections (manifested as branch mispredictions and cache misses). *Look-ahead* is a tried-and-true strategy to exploit implicit parallelism, but can have resource-inefficient implementations such as in a conventional, monolithic out-of-order core. A more decoupled approach with an independent, dedicated look-ahead thread on a separate thread context can be a more flexible and effective implementation, especially in a multi-core environment. While capable of generating significant performance gains, the look-ahead agent often becomes the new speed limit; thus, we explore a range of software and hardware based techniques for accelerating the look-ahead agent to exploit implicit parallelism.

Fortunately, the look-ahead thread has no hard correctness constraints and presents new opportunities for optimizations which are not present in traditional architecture. First, we explore speculative parallelization in the look-ahead thread which is especially suited for the task of accelerating the look-ahead agent. Second, we observe that not all dependences are equal, and some links in a dependence chain are *weak* enough that removing them in the look-ahead thread does not materially affect the quality of look-ahead. A trial-and-error approach and a genetic algorithm based *self-tuning* framework can reliably identify weak instructions to improve the speed of the look-ahead thread. We further tune look-ahead code via skipping side-effect free, non-critical (we call them *Do-It-Yourself* or DIY) branches. Finally, we apply self-tuning principles in a multi-program environment to improve overall protection and utilization of shared caches which are often shared among multiple competing programs.

With a series of faithful simulation experiments and detailed insightful analysis, we show that while the two main drivers for single-thread performance – faster clocks and advancements in microarchitecture – have all but stopped in recent years, we can still uncover significant implicit parallelism using *intelligent* look-ahead techniques which brings impressive performance gain at relatively low cost.
Contributors and Funding Sources

While I am the author of this dissertation, this work would not have been possible without the collaboration with various students and professors. I would like to thank my adviser, Prof. Michael C. Huang, for providing valuable suggestions and technical guidance throughout the graduate studies and research. This work was supported by a dissertation committee consisting of Prof. Michael C. Huang (adviser) of the Electrical and Computer Engineering Department, Prof. Chen Ding of the Computer Science Department, Prof. Engin Ipek of the Computer Science and Electrical Engineering Departments, Dr. Jose Moreira of IBM Thomas J. Watson Research Center, and Prof. Sandhya Dwarkadas of the Computer Science Department at University of Rochester.

The speculative parallelization for decoupled look-ahead, described in Chapter 3, is a result of collaboration with Alok Garg and Michael C. Huang and was published in PACT’11. My contribution to this work was to generalize the overall framework of speculative parallelization and use a credit based mechanism that improves the overall efficiency.

Removal of weak dependences to accelerate the decoupled look-ahead, presented in Chapter 4, is a result of numerous discussions with Michael C. Huang and was published in HPCA’14. This exploration would not have been possible without the support of CIRC computing platforms (Bluehive) at University of Rochester.

Look-ahead skeleton tuning and Do-It-Yourself (DIY) branches to further accelerate the look-ahead agent, described in Chapter 5, is also a result of discussions with Michael C. Huang. A short paper (DIY Branches) has been published in ACM Student Research Competition held with PACT’15. A full-length paper has been submitted to HPCA’17 and is under review.

Self-tuning in shared caches via rationing, presented in Chapter 6, is a result of collaboration with Jacob Brock, Chen Ding and Michael C. Huang. Collaborative caching, software based hints and integration of these ideas with Rationing was contributed by Jacob Brock. A short paper has been published in PACT’14 and a full-length paper has been accepted in ISMM’16 which won the best student paper award as well.
Detailed analysis of various performance bottlenecks, potentials of baseline and decoupled look-ahead systems are the culmination of a series of discussions with Michael C. Huang, Alok Garg and Shushant Kondguli.

Finally, decoupled look-ahead research is supported by NSF CAREER award CCF-0747324 and also in part by NSFC under the grant 61028004.
# Table of Contents

Dedication ii

Biographical Sketch iii

Acknowledgments v

Abstract vi

Contributors and Funding Sources vii

List of Tables xvii

List of Figures xx

1 Introduction 1
   1.1 Motivation and Context of Work 1
   1.2 The Problem: Speed of the look-ahead agent is often the bottleneck in the decoupled look-ahead 3
   1.3 Thesis Statement 5
   1.4 Contributions 5
   1.5 Thesis Organization 6

2 Background and Related Work 7
   2.1 Decoupled Architectures 8
      2.1.1 Decoupled Access/Execute Architecture and Early Work 8
      2.1.2 Decoupling to Mitigate the Memory Latency 10
      2.1.3 Control Flow based Decoupling 10
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1.4 Recent Decoupling Proposals</td>
<td>11</td>
</tr>
<tr>
<td>2.2 Helper Threading</td>
<td>11</td>
</tr>
<tr>
<td>2.2.1 Lightweight Microthreads or Subordinate Threads</td>
<td>13</td>
</tr>
<tr>
<td>2.2.2 Decoupled Look-ahead Paradigm</td>
<td>15</td>
</tr>
<tr>
<td>2.2.3 Microthreads vs Decoupled Look-ahead</td>
<td>16</td>
</tr>
<tr>
<td>2.3 Program Slicing for Helper Threading</td>
<td>18</td>
</tr>
<tr>
<td>2.4 Baseline Decoupled Look-ahead Architecture</td>
<td>19</td>
</tr>
<tr>
<td>2.4.1 Basic Hardware Support</td>
<td>19</td>
</tr>
<tr>
<td>2.4.2 Software Support for Look-ahead</td>
<td>21</td>
</tr>
<tr>
<td>2.4.3 Code Address Space and Runtime Recoveries</td>
<td>22</td>
</tr>
<tr>
<td>2.4.4 Summary of Performance Benefits</td>
<td>22</td>
</tr>
<tr>
<td>2.5 Bottlenecks in Baseline Decoupled Look-ahead</td>
<td>24</td>
</tr>
<tr>
<td>2.6 Summary</td>
<td>26</td>
</tr>
<tr>
<td>3 Speculative Parallelization in Decoupled Look-ahead</td>
<td>27</td>
</tr>
<tr>
<td>3.1 Background: Speculative Parallelization</td>
<td>28</td>
</tr>
<tr>
<td>3.2 Motivation for Speculative Parallelization</td>
<td>29</td>
</tr>
<tr>
<td>3.3 Software Support</td>
<td>32</td>
</tr>
<tr>
<td>3.3.1 Dependence Analysis</td>
<td>32</td>
</tr>
<tr>
<td>3.3.2 Selecting Spawn and Target Points</td>
<td>33</td>
</tr>
<tr>
<td>3.3.3 Loop-level Parallelism</td>
<td>34</td>
</tr>
<tr>
<td>3.3.4 Available Parallelism</td>
<td>34</td>
</tr>
<tr>
<td>3.4 Hardware Support</td>
<td>35</td>
</tr>
<tr>
<td>3.4.1 Spawning a New Look-ahead Thread</td>
<td>36</td>
</tr>
<tr>
<td>3.4.2 Support for Register Access</td>
<td>36</td>
</tr>
<tr>
<td>3.4.3 Support for Memory Access</td>
<td>38</td>
</tr>
</tbody>
</table>
3.5 Runtime Management and Control ........................................... 39
3.5.1 Spawning Thread and Management ................................... 39
3.5.2 Communicating Branch Predictions to Primary Thread .......... 40
3.6 Experimental Setup .......................................................... 41
3.6.1 Microarchitecture and Configuration ................................. 41
3.6.2 Applications and Inputs .................................................. 42
3.7 Experimental Analysis ....................................................... 42
3.7.1 Performance Analysis ..................................................... 42
3.7.2 Microarchitectural Sensitivity .......................................... 44
3.7.3 Comparison with Conventional Speculative Parallelization ..... 45
3.7.4 Recoveries ................................................................. 46
3.7.5 Partial Recoveries .......................................................... 47
3.7.6 Quality of Spawns ........................................................ 48
3.7.7 Effect of Speculation Support ......................................... 49
3.7.8 Potential for Multiple Look-ahead Threads ......................... 51
3.7.9 Reasons for Extra Recoveries ........................................... 52
3.8 Future Work ................................................................. 53
3.8.1 Intelligent Runtime Spawning Management ......................... 53
3.8.2 Credit Based Spawn-Target Pairs ..................................... 54
3.8.3 Simplified Design with Stable and Stipulated Pairs: ............ 55
3.8.4 Potential and Performance Mismatch Exploration ................ 56
3.8.5 Additional Insights and Explorations ................................. 56
3.9 Summary ................................................................. 57

4 Weak Dependence Removal in Decoupled Look-ahead .......... 58
4.1 Background and Related Work ........................................... 59
5.1 Motivation .................................................. 108
  5.1.1 Idea of Do-It-Yourself Branches ..................... 109
  5.1.2 Customized Skeletons ................................ 110
5.2 Performance Potentials ................................. 111
5.3 Methodology and Tuning Framework ..................... 113
  5.3.1 Static DIY Transformations ........................... 113
  5.3.2 Temporal Slicing of DIY Branches ................... 114
  5.3.3 Combining DIY Branches ............................... 116
  5.3.4 Skeleton Tuning and Customization .................. 116
  5.3.5 Putting It Together .................................. 118
5.4 Architectural Design and Support ....................... 118
  5.4.1 Software Analysis and Support ....................... 119
  5.4.2 Additional Hardware Support ......................... 119
  5.4.3 Runtime Management Support ......................... 121
  5.4.4 In a Nutshell ........................................ 121
5.5 Experimental Setup ...................................... 122
  5.5.1 Applications and Inputs .............................. 122
5.6 Performance Analysis .................................... 123
  5.6.1 Overall Performance .................................. 123
  5.6.2 Comparison with Weak Dependence ................... 125
  5.6.3 Comparison with Turbo Boost Technology ............ 126
5.7 Diagnostics Analysis ..................................... 128
  5.7.1 Look-ahead Core Simplification ...................... 128
  5.7.2 Optimal Skeleton Distribution ....................... 130
  5.7.3 Static Code and Skeleton Affinity ................... 131
  5.7.4 Duty Cycle Sensitivity ............................... 131
6.6.4 Hardware-software Collaborative Caching ........................................ 163
6.7 Related Work ...................................................................................... 165
  6.7.1 Cache Partitioning ................................................................. 165
  6.7.2 Cache Management ............................................................. 167
  6.7.3 Collaborative Caching ............................................................ 168
  6.7.4 Resource Allocation ............................................................... 168
6.8 Summary ......................................................................................... 169

7 Summary and Future Work .......................................................... 170
  7.1 Summary ...................................................................................... 170
    7.1.1 Speculative Parallelization in Decoupled Look-ahead .............. 171
    7.1.2 Weak Dependence Removal in Decoupled Look-ahead .......... 171
    7.1.3 Look-ahead Skeleton Tuning and Do-It-Yourself Branches ....... 172
    7.1.4 Self Tuning (Rationing) in Shared Cache Management ............ 172
  7.2 Future Explorations ......................................................................... 173
    7.2.1 SMT based Single-core Decoupled Look-ahead ....................... 173
    7.2.2 Machine Learning based Self-Tuning Look-ahead Process ....... 174
    7.2.3 Cost Effective Heuristics for Skeletons ................................... 177
    7.2.4 Value Prediction Using Look-ahead Thread ......................... 177
  7.3 Closing Remarks: Potential Impact .............................................. 178

Bibliography ......................................................................................... 180
List of Tables

2.1 Summary of top instructions accountable for 90% and 95% of all last-level cache misses and branch mispredictions. Stats are collected on entire run of ref input. DI is the total dynamic instances (% of total dynamic instructions). SI is total number of static instructions. .................................................. 17

2.2 Summary of performance benefits of the baseline decoupled look-ahead system. .................................................. 23

3.1 Core configuration. .................................................. 42

3.2 IPC of baseline (BL), baseline decoupled look-ahead (DLA), and speculatively parallel look-ahead (SP LA). .................................................. 44

3.3 Microarchitectural parameters of Cortex-A9 and POWER5 cores. .................................................. 44

3.4 Recovery rate for the baseline and speculatively parallel look-ahead systems. Rates are measured by the number of recoveries per 10,000 committed instructions in the main thread. .................................................. 47

3.5 Partial recoveries in the speculative look-ahead. Recv-Merge are the instances when a recovery happens in the main look-ahead thread and spawned thread merges later. Live 200 and Live 1000 are the instances when a Recv-Merge occurred and merged thread didn’t experience a new recovery for at least 200 and 1000 instructions respectively. .................................................. 48

3.6 Breakdown of all the spawns. Wrong Path refers to the case the spawn happens when the primary look-ahead thread has veered off the right control flow and will eventually encounter a recovery. For crafty, eon, and gzip we do not see any spawns in our experiments. .................................................. 49
3.7 Slowdown compared to baseline speculative parallelization and recoveries rate per 100,000 instructions due to not passing of values produced in primary thread to secondary look-ahead thread after spawning. ................................. 50

4.1 Microarchitectural configurations. ................................................................. 84

4.2 IPC of single-threaded baseline, baseline decoupled look-ahead and genetic algorithm based self-tuned look-ahead. ......................................................... 87

4.3 Dynamic size of baseline and GA based self-tuned look-ahead thread relative to the original program binary and static instruction count of the original program and those removed by the framework (last row). A - Baseline look-ahead skeleton (%dyn), B - GA tuned look-ahead skeleton (%dyn), C - Total program instructions (static), D - Instructions in 100m window (static), E - Individual weak instructions (static), E - Instructions removed using GA (static). ........ 90

4.4 Reboot rate for baseline and the self-tuned decoupled look-ahead systems. The rates are measured by the number of reboots per 100,000 committed insts in the main thread. ................................................................. 90

4.5 Program execution and profile time for offline GA based system. ................. 92

4.6 Average number of modules, their dynamic instance size (in 1000 instructions) and total number of genes per module in SPEC 2000 benchmarks for complete run using ref inputs. ................................................................. 96

4.7 Top 5 weak instructions and their respective cycles savings (Fitness) normalized to baseline look-ahead system’s cycles; also showing their dynamic code contributions (Freq) to the program execution of 100 million instructions. .... 97

4.8 Slices containing the weak instructions. Weak instructions are shown in bold. .. 100

5.1 Total DIY branches (Seeds), good DIY seeds (Good) and upper bound of performance potential per application. ................................................................. 113

5.2 Skeleton identifiers and associated transformations. ........................................ 116

5.3 Microarchitectural configurations. ................................................................. 123
5.4 IPC of single-thread (ST), baseline look-ahead (LA), and DIY+skeleton tuned
decoupled look-ahead (LB) .................................................... 123
5.5 Performance and power overhead comparison of Turbo Boost systems over
baseline. ............................................................................ 127
5.6 A few examples of DIY code modules. .................................... 133
5.7 Profile overhead comparison of DIY branches and skeleton tuning with weak
dependence removal. ............................................................ 133
5.8 Comparison of DLA configs with idealized systems. ....................... 135
6.1 Microarchitectural configurations. ............................................ 152
6.2 SPEC 2000 applications used in experiments and their index used in the result
figures. .................................................................................. 153
6.3 Overall protection and utilization for all test suites. ......................... 158
List of Figures

1.1 Implicit parallelism of integer applications. The first set of three bars on the left indicate the amount of available parallelism (instructions/cycle) when inspected with a moving window of 128, 512, or 2048 instructions. The three bars on the right repeat the same experiments only this time under realistic branch misprediction and cache miss situations that further constrain the available parallelism to be exploited. Vertical axis is in log scale. 2

2.1 A generic Decoupled Access/Execute (DAE) architecture [1]. 8

2.2 Depiction of generic microthread based system (A), and possible implementation of shared register file (B) [2]. 14

2.3 A generic decoupled look-ahead architecture [3]. 20

2.4 Illustration of biased branch and long distance store-load pair optimization in the look-ahead binary (skeleton). 21

2.5 Comparison of various performance potentials and projected decoupled look-ahead speedup with an RTL/FPGA accurate performance simulation (IMG-psim) framework. 24

2.6 Performance comparison of 4 configurations. Shown in the bars are baseline single core (left) and a decoupled look-ahead system (right). Two upper-bounds are shown: the performance of a single core with idealized branch predictions and perfect cache accesses (curve with circles), and the approximate speed limit of the look-ahead thread (gray wide curve indicating approximation). The applications are sorted with increasing performance gap between the decoupled look-ahead system and the prediction- and accesses-idealized single-core system. 25
3.1 Dependence characteristic changes due to skeleton construction. In this example from *quake*, each box represents a basic block. For clarity, only a subset of instructions are shown. Instructions that are not on the skeleton are shown in gray color. The arcs show the dependences and their distances measured in numbers of basic blocks.

3.2 Illustration of how spawning a secondary thread naturally preserves look-ahead.

3.3 Example of basic-block level parallelization.

3.4 An approximate measure of available parallelism in the trace of the look-ahead thread and the main program thread (top figure). A more constrained measure of parallelism that is likely to be exploited (bottom figure).

3.5 Illustration of runtime look-ahead thread spawn and merge.

3.6 Register renaming for the spawned thread. ‘O’ in map refers to “ownership” bit.

3.7 Example of a banked branch queue. Bank 0 and bank 1 are written by primary (older) look-ahead and spawned (younger) look-ahead threads respectively. Primary thread uses global head pointer, currently pointing to an entry in bank 0, to read branch predictions.

3.8 Speedup of baseline look-ahead and speculatively parallel look-ahead over single-core baseline architecture. Because of the uneven speedups, the vertical axis is log-scale.

3.9 Speculative parallelization performance improvement for a simple (Cortex-A9) and a more aggressive (POWER5) out-of-order microarchitecture.

3.10 Comparison of the effect of our speculative parallelization mechanism on the look-ahead thread and on the main thread.

3.11 Speedup comparison of regular support and two other alternatives, one removing the partial versioning support altogether, the other adding dependence violation detection to squash the spawned thread.
3.12 Number of instructions executed in two look-ahead thread mode and number of instructions missed the spawning opportunity because of limited hardware resources in 100 million simulation window. .................................................. 51

3.13 Number of average spawning opportunities over time in INT and FP applications. .............................................................. 52

4.1 Various phases in a genetic algorithm based evolution process. .................. 60

4.2 Example of weak dependences in the original program binary. In these examples from applications vpr and mcf, each box represents a basic block. For clarity, only a subset of instructions are shown. Frequent values of inputs and output registers – captured using a profiler – are shown on the right hand side. Shown in bold are instructions that are weak and can be safely removed without impacting the quality of look-ahead. ......................... 62

4.3 Performance impact of cumulatively removing instructions identified as weak in isolation. ................................................................. 65

4.4 Multi-instruction based genes from applications vpr and gzip. (A) Removal of either of the bold instruction results into large value of $t_{10}$ register which is “very different from actual value of $t_{10}$ on which the branch outcome depends. (B) The branch outcome depends upon the $s_3$ register which is indirectly dependent upon $gp$ register through certain offset based loading. Removal of either of the bold instruction again results into large value of $s_3$ register which is quite different from actual value of $s_3$ that is close to 0. ........................................... 71

4.5 Look-ahead chromosome representation. Genes are basic blocks of chromosomes and can be either single-instruction or multi-instruction. Marking of a gene (shown as X) knocks out the associated instructions from the chromosome (look-ahead binary). ................................................................. 72

4.6 Genetic algorithm based framework to refine look-ahead skeleton by removing weak dependences. ................................................................. 75
4.7 Correlation between speedups and norm. recoveries of initial genes/chromosomes. .................................................. 81
4.8 Simulation framework to refine look-ahead skeleton. ....................... 85
4.9 Speedup of baseline look-ahead and genetic algorithm-based self-tuned look-ahead over single-core baseline architecture. The vertical axis is log scale. . 86
4.10 Speedup of baseline look-ahead and genetic algorithm based self-tuned look-ahead over single-core baseline architecture for SPEC 2006 applications. The vertical axis is log scale. ........................................ 87
4.11 Speedup of speculative parallel look-ahead before and after applying GA based weak dependence removal mechanism. We also show the stand alone speedup of GA based baseline look-ahead. ............................. 88
4.12 Normalized gains achieved through generations of evolution. ............ 91
4.13 Comparison of single-thread, baseline look-ahead and self-tuned look-ahead using online evolution for *equake* (faster evolution) and *art* (slower evolution). X-axis represents the number of instructions. Note that the x-axis scale is different in two cases. ........................................ 93
4.14 Impact of code size on relative performance gain (in %) and saturation generation. ......................................................... 94
4.15 Speedup of various configurations: baseline (BL) baseline decoupled look-ahead (BL-DLA), and genetic algorithm tuned decoupled look-ahead (GA-DLA) for various L2 cache sizes. All results are geometric means of speedup relative to single-threaded execution on a 1MB L2 system. ................ 95
4.16 Comparison of generational solutions using full-run fitness tests (precise) and sampling based fitness test (sampling) for two representative applications. . . 98
4.17 Distribution of weak (a) and regular (b) insts by their static attributes (opcode and no. of operands). Radius of circle indicates the relative weight of specific type of instruction. .................................................. 99
4.18 Comparison of state-of-art decoupled look-ahead system with previous proposals. The latency to copy register file is set to 64 cycles for DCE. Speedups are normalized to a 4-wide, out-of-order single-thread baseline with a state-of-art L2 stream prefetcher.

4.19 Speedup of baseline ZVS heuristics and GA based ZVS scheme over single-thread.

4.20 Redundant loads in INT applications.

5.1 A DIY branch based code example from 176.gcc.

5.2 Performance potential of skeleton tuning as a function of epoch size – normalized to baseline look-ahead. We omit one high potential application equake from this plot (not from the Geomean) for better visualization.

5.3 DIY branch code transformations of conditional branches (A), and loops (B) in the look-ahead thread. Broken lines indicate NOP instructions or unreachable control paths.

5.4 Various steps of skeleton tuning process.

5.5 Additional hardware support needed to handle DIY branches is shown in shaded (blue) color.

5.6 Performance speedup of baseline decoupled look-ahead, DIY branch enabled look-ahead, skeleton tuned look-ahead and combination of DIY branches + skeleton tuned look-ahead over single-thread baseline.

5.7 Performance improvement of skeleton tuning + DIY branch based look-ahead applied on top of weak dependence removed look-ahead. Also showing the weak dependence removed look-ahead for individual comparisons.

5.8 Performance comparison of load balancing (DIY + skeleton tuned) DLA with two flavors of turbo boosted single-thread systems.

5.9 Performance impact of simpler look-ahead core pipeline on baseline and final decoupled look-ahead system. X-axis shows the back-end pipeline width of look-ahead core.
5.10 Power consumption of 3-wide and 2-wide look-ahead core components over the 4-wide baseline look-ahead core. Also showing the power reductions of entire decoupled look-ahead system (last set of bars labeled as “Total”). 130

5.11 Optimality distribution of various types of skeletons in all applications. 131

5.12 Duty cycle sensitivity of WeightAdj() from *art*. 132

5.13 Performance potential of idealized system and comparison with various decoupled look-ahead systems (for SPEC apps). 134

6.1 Space diagram of ratiored cache showing ration trackers (1/set per core), block owner and access bit (1 per cache block). 142

6.2 Energy efficient rationing: after all the ration counters and access-bits are reset, a still zero-valued ration counter at next reset point can be used to either evict the lines belonging to respective core or send them to a power-saving mode to reduce leakage. 146

6.3 An example of cache rationing with an MRU-hint bit. If the hint bit is set, the access bit is not set so that the accessed blocks will not be kept in the cache. The contents of a size 4 cache are shown after each pair of accesses, and blocks with their access bit zeroed are shown as outlined letters. 148

6.4 Illustration of resource protection (a) and capacity utilization (b) in evenly rationed cache in comparison with equal partitioning and no partitioning policies for a 4-way cache shared among 2 cores. Core access patterns are shown in the left and the accesses that experienced misses are shaded in the red. 150

6.5 Comparison of cache rationing with baseline PIPP policy. PIPP-equal (PIPP with equal partitioning) for symmetric co-run incurs more misses (shaded in red), and allows a dead line to stay, increasing contention on the second half of the set. 151
6.6 SPEC 2000 symmetric co-runs of 2- (upper) and 4- (lower) programs. In the 2-program co-runs, IPCs are normalized to respective solo runs with half the cache. In the 4-program co-runs, IPCs are normalized to equal partitioning. Co-running applications (index to name mapping from Table 6.2) are adjacent to each other (shown in the same shade) and different co-runs are separated by alternating shades.

6.7 SPEC 2000 co-run with low cache demand application eon. First bar in each pair represents eon and the second bar represents one of the spec program (name is in the Table 6.2). Performance of co-running pairs are normalized to respective solo runs with half the cache.

6.8 SPEC 2000 co-run with extremely high cache pressure mcf. First bar in each pair represents mcf and second bar represents one of the spec program (name is in the Table 6.2). Performance is normalized to their respective solo runs with half the cache.

6.9 Access-bit reset interval sensitivity for equake pair-run.

6.10 Comparison of fine-grained (per set) and coarse-grained (global) rationing for symmetric co-runs.

6.11 equake and lucas co-run. Collaborative rationing obtains full protection while the hardware-only rationing does not.

6.12 Pacman miss ratios for cache rationed and cache partitioned simulations of Streaming and SOR. Partitioned-cache points are plotted for cache sizes of 0.125MB, 0.25MB, 0.5MB, and 1MB. For rationed-cache points, the horizontal axis represents the cache ration as a percentage of the total 1MB cache.

7.1 Random forest based learning framework to identify and predict weak instructions. Output of the model is a Boolean information whether the instruction is weak or strong. Input to the model is instruction under test, its attributes along with limited context information in which the instruction exists.
Chapter 1

Introduction

1.1 Motivation and Context of Work

As CMOS technology edging towards the end of the projected ITRS roadmap, scaling no longer brings significant device performance improvements. At the same time, increasing transistor budgets are allocated mostly towards improving throughput and integrating more functionality on the chip. Without these traditional driving forces, improving single-thread performance for general-purpose programs is undoubtedly more challenging. Despite the ubiquity of multi-core and multi-threaded architectures, high single-thread performance is still an important processor design goal. It not only provides *automatic* performance benefits and thereby improves productivity, but is also important for overall performance even for explicitly parallel programs in speeding up serial sections to avoid the Amdahl’s bottleneck.

Unfortunately, the two main drivers for single-thread performance – faster clocks and advancements in microarchitecture – have all but stopped in recent years. The future for single-thread performance might appear bleak. However, that appearance can be misleading. There is no evidence of a fundamental lack of parallelism in sequential codes. In fact, limit studies (similar to [4]) that we have conducted confirm that modern codes are just like older codes and have significant potential in implicit parallelism: As shown in Figure 1.1, even the integer codes have a geometric mean parallelism of about 20 insts/cycle even if we only look ahead a few 100 instructions. The real question is whether we can realize the potential without undue costs.
Out-of-order microarchitecture can effectively exploit inherent parallelism in sequential programs, but its capability is often hindered by imperfect instruction and data supply: branch mispredictions inject useless, wrong-path instructions into the execution engine; while cache misses can stall the engine for extended periods of time, reducing effective throughput. Elaborate branch predictors, deep memory hierarchies, and sophisticated prefetching engines are routinely adopted in high-performance processors. Yet, cache misses and branch mispredictions continue to be important performance hurdles. One general approach to mitigate their impact is to enable deep look-ahead so as to overlap instruction and data supply activities with the instruction processing and execution.

The sequential code structure, rich with control flow instructions, is an inherently poor format to reveal the independence of computation. Without an external agent to mark up the code and reveal independence, an execution system trying to exploit implicit parallelism has little choice but to inspect the instruction stream according to its sequential order. This inspection process can reveal many data and control dependences and indirectly deduce the lack thereof. Since this inspection can be done without actually executing instructions, in theory we can look arbitrarily far ahead of actual execution, and thus search for long-range opportunities for parallelism. In practice, in a conventional out-of-order microarchitecture, look-ahead is tightly coupled with the rest of the processing. While inspecting the instruction, the core already starts to allocate significant resources related to various stages of instruction processing (such as issue
queue entries and physical registers). As a result, looking too far ahead quickly becomes too expensive to be practical!

One way to address the resource constraint is to decouple the inspection activities for look-ahead purposes from the normal processing. In particular, given the proliferation of multicore architectures, it is conceivable to launch a look-ahead thread on a different core and pass on relevant information to the original program thread [3, 5]. This certainly comes with its own costs as decoupling inevitably leads to some duplication. However, it certainly allows long-range look-ahead and can be particularly effective in mitigating long-latency cache misses. Indeed, a number of such decoupled look-ahead designs have been proposed specifically to hide the off-chip memory access latency [6–8]. Clearly, for a decoupled look-ahead system to be effective, the look-ahead thread needs to be fast without compromising the quality of the information passed on.

1.2 The Problem: Speed of the look-ahead agent is often the bottleneck in the decoupled look-ahead

An important challenge in achieving effective look-ahead is to be deep and accurate at the same time. Simple, state machine-controlled mechanisms such as hardware prefetchers can easily achieve “depth” by prefetching data far into the future access stream. However, these mechanisms fall short in terms of accuracy when the access pattern defies simple generalization. On the other hand, a more general form of look-ahead (we call it decoupled look-ahead) executes all or part of the program to ensure the look-ahead control flow and data access streams accurately reflect those of the original program. However, such a look-ahead agent needs to be fast enough to provide any substantial performance benefit. This thesis investigates a range of hardware/software techniques to accelerate the look-ahead agent to speed up the entire system.

First of all, we explore speculative parallelization in a decoupled look-ahead system. Intuitively, speculative parallelization is aptly suited to the task of boosting the speed of the decoupled look-ahead agent for two reasons. First, the code slice responsible for look-ahead (also known as the skeleton) does not contain all the data dependences embedded in the original pro-
gram, providing more opportunities for speculative parallelization. Second, the execution of the skeleton is only for look-ahead purposes, and thus the environment is inherently more tolerant to dependence violations. We find these intuitions to be largely borne out by experiments, and speculative parallelization can achieve significant performance benefits at a much lower cost than needed in a general-purpose speculative parallelization system [9].

A second technique targets the unique opportunity that, unlike regular threads, the look-ahead thread is not bounded by unyielding correctness constraints. In particular, we hypothesize that among the apparent dependences in a program, there are plenty of weak links. Cutting them in the look-ahead thread allows us to speed it up without much loss of effectiveness. However, this is not a trivial task. Intuitively, weakness is not absolute: the effect of removing a weak link depends on whether other links are removed, much like in the game Jenga. We design a genetic algorithm based self-tuning framework which can reliably identify and eliminate weak instructions from the look-ahead thread. Weak instruction removal framework can be implemented either as offline or online self-tuning system. We show that significant performance improvement can be obtained by eliminating weak instructions from the skeleton [10].

Extending the idea of self-tuning further, we propose two novel orthogonal techniques to tune the look-ahead thread’s workload. First, we use a static, profile-driven technique to tune skeleton for various code regions. Second, we accelerate sluggish look-ahead by skipping branch based, non-critical, side-effect free code modules that do not contribute to the effectiveness of look-ahead. We call them Do-It-Yourself (or DIY) branches for which the main thread does not get any help from the look-ahead thread, instead relies on its own branch predictor and data prefetcher. As a result, look-ahead thread skips DIY code regions and provides performance-critical assistance to improve the overall performance.

Finally, we present a detailed analysis of interaction among various techniques that we explored in this work. We conclude with a holistic view of numerous performance bottlenecks and potentials, before and after applying our techniques, in decoupled look-ahead system. As a future work, we plan to explore solutions to address the remaining bottlenecks and simplify implementation so that decoupled look-ahead can be easily adopted as a universal turbo boosting mechanism across various platforms – ranging from handheld devices to high end servers.
1.3 Thesis Statement

With the significant slowdown in clock speed and device scaling, intelligent look-ahead techniques offer an attractive microarchitectural solution and effective performance boosting mechanism to achieve much needed high single-thread performance by extracting significant implicit parallelism in the general-purpose sequential programs with moderate hardware support.

1.4 Contributions

This work makes the following major contributions:

- A framework to explore and apply speculative parallelization in the decoupled look-ahead to achieve deep and accurate look-ahead is presented. We demonstrate that a look-ahead binary encompasses relatively more data independence compared to regular programs; thus, lends itself to better thread level speculation. We also show that, due to the lack of correctness constraints, the hardware support needed for speculation and rollback in the look-ahead thread is much simpler than in a conventional architecture.

- With no hard requirement for correctness, we demonstrate that “weak” dependences can be safely eliminated in the look-ahead thread without considerably affecting the control flow and prefetch streams. A genetic algorithm based self-tuning framework has been developed which reliably identifies and eliminates the weak instructions to accelerate the look-ahead agent by reducing the length of critical sections. Additionally, the self-tuning framework can also be used effectively to refine baseline look-ahead heuristics, such as to identify useful prefetches, selective value prediction.

- We identify non-critical (DIY) branches and code regions empirically which can be safely dithered or skipped in the decoupled look-ahead thread to achieve better load balancing. We also show that the look-ahead skeleton can be tuned for various program phases using a simple static, profile-driven technique to maintain appropriate look-ahead distance that is crucial for the overall performance. Performance gains from DIY branches and skeleton tuning are easily combined together.
• We extend the idea of self-tuning to shared caches (we call it rationing) to improve overall protection and utilization in a multi-program environment where multiple programs often compete for the same shared resources. We show that rationing and collaborative caching can be easily combined together to reap the further benefits.

• Finally, we envision the look-ahead agent as a self-evolving, self-optimizing process which learns from its own execution in the past and remembers by keeping metadata information. Look-ahead agent can employ machine learning based prediction mechanisms to predict and identify weak instructions, branches, and less critical code regions to exploit them with reasonably smaller size training input.

1.5 Thesis Organization

The rest of the thesis is organized as follows: Chapter 2 discusses background and related work; Chapter 3 explores speculative parallelization in a decoupled look-ahead agent; Chapter 4 discusses the concepts of weak dependences in the context of the look-ahead threads and how they can be exploited to accelerate the look-ahead agent; Chapter 5 extends the idea of weak instructions to weak, less critical branches and code regions to achieve load balancing by dithering them in the look-ahead thread. We also explore techniques to further tune the look-ahead skeleton. Chapter 6 explores a form of self-tuning (known as rationing) to improve the overall protection and utilization of shared caches in a multi-program environment. Finally, Chapter 7 summarizes our insights and findings with a brief discussion of potential avenues for future explorations.
Chapter 2

Background and Related Work

Reducing program execution time has been a central goal of architectural research for decades. Microarchitectural optimizations have enabled current systems to run orders of magnitude faster than their predecessors. Traditional optimizations to achieve high performance have to be correct under all corner cases. This hard correctness requirement can significantly increase the design complexity and verification effort. Decoupled architectures have been used to achieve good performance while reducing the overall microarchitectural complexity compared to monolithic designs. To counter the correctness constraints, various proposals designed part of the microarchitecture in approximate fashion and used it primarily to provide hints. Helper threading based system improved the system performance by providing the high quality hints to the main thread. Look-ahead is a specific kind of helper threading in which a representative slice of the original program runs on a separate thread/core and provides execution based, high quality hints to the main architectural thread.

There have been numerous attempts in the past to improve the performance of a single semantic thread through decoupling, helper threading, and look-ahead. In this chapter, we summarize various proposals, designs, simulation methodologies and techniques for improving single-thread performance through helper threading and look-ahead techniques (Sections 2.1, 2.2, 2.3). We also discuss the underlying principles behind these approaches and key differences among them. Finally, we present a brief discussion of our baseline look-ahead architecture and its salient features – along with bottlenecks present in it – in Sections 2.4 and 2.5.
2.1 Decoupled Architectures

2.1.1 Decoupled Access/Execute Architecture and Early Work

Decoupling is an age old concept in computer architecture. One of the very first well known decoupled architecture designs was presented by Smith in the early 80s, which he termed Decoupled Access/Execute (DAE) architecture [1]. The DAE design tried to decouple memory accesses and operand fetching (executed on A-processor) from real computation (executed on E-processor) as depicted in Figure 2.1. In DAE architecture, A-processor executes ahead and results are communicated through hardware queues to E-processor. Branches are executed on both the processors to ensure that A-processor fetches the instructions and data from the correct control path. Smith and his colleagues subsequently studied several other variations of decoupled architecture, and presented the performance results for numerous decoupled designs and configurations [11]. In their study, the Lawrence Livermore loops were used as the workload which were executed on a decoupled architecture based on the CRAY-1 scalar architecture. Later on, Palacharla and Smith also proposed and studied the decoupling of integer execution unit in superscalar processors [12].

Figure 2.1: A generic Decoupled Access/Execute (DAE) architecture [1].
Goodman et al. presented PIPE (Parallel Instructions and Pipelined Execution) work which provides more thorough evaluation of the DAE architecture. PIPE architecture processors make extensive use of hardware queues in which two processors cooperate in executing the same task and communicate via hardware queues [13]. The ACRI work [14] was another early proposal for decoupled architecture which investigates the applicability of access and control decoupling in real world codes. Subsequent ACRI work, measured the loss of decoupling (LOD) and showed that some codes exhibit performance potential close to vector machines. They derived bounds for the performance of these codes and showed that, whilst some exhibit performance roughly equivalent to that on vector computers, others exhibit considerably higher performance potential in a decoupled system.

Farranes et al. studied and compared superscalar processors with the DAE architecture [15]. They presented simulation results for four different configurations and demonstrated that the architectural queues of the decoupled machines provide similar functionality to register renaming, dynamic loop unrolling, and out-of-order execution of the superscalar machines with significantly less complexity. They also showed that DAE architecture even outperforms the unconstrained superscalar on several loops. This can be explained by the fact that DAE architecture hides the memory latency naturally, which superscalar machines cannot achieve without loop unrolling or other compiler techniques.

Parcerisa et al. presented and evaluated a novel processor microarchitecture which combines two paradigms: access/execute decoupling and simultaneous multithreading [16]. They investigate the interaction of two techniques and how they complement each other. While decoupling features an excellent memory latency hiding efficiency, multithreading supplies the in-order issue stage with enough instruction level parallelism (ILP) to hide the functional unit latencies. Dorojevets et al. extended the idea of decoupled architecture to multi-threaded decoupled architecture [17]. Sung et al. also proposed a multi-threaded version of decoupled control/access/execute architecture and argued that such a decoupled architecture is more complexity-effective and scalable than comparable superscalar processors, which incorporate enormous amounts of complexity for modest performance gains [18]. Balasubramonian et al. evaluated a microarchitecture that dynamically allocates a portion of the processor’s physical resources to a future thread in order to exploit distant ILP in addition to nearby ILP [19].
2.1.2 Decoupling to Mitigate the Memory Latency

Numerous proposals studied the decoupled architecture to achieve efficient cache and memory prefetching. One design in this category is HiDISC (Hierarchical Decoupled Instruction Stream Computer) architecture which achieves high performance for loop-based scientific and signal processing programs by exploiting ILP and improving memory system performance through decoupled prefetching [20]. HiDISC work proposes to extract the cache accessing instructions from the original program and execute them on dedicated processors (known as cache management processors or CMPs) for each level of cache in the memory hierarchy.

Veidenbaum et al. proposed to mitigate the memory latency by placing a simple access processor in the memory itself [21]. A program is compiled to run as a computational process and several access processes with the latter executing in the DRAM core. Using multi-level branch prediction the access processor stays ahead of the execute processor and keeps the latter supplied with data. This system reduces latency by moving address computation to the memory, and thus avoids sending addresses by the computational processor. Jones et al. studied the data prefetching aspect of DAE architectures and compared them with superscalar machines [22].

Kurian et al. studied the memory latency effects in the decoupled architecture through detailed simulation and observed that decoupled architectures can reduce the peak memory bandwidth requirement, but not the total bandwidth [23]. On the other hand, data caches can reduce the total bandwidth by capturing locality. They concluded that despite the capability to partially mask the effects of memory latency, decoupled architectures still need a data cache.

2.1.3 Control Flow based Decoupling

Bird et al. introduced the term control decoupling proposing to not only decouple the memory address stream, but also the control stream [24]. More recent work on this, Dynamic Branch Decoupled Architecture, proposed dividing a program into two streams [25]. One stream is solely dedicated to resolving the branch outcome and target as soon as possible. Once the branch outcome is resolved it is passed to the other stream through a queue. The key motivation to execute the branch streams in advance is to eliminate the need for prediction by resolving the branches ahead of time.
2.1.4 Recent Decoupling Proposals

Recent proposals used decoupling to improve the reliability and functional correctness of the processors. In a conventional design, the reliability against errors due to particle strikes or circuit glitches is conservatively built into the hardware. Dynamic Implementation Verification Architecture (DIVA) is one of the first proposals that decoupled the implementation of correctness guarantee circuitry from each individual hardware structure to a separate functional checker unit [26].

On the performance front, Reinman et al. studied a range of possible optimizations which are enabled by a decoupled front end architecture [27]. In their proposal, a Fetch Target Queue (FTQ) is inserted between the branch predictor and instruction cache. This allows the branch predictor to run far in advance of the address currently being fetched by the cache. The decoupling enables a number of architectural optimizations, including multi-level branch predictor design, fetch-directed instruction prefetching, and easier pipelining of the instruction cache.

2.2 Helper Threading

Two major roadblocks in achieving near ideal performance in a single-thread system are memory stalls and useless speculations. Recent proposals target these performance bottlenecks by implementing a range of helper threading techniques utilize the idle resources in a multi-core or multi-threaded environment. Helper threading is a promising alternative to the conventional multi-threading computing paradigm in which a complete or partial copy of the original program pre-executes and assists the main thread to uncover implicit parallelism. The common purpose of helper threading is to mitigate branch misprediction and cache miss penalties by essentially overlapping branch and cache miss penalties (or “bubbles”) with normal execution. Numerous helper threading based techniques have been employed in many conventional architectures to achieve higher performance at reasonably low complexity [28–34].

Although helper threads can be used to extract all types of information to help exploit implicit parallelism, using them to mitigate cache misses and branch mispredictions will likely be the near-term research focus. First, the potential benefits of these targets are quite significant.
Many general-purpose programs see dramatic performance improvement if mispredictions and
cache misses are eliminated or reduced. From the ILP limit studies we conducted, it is clear that
by eliminating cache misses and branch mispredictions the amount of exploitable parallelism
significantly increases (Figure 1.1). Second, targeting mispredictions and cache misses is rel-
atively easier to accomplish and can be a cost-effective option. It takes little extra hardware
support to allow helper threads to warm up the shared caches and pass on the branch hints to
the main thread. Compared to single-purpose complex branch predictors and data prefetchers,
the design is more general-purpose and easy to suit different runtime demands.

From the correctness point of view, helper threading is quite different from traditional multi-
threading. In conventional multi-threading, every thread executes and commits in pre-defined
order to guarantee the correctness. On the other hand, helper threads only affect the performance
of the applications and therefore present more opportunities to apply aggressive optimizations
which are not possible in conventional systems. These helper threads can be created and opti-
mized either in static or in runtime [29, 35–41]. The strict correctness requirement mandates
that conventional architectures carefully track all the memory accesses to detect (potential) de-
pendence violations. In the case of helper threading, some dependences will also be violated,
but the consequence is less severe: (e.g., less effective latency hiding). This makes track-
ing of dependence violations potentially less critical or even unnecessary in helper threading
paradigm.

If there are additional cores or hardware thread contexts available, the helper threading
effort can be carried out in parallel with the program execution, rather than being triggered
when the processor is stalled. Helper threads can be targeted to specific instructions, such as
so-called delinquent loads [40], or can become a continuous process that intends to mitigate
all misses and mispredictions present in the original program. In the former case, the actions
are guided by backward slices leading to the target instructions and are spawned as individual
short threads, often called microthreads or subordinate threads [29, 35–41]. In the latter case,
a dedicated thread runs ahead of the main thread and can be a complete copy of the original
program [5, 6, 42–44] or a reduced version that only serves to prefetch and to help branch
predictions [3]. In the subsequent sections we will discuss these two flavors of helper threading
and point out the key differences between them.
2.2.1 Lightweight Microthreads or Subordinate Threads

Early work in helper threading explored a range of solutions to implement lightweight helper threads, also known as microthreads or subordinate threads, to target a handful of problematic instructions i.e. cache missing loads and hard-to-predict branches [2, 29, 35, 36, 36–41, 45–51]. Most of these works proposed to spawn a lightweight thread to issue prefetches or resolve branch outcomes before their actual execution in the main thread. Compared to the conventional single-thread out-of-order systems with aggressive prefetchers and branch predictors, they are flexible in target benefits (eliminating mispredictions or cache misses even with irregular patterns) and in the target distance. These microthreads can use regular thread contexts, but are perhaps more suited to thread contexts designed specifically for them [36]. Microthreads, sometime also known as nanothreads, are mostly hand-optimized helper threads or in some cases are extracted from the original program using a compiler. In most of these designs, microthread code is inserted into the main thread and shares execution resources with the main thread [2], as depicted in Figure 2.2.

Most of the microthreads are constructed in an independent manner and make only local decisions. While there are many proposals of microthreads, in subsequent paragraphs, we discuss only some of the well known work and their distinguishing features. Zilles et al. make use of speculative slices to drive lightweight helper threads which predict the branch outcome in runtime. Their design skips or invalidates the static branch predictions in runtime when they are certain that static prediction is not going to be useful [37, 45]. Luk et al. proposes to mitigate the memory latency through software controlled pre-execution in SMT processors [39]. Annavaram et al. devised a runtime dependence graph precomputation scheme to dynamically identify and precompute the addresses of cache missing loads and stores [38]. Similarly, Collins et al. proposed to speculatively identify and precompute delinquent load slices to prefetch using lightweight threads [40, 49].

Chappel et al. introduced the term Subordinate Simultaneous Multithreading (SSMT) which employs subordinate threads to improve the main thread’s branch prediction accuracy, cache hit rates, and prefetch effectiveness [36, 47, 48]. They also proposed to tweak the SSMT framework to target difficult-path branch prediction [47]. Finally, they discuss the microarchitectural
support needed for SSMT and address the issue of dynamically identifying and aborting the useless microthreads [48]. Zhang et al. proposed to create pre-computation threads that are dynamically constructed by a runtime compiler from the programs frequently executed hot traces, and are adapted to the memory behavior automatically. In their system, the construction and execution of the prefetching threads happens in another thread and imposes little overhead on the main thread [50].

Recent work on microthread based helper threading tries to extend the whole framework to the multi-core environment [52, 53]. As opposed to extracting a complete program slice for delinquent load, the helper thread extracts the slice for large loop based code sections [52]. Inter-core prefetching work exploits multiple cores to accelerate a single thread [53]. It also extends existing work on helper threads for SMT and multicore machines. They use one compute thread and one or more prefetching threads. Prefetching threads execute on the cores that would otherwise be idle, prefetching the data that the compute thread will need. The compute thread then migrates between cores, following the path of the prefetch threads, and finds the data already waiting for it.

Figure 2.2: Depiction of generic microthread based system (A), and possible implementation of shared register file (B) [2].
2.2.2 Decoupled Look-ahead Paradigm

As opposed to targeting only a handful of instructions through microthreads, another flavor of helper threading is to use either a complete or reduced version of the original program to guide the main thread execution by passing high quality speculative hints. Recent proposals including Slipstream [5, 54], Flea-flickers [8, 42], Dual-core execution [6], Tandem [43], Paceline [44] and Explicit Decoupled Architecture [3, 9] belong to this category. These proposals primarily differ in the way the look-ahead thread is created and the mechanisms they use to accelerate it. The helper threading model in which the look-ahead is a continuous, monolithic process and is executed on a separate general purpose thread context we call decoupled look-ahead.

The main attraction of the decoupled look-ahead approach is that it sidesteps some of the subtle implementation issues of the microthread approach. For instance, microthreads are often hand crafted and carefully inserted at the right location. A fully automated mechanism to make these decisions may have difficulty achieving similar effectiveness as the hand-crafted versions. Microthreads are numerous, and without substantial hardware support spawning these threads, passing necessary initial register and memory states, and receiving results from them can create significant overheads for the main thread (which offset their benefits). In contrast, a decoupled look-ahead thread is simple to generate and relies on less extra hardware support or micro-management from the main thread. Some of the well known designs of the decoupled look-ahead paradigm are discussed in the remainder of this section.

Slipstream divides a program into two streams: A-stream which is the advance thread and R-stream also known as the redundant thread [5, 54]. Slipstream tries to achieve forward progress by skipping non-essential computations. A-stream passes data and control flow outcomes to the full program thread, which is only “marginally” behind and known as R-stream. Outcomes of A-stream are checked against R-stream, and if a deviation occurs then R-stream’s architectural state is used to repair the A-stream’s corrupted architectural state. Dual-core execution, as the name suggests, uses a complete copy of the original program to provide look-ahead, and performs speculation on stalls due to long latency memory access [6]. Tandem and Paceline achieve acceleration of the look-ahead thread by improving the operating clock frequency of the look-ahead agent [43, 44].
Explicitly Decoupled Architecture proposed by Garg et al. tries to decouple the performance and correctness domains [3, 9, 55]. Information from the performance domain is communicated to the correctness domain through hardware-based queues and is treated as fundamentally speculative. The performance domain never writes to architectural states, thus removes the need for cache versioning. Also, the performance domain only runs a reduced version of the program (known as a skeleton) which mainly consists of hard-to-predict branches, second level cache misses and their backward data dependence chain. An attempt to make the look-ahead thread faster (which is often the bottleneck) was done in their subsequent work [9].

2.2.3 Microthreads vs Decoupled Look-ahead

While the two approaches are similar in principle, there are numerous practical advantages of using a single, continuous thread for look-ahead. First, as shown in Figure 2.3, the look-ahead thread is an independent entity and its execution and control is largely decoupled from the main thread. In contrast, embodying the look-ahead activities into a large number of short microthreads, as shown in Figure 2.2, inevitably requires micro-management from the main thread and entails extra implementation complexities. For instance, using extra instructions to spawn microthreads requires modification to the program and adds unnecessary overhead when the runtime system does not perform look-ahead (in multithreaded throughput mode).

Although it seems wasteful to run the program twice, in reality, the overhead is far smaller for decoupled look-ahead [3]. First, the look-ahead thread can be a stripped down version of the main thread, reducing redundancy. Second, successful look-ahead prevents the main thread from wasting resources on wrong-path instructions or idling. An optimal look-ahead thread provides large performance benefits at small energy cost. Because the look-ahead code mirrors the main program, there is a one-to-one mapping between the branch streams. This makes the passing of branch hints straightforward in the decoupled look-ahead. Using individual microthreads to selectively precompute certain branch outcomes requires extra support to match the producer and the consumer of individual branch hints. Third, prefetching too early can be counter-productive and should be avoided in any kind of helper threading. This becomes an issue when helper threads can also spawn other helper threads to lower the overhead on the main
Table 2.1: Summary of top instructions accountable for 90% and 95% of all last-level cache misses and branch mispredictions. Stats are collected on entire run of ref input. DI is the total dynamic instances (% of total dynamic instructions). SI is total number of static instructions.

<table>
<thead>
<tr>
<th></th>
<th>Memory References</th>
<th>Conditional Branches</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Top 90%</td>
<td>Top 95%</td>
</tr>
<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>gcc</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>perlbmk</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>

thread [40]. In decoupled look-ahead, since the look-ahead thread pipes its branch outcome through a FIFO to serve as hints to the main thread, it naturally serves as a throttling mechanism, stalling the look-ahead thread before it runs too far ahead. Additionally, addresses can go through a delayed-release FIFO buffer for better just-in-time prefetches for time-sensitive prefetching in first level caches.

Finally, as a program becomes complex and uses more ready-made code modules, “problematic” instructions will be more spread out, calling for more helper threads. The individual microthreads quickly add up, making the execution overhead comparable to a whole-program based decoupled look-ahead thread. As an illustration, Table 2.1 summarizes the statistics of instructions which are most accountable for last-level cache misses and branch mispredictions. For instance, the top static instructions that generated 90% of the misses and mispredictions in a generic baseline processor accounted for 8.7% (3.6%+5.1%) of the total dynamic instruction count. Assuming each problematic instruction instance is being targeted by a very brief 10-instruction long helper thread, the total dynamic instruction count for all the helper threads becomes comparable to the program size. If we target more problematic instances (e.g., 95%), the cost gets even higher. Due to these numerous advantages, we choose a decoupled look-ahead system for our exploration (which is also the more generic form of helper threading).
2.3 Program Slicing for Helper Threading

A program slice is itself an executable program whose behavior must be identical to the specified subset of the original program’s behavior. Program slicing [56] was originally proposed by Weiser in the early 80s and has seen a rapid development since the original definition. Two well known and popular program slicing techniques are static and dynamic program slicing [56–64]. Static program slicing, a compile-time version of the analysis, was first introduced in 1982, whereas runtime based dynamic slicing systems appeared in around 1988.

At first, slicing was only static and was applied on the source code with no other information than the source code. A static slice of a program is computed without making any assumptions regarding a program’s input, whereas dynamic program slicing relies on some specific test cases and inputs. Korel et al. introduced dynamic slicing [59], which works on a specific execution of the program for a given execution trace. Hall et al. proposed automatic extraction of executable program slices from the programs [65]. Slicing techniques have found applications in various areas such as debugging, program integration, and testing. Other applications include program comprehension, restructuring, downsizing, and parallelization.

Speculative slices have been used in the context of helper threading and data prefetching [37, 38, 40, 45, 46, 66]. From a helper threading point of view, the slices which are fast to execute, even with slight compromise in the correctness aspect, are the interesting slices. Many proposals have employed such slices to improve overall system performance by doing helper threading through them or precomputing the memory addresses in advance.

Zilles et al. proposed execution based prediction using speculative slices to improve the branch prediction accuracy [37]. In their proposal, correctness constraints of slices were relaxed and these slices were executed in a separate helper thread. This avoids the unnecessary stalls and squashes in the main thread. However, it was challenging to figure out the profitability of a slice before executing it. Collins et al. proposed to perform runtime memory address precomputation through back-end instruction analysis hardware, located off the processor’s critical path. Slice-processor [41] work dynamically identifies the frequently missing loads and extracts the relevant address computation slice on-the-fly. Such slices are then executed in parallel with the main sequential thread prefetching memory data. Annavaram et al. proposed to do data prefetching
by performing the pre-computation of the prefetch addresses for irregular access patterns, using dependence graph precomputation schemes [38].

Kim et al. proposed and designed a set of pre-execution algorithms to implement helper threading [51]. In their system, the compiler extracts pre-execution thread code from application code using static analysis which is guided by profile information. They primarily target load prefetching in their framework. One major difference of their technique was that pre-execution threads/slices were extracted using an automated system as opposed to manually constructing slices as in many prior proposals.

2.4 Baseline Decoupled Look-ahead Architecture

Due to numerous advantages of decoupled look-ahead, as pointed out in Section 2.2.3, we choose this kind of helper threading over microthreads for our explorations. This thesis proposal is built upon the idea of one particular design of decoupled look-ahead architecture. While a more detailed discussion of this design can be found in [3, 55, 67], a brief discussion of its salient features is called for. We discuss the basic hardware and software support needed for this decoupled look-ahead system. In this system, a statically generated look-ahead thread executes on a separate core and provides branch prediction and prefetching assistance to the main thread. We also summarize the performance benefits and other microarchitectural characteristics i.e. cache and branch mispredictions, and energy consumption. Finally, we present our analysis of performance bottlenecks present in the baseline decoupled look-ahead system and the motivations for look-ahead thread acceleration.

2.4.1 Basic Hardware Support

Like many decoupled look-ahead systems [3, 6, 54], our baseline hardware requires modest support on top of a generic multi-core system. Specifically, a FIFO queue (we call it branch queue or BOQ) is used to pipe the outcome of committed branch instructions to the main thread to be used as the branch predictions. These execution based branch hints are significantly more accurate compared to any kind of modern branch prediction mechanisms – including TAGE
branch predictors\(^1\). We also pass branch target address for those branches where the target was mispredicted in the look-ahead thread. The BOQ contains two bits per branch, one indicates the direction outcome, the other indicates whether a target is associated with this branch. If so, the main thread dequeues an entry from another shallower queue that contains the target address.

The memory hierarchy includes support to confine the state of the look-ahead thread to its private cache (to distinguish from the main core’s private cache, we call it L0 cache). Specifically, a dirty line evicted in the look-ahead thread is simply discarded and not written back to L2. A cache miss will only return the up-to-date non-speculative version from the main thread. Simple architectural support to allow faster execution of the look-ahead thread is used to reduce stalling due to L2 misses. In some designs, this is the major, if not the only, mechanism to propel the look-ahead thread to run ahead of the main thread [6, 42]. We use the same simple heuristics from [3]: if the look-ahead does not have a sufficient lead over the main thread, we feed 0 to the load that misses in the L2 cache. This is easier to implement than to tag and propagate poison. Finally, prefetching all the way to the L1 is understandably better than prefetching only to L2. We find that a FIFO queue to delay-release the final leg of prefetch (from L2 to L1) is a worthwhile investment as it can significantly reduce pollution.

\(^1\)One of the major sources of remaining mispredictions in modern branch predictors are data-dependent branches [68] which are hard to predict based on branch history. Because the branch outcome heavily depends upon the value of short distance store-load pairs, decoupled look-ahead is able to eliminate almost all of them, as we will see in the evaluation section later.
2.4.2 Software Support for Look-ahead

For the look-ahead thread generation from the original program binary, we use an approach similar to the one proposed by Garg et al. in [3]. A binary analyzer, based on alto [69], creates a look-ahead binary (skeleton) from the original program binary using train input based profiling information. Due to lack of correctness constraints in skeleton, we focus on common cases and ignore many corner cases which results in an optimized look-ahead skeleton. Specifically, biased branches (with over 99.9% bias towards one direction) are converted into unconditional branches as illustrated in the Figure 2.4-(A). Targets of this particular optimization are most of the loops in general. All other branches and their backward slices are included in the skeleton.

Memory dependences are profiled to exclude long-distance dependences: if a load depends on a store that consistently (more than 99.9%) has long def-to-use distance (> 5000 instructions) in the profiling run, backward slicing will terminate at the load (as shown in the Figure 2.4-(B)). These stores are too far away from the consumer load that even if they are included in the look-ahead thread, the result would likely be evicted from the L0 cache before the load executes. We also remove the dynamically dead and rare path instructions and their exclusive backward dependence chain.

Finally, memory instructions that often miss the last-level cache that are not already included in the skeleton are converted to prefetch instructions and added to the skeleton. All other
instructions on the original program binary become NOPs. Complete skeleton building is done using backward data dependence based slicing. Control dependences have already been taken care of because all the branch instructions are included in the skeleton. For DIY branch explorations, we also implement a lightweight version of control dependence slicing where not all branches are included in the skeleton.

2.4.3 Code Address Space and Runtime Recoveries

The resulting skeleton code body has the same basic block structure and branch offsets. Therefore, the two code segments (original program and the skeleton) can be laid out in two separate code address spaces or in the same code address space (which will leave a fixed offset between any instruction in the skeleton and its original copy in the original code). In either case, a simple support in the instruction TLB can allow two threads to have the exact same virtual addresses and yet fetch instructions from different places in the main memory. As such, when the main thread is being context switched-in, the same initialization can be applied to its look-ahead thread. Similarly, when the branch outcomes from the look-ahead threads deviate from that of the main thread, it can be “rebooted” with a checkpoint register state from the main thread. We call such an event a recovery which is a relatively rare operation.

2.4.4 Summary of Performance Benefits

Speedup and Energy Overhead:

In this section, we summarize the performance benefits of a decoupled look-ahead system over the single-thread baseline (Table 2.2). The decoupled look-ahead achieves 1.29x speedup over single thread for INT applications and 1.34x speedup for FP. It may appear that running two cores may roughly double the energy consumption compared to the baseline. This two-core multiprocessor view of the decoupled look-ahead is grossly misleading, and in reality the energy overhead can be far less due to a number of facts. First of all, the look-ahead thread only executes a subset of instructions compared to the main thread. Second, many energy components do not double. For example, the baseline single-thread wastes most of the energy in waiting for caches and in wrong path execution when a speculation fails. The look-ahead thread
Table 2.2: Summary of performance benefits of the baseline decoupled look-ahead system.

<table>
<thead>
<tr>
<th>Performance Stats</th>
<th>INT (11 apps)</th>
<th>FP (14 apps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speedup over single-thread baseline</td>
<td>1.95</td>
<td>2.12</td>
</tr>
<tr>
<td></td>
<td>1.29</td>
<td>1.34</td>
</tr>
<tr>
<td></td>
<td>1.07</td>
<td>1.05</td>
</tr>
<tr>
<td>Energy consumption w.r.t. baseline (%)</td>
<td>150</td>
<td>155</td>
</tr>
<tr>
<td></td>
<td>110</td>
<td>107</td>
</tr>
<tr>
<td></td>
<td>55</td>
<td>49</td>
</tr>
<tr>
<td>Look-ahead dynamic skeleton size (%)</td>
<td>92.2</td>
<td>83.6</td>
</tr>
<tr>
<td></td>
<td>75.2</td>
<td>42.1</td>
</tr>
<tr>
<td></td>
<td>17.1</td>
<td>10.3</td>
</tr>
<tr>
<td>Recoveries per 10K instructions</td>
<td>10.4</td>
<td>4.22</td>
</tr>
<tr>
<td></td>
<td>3.27</td>
<td>1.03</td>
</tr>
<tr>
<td></td>
<td>0.06</td>
<td>0.01</td>
</tr>
<tr>
<td>L2 miss reductions from baseline (%)</td>
<td>99.5</td>
<td>99.7</td>
</tr>
<tr>
<td></td>
<td>91.9</td>
<td>95.3</td>
</tr>
<tr>
<td></td>
<td>75.2</td>
<td>80.5</td>
</tr>
</tbody>
</table>

helps the main thread to avoid both of these events to good extent, and thus helps in saving a significant fraction of these energy wastes. With our detailed experiments we found out that for INT applications the energy overhead is only about 10% compared to the baseline system.

Additional Details:

After applying various optimizations to the skeleton, discussed in the Section 2.4.2, the average dynamic skeleton size is 75.2% for INT and 42.1% for FP applications compared to original program size. If look-ahead thread veers off from correct control path, the main thread – upon committing that branch – corrects the look-ahead thread by re-initializing the architectural states. This operation is known as recovery which is an expensive operation and typically costs about 200 execution cycles. Recoveries in the decoupled look-ahead system are rare: on an average about 3.27 out of 10K insts with a maximum of 10.4. Additionally, due to good prefetching effect, the reduction in L2 misses is 92% (INT) and 95% (FP) compared to the baseline system.

Correlation with an RTL/FPGA Accurate Simulator:

We compare various performance potentials with an RTL/FPGA accurate performance simulator. In this case RTL/FPGA accurate performance simulator is from Imagination Technology that is being used for modeling next generation MIPS P-series Warrior cores. As you can see from the Figure 2.5, performance numbers from our current simulation framework (SimpleScalar based) are quite a bit “pessimistic”. If we assume that baseline decoupled look-ahead will reduce the same fraction of branch mispredictions and L2 cache misses when implemented
Figure 2.5: Comparison of various performance potentials and projected decoupled look-ahead speedup with an RTL/FPGA accurate performance simulation (IMG-psim) framework.

in a more realistic framework or in real hardware we will gain 1.56x performance speedup compared to single thread baseline. Because per branch misprediction and cache misses penalty is higher in RTL/FPGA accurate simulator, we expect to achieve more performance gain compared to what we report in the subsequent chapters of this thesis. These performance gains are the lower bound of what our current infrastructure allows us and not what can be achieved when the decoupled look-ahead is implemented in a real hardware.

2.5 Bottlenecks in Baseline Decoupled Look-ahead

While decoupled look-ahead system achieves good performance over the single-thread baseline there is still a plenty of room for further improvements. Intuitively, there is a fundamental tradeoff between the speed and helpfulness of the look-ahead thread. In this section, we analyze the bottlenecks present in the current version of decoupled look-ahead system. With simple transformations, a look-ahead thread can be constructed from the original program thread to successfully reduce misprediction and cache misses for the main thread. With two threads running together (as shown in Figure 2.3), the performance is dictated by whichever runs slower. If the look-ahead thread is doing a good enough job, the speed of the duo will be determined by

---

2This performance was computed by assuming the reductions in L2 misses and branch mispredictions are similar to Table 2.2. This is a rough estimate and eventual performance gain maybe slightly lower due to implementation complexities which might not have been taken into account in the performance simulator.
Figure 2.6: Performance comparison of 4 configurations. Shown in the bars are baseline single core (left) and a decoupled look-ahead system (right). Two upper-bounds are shown: the performance of a single core with idealized branch predictions and perfect cache accesses (curve with circles), and the the approximate speed limit of the look-ahead thread (gray wide curve indicating approximation). The applications are sorted with increasing performance gap between the decoupled look-ahead system and the prediction- and accesses-idealized single-core system.

how much execution parallelism can be extracted by the main thread. The performance potential can be approximated by idealizing cache hierarchy and branch predictions. On the other hand, when the look-ahead thread is the bottleneck, the main thread can only be accelerated up to the speed of the look-ahead thread running alone, which is another performance upper bound.

To understand the performance impact of this decoupled look-ahead system, in Figure 2.6, we show the speed of benchmarks in four configurations (the details of the experimental setup are presented in Section 3.6). The first configuration is the baseline where the benchmarks run on a single core. The second configuration is the decoupled look-ahead system. One performance upperbound of this second configuration is the speed of an ideal system without any mispredictions or cache misses (shown as curves with dots). The applications are sorted from left to right with increasing gap from this upper-bound. As we can see, for 9 benchmarks (on the left of the dashed line), the potential is small (<5%), showing that the decoupled look-ahead architecture has successfully achieved the goal. We can also see that, after using decoupled look-ahead, these benchmarks are reaching a high sustained IPC (around 3 or more). Further increasing the speed probably requires addressing the throughput bottleneck of the pipeline.

For the remaining benchmarks, the potential can be quite large. To understand the reason why the decoupled look-ahead system falls short of the potential, we show the result of a fourth

\[ \text{Note that idealization tends to produce a loose upper-bound.} \]
configuration, where we measure the (approximate) speed of the look-ahead thread on its own. Because of the approximation involved\(^4\), this upperbound is not exact. Faithful modeling of decoupled look-ahead execution requires true execution-driven simulation with values and timing faithfully modeled. Idealization can make an unreliable approximation in these cases. We indicate this inexactness with shaded sidebands around the simulated results.

Figure 2.6 clearly suggests that for the vast majority of the remaining cases, the look-ahead thread is slowing the overall system. Indeed, among the 16 benchmarks, the decoupled look-ahead architecture can run 14 of them (with the exception of *apsi* and *bzip2*) almost exactly at the speed of running look-ahead thread alone. This clearly suggests that the speed of the look-ahead thread is often the new bottleneck. Also the difference between the decoupled look-ahead (darker bars) and ideal (cache, branch) curve suggests that if the look-ahead thread is accelerated there is significant performance potential. We will primarily focus on right hand side applications when we apply techniques to improve the overall speed by accelerating the look-ahead agent. We revisit these potential study and bottleneck experiments after applying our techniques and present the findings in the tail end of this thesis.

### 2.6 Summary

There has been plethora of work targeting problematic instructions through helper threading. Two flavors of helper threading are lightweight microthreads and monolithic continuous decoupled look-ahead threads. While decoupled look-ahead thread provides enough performance potential, not all the potential is converted into real performance. One of the major bottlenecks is that the look-ahead thread itself is not fast enough to provide deep look-ahead. In other words, a slower look-ahead thread slows the whole system down.

In subsequent chapters, we present a range of techniques and hardware/software mechanisms to accelerate the look-ahead thread in order to speed up the overall system. We envision the look-ahead thread as a self-tuning process which makes use of offline profiling and makes itself more efficient throughout the execution.

\(^4\)In this configuration, the trailing main thread is idealized such that it can saturate the pipeline (limited by a decode and rename bandwidth of 4 insts/cycle).
In this chapter, we explore speculative parallelization in a decoupled look-ahead system in order to accelerate the look-ahead agent. Intuitively, speculative parallelization is aptly suited for the task of boosting speed of the decoupled look-ahead agent for two reasons. First, the look-ahead binary does not contain all the data dependences embedded in the original program, providing more opportunities for speculative parallelization. Second, the execution of the look-ahead slice is only for look-ahead purposes, thus the environment is inherently more tolerant to dependence violations and occasional errors which simplifies the hardware support needed for the speculative parallelization in the look-ahead thread. We find these intuitions to be largely borne out by experiments, and speculative parallelization can achieve significant performance benefits at a much lower cost than needed in a general purpose environment.

The rest of this chapter is organized as follows: First, we discuss the background and related work in Section 3.1, and the motivation for using speculative parallelization in Section 3.2. Sections 3.3 and 3.4 provide details of the software and hardware support needed to enable speculative parallelization in the decoupled look-ahead. Next, we present the runtime management and support for spawning a thread in Section 3.5. Sections 3.6 and 3.7 then show the experimental setup and discuss experimental analysis of the design. Finally, Section 3.9 summarizes the key findings and insights.
3.1 Background: Speculative Parallelization

To exploit implicit parallelism, most of the past proposals speculatively parallelize the main thread. In recent years, many schemes with hardware support for speculative parallelization have been proposed [28, 30, 31, 33, 34, 70–72]. Due unyielding correctness constraint, these proposals also have to employ a full-fledged mechanism to correct the execution and recover the correct architectural states when then speculation fails. In general, there are three things these proposals need to support: support for spawning a thread, detecting a dependence violation, and finally, if dependence violation is detected then to rollback to the point where the execution is non-speculative. Detecting a dependence violation and a rollback support mechanism require complex hardware that must catch every single violation. In most cases, the rollback requirement is challenging and costly as well.

Success of speculative parallelization depends on capability of software or hardware mechanisms to find out tasks with little inter-task dependencies. Recent proposals mainly differ in how the violations are detected and updates from the parallel tasks are merged [28, 30, 31, 33, 34, 70, 72]. Instead of applying thread level speculation on main thread, we propose to apply it to the look-ahead thread [9]. Speculative parallelization in look-ahead thread has a few distinct advantages, such as no hard requirement of rollback and the increased parallelism due to relatively fewer dependencies in the skeleton. By speculatively parallelizing the look-ahead thread, some dependences will also be violated, but the consequence is (much) less severe: (e.g., no impact on correctness). This makes tracking of dependence violations less critical.

Xekalakis et al. proposed to combine thread level speculation (TLS), helper threading (HT) and runahead execution (RA) in a single framework [73]. Similar to previous proposals, they also apply the TLS to the main thread. In their design, thread level parallelism (through TLS) and instruction level parallelism (through HT and RA) are exploited at the same time, depending upon the applications and availability of TLP and ILP in a given phase. They mainly focus on existing TLS systems and implement microthread kind of helper threads on top of it. Follow up work addresses the challenges in handling branches in TLS systems with multi-path execution [74]. They claim that branch prediction in TLS systems are harder and more important compared to sequential execution.
3.2 Motivation for Speculative Parallelization

From Section 2.5 and Figure 2.6, it is clear that for the right hand side applications, the overall speedup is mainly limited by the speed of the slow look-ahead thread, indicating potential performance gain when the look-ahead thread is accelerated. There are two unique opportunities in the look-ahead environment for applying speculative parallelization to accelerate the look-ahead thread. First, the construction of the skeleton removes some instructions from the original binary, as shown in Figure 2.4, and thereby removes some dependences too. Our manual inspection of a few benchmarks finds a repeating pattern of complex, loop-carried dependences that are naturally removed as the skeleton is being constructed, resulting into more loop-level parallelism. Another pattern is the increase in dependence distance as a result of removing instructions in short-distance dependence chains. As shown in the example in Figure 3.1, in the original code, almost every basic block depends on its immediate predecessor basic block. When constructing the skeleton, the removal of some instructions from the basic blocks 5 and 6 breaks the chain of dependences, leaving only a long-distance dependence which provides exploitable parallelism that wasn’t present otherwise.

A second opportunity is the lowered requirement for speculative parallelization. When two sequential code sections $A$ and $B$ are executed in parallel speculatively, register and memory dependences that flow from $A$ to $B$ are preserved by a combination of explicit synchronization and squashing of instructions detected to have violated the dependence. Because register dependences are explicit in the instruction stream, explicit synchronization can be used to enforce them. Nevertheless, the extra synchronization adds to the overhead of execution and demands special hardware support. For memory dependences, without complete static knowledge of the exact addresses, compiler cannot identify all possible dependences. As a result, the runtime tracking becomes necessary.

In contrast, when we try to exploit speculative parallelization in the inherently non-critical look-ahead thread, correctness is no longer a must, but rather a quality of service issue: unenforced dependences only presumably reduce the effectiveness of look-ahead and affect performance slightly, and may not be worth fixing. Indeed, we will show later in Section 3.7, fixing a dependence violation using rollback slows down the look-ahead and is more detrimental than
Figure 3.1: Dependence characteristic changes due to skeleton construction. In this example from *equake*, each box represents a basic block. For clarity, only a subset of instructions are shown. Instructions that are not on the skeleton are shown in gray color. The arcs show the dependences and their distances measured in numbers of basic blocks.

the dependence violation itself. Thus, it becomes a matter of choice how much hardware and runtime support we need to detect violations and repair them. In the extreme case, we can forgo all conventional hardware support for speculative parallelization and only rely on probabilistic analysis of the dependence to minimize violations.

Finally, in hindsight, we realized that speculative parallelization of the look-ahead thread also has a desired side effect of reducing the cost of recoveries. A recovery happens when the branch outcome from the look-ahead thread deviates from that of the main thread. For simplicity, we reboot the look-ahead thread rather than trying to repair the state. Because the look-ahead threads can be running far ahead of the main thread, such a recovery can wipe out the lead look-ahead thread accumulated over a period of time. Spawning a secondary thread
provides a natural mechanism to preserve part of the look-ahead that has already been done. Thus, we can reboot one thread while keep the other thread running as shown in the Figure 3.2.

We primarily use two mechanisms to expose parallelization in skeleton. First, we identify sections of the code that are either independent or connected by long dependencies. These long dependencies allow us to execute a part of the code section in parallel until the dependence needs to be resolved for correct execution and are referred to as synchronization points for discussion purposes. A spawned thread might have many synchronization points. Note that if we choose to use this mechanism to exploit parallelism in the baseline processor, to ensure correctness, these synchronization points have to be satisfied. However, for the look-ahead thread, we can afford to be relaxed and hope that these dependencies would not be violated or rather any violation would rarely affect the execution of skeleton (and the quality of look-ahead).
3.3 Software Support

3.3.1 Dependence Analysis

To detect coarse-grain parallelism suitable for thread-level exploitation, we use a profile guided analysis tool. The look-ahead thread binary is first profiled to identify dependences and their distances. To simplify the subsequent analysis, we collapse the basic block into a single node, and represent the entire execution trace as a interconnected graph of these nodes with dependences as edges. Dependences are therefore between basic blocks, and the distance can be measured by the distance of nodes in graph as shown in Figure 3.3-(a).

Given these nodes and arcs representing dependences between them, we can make a cut before each node and find the minimum dependence distance among all the arcs that pass through the cut. This minimum dependence distance, or $D_{\text{min}}$, represents an approximation of parallelization opportunity as can be explained by the simple example in Figure 3.3. Suppose, for the time being, the execution of a basic block takes one unit of time, and there is no overlapping of basic block execution. Node $d$ in Figure 3.3, which has a $D_{\text{min}}$ of 3, can therefore be scheduled to execute 2 units of time ($D_{\text{min}} - 1$) earlier than its current position in the trace – in parallel with node $b$. All subsequent nodes can be scheduled 2 units of time earlier as well, without reverting the direction of any arc as shown in Figure 3.3-(b).

Of course, the distance in basic blocks is only a very crude estimate of the actual time lapse between the execution of the producer and consumer instructions. In reality, the size and execution speed of different basic blocks are different, and their executions overlap. Furthermore, depending on the architectural detail, executing the producer instructions earlier than the consumer does not necessarily guarantee the result will be forwarded properly. Therefore, for nodes with small $D_{\text{min}}$ there is little chance to exploit parallelism. We set a threshold on $D_{\text{min}}$ (in Figure 3.3) to find all candidate locations for a spawned thread to start its execution. $D_{\text{min}}$ threshold in our study is 15 basic blocks which is approx. 120 instructions – same as the maximum possible in-flight instructions in the pipeline at any given moment.

1 A somewhat more appropriate notion of dependence distance that we use is the actual number of instructions in between the source and destination basic blocks.
3.3.2 Selecting Spawn and Target Points

With the candidates selected from profile-based analysis, we need to find static code locations to spawn off parallel threads (e.g., node a in Figure 3.3-b), and locations where the new threads can start their execution (e.g., node d in Figure 3.3-b). We call the former spawn points and the latter target points. We first select only those target points whose dynamic instances consistently show a $D_{\text{min}}$ larger than the threshold value. Next, we search for the corresponding spawn points. The choices are numerous. In the example shown in Figure 3.3, if we ignore the threshold of $D_{\text{min}}$, the spawn point of node d can be either a or b. Given the collection of many different instances of a static target point, the possibilities are even more numerous. The selection needs to balance cost and benefit. In general, as a spawn becomes more successful and as the distance between the spawn and target points increases the potential benefit increase as well. On the other hand, every spawn point will have some unsuccessful spawns, incurring costs. We use a cost benefit ratio ($\Sigma \text{distances}/\# \text{false spawns}$) to sort and select the spawn points. Note that a target point can have more than one spawn points.
3.3.3 Loop-level Parallelism

In general, frequent memory accessing loops are responsible for slowdown in loop dominated applications. These loops consist of complex cross-iteration dependencies that are usually hard to exploit for loop-level parallelization by the state-of-art compiler. The skeleton version of these loops are still slow due to cache misses; however, they are easy to parallelize because most of the computation resulting in complex cross-iteration dependencies is removed.

Without special processing, a typical loop using index variable can project a false loop-carried dependence on the index variable and can mask the potential parallelism from our profiling mechanism. After proper adjustments, parallel loops will present a special case for our selection mechanism. The appropriate spawn and target points will be the same static node. The number of iterations to jump ahead is selected to make the number of instructions approach a target number (1000 instructions in this study).

3.3.4 Available Parallelism

A successful speculative parallelization system maximizes the opportunities to execute code in parallel (even when there are apparent dependences) and yet does not create too many squashes at runtime. In that regard, our methodology has a long way to go. Our goal in this study is to show that there are significant opportunities for speculative parallelization in the special environment of look-ahead, even without a sophisticated analyzer. Figure 3.4 shows an approximate measure of available parallelism recognized by our analysis mechanism. The measure is simply that of the “height” of a basic block schedule as shown in Figure 3.3. For instance, the schedule in Figure 3.3-a has a height of 6 and the schedule in Figure 3.3-b has a height of 4, resulting in a parallelism of 1.5 (6/4). Note that in reality, node d’s $D_{min}$ is too small to be hoisted up for parallel execution. For simplicity, at most two nodes can be in the same time slot, resulting in a maximum parallelism of 2. For comparison, we also show the result of using the same mechanism to analyze the full program trace.

The result shows that there is a significant amount of parallelism, even in integer code. Also, in general, there is more parallelism in the look-ahead thread than in the main program thread. Figure 3.4-(b) shows the amount of parallelism when we impose further constraints,
Figure 3.4: An approximate measure of available parallelism in the trace of the look-ahead thread and the main program thread (top figure). A more constrained measure of parallelism that is likely to be exploited (bottom figure).

namely finding stable spawn-target point pairs to minimize the spurious spawns. The result suggests that some parallelism opportunities are harder to extract than others because there are not always spawn points that consistently lead to the execution of the target point. Future work will explore cost-effective solutions to this problem.

3.4 Hardware Support

Typical speculative parallelization requires a whole host of architectural support such as data versioning and cross-task dependence violation detection [75]. Since look-ahead does not require correctness guarantee, we are interested in exploring a design that minimizes intrusion.

To enable the speculative parallelization discussed so far, there are three essential elements: we need to (1) spawn a thread, (2) communicate values to the new thread, and (3) properly merge the threads.
3.4.1 Spawning a New Look-ahead Thread

The support we need is not very different from existing implementation to spawn a new thread in a multi-threaded processor. For example, the main thread will set up the context of the newly spawned thread (Figure 3.5). A key difference is that the spawned thread will execute a future code segment in the same logical thread. If speculation is successful, the primary look-ahead thread is expected to reach where the spawned thread has already started and “merge” with the spawned thread (Figure 3.5). Therefore, the starting PC of the newly spawned thread needs to be recorded. When the primary look-ahead thread reaches that same PC – more specifically, when the instruction under that PC is about to retire – the primary look-ahead thread simply terminates, without retiring that instruction, since it has been already executed and retired by the spawned look-ahead thread.

3.4.2 Support for Register Access

When a new look-ahead thread is spawned, it inherits the architectural state including memory content and register state. While this can be implemented in a variety of environments, it is most
straightforward to support in a multithreaded processor. Thus we focus on this environment.

When the spawning instruction is dispatched, the register renaming map is duplicated for the spawned thread. With this action, the spawned thread is able to access register results defined by instructions in the primary thread prior to the spawn point. Note that if an instruction (with a destination register of say, \( R_2 \)) prior to the spawn point has yet to execute, the issue logic naturally guarantees that any subsequent instruction depending on \( R_2 \) will receive the proper value regardless of which thread the instruction belongs to (e.g., register \( r_6 \) in Figure 3.6).

When a rename table entry is duplicated, a physical register is mapped in two entries and both entries can result in the release of the register in the future. A single bit per rename table entry is therefore added to track “ownership” (‘O’ bit in Figure 3.6). When the spawned thread copies the rename table, the ownership bits are set to 0. A subsequent instruction overwriting the entry will not release the old register, but will set the ownership bit to 1 (e.g., in Figure 3.6, ‘O’ bit of register \( r_6 \) in spawned thread’s map table is set to 1 after renaming).

Since the bit indicates whether the spawned thread has defined its own version of a particular
architectural register, a 0 means the corresponding register should carry the last update to that architectural register made by the primary thread. This is approximated by updating the rename mapping of the architectural register as the primary thread updates its corresponding entry. This is illustrated in Figure 3.6, where update of \( r7 \) in the primary thread changes both threads’ \( r7 \) mapping to \( p9 \). In other words, after the thread is spawned, it may still receive register inputs from the primary thread. A slightly simpler alternative is not to support such inputs. We find that to be also working fine in the vast majority of cases with negligible performance impact, but it does result in a significant (16%) performance degradation in one application.

Finally, when the primary thread terminates (at merge point), any physical register that is not mapped in the secondary thread’s rename table can be recycled. This can be easily found out from the ownership bit vector: any bit of 1 indicates the threads both have their thread-local mapping and thus the register is private to the primary thread and can be recycled.

### 3.4.3 Support for Memory Access

Getting memory content from the primary thread is also simplified in the multithreaded processor environment since the threads share the same L1 cache. An extreme option is to not differentiate between the primary look-ahead thread and the spawned thread in cache accesses. This option incurs the danger that write operations to the same memory location from different threads will not be differentiated and subsequent reads will get wrong values. However, even this most basic support is a possible option, though the performance benefit of parallel look-ahead is diminished as we will show later.

A more complete versioning support involves tagging each cache line with thread ID and returning the data with the most recent version for any request. For conventional speculative parallelization, this versioning support is usually done at a fine access granularity to reduce false dependence violation detections [71]. In our case, we use a simplified, coarse-grain versioning support without violation detection, which is a simple extension of the cache design in a basic multithreaded processor. For notational convenience we call this *partial* versioning.

The main difference from a full-blown versioning support is two-fold. First, the version is only attached to the cache line as in the baseline cache in a multithreaded processor. No per-
word tracking is done. Similar to versioning cache, a read from thread $i$ returns the most recent version no later than $i$. A write from thread $i$ creates a version $i$ from the most recent version if version $i$ does not already exist. The old version is tagged (by setting a bit) as being replaced by a new version. This bit is later used to gang-invalidate replaced lines. Second, no violation detection is done. When a write happens, it does not search for premature reads from a future thread. The cache therefore does not track whether any words in a line have been read.

3.5 Runtime Management and Control

In this section, we present the runtime support and management needed for the speculative parallelization in the decoupled look-ahead system. These set of policies are easy to implement in real hardware, and later we discuss refinements and optimizations on top of these policies.

3.5.1 Spawning Thread and Management

Based on the result of our analysis and to minimize unnecessary complexities, we opt to limit the number of threads spawned at any time to only one. This simplifies hardware control such as when to terminate a running thread and partial versioning support. It also simplifies the requirement on dependence analysis.

At runtime, two thread contexts are reserved (in a multithreaded core) for look-ahead. There is always a primary look-ahead thread. A spawn instruction is handled at dispatch time and will freeze the pipeline front end until the rename table is duplicated and a new context for the spawned look-ahead thread is set up. If another thread is already occupying the context, the spawn instruction is discarded. Since the spawn happens at dispatch, it is a speculative action and is subject to a branch misprediction squash. Therefore, we do not start the execution immediately and wait for a short period of time. This waiting also makes it less likely that a cross-thread read-after-write dependence is violated. When the spawn instruction is indeed squashed due to branch misprediction, we terminate the spawned thread.

When the primary thread reaches the point where the spawned thread started the execution, the two successfully merge. The primary thread is terminated and the context is available for
Figure 3.7: Example of a banked branch queue. Bank 0 and bank 1 are written by primary (older) look-ahead and spawned (younger) look-ahead threads respectively. Primary thread uses global head pointer, currently pointing to an entry in bank 0, to read branch predictions.

another spawn. The spawned thread essentially carries on as the primary look-ahead thread. At this point, we gang invalidate replaced cache lines from the old primary thread and consolidate the remaining lines into the new thread ID. When the primary thread and the spawned thread deviate from each other, they may not merge for a long time. If this happens, keeping the spawned thread will prevent new threads from being spawned and limit performance. Therefore, run-away spawns are terminated after a fixed number of instructions suggested by the software.

3.5.2 Communicating Branch Predictions to Primary Thread

Branch predictions produced by look-ahead thread(s) are deposited in an ordered queue called branch queue. There are many options to enforce semantic sequential ordering among branch predictions despite that they might be produced in different order. One of the simpler options we explored in this paper is to segment the branch queue in a few banks of equal size as shown in Figure 3.7.

The bank management is straightforward. When a thread is spawned a new bank is assigned in sequential order. A bank is also de-allocated sequentially as soon as the primary thread consumes all branch predictions form that bank. If a thread exhausts the entries in its current bank, the next sequential bank is used. It is possible, but very rare, that the next bank has already been allocated to a spawned thread. In such a case, to maintain the simplicity of sequential allocation, we kill the younger thread and reclaim bank for re-assignment.
3.6 Experimental Setup

We perform our experiments using an extensively modified version of SimpleScalar [76]. Support was added for Simultaneous Multi-Threading (SMT), decoupled look-ahead, and speculative parallelization. Because of the approximate nature of the architecture design for look-ahead, the result of the semantic execution is dependent on the microarchitectural state. For example, a load from the look-ahead thread does not always return the latest value stored by that thread because that cache line may have been evicted. Therefore, our modified simulator uses true execution-driven simulation where values in the caches and other structures are faithfully modeled. The values are carried along with instructions in the pipeline and their semantic execution are emulated on the fly to correctly model the real execution flow of the look-ahead thread.

3.6.1 Microarchitecture and Configuration

The simulator is also enhanced to faithfully model issue queues, register renaming, ROB, and LSQ. Features such as load-hit speculation (and scheduling replay), load-store replays, and keeping a store miss in the SQ while retiring it from ROB, are all faithfully modeled [77]. We also changed the handling of prefetch instructions (load to ZERO register – R31). By default, the original simulator not only unnecessarily allocates an entry in the LQ, but fails to retire the instruction immediately upon execution as indicated in the alpha processor manual [77]. In our simulator, a prefetch neither stalls nor takes resource in the LQ. Our baseline core is a generic out-of-order microarchitecture loosely modeled after POWER5 [78]. Details of the configurations are shown in Table 6.1.

An advanced hardware-based global stream prefetcher based on [79, 80] is also implemented between the L2 cache and the main memory. On an L2 miss, the stream prefetcher detects an arbitrarily sized stride by looking at the history of past 16 L2 misses. If the stride is detected twice in the history buffer, an entry is allocated on the stream table and prefetch is generated for the next 16 addresses. Stream table can simultaneously track 8 different streams. For a particular stream, it issues a next prefetch only when it detects the use of previously prefetched cache line by the processor.
### Baseline core

<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch/Decode/Commit</td>
<td>8 / 4 / 6</td>
</tr>
<tr>
<td>ROB</td>
<td>128</td>
</tr>
<tr>
<td>Functional units</td>
<td>INT 2+1 mul +1 div, FP 2+1 mul +1 div</td>
</tr>
<tr>
<td>Issue Q / Reg. (int,fp)</td>
<td>(32, 32) / (120, 120)</td>
</tr>
<tr>
<td>LSQ(LQ,SQ)</td>
<td>64 (32,32) 2 search ports</td>
</tr>
<tr>
<td>Branch predictor</td>
<td>Bimodal + Gshare</td>
</tr>
<tr>
<td>- Gshare</td>
<td>8K entries, 13 bit history</td>
</tr>
<tr>
<td>- Bimodal/Meta/BTB</td>
<td>4K/8K/4K (4-way) entries</td>
</tr>
<tr>
<td>Br. mispred. penalty</td>
<td>at least 7 cycles</td>
</tr>
<tr>
<td>L1 data cache</td>
<td>32KB, 4-way, 64B line, 2 cycles, 2 ports</td>
</tr>
<tr>
<td>L1 I cache (not shared)</td>
<td>64KB, 1-way, 128B, 2 cyc</td>
</tr>
<tr>
<td>L2 cache (uni. shared)</td>
<td>1MB, 8-way, 128B, 15 cyc</td>
</tr>
<tr>
<td>Memory access latency</td>
<td>400 cycles</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

Table 3.1: Core configuration.

#### 3.6.2 Applications and Inputs

We use SPEC CPU2000 benchmarks compiled for Alpha. We use the *train* input for profiling, and run the applications to completion. For evaluation, we use *ref* inputs. We simulate 100 million instructions after skipping over the initialization portion as indicated in [81].

#### 3.7 Experimental Analysis

We first look at the end result of using our speculative parallelization mechanism in the decoupled look-ahead. We also conduct some additional experiments that shed light on how the system works. Finally, we list the future explorations to improve the design’s efficacy.

#### 3.7.1 Performance Analysis

Recall that a baseline decoupled look-ahead system can already help many applications to execute at high throughput that is close to saturating the baseline out-of-order engine. For these applications, the bottleneck is not the look-ahead mechanism. Designing efficient wide-issue execution engine or look-ahead to assist speculative parallelization of the main thread are direc-
tions to further improve their performance. For the remaining applications, Figure 3.8 shows the relative performance of a baseline, single-threaded look-ahead system and our speculative, dual-threaded look-ahead system. All results are normalized to the single-core baseline system.

Our speculative parallelization strategy provides up to 1.39x speedup over baseline look-ahead. On average, measured against the baseline look-ahead system, the contribution of speculative parallelization is a speedup of 1.13x. As a result, the speedup of look-ahead over a single core improves from 1.61x for single look-ahead thread to 1.75x for two look-ahead threads (in the applications where look-ahead is the bottleneck). If we only look at the integer benchmarks in this set of applications, that are traditionally considered more difficult to parallelize, the speedup over the baseline look-ahead system is 1.11x.

It is worth noting that the quantitative results here represent what our current system allows us to achieve. It does not represent what could be achieved. With more refinement and trial-and-error, we believe more opportunities can be explored. Even with these current results, it is clear that speeding up sequential code sections via decoupled look-ahead is a viable approach for many applications.

Finally, for those applications where the main thread has (nearly) saturated the pipeline, this mechanism does not slow down the program execution. The detailed IPC results for all applications are shown in Table 3.2.
5.7.2 Microarchitectural Sensitivity

To measure the robustness of speculative parallelization, we conducted detailed experiments on two different kinds of microarchitecture. First microarchitecture is a simple 3-way out-of-order core that is modeled after ARM Cortex-A9 processor. The second microarchitecture is a more aggressive microarchitecture that is modeled after IBM POWER5, a 4-way out-of-order core. Microarchitectural parameters of these two architectures are summarized in the Table 3.3.

<table>
<thead>
<tr>
<th></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>

Table 3.3: Microarchitectural parameters of Cortex-A9 and POWER5 cores.

In Figure 3.9, we show the speedup of our speculative parallelization for two different microarchitecture over their respective single-thread baseline systems. For a simple Cortex-A9 kind of core, the speculative parallelization achieves 1.78x speedup over its single-thread baseline. For more aggressive POWER5 kind of core, the speedup of speculative parallelization
Figure 3.9: Speculative parallelization performance improvement for a simple (Cortex-A9) and a more aggressive (POWER5) out-of-order microarchitecture.

is 1.81x over baseline single-thread. These results suggest that our speculative parallelization technique is robust and not sensitive to any specific microarchitectural configurations. It reaps good benefits in a simple OoO engine and state-of-art aggressive OoO core as well.

### 3.7.3 Comparison with Conventional Speculative Parallelization

As discussed earlier, the look-ahead environment offers a unique opportunity to apply speculative parallelization technique partly because the code to drive look-ahead activities removes certain instructions and therefore, provides more potential for parallelism. However, an improvement in the speed of the look-ahead only indirectly translates into the overall performance. Here, we perform few experiments to inspect the impact.

We use the same methodology on the original program binary and support the speculative threads to execute on a multi-core system. Again, our methodology needs further improvement to fully exploit the available parallelism. Thus the absolute results are almost certainly underestimating the real potential. However, the relative comparison can still serve to contrast the difference in the two setups. Figure 3.10 shows the results. For a more relevant comparison, conventional speculative parallelization is executed on two cores to prevent execution resource from becoming the bottleneck. It is worth noting, the base of normalization is not the same. For the conventional system, the speedup is over a single-thread baseline. For our system, the speedup is over a baseline look-ahead system, which has much higher performance.
As we showed earlier using a simple model of potential parallelism (Figure 3.4), there is more parallelism in the look-ahead binary (the skeleton) than in the full program binary. In Figure 3.10, we see that in many cases, speculative parallelization translates into more performance gain in the end for the look-ahead system. However, there are phases of execution where the look-ahead speed is not the bottleneck. A faster look-ahead thread only leads to filling the queues faster. Once the queues that pass information to the main thread fill up, look-ahead stalls. Therefore, in some cases, the advantage in more potential parallelism does not translate into more performance gain.

### 3.7.4 Recoveries

When the look-ahead thread’s control flow deviates from that of the main thread, the difference in branch outcome eventually causes a recovery. The ability of the look-ahead thread to run far ahead of the main thread is crucial for performing useful help. This ability is a direct result of ignoring uncommon cases and other approximations that comes at the price of recoveries. Speculative parallelization in the look-ahead environment also introduces its own sets of approximations. Otherwise, the opportunities will be insufficient, and the implementation barrier will be too high. However, too much corner-cutting can be counter-productive. Table 4.4 summarizes the maximum, average, and minimum recovery rate for integer (INT) and floating-point (FP) applications.
Recoveries per 10K insts

<table>
<thead>
<tr>
<th></th>
<th>INT</th>
<th>FP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Max</td>
<td>Avg</td>
</tr>
<tr>
<td>Baseline decoupled look-ahead</td>
<td>3.21</td>
<td>1.24</td>
</tr>
<tr>
<td>Speculative parallel look-ahead</td>
<td>6.55</td>
<td>1.21</td>
</tr>
</tbody>
</table>

Table 3.4: Recovery rate for the baseline and speculatively parallel look-ahead systems. Rates are measured by the number of recoveries per 10,000 committed instructions in the main thread.

For most applications, the recovery rate stays essentially the same. For some applications, the rate actually reduces (e.g., perlbench, from 3.21 to 0.43). Recall that skipping an L2 miss (by returning a 0 to the load instruction) is an effective approach to help the look-ahead thread stay ahead of the main thread. Frequent applications of this technique inevitably increase recoveries. In our system, this technique is only applied when the trailing main thread gets too close. With speculative parallelization, the need to use this technique decreases, as well as the number of resulting recoveries.

### 3.7.5 Partial Recoveries

As discussed earlier, when a recovery happens, we reboot the primary look-ahead thread. However, if there is another look-ahead thread spawned, we do not terminate the spawned thread, even though the recovery indicates that some state in the look-ahead threads is corrupted. We have this option because the look-ahead activities are not correctness critical. This essentially allows a partial recovery (without any extra hardware support) and maintains the lead of look-ahead. Nevertheless, it is possible that the spawned thread is corrupt and this policy only delays the inevitable.

Table 3.5 shows that this is not the case. The first row of numbers indicate how often a spawned thread successfully merged with the rebooted main look-ahead thread, indicating the spawned thread is still on the right path. The next two rows show in how many of these cases the spawned thread is still alive (has not encountered its recovery) after 200 and 1000 instructions. In many applications, almost all instances of the spawned thread are alive and well pass 1000 instructions, indicating that indeed they deserve to be kept alive at the recovery point. Also, the difference between the number of instances alive at 200 and 1000 instructions point is very
48

Table 3.5: Partial recoveries in the speculative look-ahead. Recv-Merge are the instances when a recovery happens in the main look-ahead thread and spawned thread merges later. Live 200 and Live 1000 are the instances when a Recv-Merge occurred and merged thread didn’t experience a new recovery for at least 200 and 1000 instructions respectively.

small, indicating that those that do not survive long actually terminate rather early. All in all, it is clear that keeping these spawned threads live has low risks and can achieve up to 3% performance improvement.

3.7.6 Quality of Spawns

Table 3.6 shows statistics about the number of spawns in different categories. The top half shows the cases where a spawn happens on the right path. They are predominantly successful. Only a handful of spawns had to be killed because the main thread has not merged with the spawned thread after a long time. The bottom half shows spawns on the wrong path due to branch mispredictions or because the primary look-ahead thread veers off the right control flow.

We can see that most of these spurious spawns happen because of branch misprediction that would be subsequently corrected. Recall that the spawned thread does not execute immediately after the spawn, but wait for a small period of time (to minimize unnecessary execution due to branch misprediction-triggered spawns and also to reduce violation of dependence). As a result of this small delay, many spawned threads have not dispatched any instruction before the branch is resolved and the spawn squashed. Almost all of these spurious spawns are short lived, even for those cases where some instructions on the spawned thread have been dispatched. In summary, speculative parallelization does not significantly increase the energy cost because the waste is small. Our speculatively parallel look-ahead executes on average 1.5% more instructions than sequential look-ahead due to few failed spawns.
### Spawns invoked under correct path

<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>
<tbody>
<tr>
<td>Successful</td>
<td>2297</td>
<td>26873</td>
<td>21067</td>
<td>1273</td>
<td>42082</td>
<td>6328</td>
<td>29598</td>
<td>16676</td>
<td>9687</td>
<td>20997</td>
<td>24022</td>
</tr>
<tr>
<td>Runaway</td>
<td>257</td>
<td>245</td>
<td>1738</td>
<td>37</td>
<td>409</td>
<td>3542</td>
<td>363</td>
<td>0</td>
<td>3965</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

### Spawns invoked under incorrect path

<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>
<tbody>
<tr>
<td>No dispatch</td>
<td>11</td>
<td>707</td>
<td>2837</td>
<td>96</td>
<td>1633</td>
<td>26</td>
<td>29</td>
<td>245</td>
<td>363</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Few dispatch</td>
<td>28</td>
<td>69</td>
<td>1803</td>
<td>6</td>
<td>273</td>
<td>45</td>
<td>116</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Wrong Path</td>
<td>11</td>
<td>184</td>
<td>2997</td>
<td>152</td>
<td>111</td>
<td>339</td>
<td>6</td>
<td>62</td>
<td>4</td>
<td>17</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 3.6: Breakdown of all the spawns. Wrong Path refers to the case the spawn happens when the primary look-ahead thread has veered off the right control flow and will eventually encounter a recovery. For crafty, eon, and gzip we do not see any spawns in our experiments.

### 3.7.7 Effect of Speculation Support

Because look-ahead thread is not critical for correctness, supporting speculative parallelization can be a lot less demanding than otherwise – in theory. In practice, there is no appeal for complexity reduction if it brings disproportionate performance loss. Section 3.4 described a design that does not require full-blown versioning and has no dependence violation tracking. The cache support required is a much more modest extension of cache for multithreaded core. In Figure 3.11, we compare this design to one that is even more relaxed: the data cache has no versioning support and, in fact, is completely unaware of the distinction between the primary look-ahead thread and the spawned one. As the figure shows, two applications (vpr and equake) suffer significant performance losses. However, for all other applications the degradation is insignificant. Since L1 caches are critical for performance and is often the subject of intense circuit timing optimization, any significant complication can create practical issues in a high-end product. Speculative parallelization in look-ahead, however, gives the designers the capability to choose incremental complexity with different performance benefits, rather than an all-or-nothing option as in conventional speculative parallelization.

Another example of the design flexibility is about whether to detect dependence violations. Dependence violation detection also requires intrusive modifications. Thus the flexibility of not having to support it is valuable. Figure 3.11 also compares our design to another one where accesses are carefully tracked and, when a dependence violation happens, the spawned thread
is squashed. This policy provides no benefit in any application window we simulated, and degrades performance significantly in one case (**vortex**). Intuitively, the look-ahead process is somewhat error tolerant. Being optimistic and ignore the occasional errors is, on the balance, a good approach.

![Bar chart showing speedup comparison](image)

**Figure 3.11**: Speedup comparison of regular support and two other alternatives, one removing the partial versioning support altogether, the other adding dependence violation detection to squash the spawned thread.

Table 3.7: Slowdown compared to baseline speculative parallelization and recoveries rate per 100,000 instructions due to not passing of values produced in primary thread to secondary look-ahead thread after spawning.

<table>
<thead>
<tr>
<th></th>
<th>crfy</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>eqk</th>
<th>fma3d</th>
<th>galgel</th>
<th>lucas</th>
<th>gmean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slowdown (in %)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>0</td>
<td>16</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Recv rate</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.8</td>
<td>1.6</td>
<td>3.6</td>
<td>7.7</td>
<td>8.1</td>
<td>0</td>
<td>33.9</td>
<td>3.6</td>
<td>0.4</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
Figure 3.12: Number of instructions executed in two look-ahead thread mode and number of instructions missed the spawning opportunity because of limited hardware resources in 100 million simulation window.

### 3.7.8 Potential for Multiple Look-ahead Threads

In the current system, we explored the speculative parallelization using only two hardware contexts for the look-ahead threads. When a secondary look-ahead thread is already spawned and a new spawning opportunity arrives, it is lost. However, this is purely an artifact and an arbitrary design choice. In this section, we show potential of multiple hardware thread contexts. From Figure 3.12, we can see that for each application there are significantly more instructions that missed the spawning opportunities than the ones that were executed in the spawned thread mode. On an average, an order of instructions missed the spawning opportunities due to limited hardware resource in the current version of speculative parallelization.

We present preliminary results to show the distribution of number of contexts required for look-ahead on a cycle-by-cycle basis. In this study, we simply count the active number of spawning assuming that unlimited hardware thread contexts are available. From Figure 3.13, it is evident that most of the time, on an average, 4 spawning opportunities are available. This clearly suggests that there are more opportunities to be exploited when we have more than two hardware contexts available/reserved for look-ahead purpose.

Also, note that these extra thread contexts do not have to be active contexts. Our understanding is that a few passive contexts that can save the architectural state and later help the active thread contexts to spawn threads would be sufficient. A passive context can be as simple as a table which records the spawn-target PCs and a copy of register file at the time of the original spawning. From Table 3.7, we know that not passing the exact registers among spawned
thread does not lead to significant slowdown. Once the active thread merges, it can pick an entry from passive thread table and start executing the instructions from corresponding target point onward. Future work will extend the current framework to support these techniques.

3.7.9 Reasons for Extra Recoveries

Here we list some of the possible reasons for extra recoveries in our current system.

- **L0 cache pressure:** With two look-ahead threads, the L0 evictions are more frequent and the recoveries due to additional evictions increase. To gauge the impact due to cache pressure, we increased the size of L0 cache by 2x and noticed that several recoveries that were due to cache pressure were eliminated. While a larger L0 cache eliminates recoveries due to evictions it introduces new recoveries due to stale data.

- **Faulty startup of secondary look-ahead:** In some cases, at the time of thread spawning, state of the primary look-ahead thread is corrupted. When the secondary look-ahead thread copies the states, it inherits a lot of wrong values that might lead to increased recoveries. However, in most of applications, this has negligible impact except one.

- **Impact of runaway spawns:** In general, runaway spawns shouldn’t lead to an increase in the number of recoveries. However, when a thread does not terminate successfully or violates the dependencies, the work done by that thread is thrown away, and the primary look-ahead thread takes over and does the extra task. Runaway spawn might pollute the shared DLO cache in addition to creating extra cache pressure.
3.8 Future Work

Quantitative results, shown in Section 3.7, are what our current system allows us to achieve and not necessarily what can be achieved by speculative parallelization in the decoupled look-ahead. With more trail-and-error of various parameters and sophisticated runtime control policies, speculative parallelization can further improve the performance. There are several interesting future avenues and extensions to this work. In this section, we summarize some of the key ideas which can enhance and optimize the speculative parallelization framework.

3.8.1 Intelligent Runtime Spawning Management

Here we list a few improvements over current thread spawning and management policies.

- **Prioritized spawns**: We observed that the current framework with naive runtime support is not able to fully exploit all spawning opportunities identified by the compile time system. To employ better runtime policies, we should classify the quality of spawn-target pairs based on their dependence distance, robustness, and successes. Dependence distance and successful spawn index product can be used to prioritize the spawns-target pairs by runtime system as they arrive.

- **Dependence classification**: We can classify the dependencies as *weak* or *strong* depending upon how much influence a particular dependence has in deciding the branch outcome and prefetching down the instruction stream. For example, if a load value only determines the outcome of a highly biased branch, we can classify this load instruction as weak dependence and safely remove the dependence arc. This optimization will allow us to create more parallelism in the skeleton binary which is helpful in speculative parallelization.

- **Preserving unspawned opportunities**: In the current framework, when a spawning opportunity arrives while the execution is already in spawned mode, the opportunity is lost. An intelligent choice would be to remember these spawning opportunities as they arrive and once the current spawn terminates, we can pick up the best high-priority spawn available and start execution from there.
3.8.2 Credit Based Spawn-Target Pairs

In the current framework, only highly stable spawn-target pairs are candidates for spawning. These highly stable target points have consistent high $D_{\text{min}}$ and also same fixed spawn point. This naive policy may deny considerable opportunities of spawning when the application does not have highly stable spawn-target pairs. To reap further benefits, we can relax this requirement and implement a more generic solution that will allow us to exploit some parallel execution even in the absence of highly stable pairs. In the following paragraphs, we describe the policies and strategies pertaining to this new solution.

Static/compile time analysis:

Instead of finding only a few highly stable spawn-target pairs, we can assign weight to each feasible spawn-target pair. Let’s assume, the number of instructions between the $i$th spawn-target pair is $l(i)$. This can also serve as the weight for the spawn-target pair. Because we want to exploit the long-range parallelism, the longer the distance the fewer the chances of premature value communication between the two threads would be. In static time, we also calculate the probability of a spawn being a successful which we call probability of success or $p_s(i)$. Static time priority of a pair, $P_{\text{stc}}(i)$, is the product of these two parameters and can be expressed as follows:

$$P_{\text{stc}}(i) = p_s(i) \times l(i) \quad (3.1)$$

Runtime support:

The priority of a spawn-target pair can be used in addition to runtime credits to decide when to pre-empt a pair and give the resources to other stable pairs waiting in the queue for execution. In addition to $P_{\text{stc}}(i)$, we also calculate initial credit for each pair, denoted by $C_P(i)$, in static time. Initial $C_P(i)$ is used in runtime to subtract the credit from a running pair. For simplicity, we define $C_P(i)$ as the $Kth$ fraction of $P_{\text{stc}}(i)$.

$$C_P(i) = K \times P_{\text{stc}}(i) \quad (3.2)$$
K is more likely to be less than 1 and typical values could lie between 0.1 to 0.2. Each time a new spawning opportunity arrives, it subtracts the designated $C_P(i)$ from the $P_{stc}(i)$ of the running pair. We define runtime priority of pairs (denoted by $P_{dyn}(i)$) as following:

$$P_{dyn}(i) = P_{stc}(i) - \sum C_P(i)$$

(3.3)

where $\sum C_P(i)$ is the credit of all the spawn-target pairs which appeared during the execution of current pair. Once $P_{dyn}(i)$ becomes zero for the current running pair, it empties the resources and the next arriving/waiting spawn-target pair gets the opportunity to spawn. If there are spawns waiting in the queue, the pair with highest $P_{stc}(i)$ gets the resources.

### 3.8.3 Simplified Design with Stable and Stipulated Pairs:

In the previous section, we described a generic framework to manage resources among multiple spawn-target pairs. Alternatively, we propose to simplify the framework that is easy to implement and yet capable of exploiting less stable spawn-target pairs as well. We classify the spawn-target pairs into two categories: stable and stipulated. Stable pairs have highly consistent $D_{min}$ and fixed spawn-target pairs. Stipulated pairs have multiple spawn points for single target and moderately consistent $D_{min}$. Runtime system implements following policies to share execution resources among these two kinds of pairs.

- A stable pair can pre-empt a stipulated pair when it arrives and takes the execution resources occupied by stipulated pairs.

- Stipulated pair stores the partial result into passive thread context queue and wait for the stable pair to finish execution so that it can resume its execution.

- Stable pairs subtract credit of 0.5, while stipulated pairs subtract credit of 0.1 from a currently running spawn-target pair.

- When the credit against a stable running spawn-target pair reaches 1, the currently running pair is preempted, and resources are given to the next spawn-target pair.
3.8.4 Potential and Performance Mismatch Exploration

There are various reasons for lower performance in a phase when static time analyzer indicates the high potential for the same phase. Compile time analyzer treats all the basic blocks same, whereas in runtime the execution time of each basic blocks varies depending on the types of instructions present. For example, a basic block with more arithmetic instructions would take relatively less time to execute; whereas a basic block with memory operations that miss in the lower-level caches would take 100s of cycles, if not 1000s.

The above mentioned scenario has interesting implication on our speculative parallelization framework. We assume two basic blocks when they can be statically overlapped, their results would be available when next basic block is executed. In reality, a basic block may take more time to execute, while spawned path basic blocks may execute earlier. The result needed for the downstream basic blocks is not available, which may result into recoveries. This would create a performance gap between the compile time potential and runtime performance. A smarter compile time analyzer should assign weight for each basic block, which is rough measurement of the time it takes to execute. This information can be used by runtime system to spawn only those pairs for which it ensures that results from previous basic blocks would be available when the execution of downstream basic block begins.

3.8.5 Additional Insights and Explorations

In this section we point out some of the additional findings and possible explorations.

- **Small window profiling:** In some cases, small window profiling works well. For example, in *art* we collected profile for 10 million instructions and simulated a much larger window, which still resulted into good performance gain.

- **Ideal study of spawn-target pairs:** In this study, we want to know ideally when a pair is spawned whether it would be successful or not. If so then only we spawn; otherwise we do not spawn in runtime. This would give the upper bound on how much potential we have which can be exploited using the various credit and priority based mechanism, as described in Section 3.8.2.
3.9 Summary

In this chapter, we have proposed a mechanism to apply speculative parallelization to the look-ahead thread. This approach is motivated by two intuitions: (1) look-ahead code contains fewer dependences, and thus, lends itself to (speculative) parallelization; (2) without correctness constraints, hardware support for speculative parallelization of the look-ahead thread can be much less demanding. We have presented a software mechanism to probabilistically extract parallelism and have shown that indeed the look-ahead code affords more opportunities. We have also presented a hardware design that does not contain the support needed for the conventional speculative parallelization (such as dependence tracking, complex versioning, and rollback). For an array of 14 applications, where the speed of the look-ahead thread is the bottleneck, the proposed mechanism speeds up the baseline, single-threaded look-ahead system by up to 1.39x with a geometric mean of 1.13x.

Experimental data also suggest that there is further performance potential to be extracted, if we have more hardware resources to spawn additional look-ahead threads. Current design considers only highly stable parallel regions to be a candidate for the speculative parallelization with equal priority. With a slight modification to our current scheme, augmented with a credit based policy, we can prioritize parallel regions and achieve a much better coverage by considering conditionally stable parallel regions as well.
Weak Dependence Removal in Decoupled Look-ahead

A unique opportunity for optimization in the look-ahead thread is that unlike conventional threads, it is not bounded by unyielding correctness constraints. In particular, we hypothesize that among the apparent dependences in a thread, there are plenty of *weak* links. Cutting them from the skeleton allows us to speed it up without much loss of effectiveness. However, it is not a simple task to identify weak links as the effect of removing a weak link often depends on the dynamic context and whether other links are removed, much like in a Jenga game. Through our explorations, we found that simple heuristics based on the static code and attributes are unlikely to be an acceptable solution as they lead to non-negligible false positives with considerable performance costs – akin to taking out the wrong block in the Jenga game.

Verifying the suspected weakness by actual measurements is probably an indispensable component of any heuristic that is aimed to improve look-ahead thread performance by removing weak instructions. Given such a framework of trial-and-error, we argue that human analysis to find heuristics in code patterns becomes unnecessary. Instead, we can apply a metaheuristic approach and let the system perform the search. A framework based on genetic algorithm can help search for the right set of changes to the look-ahead thread to achieve significant performance gain at relatively low cost without altering the quality of the overall look-ahead process.
We will discuss our proposed solution and methodology in Sections 4.4, 4.5, 4.6, and 4.7. Then we present the experimental setup (Section 4.8) and detailed performance analysis in Sections 4.9, 4.10 and 4.11. But first, we will discuss the motivation for our approach and challenges in identifying weak dependences in more detail in Sections 4.2 and 4.3 respectively. Finally, we present the future work in Section 4.12 and summarize the key findings and insights in Section 4.13.

4.1 Background and Related Work

Although there have been some explorations to skip redundant loads and computations driven by such loads [82, 83], we do not find any work that is close to identifying weak instructions based on the dependence relationships in general-purpose programs. Tseng et al. proposed data-triggered thread programming model in which threads are initiated only on a change of memory locations which drive loads. If a memory location does not change then the whole computation is skipped and the results from the previous dynamic instance of the same computation are used [82, 83]. This technique requires support to remember values from past execution of same static code. Several other proposals describe various techniques to exploit redundant computations. These proposals include value prediction [84], dynamic instruction reuse [85], block reuse [86], and silent stores elimination [87, 88].

More recent work transforms approximable code sections of general-purpose programs to execute them on neural network based accelerators without compromising the quality of the outcome significantly [89]. Arbelaez et al. heuristically discover a simplified form of functional weak dependencies between variables to speedup the search by reducing the size of search trees in tree-based search [90]. One approach to speedup the look-ahead thread by dependence elision is recovery free value prediction [91]. Garg et al. proposed to parallelize look-ahead thread by exploiting more TLP inherent in the look-ahead binary [9]. Certain dependence violations were ignored because the cost of rollback out-weighed the performance gain achieved by correcting dependence violations. Our proposal of weak dependence removal improves ILP of the look-ahead thread and can be synergistically combined with speculative parallelization of look-ahead.
4.1.1 Evolutionary Genetic Algorithms

Genetic algorithms (GA), a subset of much larger class known as evolutionary algorithms (EA), were proposed by John Holland in the early 70’s to solve optimization problems which involved tricky trade-offs [92, 93]. They were inspired from the natural evolution of biological system and tried to mimic the same behavior in machines. Genetic algorithms have shown great potential to solve various inter-disciplinary problems [94]. The actual derivation of new solutions involves manipulation of schemata (building blocks of good solutions). Schemata are spread in the population in proportion to their relative fitness. Two basic operations involved in the genetic algorithms are crossovers and mutations. Single-point crossovers tend to saturate the population and are relatively slower. Uniform crossover and fusion operator have been proposed which tend to create more diverse new solutions compared to single-point crossover [95]. Mutation plays an important role in evolution, especially towards the later generations, and variable mutation rate helps in keeping the population pool diverse [96].

![Flowchart of Genetic Algorithm Process](image)

Figure 4.1: Various phases in a genetic algorithm based evolution process.
A simple genetic algorithm based evolution process flow is depicted in Figure 4.1. Initial solutions are combined using crossover and mutation to create newer solutions. Fitter candidates are promoted to next generations based on a probabilistic model. The process continues until we achieve desired traits in the current population or we exhaust the allocated resources.

4.1.2 Genetic Algorithms in Computer Architecture

There has been very limited use of genetic algorithms to optimize various aspects of microarchitecture. In particular, Abts et al. explored the design space of on-chip fabrics to find optimal memory controller placement relative to different topologies (i.e. mesh and torus), routing algorithms, and workloads using genetic algorithms [97]. They argue that the location of the memory controllers can reduce contention (hotspots) in the on-chip fabric and lower the variance in reference latency. Limited pin bandwidth prevents the integration of a large number of memory controllers on-chip. With many cores, and few memory controllers, the location of controllers in the on-chip interconnection fabric becomes an important and yet unexplored question. Pierre et al. and Kumar et al. proposed genetic algorithm based solutions to generate low-cost feasible computer network topologies subject to various constraints [98, 99]. Their results confirm that the efficiency of genetic algorithm to provide good solutions for medium-size computer networks, in comparison with well-tried conventional methods is indeed very high.

4.2 Motivation for Exploiting Weak Dependences

As pointed out in Section 2.5, in the decoupled look-ahead system, it is the speed of the look-ahead agent which is the bottleneck and slows down the overall system. To further speed up the look-ahead thread, without affecting the quality of the look-ahead process, a number of approaches can be taken. Speculations commonly used in helper threads [37, 45, 47] are an example. In theory, deep transformations beyond mechanistic slicing can be performed to generate an optimal skeleton code solely for the purpose of look-ahead and we present one such case here. In this work, we search for solutions that do not require rewriting the skeleton code, but simply skip those instructions that contribute least to its look-ahead purposes.
Figure 4.2: Example of weak dependences in the original program binary. In these examples from applications vpr and mcf, each box represents a basic block. For clarity, only a subset of instructions are shown. Frequent values of inputs and output registers – captured using a profiler – are shown on the right hand side. Shown in bold are instructions that are weak and can be safely removed without impacting the quality of look-ahead.

Intuitively, not all computations are equal. For instance, some merely add small adjustments to addresses which are inconsequential if all we wanted to do is to prefetch the right cache line. Put in a different way, for our purpose of look-ahead, the final address value weakly depends on such small adjustments. However, if we use traditional dependence analysis on the static code, we are bound by the mere appearance of dependence and would include all instructions on the backward dependence chain, regardless of the strength of each link. Dependence strength is not a consideration in a traditional analysis, which we used to generate the look-ahead thread.

Weak dependence is not a mere possibility. We have observed many examples in actual code. A simple litmus test can be used to identify a weak dependence in isolation: if we remove the instruction(s) from the look-ahead thread and the overall system runs faster, that instruction (chain) can be considered to be weakly depended upon (or a weak instruction). In Figure 4.2, we list a number of easy-to-explain examples from two applications (vpr and mcf). Note that
these examples are *a posteriori* after experimentally identifying weak instructions.

- **Mostly silent loads (and stores):** These instructions often load or store the same values to their target register or memory location. In example ①, the value loaded from memory location happens to match the content of the register very often. One particular root cause of such behavior is unnecessary register spilling that led to silent loading back later.

- **Inconsequential adjustments:** In both ② and ④, the value was incremented by 1. In these two cases, updates are inconsequential and removing them ended up being helpful.

- **Dynamic NOP:** Similar to the silent loads/stores, an arithmetic and logical instruction may end up not changing the result most of the time. Particular examples are sign extension as in ⑤ and logical AND operation in ⑥: When the value is predominantly non-negative, sign extension has no effect. Similarly, when one input is zero the other input does not matter for logical AND operation.

These are only few instances of weak dependences among many present in a program. To guarantee the *absolute* correctness, conventional architecture has to always execute these instructions and even an aggressive state-of-art compiler can not optimize them. However, the look-ahead thread is not constrained by the correctness requirement, and a violation only results in slight performance degradation (if at all).

Not only do weak dependences exist, they also have non-trivial impact on system performance. In fact, in one particular application (*art*), we found that removing just a single instruction from the look-ahead thread resulted in a 9.4% performance improvement of the entire system! Clearly, systematically identifying and exploiting weak dependences is useful. Although *art* belongs to the category of typical loop based applications in which few loops can dominate the whole execution, even after barring these aberrations there are still plenty of weak instructions present in programs.

These evidences motivate us to identify and exploit the weak dependences to improve the speed of the look-ahead agent at very little or no degradation of quality of look-ahead. Most importantly, to exploit the weak dependences we require very minimal additional hardware support on top of the baseline look-ahead support.
4.3 Challenges of Identifying Weak Dependences

The very concept of weakness is context-dependent. Fundamentally, we are performing a trade-off: removing some instructions can shorten the dependence chain for the look-ahead thread, but the resulting approximation leads to other costs. For instance, an incorrectly prefetched cache line can hurt performance through pollution. In particular, when the approximation eventually causes the look-ahead thread to deviate from the original program’s control flow, we need to “reboot” (or re-initialize) it which is a costly operation. In the look-ahead thread, it is all but impossible to analytically attribute the true cost to the exact root cause. In this section, we list a few major challenges that are involved in identifying the weak instructions accurately.

4.3.1 Weak Instructions Do Not Look Different

While weak dependences can be experimentally identified and we can assign reasons to explain them after the fact (as in the examples above), predicting them ahead of time is an entirely different matter. After comprehensive testing of individual instructions to round out all single weak instructions, we attempted to identify their uniqueness based on the static instruction. Unfortunately, they consist of all types of instructions and do not appear to be special at all. We present a detailed analysis in Section 4.10.4. Is it the specific computation and value involved that make their computation less consequential than others? We can not say for sure. To make things more difficult, removing a critical instruction incorrectly identified as a weak instruction from the computation often leads to significant performance penalties. Ultimately, it is the specific computation and values involved that make their computation less consequential than others. This makes a static, heuristic-based approach to identify weak dependences very unlikely to be successful, in our opinion.

4.3.2 False Positives Are Extremely Costly

Admittedly, some instructions are more likely than others to be weak, but even in those cases, a single false positive can negate all gains from correct predictions. Case in point: instruction
zapnot\textsuperscript{1} is a common occurrence in experimentally identified weak instructions. In \textit{gap}, a whopping 83.3\% of all zapnot (static) instructions are weak in isolation. Removing all of them allows the program to speed up by 3.4\%. Unfortunately, if we falsely identify even one zapnot from the remaining cases as weak, we can lose up to 5.9\% performance, more than wiping out all the gain from the right selections. If we have more than one false positives, the performance degradation can go up to 13\%.

4.3.3 Weakness Is Neither Absolute Nor Additive

Finally, even if it is possible to identify weak dependences in isolation, the effect of removing multiple instructions is certainly complex and non-linear. To illustrate this effect, we experimentally identify all weak instructions (in isolation) in application \textit{perlbmk} and sort them by the number of cycles saved due to their removal. In Figure 4.3, we show the cumulative effect of removing these 300+ instructions one by one. Note that all 300 instructions are by themselves weak, but once we pile on about 50 of them, the overall impact becomes negative. It goes from bad to worse, slowing down the program by almost 40\% at some point. To continue the Jenga analogy, the tower clearly collapsed when we removed a collection of individually safe-to-remove blocks.

![Figure 4.3: Performance impact of cumulatively removing instructions identified as weak in isolation.](image)

\textsuperscript{1}It sets selected bytes of the source register to zero and copies the result into the destination register.
Although each application has its own shape in a plot like this, there are certain commonalities such as adding one modification (again by itself positive) can significantly decrease performance. This underlines the challenge of a code analysis-driven approach to identifying weak dependences: more often than not, whether a chain of instructions can be removed to improve performance is not an intrinsic property of the computation, but depends a lot on what is happening elsewhere in the code.

4.3.4 Putting It Together

Given the ad hoc nature of dependence strength, the non-linearity of performance effects, and the interdependence of individual chains, using heuristics and code analysis to predict weakness is at best a non-robust mechanism – in our opinion. However, regardless of the root cause of weakness, the effect of removing instructions is experimentally measurable. Moreover, it appears to be reasonably stable across time and different inputs. The performance impact of removing an instruction can be measured quite early without the need to wait for long-term observation. Therefore, we can easily envision a self-tuning system that identifies weak instructions through trial-and-error and figures out – via metaheuristics – the right combination of these instructions to remove in order to maximize performance gain.

Genetic algorithm is a good metaheuristic method to use in such a self-tuning system. Developed in the 1970s to solve optimization problems, genetic algorithm mimics natural evolution to adapt solutions slowly but steadily towards more optimal versions [92, 93]. While genetic algorithms can be a slow mechanism to derive solutions, they have shown great potentials in finding intelligent solutions of conventional optimizations and problems involving tricky trade-offs [94]. Finally, genetic algorithm is a natural choice for our system: We use a bit mask in our design to indicate which instructions in the binary are part of the skeleton to be executed in the look-ahead thread. The way the bits govern the behavior of the skeleton is analogous to genes governing an organism, making genetic algorithm a natural choice.
4.4 Genetic Algorithm Based Framework

4.4.1 Basic Design

Our look-ahead skeleton optimization problem maps naturally to a genetic algorithm framework. The individual weak instructions are the genes and the genetic algorithm will find a collection of them (chromosome) that, when removed from the look-ahead thread, maximize performance. In fact, the genes can be broader than individual weak instructions. They can be any kind of modifications to the look-ahead thread, including removing a group of instructions as an atomic unit, or adding instructions that do not appear to be depended upon back to the look-ahead thread.

Our goal is to find a skeleton that maximizes performance. In our current design, the skeleton is a masked version of the program binary: some instructions are dynamically suppressed at fetch time. The task of the genetic algorithm is to find the best mask. The initial candidate solutions can be generated from any heuristics. Indeed, they can be completely random. From pairs of existing solutions, we use crossover and mutation to create pairs of children solutions as part of the next generation. To guide the evolution towards better solutions, we need to assign better (faster) solutions with a higher fitness score. Evolution can be stopped after a certain number of generations or upon seeing diminishing return.

In a most straightforward case of fitness testing, given a mask to be tested, we run the application with the corresponding skeleton as the look-ahead thread, measure the new execution time, and use the number of cycles saved (compared to the reference run using the default skeleton and the same training input) as the fitness score. Fitness tests can also be performed online while the application is running (Section 4.6). As often is the case with genetic algorithm-based problems, fitness tests are time-consuming. Given the size of the gene pool, the number of fitness tests needed to reach a reasonably good solution is likely to be in the thousands. However, with sampling-based performance measurement, in a single second of real system time, 10s, if not 100s, of these tests can be performed. Even with a fully online system, most applications would have meaningfully evolved in the first minutes of their cumulative execution, given proper system support such as keeping metadata of evolution persistent.
During this period, the runtime system only needs to perform minimum bookkeeping and incurs insignificant overhead. Program performance can be accurately estimated by decomposing it into subroutine-level modules, which are highly repetitive and lend themselves to efficient sampling-based measurements [100].

As typical in genetic algorithms, given one generation of solutions (parent chromosomes) we use mutation and crossover to create children (new solutions) that differ from their parents. In the reproduction process, genetic algorithms mimic the process of natural selection by giving the fittest individuals more chances to survive to the next generation. The fitness test is the measure of the performance impact of said modifications. As we will discuss later, this test can of course be done offline but can also be performed online.

In the following, we discuss the genetic algorithm framework (Section 4.5), implementation issues regarding fitness tests (Section 4.6), and the sampling framework that accelerates the fitness tests (Section 4.6.2). But before that we review some of the techniques that we can leverage to speedup the baseline genetic evolution process.

### 4.4.2 Optimizations to Baseline Genetic Evolution

The baseline genetic evolution is computationally most expensive step in whole evolution process and many techniques to speed up the overall evolution process exist. They are based on certain heuristics that allow the search to more quickly converge to optimal solutions. A number of them can be applied in our case and we present a brief overview of some of them here.

- **Hybridization:** Hybridization makes use of known good solutions via another algorithm (or heuristic) to help “jump start” the evolutionary process. We use two heuristics to derive initial solutions, which we will discuss in detail later. As an example, we assume the modifications to different subroutines are more or less orthogonal in terms of their impact to program performance. Therefore, we can pick one known good modification per subroutine and combine them into an initial solution.

- **Fusion operator:** Increasing the speed of change in each generation is one way to more quickly evolve to the optimal solution. *Uniform* and *multi-point* crossovers are attrac-
tive alternatives to single-point crossover as they tend to produce very diverse solutions from one generation to another [95]. Fusion operator is an enhancement over uniform crossover where the shares of parents are biased according to their fitness.

- **Elitism**: Elitism ensures that the best solution(s) from previous generations are shielded from the destructive reproduction. There are various flavors of elitism policy and an elitist does not need to participate in the reproduction process; it simply gets promoted to the next generation.

- **Adaptive mutation rate**: When a population set contains very similar chromosomes, perhaps after several generations of evolution, creating very diverse chromosomes just from crossovers becomes challenging. This translates to solutions that are local optima. One way to get out of these local optima is to increase the mutation rate. It has been shown that variable and adaptive mutation rate help keep the generation pool diverse [96].

- **Unique chromosomes**: By keeping the chromosomes unique (without duplication), we avoid wasting resources on repetitive fitness tests of the same solution.

### 4.5 Framework of Evolution

In this section, we describe our framework of evolution to refine look-ahead skeleton by identifying the weak instructions and removing them from the look-ahead thread.

#### 4.5.1 Genes

In our framework, a gene is the fundamental unit of modifications to the skeleton. For the most part, such a modification is masking off one (static) instruction. A “good” gene is one that improves the system’s overall performance, i.e., one that removes a weak dependence. Since our baseline system depends on the fact that both threads execute all branches, in this work, we will not attempt to remove branches. Therefore, for a skeleton with $n$ static non-branch instructions, we will have $n$ possible genes. Out of these $n$ genes, there is a much smaller subset of good genes that speed up the execution. To expedite the convergence to good solutions, we only use those to form chromosomes.
It is entirely possible that removing two (or more) instructions together can improve performance when removing any individual instruction alone slows down the program. The reason is that a weak dependence can be a chain of instructions, of which, any individual instruction will not appear to be weak. In general, if we remove an instruction $I$ from the code, we should remove all instructions on $I$'s exclusive backward dependence chain, i.e., all instructions that are only helping instruction $I$ and nothing else. Unfortunately, identifying the entire exclusive backward dependence chain is not easy as we need either complex hardware support or slow software analysis. Note that letting genetic algorithm stumble upon these chains is theoretically possible, but exceedingly inefficient. For simplicity, we choose to use an approximate approach and form multi-instruction genes as possible weak dependence chains to be removed. To limit the number of genes for testing, we only form genes from consecutive instructions and up to five in one gene.

Finally, at the end of the genetic algorithm-based search, we can take the result, feed it to the static analyzer and do a pass to find out other instructions that could be removed. Consider the contrived example of insert a chain of two instruction that are useless to the code: the first added a large number to R1, and the next subtracted the same number from R1. Taking away only one instruction at time, however, the code may behave erratically, incorrectly suggesting the removed instruction to be essential.

### 4.5.2 Multi-Instruction Genes

As mentioned before, in some cases it is useful to remove multiple instructions in a group as opposed to single instruction. Removing of single instruction performs partial computation which is more harmful than either removing the whole computation chain or performing the complete computation. In these two particular examples, shown in Figure 4.4, it is evident that removing one instruction only in isolation results into a large value into the register on which a following branch depends upon and recovers often. If both the instructions are removed together then the branch input register value is same/similar to the baseline look-ahead execution.
Figure 4.4: Multi-instruction based genes from applications vpr and gzip. (A) Removal of either of the bold instruction results into large value of \(t_{10}\) register which is “very different from actual value of \(t_{10}\) on which the branch outcome depends. (B) The branch outcome depends upon the \(s_{3}\) register which is indirectly dependent upon \(gp\) register through certain offset based loading. Removal of either of the bold instruction again results into large value of \(s_{3}\) register which is quite different from actual value of \(s_{3}\) that is close to 0.

4.5.3 Prescreening and Fitness Tests

To measure the impact of a gene, we activate it (masking off the corresponding instructions), measure the speed, and compare that to the original system without the modification. Again, in the simpler example of off-line profiling, the fitness score of the gene is the number of cycles saved. Note that the fitness score is only used in a probabilistic way later on. Measurement precision requirement is not high. In particular, in the first step of operation, we will filter out the bad genes from all possible genes. We do not need exact fitness score of a bad gene. In many cases, it is very clear early on that a gene is a bad one (e.g., causing too many look-ahead thread reboot). We can thus terminate the measurement early.

Each gene is given a fitness test to estimate its performance impact on the whole system. In the most straightforward design, the fitness test is measuring the difference of program performance using a training input under the new look-ahead thread compared with that under the baseline look-ahead thread. As we will see later, this can be significantly accelerated. After the fitness measurement, a minority of these genes have a positive fitness score. After pre-
screening single-instruction genes, we screen multi-instruction genes. A multi-instruction gene is only considered positive if its performance benefit is higher than the sum of all constituent instructions’ individual benefit.

4.5.4 Initial Chromosome Pool

Given $N$ positive genes in an application, the chromosome is represented as an $N$-bit vector. In our experiments, $N$ ranges between 30 and 300. To “jump start” the evolutionary process, in addition to $N$ single-gene chromosomes (Figure 4.5-a), we seed the initial pool of chromosomes with some heuristically-derived solutions. This is known as hybridization.

The first heuristic is simply to turn on many genes and create a “superposition” of genes. Not surprisingly, turning on all genes is almost never a good design, as too many approximations weaken the function of the look-ahead thread beyond repair. To create partial superpositions, we sort all $N$ genes based on their fitness score and create $N - 1$ different chromosomes starting from one that contains the two top-ranking genes, and going down the list, adding one more gene at a time to the previous chromosome (Figure 4.5-b).

<table>
<thead>
<tr>
<th>Chromosome</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Chromosome Diagram" /></td>
</tr>
</tbody>
</table>

Figure 4.5: Look-ahead chromosome representation. Genes are basic blocks of chromosomes and can be either single-instruction or multi-instruction. Marking of a gene (shown as X) knocks out the associated instructions from the chromosome (look-ahead binary).
Lastly, we use the heuristic that modifications to different subroutines are likely to be more orthogonal to each other than are those in the same subroutine. We thus select one gene from each subroutine to form a chromosome. Specifically, we sort the genes from each subroutine separately based on their fitness score. We pick the top-ranking gene from each subroutine to form the first chromosome; the second-ranking genes for the second chromosome, and so on. If all the genes from a subroutine have been exhausted, that subroutine will not contribute to later chromosomes. Finally when all but one subroutine have exhausted their genes, the process stops (Figure 4.5-c).

4.5.5 Population Size and Parent Selection

For simplicity of implementation, we use a fixed population size. Based on several factors such as the number of positive genes, the cost of fitness tests, and convenience of experiments, we chose 96 as the population size. Of all the initial chromosomes formed, only 96 unique members will be selected as generation-1 population. To select these members, we use an approach that can be likened to repeatedly spinning a roulette wheel with a large number of slots to select one winner at a time. Each member occupies a number of slots proportional to its fitness. Given one generation of population, we go through the parent selection process in order to reproduce. This is again done with the roulette wheel approach. We go through iterations, each time selecting two parents to do crossover and mutation to generate two offsprings.

We also conducted limited experiments with doubling the population size and reducing it to 30 unique chromosomes. Applications that have fewer good genes (in the range of 50 or less), a population of 30 chromosomes essentially resulted in the same level of performance after a fixed number of generations (7 in this case). However, applications with larger number of genes (more than 200), doubling the population size improved the performance of best chromosome in earlier generation (up to generation 4). Towards the later generation (7 and later), the performance gain was almost similar. This suggests that, by and large, a population of 96 unique chromosomes for our exploration was a good choice overall.
4.5.6 Crossover and Adaptive Mutation

Crossover is the process of multiple (two in our case) parents swapping parts to form the same number of offsprings. Uniform and multi-point crossover are attractive alternatives to one-point crossover as they tend to produce very different solutions from one generation to another [95]. We experimented with single-point, multi-point, uniform mask based, fusion operator, and xor based crossover. We chose fusion operator based on limited experiments. Fusion operator is an enhancement over uniform crossover where each bit of the vector is randomly decided to follow one of the parents based on a probability proportional to that parent’s fitness score.

Crossovers allow chromosomes to exchange their genes but does not create diversity outside gene patterns present in the current generation. Mutation is thus added, which randomly flips individual genes based on certain probability. This probability increases in each generation to lower the chance that the algorithm gets stuck at local optima [96].

4.5.7 Survival and Uniqueness Test

It is possible to go through iterations of reproduction and produce more offsprings than parents. In that case, after performing fitness tests on all offsprings, a mechanism similar to the roulette wheel can be used to determine which ones survive into the next generation to keep the population size fixed. In our case, to minimize expensive fitness tests, we do not generate superfluous offsprings. We perform tests to maintain unique offsprings and all of them survive into the next generation. We also incorporate elitism, which is a type of mechanisms to shield the best solution(s) from destructive evolution. We use one flavor of elitism which remembers the currently known best chromosome. This chromosome may not be present in the current population.

4.5.8 Putting It Together

The complete process flow is shown in Figure 4.6 and can be divided into three parts: baseline look-ahead thread construction, initial chromosome generation, and evolution. We start with program binary and build baseline unoptimized look-ahead binary using conventional approach [3]. We then select positive single- and multi-instruction genes, i.e., weak dependences
in isolation. We form some initial chromosomes from these genes based on some heuristics. Finally, we start the genetic algorithm to gradually fine-tune the solutions. Various phases of complete genetic evolution framework are described in detail here:

1. *Baseline skeleton creation*: We assume a baseline look-ahead binary (skeleton) is generated using a binary parser, similar to [3].

2. *Seed evaluation and gene creation*: For the pre-generation, framework takes un-optimized skeleton and computes individual seed’s (instruction’s) fitness and assembles them in a pool. Some individual seeds are combined to create multi-seed genes for the very reason that they work well in a group as opposed to in isolation.

3. *Heuristic based chromosomes*: Genes from different subroutines are combined to create orthogonal chromosomes. We also combine genes in superposition manner to create few superposition chromosomes.

4. *Chromosomes pool creation*: Single-gene based chromosomes are created and are pooled along with heuristic based chromosomes.

5. *Parent selection*: Once we have various kinds of chromosomes we use a fitness favoring mechanism to pick fittest chromosomes as parents for reproduction. Due to experimental
evidences and execution resource limitation we limit the number of parents to 96 and keep the subsequent generations’ population constant.

6. **Reproduction**: Two members from the population are picked randomly to create two offsprings using uniform crossover and random mutation. We ensure that no duplicate children are produced at this point for optimization efficiency.

7. **Children fitness test**: All children are supposedly different from their parents and to know the exact fitness score we perform a sampling based fitness test to know for each child.

8. **Replacing parents**: Once we have the children with known fitness we use the same parent selection mechanism to graduate the children to become parents for next generations.

Our evolution framework iterates through above steps and typically runs for 5-7 generations. We also incorporate elitism and de-duplication policies into our framework for better coverage and faster convergence.

### 4.6 Optimizations to Fitness Tests

In our current evolution framework, fitness tests are one of the most time consuming steps. Thus, in this section, we propose methods to either speed up the fitness tests or utilize profiling info from the past executions.

#### 4.6.1 Online and Offline Fitness Tests

The tuning performed by the genetic algorithm can be done either offline or online. In the offline case, the algorithm will measure the execution time of the same application (window) many times, each time with a different skeleton\(^2\). We found that a few generations of evolution is sufficient to significantly improve the performance. The control program running the genetic algorithm only needs to manipulate bit masks for skeletons to be tested, launch the program, read performance counters, and perform simple calculations and book-keepings. It thus incurs

\(^2\)For those executions faster than the system using the baseline skeleton, the fitness score is simply the cycles saved. For the execution slower than baseline, the exact speed is unimportant as we set their fitness score to 0.
negligible overheads. The bulk of the tuning delay comes from the actual profiling runs. But these runs need not to be extensive and there is significant parallelism in administering the fitness tests. Overall, it is feasible to have a fully automated self-tuning process complete in minutes. Such offline-tuning can be done at software release or install time. The cost of tuning is easily amortized over the life cycle of the particular release.

With this evolution procedure, we can evolve the design either offline or online as the application executes. The only difference is how the fitness tests are administered. For an offline implementation, a control program implements the evolution process and spins off profiling runs with different versions of the look-ahead thread to measure their performance. In our experiments, we found that the impact of a modification (especially its polarity) is clearly established very early on. With 10s to 100s of millions of cycles, the result is robust enough to be used to predict the code’s general behavior under different inputs. The task to generate different versions of the look-ahead thread is almost trivial: we exchange instructions to be removed with NOPs. Overall, the profiling latency is dominated by actual measurement time.

In addition to offline tuning, we can also perform the evolution online as the program runs. For online evolution, the system is similar to diagnosis embedded in software that we see today. Instead of sending certain statistics to the software vendor to improve the next release (or to mine the data for commercial purposes), we keep the statistics as the application’s metadata to improve its speed. The metadata will be kept on persistent storage so that the evolution process can span over multiple runs, if needed.

During the initial stage of evolution, the program will likely run slower than without the self-tuning. This is mostly because we will inevitably try bad solutions. But this is only a short-term cost that will be more than compensated for in the later stage of evolution (Section 4.10). Another source of slowdown is the intermittent execution of the control algorithm, which reads counters, calculate probabilities, and so on. However, this overhead is largely theoretical as the magnitude is minuscule. We have measured our implementation of the control program and found that it executes about 16 million instructions for the entire evolution process that can last

---

3More precisely, we do not have a separate code for the look-ahead thread. It is merely represented by a bit mask indicating which instructions of the main thread are on the look-ahead thread. This not only saves space, but also avoids a number of subtle implementation complexities. We assume the architectural support of the mask bits in the instruction cache and a mechanism to load them from the program binary.
hundreds of billions of instructions. Overall, the overhead for online self-tuning is negligible. The real question is how fast we can reach an optimal stage. And there are mechanisms to accelerate that as we discuss below.

4.6.2 Sampling-Based Fitness Tests

To screen for potential genes, we will test static instructions on the look-ahead thread, which can be more than a thousand in some applications. After screening for potential genes we need to perform $N_{IC} + m \times N_P$ fitness tests, where $N_{IC}$ is the number of initial chromosomes, $m$ is the number of generations of evolution, and $N_P$ is the size of population (96 in our case). All together, hundreds of fitness tests are needed in our applications. If we use naive (offline) profiling, hundreds of runs of an application are needed to reach a solution (though using smaller inputs). Significant reduction in test time can be achieved using two simplifications: sampling and multi-gene tests.

**Sampling:**

The code module (loop or subroutine) that contains the gene are invoked countless times during one run of the application. These instances are rather self-similar [100]. Taking enough sample instances will provide a reasonably accurate estimate of the impact of the modification, especially because the sign of performance impact is more important than the exact magnitude – the magnitudes are only probabilistic heuristics. And indeed, as we will show later, sampling errors do not affect the quality of the solution.

Module-based sampling is especially useful for the online version. Unlike offline profiling where we can control what to run when performing fitness tests, in an online system we do not have the choice. Since we need to measure execution speeds with and without a modification at two different time points, we would need the system to behave the same way at these two points except for the modification. This way, the difference in measurement can be attributed to that modification alone, not random variation. If we choose the two time points to be two instances of the same code module (a form of stratified sampling), the random variation will be far lower than if we choose two time windows [100]. Of course, the impact will still be calculated from
the measurement of many samples. Once we know the performance impact of a particular chromosome on every module, the program-wide impact is estimated as the weighted-average of the per-module impacts, the weight being that of the execution time for each module.

As opposed to all-instance testing for a gene we tried only a sample of instances and then measured their fitness which was used to predict the overall fitness of the gene. For example if a gene $g_i$ has $n$ dynamic instances, we tried only $m$ (where $m < n$ and $m$ is factor of $n$) and then try to extrapolate the fitness score for all $n$ instances. This works in general because from the genetic test we roughly need to know the orientation of gene (whether good or bad) as oppose to exact fitness score. In our experimental setup, with trial and error we achieved a sampling rate 1 out of 30 to yield similar performance as full fledged simulation. This means a gene candidate which had been selected to be removed would only be measured 1 out of 30 instances to determine the overall fitness.

**Multi-gene fitness tests:**

When we make a modification to some part of the code, intuitively its performance impact is localized to a certain extent. If we assume that the impact is limited to the code module that contains the modification (or gene), then we can perform tests on different genes that appear in different code modules simultaneously. All we need to do is to measure performance (say IPC) of the instances of code modules and attribute the change in performance to the tested gene within the corresponding code module. When we use both simplifications, we will be able to perform a fitness test not for every profiling run of an application, but for every batch of instances of a code module.

With appropriate runtime support, the test results and intermediate genetic algorithm search results can be recorded as metadata of the program. That way, the search process can span multiple runs of the application if necessary. The size of code module instances ranges between tens of thousands of instructions to a few million [100]. Even testing for thousands of genes will only lasts on the orders of 100 billion instructions, or minutes of actual wall clock time. Note that during this period, the application is still fully productive. It may be slowed down occasionally due to testing of a bad gene.
4.7 Hierarchical Online Evolution Framework

In this section, we describe the online version of hierarchical genetic algorithm based framework to optimize look-ahead thread by removing weak dependences in runtime. Our system opportunistically tests individual genes for their fitness orientation, either good or bad, and then combines the good genes using hierarchical module based genetic evolution.

4.7.1 Baseline System Performance

Because the fitness test is the most time-consuming step in whole genetic evolution process we apply numerous optimizations to the fitness test. Unlike offline version, we have limited computational resource which sequentially tests genes and evolves chromosomes, thus we are constrained to optimize the fitness tests. To measure the baseline performance, programs are broken into modules which are either large subroutines or loops. A subroutine/loop smaller than 100,000 instructions per instance is considered a part of caller module. A larger code module with size bigger than 1 million instructions per instance is broken into smaller modules. We sample each module enough number of times, before it becomes stable, and then only we make a decision about it. According to our criteria, the lower bound of sampling is 10 instances per module and upper bound is when the ratio of coefficient of variance (CoV) and the sample size \( n \) becomes less than 1% of the mean IPC of instances accumulated so far.

We try two sampling strategies: clustered, in which consecutive instances are used to measure the gene fitness; and distributed in which interleaved instances are used to measure the gene fitness. First \( n \) instances of a module are used to measure the cycles and instructions for the baseline system’s performance. This is also called the native execution speed of the module.

4.7.2 Initial Gene Evaluation

Once we know the native execution speed of a module, we try individual genes one after another (one at a time) in each module. It is possible to carry out gene testing in one module while other module is going through chromosome formation and genetic evolution. We make the assumption that removing a gene will make a difference only to the module which it belongs
Figure 4.7: Correlation between speedups and norm. recoveries of initial genes/chromosomes.

to. For large code modules, this assumption is quite effective and reasonable. After recording enough instances of a module, enforced by stability criteria, we compute the orientation of a gene by comparing the fitness with the baseline native execution.

4.7.3 Early Termination of Bad Genes

An interesting optimization to fitness test is to determine the orientation of a gene early by system parameters such as branch recoveries. Our observation is that when, at any point of time, a gene removal results into 20% extra recoveries then it is highly unlikely for that gene to be good. As shown in Figure 4.7, it is quite evident that speedup of a solution is strongly correlated with the number of extra recoveries it has compared to the baseline decoupled look-ahead system. In our framework, we exploit this fact by measuring the average number of recoveries per module instance. After the minimal number of samples are taken if we find a substantial difference in the number of recoveries in two system we skip the further testing of the gene and mark it as bad gene.

4.7.4 Module-level Genetic Algorithm

Once we know all the good genes from a module, we construct the local module-level chromosomes using genetic algorithm operations. The length of the chromosome is same as the number of good genes in the module. The number of local chromosomes is also same as the
total number of good genes in the module. These single-gene chromosomes take part in local genetic evolution, through crossovers and mutations, and generate new set of chromosomes. We perform this process until the generational process saturates. These conditions are used to determine whether a module’s chromosome population has saturated.

- The best chromosome reaches within 10% of the total fitness of individual genes.
- Average fitness of chromosomes has reached within 10% of the maximum fitness.
- Improvement in the best chromosome from previous two generations was less than 1%.
- A total of 10 generations in local genetic evolutions are reached.

The module-level genetic evolution process to determine the best chromosome stops when any of the above condition is satisfied. In a local, module level, genetic algorithm following attributes of chromosome pool are recorded.

- The best, and the average chromosome fitness of each generation is recorded.
- Top 10% chromosomes with their fitness scores and generation in which they were produced are also recorded as metadata.

### 4.7.5 Global Genetic Evolution

After 90% of modules have evolved their local chromosomes, they are combined using the orthogonal property across the modules. Various locally optimized chromosomes are turned on in combination with others and their fitness is compared by merging their effects on the subroutine’s in which they exist. However, there is a problem: if a section of chromosome never executed again then we don’t know the exact fitness of the chromosome. Fitness of a global chromosome is determined by combining the native execution speed of each module involved and subtracting the new combined execution speed from native execution speed. After 5 generations, when the evolution is saturating we record the best chromosome as metadata and assign it to be the final chromosome. However, this can be subject to further optimization.
4.8 Experimental Setup

We perform our experiments using a cycle-level, execution-driven in-house simulator. We faithfully model support for a decoupled look-ahead system, including when the lead thread diverge from the actual program’s control flow. The simulator also faithfully model a number of details in advanced designs such as load-hit speculation (and scheduling replay), load-store replays, keeping a store miss in the SQ while retiring it from ROB [77]. Our baseline core is a generic out-of-order microarchitecture with parameters loosely modeled after POWER5 [78]. The configuration parameters are shown in Table 6.1.

An advanced hardware-based global stream prefetcher based on [79, 80] is also implemented between the L2 cache and the main memory: On an L2 miss, the stream prefetcher detects an arbitrarily sized stride by looking at the history of the past 16 L2 misses. If the stride is detected twice in the history buffer, an entry is allocated on the stream table and prefetch is generated for the next 16 addresses. Stream table can simultaneously track 8 different streams. For a particular stream, it issues a next prefetch only when it detects the use of previously prefetched cache line by the processor. To compute the energy and power, our system uses Wattch [101] for dynamic power modeling and HotSpot [102] with BSIM3 models [103] for leakage power modeling.

4.8.1 Applications and Inputs

We use applications from SPEC CPU2000 benchmark suite compiled with optimization flag -O3 for Alpha using a cross-compiler built on gcc-4.2.1. We use the train input for profiling, and run the applications for 500 million instructions, which generally cover the exercised code region in later non-profiling runs. After the offline evolution, the modified look-ahead thread will be used for performance benefit analysis. We use ref input and simulate 100 million instructions after skipping over the initialization portion as indicated in [81].

---

4Instruction interpretation and Linux system call emulation are partially borrowed from [76].
### Table 4.1: Microarchitectural configurations.

<table>
<thead>
<tr>
<th></th>
<th>Baseline core</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch/Decode/Issue/Commit</td>
<td>8 / 4 / 6 / 6</td>
</tr>
<tr>
<td>ROB</td>
<td>128</td>
</tr>
<tr>
<td>Functional units</td>
<td>INT 2+1 mul +1 div, FP 2+1 mul +1 div</td>
</tr>
<tr>
<td>Fetch Q/ Issue Q / Reg. (int,fp)</td>
<td>(32, 32) / (32, 32) / (80, 80)</td>
</tr>
<tr>
<td>LSQ(LQ,SQ)</td>
<td>64 (32,32) 2 search ports</td>
</tr>
<tr>
<td>Branch predictor</td>
<td>Gshare – 8K entries, 13 bit history</td>
</tr>
<tr>
<td>Br. mispred. penalty</td>
<td>at least 7 cycles</td>
</tr>
<tr>
<td>L1 data cache (private)</td>
<td>32KB, 4-way, 64B line, 2 cycles, 2 ports</td>
</tr>
<tr>
<td>L1 inst cache (private)</td>
<td>64KB, 2-way, 128B, 2 cycles</td>
</tr>
<tr>
<td>L2 cache (shared)</td>
<td>1MB, 8-way, 128B, 15 cycles</td>
</tr>
<tr>
<td>Memory access latency</td>
<td>200 cycles</td>
</tr>
<tr>
<td>Look-ahead core:</td>
<td>Baseline core with only LQ, no SQ</td>
</tr>
<tr>
<td>L0 cache:</td>
<td>32KB, 4-way, 64B line, 2 cycles</td>
</tr>
<tr>
<td>Round trip latency to L1:</td>
<td>6 cycles</td>
</tr>
<tr>
<td>Communication:</td>
<td>Branch Output Queue: 512 entries</td>
</tr>
<tr>
<td></td>
<td>Reg copy latency (recovery): 64 cycles</td>
</tr>
</tbody>
</table>

#### 4.8.2 Genetic Evolution Setup

To remove weak instructions from baseline look-ahead binary (based on the strength of dependences) we implement and modify range of tools and automation software. Our complete framework is depicted in Figure 4.8. Core of simulation framework is an automated supervisor program which models an extensive genetic algorithm using C/C++. This supervisor takes look-ahead binary as input and extracts modification units and combines them in group also known as initial candidate genes, for removal from the look-ahead binary. These initial genes are removed one after another and their impact on the overall system’s speed is measured using sampling based out-of-order microarchitectural simulator.

Various components of genetic algorithm such as crossover, mutations and parent selection are also modeled. We have also modeled and tried various crossovers techniques (i.e. single-point, multi-point, uniform, fitness based, xor based etc.) and mutations (fixed, variable). We also mixed the user based heuristics such as orthogonal and superposition combinations for reproductions. Finally, genetic supervisor collects the initial genes and their fitness scores. Good genes are combined by genetic supervisor using various crossovers and mutation operations through genetic evolution process.
4.8.3 Phase Based Simulations

In order to speedup the microarchitectural simulation there are numerous proposals which try to speedup the simulation time by doing temporal sampling and skipping large regions of codes after initial stages [104–106]. This temporally uniform behavior is exploited by various proposals and most of them claim to achieve about 10 to 1000x simulation speedup with less than single digit % of inaccuracy in range of metrics. While the above methodology works well for uniform programs it suffers for irregular codes.

Another more realistic and accurate methodology makes use of phase behaviors and speeds up the overall simulation by only skipping the phases which have been recorded in the past with reasonable stability [100, 107, 108]. General purpose programs exhibit phases in which the characteristics (i.e. branch mispredictions, cache misses etc.) change quite drastically. However, within the same phase certain attributes remain similar. We develop our simulation methodology based on EXPERT [100] which avoids simulating the repetitive code sections that demonstrate stable behavior. In this system, simulation time reduces from 100s of hours to few hours whereas error incurred is about 1% or less for range of metrics.
4.9 Overall Performance Impact

Using our experimental setup (described in Section 4.8), we first show the ultimate performance benefit of weak dependence removal in Section 4.9. We then offer some in-depth discussion in Section 4.10 and compare our proposal with other similar designs in Section 4.11.

Recall that out of all the benchmarks, 14 showed a bottleneck in the speed of the look-ahead thread (Section 2.5). For these applications, we compare two systems: baseline decoupled look-ahead and one that adds a step of offline self-tuning (with a different training input) to remove weak dependences. Their speedups over a single-threaded execution are shown in Figure 4.9.

Offline self-tuning provided a speedup from 1.02x to 1.48x with a geometric mean of 1.14x (or a harmonic mean of 1.13x). The detailed IPC results for all 14 applications are shown in Table 4.2. Note that the performance advantage is obtained by executing fewer instructions in the look-ahead thread – even after accounting for additional reboots which have been faithfully modeled. Thus this performance gains comes with energy savings. Our simulations show that total energy is reduced by about 11% from the baseline decoupled look-ahead system. This is mostly due to reduced execution time and less activity (the look-ahead thread retires 9.9% fewer instructions now). Due to higher ILP, the power of self-tuned look-ahead system is slightly higher compared to baseline look-ahead. Baseline look-ahead total power over single-thread is 1.53x whereas self-tuned look-ahead power over single-thread is 1.55x.

![Figure 4.9: Speedup of baseline look-ahead and genetic algorithm-based self-tuned look-ahead over single-core baseline architecture. The vertical axis is log scale.](image-url)
Table 4.2: IPC of single-threaded baseline, baseline decoupled look-ahead and genetic algorithm based self-tuned look-ahead.

<table>
<thead>
<tr>
<th>Application</th>
<th>Single-thread baseline</th>
<th>Baseline look-ahead</th>
<th>Self tuned look-ahead</th>
</tr>
</thead>
<tbody>
<tr>
<td>craf</td>
<td>2.40</td>
<td>2.47</td>
<td>2.55</td>
</tr>
<tr>
<td>eon</td>
<td>2.62</td>
<td>2.76</td>
<td>3.07</td>
</tr>
<tr>
<td>gap</td>
<td>2.19</td>
<td>2.78</td>
<td>3.21</td>
</tr>
<tr>
<td>gzip</td>
<td>2.14</td>
<td>2.37</td>
<td>2.46</td>
</tr>
<tr>
<td>mcf</td>
<td>0.80</td>
<td>1.26</td>
<td>1.62</td>
</tr>
<tr>
<td>pbm</td>
<td>1.17</td>
<td>1.45</td>
<td>1.45</td>
</tr>
<tr>
<td>two</td>
<td>0.79</td>
<td>1.06</td>
<td>1.06</td>
</tr>
<tr>
<td>vrtx</td>
<td>1.67</td>
<td>2.04</td>
<td>2.04</td>
</tr>
<tr>
<td>vpr</td>
<td>1.64</td>
<td>2.28</td>
<td>2.28</td>
</tr>
<tr>
<td>amp</td>
<td>1.17</td>
<td>1.87</td>
<td>1.87</td>
</tr>
<tr>
<td>art</td>
<td>0.47</td>
<td>2.79</td>
<td>3.34</td>
</tr>
<tr>
<td>eqk</td>
<td>1.43</td>
<td>2.73</td>
<td>2.99</td>
</tr>
<tr>
<td>fma3</td>
<td>1.46</td>
<td>2.39</td>
<td>2.39</td>
</tr>
<tr>
<td>lucas</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Compared to the single-threaded conventional execution, decoupled look-ahead achieves a speedup of 1.39x. With our proposal’s 1.14x (multiplicative) benefit, the total speedup improves to 1.58x. Such a speedup is obtained using a fully automated system without any programmer effort, little additional hardware support, and over all applications not already saturating the pipeline. It is becoming competitive with the speedup achieved using explicit parallelization and certainly more significant than what can be achieved via moderate frequency increases.

4.9.1 Self Tuning in Modern Benchmarks (SPEC 2006)

We also test the speedup of our self-tuning proposal on SPEC CPU2006 applications. The overall results are not very different from SPEC 2000 applications. As shown in the Figure 4.10, on an average we speedup the decoupled look-ahead architecture by 1.11x compared to baseline decoupled look-ahead. For a fixed evolution duration, SPEC 2006 show relatively lower speedups compared to SPEC 2000 counterparts due to larger and more complex codes. We
analyze the impact of code size on overall evolution speed and performance resulting from it in Section 4.10 to shed more light.

4.9.2 Synergy with Speculative Parallelization

Garg et al. [9] have shown that thread-level speculation can be used to significantly improve the speed of the look-ahead thread. Our genetic algorithm-based approach is a different strategy that improves the instruction level efficiency and reduces the lengths of critical paths. Note that unlike in [9], our technique to accelerate look-ahead does not require additional hardware support. We contrast the two approaches in Figure 4.11 and also show a very naive combination of the two approaches: We take the speculatively parallelized look-ahead thread from [9], and simply remove the instructions that genetic algorithm found as weak (in the regular, non-speculatively parallel version), even if they disrupt the speculative parallelization.

We see that the two approaches sometimes have different effectiveness and even when naively combined sometimes can further improve overall system speed. This naive combination achieves another 1.06x speedup on top of speculatively parallel look-ahead, suggesting potential for better integration of the two approaches.

Figure 4.11: Speedup of speculative parallel look-ahead before and after applying GA based weak dependence removal mechanism. We also show the stand alone speedup of GA based baseline look-ahead.
4.9.3 Recap

Overall, these results show that weak dependence removal is a technique with relatively significant payoff. The speedup reported here are what we achieved using our best educated guesses for genetic algorithm parameters. Nonetheless, the speedup achieved by self-tuning are significant for a range of applications from SPEC benchmarks. As another contributing piece, this technique further improves the energy efficiency of decoupled look-ahead and makes it a compelling performance boosting mechanism to add to a general-purpose multi-core architecture.

4.10 Diagnostic Analysis

As a concept, it is almost trivial that we can exploit weakness of dependence in a correctness-insensitive environment such as look-ahead. What is challenging is quantifying it precisely so that we can maximize the gain. The following analysis attempts to shed light on various aspects of the operation: how well it achieves the trade-off between speed and accuracy of look-ahead (Section 4.10.1), how long it takes to find good solutions (Section 4.10.2), and the robustness of the solutions and the entire system (Section 4.10.3). Finally, we will discuss the strengths and weakness of using a metaheuristic approach relative to relying on conventional analysis and heuristics (Section 4.10.4).

4.10.1 Tradeoff between speed and accuracy

In Table 4.3, we show the detailed instruction counts for the look-ahead thread. For these applications, the baseline look-ahead thread is about 65% that of the main thread. After weak dependence removal, the look-ahead thread is about 90% of its original size. This relatively significant dynamic size reduction is a result of removing between 12 and 56 static instructions.

The removal of these instructions will make the look-ahead thread less accurate as a predictor for future behavior of the main thread. It causes the look-ahead thread to veer off the correct program control flow more often than before and increases the number of reboot events. Table 4.4 summarizes the number of reboots in both the baseline decoupled look-ahead system and our new proposal. For brevity, we only show the range and average of the rates. We separate
Table 4.3: Dynamic size of baseline and GA based self-tuned look-ahead thread relative to the original program binary and static instruction count of the original program and those removed by the framework (last row). A - Baseline look-ahead skeleton (%dyn), B - GA tuned look-ahead skeleton (%dyn), C - Total program instructions (static), D - Instructions in 100m window (static), E - Individual weak instructions (static), E - Instructions removed using GA (static).

<table>
<thead>
<tr>
<th></th>
<th>craf</th>
<th>con</th>
<th>gap</th>
<th>gzip</th>
<th>mcf</th>
<th>phmk</th>
<th>twolf</th>
<th>vrtx</th>
<th>vpr</th>
<th>ammp</th>
<th>art</th>
<th>eqk</th>
<th>fma3</th>
<th>luc</th>
<th>Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</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>B</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>C</td>
<td>57568</td>
<td>79730</td>
<td>74650</td>
<td>23205</td>
<td>120529</td>
<td>53936</td>
<td>95121</td>
<td>42089</td>
<td>43154</td>
<td>25588</td>
<td>25639</td>
<td>103235</td>
<td>72299</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</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>958</td>
<td>582</td>
<td>1041</td>
<td>3098</td>
<td>319</td>
<td>4034</td>
<td></td>
</tr>
<tr>
<td>E</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>F</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>

Table 4.4: Reboot rate for baseline and the self-tuned decoupled look-ahead systems. The rates are measured by the number of reboots per 100,000 committed insts in the main thread.

the integer and floating-point results since they are rather different. On an average, 3-4 extra reboots are encountered for every 100,000 instructions. This is still an acceptably low rate.

Another sign of reduced accuracy of the look-ahead thread is the change in the coverage of L2 misses averted. In the baseline look-ahead, 90% of L2 misses are averted in the main thread. The rate actually improved slightly to 90.4%. This suggests that whatever imprecision in address calculation in the new look-ahead thread is more than offset by the speed increase. As a result, the L2 prefetch capability is maintained even under a more challenging environment.

4.10.2 Speed of search

Genetic algorithm convergence:

To see how quickly the genetic algorithm can find a good solution, we track the chromosomes generated through seven generations. Figure 4.12 shows this change. To show results from different applications together, we normalize the performance gain. The solutions are represented as cycles saved and normalized to that of the best solution for that application. A few applications achieved very small performance gain (<5%), making this normalization potentially misleading. They are therefore excluded from this figure.
Figure 4.12: Normalized gains achieved through generations of evolution.

From the Figure 4.12, we can see that the convergence is fairly quick: after 2 generations, all applications achieved more than half of the benefit; after 5 generations, at least 90% of benefits are achieved. Nevertheless, we can not ascertain whether these optima are local. We have observed 12 generations and so far have seen no application breaking away from the best result all found within 7 generations.\(^5\)

**Profile time:**

The amount of profiling time it would take in a real system for the complete genetic evolution process can be broken down into the control software overhead and the time it takes to executed certain amount of code in order to measure execution speed. Recall that the control software manipulates chromosomes through crossover and mutations, and performs bookkeeping on the fitness scores. Depending on the number of genes, the software executes between 4.8 (\textit{lucas}) to 43.2 million (\textit{perlbmk}) instructions with an average of 16.8 million instructions. This overhead is on the orders of millisecond for the entire evolution stage and is clearly negligible.

For offline profiling (with sampling and multi-gene tests), the complete prescreening (for an average of 862 genes) and genetic evolution (7 generations with population size 96 per

---

\(^5\)In hindsight, we found that ever-increasing mutation rate is counter productive and can degrade the quality of solutions or stagnate the evolution process itself. This could be one possible explanation of marginal improvement beyond generation 7.
<table>
<thead>
<tr>
<th>Program</th>
<th>Skeleton</th>
<th>Total</th>
<th>Dominant</th>
<th>Profile time (in min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT</td>
<td>130</td>
<td>77</td>
<td>7473</td>
<td>2591</td>
</tr>
<tr>
<td>FP</td>
<td>251</td>
<td>52</td>
<td>3393</td>
<td>1121</td>
</tr>
</tbody>
</table>

Table 4.5: Program execution and profile time for offline GA based system.

generation) process involves profiling runs between 16.9 billion (eon) and 61.5 billion (mcf) instructions with an average of 32.1 billion instructions. On the target machine, this translates to roughly 2 to 20 seconds.

The compile time actions include testings many iterations of the same code using a training input under different genes and chromosomes. With sampling as described in the paper, this process does not need to run for a very long time. According to our statistics, we need to measure between 52 and 2694 billions of instructions for the tested 14 applications, which roughly takes between 7 sec and 9.5 minutes. The runtime overhead is zero. Findings are listed in Table 4.5. On an average the typical time taken for profiling is about few minutes for complete run using ref inputs. Gene evaluation step is most time consuming in the whole profiling process and the time taken by genetic algorithm generations is about few minutes.

**Online self-tuning evolution:**

For an online self-tuning system, the overhead mainly comes from testing bad skeleton configurations that actually slow down the system. However, bad configurations are quickly discarded and therefore do no have a lasting impact. Figure 4.13 shows this effect visually with the cumulative speed (measured by IPC) of equake and art running in a single-threaded version, in the baseline decoupled look-ahead system, and in an online self-tuning system.

In the very early stage for equake, the online version is noticeably slower than the baseline decoupled look-ahead system, but the gap narrowed afterward and within 1.8 billion instructions the online version broke even and maintained the lead thereafter. By the end of 3.6 billion instructions, its overall cumulative speed was already 11% faster than the baseline decoupled look-ahead system. Our online evolution experiment is performed with brute-force simulation and is thus excruciatingly slow. An online version will also require runtime system support for persistent storage of partial evolution results.
In contrast, for *art* (shown in Figure 4.13-b) the break-even comes rather late. In the beginning self-tuned look-ahead system is almost twice as slow as baseline decoupled look-ahead. This phase represents initial exploration dominated by bad genes/chromosomes testing. By 24 billion instructions, online version is faster by 5% compared to baseline decoupled look-ahead. From the figure, it is evident that online version speedup improves quite a bit to compensate for the early slowdown incurred in the initial phases.

**Impact of code size on evolution:**

In this section, we quantify the impact of static code size on the overall evolution process and the speedup achieved in a given number of generations. In particular, we study the relationship between code size with overall speedup and the number of generations by which the bulk of speedup was achieved.
First, we study the impact of code size over relative performance gain. Relative performance gain (in %) is defined as a ratio of difference of self-tuned look-ahead IPC and baseline decoupled look-ahead IPC over the difference of ideal branch and cache system IPC and baseline look-ahead IPC. From Figure 4.14-(a), it is clear that SPEC 2000 applications have 2x smaller code size (on an average) compared to SPEC 2006 applications. As a result, relative performance gain for SPEC 2006 is lower compared to SPEC 2000. A correlation coefficient of -0.46 indicates that there is moderate negative relationship between the code size and relative performance gain.

Second, we study the correlation between saturation generation and code size. A saturation
Figure 4.15: Speedup of various configurations: baseline (BL) baseline decoupled look-ahead (BL-DLA), and genetic algorithm tuned decoupled look-ahead (GA-DLA) for various L2 cache sizes. All results are geometric means of speedup relative to single-threaded execution on a 1MB L2 system.

A generation is a generation by which 90% of the final speedup (achieved in 12 generations) is achieved. From Figure 4.14-(b), we can see that the applications with smaller code size saturate their performance rather early. A correlation of 0.56 indicates moderate positive relationship between the code size and saturation generation. In general, we observe larger static code size for SPEC 2006 applications compared to SPEC 2000 codes. Because in our evolution process we stop the evolution at a fix generation, we see relatively smaller speedups for SPEC 2006 codes. Given enough number of generation, we should be able to achieve higher performance for SPEC 2006 application than reported here.

4.10.3 Robustness

L2 cache sensitivity:

Given that L2 misses are significantly reduced, it is natural to assume that the system’s overall effect is highly sensitive to the L2 cache size. We carried out a sensitivity study where the genetic evolution was performed on our original platform (with 1MB L2 cache) but the testing of the final product was carried out with 2MB and 4MB L2 caches. The results are shown in the Figure 4.15. For brevity, we only show the geometric mean.

From the figure, we can see that having larger L2 caches simply raises the bar on all configurations. In fact, the speedup of our proposal remains surprisingly stable: it is 1.139x, 1.133x,
Table 4.6: Average number of modules, their dynamic instance size (in 1000 instructions) and total number of genes per module in SPEC 2000 benchmarks for complete run using ref inputs.

<table>
<thead>
<tr>
<th></th>
<th>Total modules</th>
<th>Module size (K)</th>
<th>Genes/module</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Max</td>
<td>Avg</td>
</tr>
<tr>
<td>INT</td>
<td>32</td>
<td>962.3</td>
<td>453.8</td>
</tr>
<tr>
<td>FP</td>
<td>18</td>
<td>921.9</td>
<td>538.7</td>
</tr>
</tbody>
</table>

and 1.131x for 1, 2, and 4MB configurations respectively. In other words, the difference is less than 1% if the L2 size is quadrupled. This experiment also clearly shows that solutions evolved from a similar but not identical system are still useful. That observation suggests opportunities to further amortize tuning costs given support at the ecosystem level.

**Simplified fitness test:**

We use two simplifications to accelerate fitness tests: sampling and multi-gene tests. In theory, these simplifications can introduce errors and potentially mislead the genetic algorithm into suboptimal solutions. To quantify the impact of these fitness test simplifications, we contrast two systems that only differ in their fitness tests: one uses complete profiling runs of 500 million instructions apiece (and costing us a tremendous amount of simulation cycles) the other with sampling based measurement and multi-gene testings discussed in Section 4.6.2. Specifically, we sample module instances at a rate of 1 per 30. For each application, we take the evolved result from each generation; measure its performance gain (cycles saved); and normalize it to that of the final (generation 7) result found in the full profiling. In Figure 4.16, we show two representative applications.

The result clearly shows that even though the evolution may go through different paths due to different fitness scores and thus assigned probabilities, the two evolutionary paths are essentially lock-stepped in its overall progress, suggesting that accelerated fitness tests make virtually no difference. The final best solutions are also almost identical in terms of their gene composition. Additionally, we found that the average of the population’s fitness also rises in similar ways in both runs. This result also shows that even though randomization is used in the evolution process, with a sufficient population size, the outcome is not chaotic.
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Fitness(%)</th>
<th>Freq(%)</th>
<th>Instruction</th>
<th>Fitness(%)</th>
<th>Freq(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>s8addq t4,v0,v0</td>
<td>0.12</td>
<td>0.03</td>
<td>ldq t3,16(t1)</td>
<td>0.71</td>
<td>0.31</td>
</tr>
<tr>
<td>ldq t1,-11616(t9)</td>
<td>0.12</td>
<td>0.04</td>
<td>stl at,0(a3)</td>
<td>0.57</td>
<td>0.03</td>
</tr>
<tr>
<td>ldq t12,-11608(t9)</td>
<td>0.12</td>
<td>0.04</td>
<td>xor t2,t4,t2</td>
<td>0.56</td>
<td>0.18</td>
</tr>
<tr>
<td>ld1 a2,252(sp)</td>
<td>0.10</td>
<td>0.04</td>
<td>lda t1,10009(a2)</td>
<td>0.41</td>
<td>0.09</td>
</tr>
<tr>
<td>ld1 a0,96(sp)</td>
<td>0.09</td>
<td>0.03</td>
<td>addq a2,a1,t12</td>
<td>0.41</td>
<td>0.02</td>
</tr>
<tr>
<td>stt f16,56(sp)</td>
<td>2.07</td>
<td>0.05</td>
<td>ldq t3,-25768(gp)</td>
<td>8.87</td>
<td>0.51</td>
</tr>
<tr>
<td>ldq f14,56(sp)</td>
<td>1.66</td>
<td>0.05</td>
<td>s4addq s1,a2,s3</td>
<td>6.27</td>
<td>0.51</td>
</tr>
<tr>
<td>stl t5,60(sp)</td>
<td>1.46</td>
<td>0.05</td>
<td>ldl s1,0(a1)</td>
<td>6.18</td>
<td>0.51</td>
</tr>
<tr>
<td>mult f13,f10,f13</td>
<td>0.13</td>
<td>0.06</td>
<td>stl zero,0(s3)</td>
<td>5.87</td>
<td>0.51</td>
</tr>
<tr>
<td>ldq s1,56(sp)</td>
<td>0.10</td>
<td>0.04</td>
<td>ldl s4,0(t0)</td>
<td>5.04</td>
<td>0.51</td>
</tr>
<tr>
<td>addl v0,a3,v0</td>
<td>0.40</td>
<td>0.09</td>
<td>divt f21,f22,f22</td>
<td>0.48</td>
<td>0.03</td>
</tr>
<tr>
<td>zapnot v0,0xf,v0</td>
<td>0.19</td>
<td>0.09</td>
<td>fneg f28,f28</td>
<td>0.30</td>
<td>0.03</td>
</tr>
<tr>
<td>srl v0,0x10,v0</td>
<td>0.19</td>
<td>0.09</td>
<td>mult f22,f24,f28</td>
<td>0.30</td>
<td>0.03</td>
</tr>
<tr>
<td>zapnot v0,0xf,v0</td>
<td>0.19</td>
<td>0.09</td>
<td>addt f22,f18,f22</td>
<td>0.26</td>
<td>0.03</td>
</tr>
<tr>
<td>srl v0,0x10,v0</td>
<td>0.19</td>
<td>0.09</td>
<td>ldl t8,280(sp)</td>
<td>0.25</td>
<td>0.27</td>
</tr>
<tr>
<td>addq t2,v0,t2</td>
<td>0.59</td>
<td>0.22</td>
<td>ldq s5,-72(s0)</td>
<td>9.42</td>
<td>0.42</td>
</tr>
<tr>
<td>ldq ra,0(sp)</td>
<td>0.58</td>
<td>0.41</td>
<td>ldl zero,128(a3)</td>
<td>9.00</td>
<td>0.42</td>
</tr>
<tr>
<td>zapnot t1,0xf,v0</td>
<td>0.57</td>
<td>0.22</td>
<td>addq s1,t10,s1</td>
<td>6.40</td>
<td>0.42</td>
</tr>
<tr>
<td>zapnot a0,0xf,a0</td>
<td>0.52</td>
<td>0.35</td>
<td>ldq s1,-16(s0)</td>
<td>6.05</td>
<td>0.42</td>
</tr>
<tr>
<td>zapnot t3,0xf,t3</td>
<td>0.46</td>
<td>0.22</td>
<td>cmptrle f10,f11,f10</td>
<td>5.25</td>
<td>0.10</td>
</tr>
<tr>
<td>stq t2,0(a0)</td>
<td>0.89</td>
<td>0.01</td>
<td>ldq t5,8(v0)</td>
<td>8.70</td>
<td>0.17</td>
</tr>
<tr>
<td>ldq t4,48(a3)</td>
<td>0.69</td>
<td>0.03</td>
<td>ldl t5,904(sp)</td>
<td>1.80</td>
<td>0.02</td>
</tr>
<tr>
<td>mov v0,a1</td>
<td>0.68</td>
<td>0.14</td>
<td>ldl t3,704(sp)</td>
<td>0.28</td>
<td>0.07</td>
</tr>
<tr>
<td>cmple t5,t2,t5</td>
<td>0.66</td>
<td>0.01</td>
<td>s4addq a2,t3,t3</td>
<td>0.28</td>
<td>0.07</td>
</tr>
<tr>
<td>lda a1,128(sp)</td>
<td>0.63</td>
<td>0.01</td>
<td>ldl t3,800(sp)</td>
<td>0.23</td>
<td>0.28</td>
</tr>
<tr>
<td>addq a0,a2,a0</td>
<td>6.12</td>
<td>0.06</td>
<td>cmovgt a5,a3,a1</td>
<td>1.41</td>
<td>1.23</td>
</tr>
<tr>
<td>subq a1,0x40,a1</td>
<td>4.70</td>
<td>0.01</td>
<td>or v0,at,at</td>
<td>1.05</td>
<td>1.23</td>
</tr>
<tr>
<td>ldq s2,24(sp)</td>
<td>2.80</td>
<td>0.03</td>
<td>subq a1,a3,a5</td>
<td>0.94</td>
<td>1.23</td>
</tr>
<tr>
<td>stq t3,0(s4)</td>
<td>2.00</td>
<td>0.06</td>
<td>extql a3,a2,a3</td>
<td>0.93</td>
<td>1.23</td>
</tr>
<tr>
<td>cmpult t8,t5,t8</td>
<td>1.66</td>
<td>0.01</td>
<td>sra a4,0x38,a4</td>
<td>0.59</td>
<td>0.17</td>
</tr>
<tr>
<td>cvtlq f0,f0</td>
<td>0.60</td>
<td>0.02</td>
<td>addq t5,t9,ra</td>
<td>3.16</td>
<td>0.26</td>
</tr>
<tr>
<td>mov s4,a1</td>
<td>0.47</td>
<td>0.02</td>
<td>srl t12,t11,s2</td>
<td>2.27</td>
<td>0.26</td>
</tr>
<tr>
<td>cir s4</td>
<td>0.44</td>
<td>0.02</td>
<td>s8addq s0,t12,s0</td>
<td>2.06</td>
<td>0.10</td>
</tr>
<tr>
<td>lda s0,-24296(gp)</td>
<td>0.27</td>
<td>0.02</td>
<td>mulq s0,a0,s0</td>
<td>2.00</td>
<td>0.10</td>
</tr>
<tr>
<td>stl s1,0(a0)</td>
<td>0.26</td>
<td>0.03</td>
<td>ldq t6,160(sp)</td>
<td>1.45</td>
<td>0.10</td>
</tr>
</tbody>
</table>

Table 4.7: Top 5 weak instructions and their respective cycles savings (Fitness) normalized to baseline look-ahead system’s cycles; also showing their dynamic code contributions (Freq) to the program execution of 100 million instructions.
Figure 4.16: Comparison of generational solutions using full-run fitness tests (precise) and sampling based fitness test (sampling) for two representative applications.

4.10.4 Strengths and Weaknesses of Metaheuristics

An alternative to using metaheuristics is a compiler analysis-based approach. Based on our experience, there are significant challenge in such an approach for our task. Although we have summarized some common patterns of weak instructions, they are hard to generalize. Table 4.7 shows the top weak instructions found in each application along with their dynamic code contributions (%dyn) and respective cycle savings (%cyc). Predicting weakness based on the instruction itself is extremely challenging if we want to avoid false positives, which are very costly. In fact, if we represent all weak instructions as a distribution based on static code attributes (opcode and number of operands), the vector shows a correlation coefficient of 0.958 with that representing all non-weak instructions (Figure 4.17). Size of each bubble represents the number of instructions belonging to the specific category. X-axis represents the opcodes sorted in alphabetical order. Just by looking at these two distributions one can infer that by look weak
instructions are not very different from strong instructions and are distributed all over the place. To us, this clearly suggests that weakness is not determined by the static instruction.

Weakness is not additive and removing multiple known weak instructions can slow down the thread as been shown in Figure 4.3. Indeed, 100s of individually weak instructions are found in almost every program. But the best solution from the evolution process removes only 12 to 56 instructions. With these instructions removed, we go back and retest the hundreds of weak instructions and have found very few to be still weak. This suggests that even if conventional analysis is used to identify potential candidates, trial-and-error is still indispensable. In the table 4.8 we show a few examples of weak instructions and their backward slices.

Figure 4.17: Distribution of weak (a) and regular (b) insts by their static attributes (opcode and no. of operands). Radius of circle indicates the relative weight of specific type of instruction.
4.10.5 Exclusive Slicing of Removed Instructions

Like a typical metaheuristic approach, our framework does not care about the meaning of the adjustments such as removing an instruction. As such, simplistic conclusions (e.g., if A is not needed nor is its exclusive backward slice) can only be stumbled upon by chance. To see how much conventional analysis can contribute to the metaheuristic search, we perform a slicing pass on top of the evolution-derived new skeleton.

After genetic evolution process discovers the set of good genes (also known as GA optimized chromosome) we do a final code slicing to further remove the exclusive backward dependence chains of already removed instructions. Because our framework operates at individual instruction level it is unaware of backward dependence chains of already removed instructions. Ideally, if a set of instructions are identified as weak instructions and removed together then their exclusive backward dependence chains should also be removed. Exclusive slicing of removed instructions results into 1% speedup over GA optimized skeleton with a maximum speedup of 3% for lucas. On an average, we removed 5 static instructions and 0.6% of the dynamic code from the GA optimized skeleton as a result of exclusive code slicing. For this specific target problem, it appears that just relying on metaheuristics is enough.
Figure 4.18: Comparison of state-of-art decoupled look-ahead system with previous proposals. The latency to copy register file is set to 64 cycles for DCE. Speedups are normalized to a 4-wide, out-of-order single-thread baseline with a state-of-art L2 stream prefetcher.

4.11 Comparison with Previous Proposals

To better understand the effectiveness of our proposal, we compare it with two existing approaches, one representing a micro helper thread approach [37, 45], the other a decoupled look-ahead approach [6]. In the former case, we follow [37] and only model the upperbound of the resulting system by idealizing the top 10 problematic instructions. This is the effect when the helper threads are always on-time, accurate, and overhead-free. We show the speedup of these configurations as well as our system in Figure 4.18. Since a more realistic implementation of microhelper thread will only achieve part of the potential (57% according to [37]), we use a horizontal line on the bar representing the average of micro helper thread approach to provide a visual reference.

Dual-core execution (DCE) [6] is similar to our proposal in terms of implementation. DCE tries to pre-execute the whole program in the front processor and passes the outcome to the back processor. Pre-executing the program achieves cache warmup and prefetching effect. We faithfully model the DCE system in our simulator. Because of fidelity improvement of simulation infrastructure the baseline system improves for few applications - most notably being swim. Fixing the modeling of prefetch instructions allows sometimes dramatic performance differences (e.g., 3x in swim). Without this change, our version of DCE obtains performance improvements that match their original proposal relatively well.

In Figure 4.18, we are comparing three systems: Left most bars are speculative slice limit speedup over single-thread. Second set of bars are DCE (with a 64-cycle latency for copying
the architectural registers from the back to front processor) speedup over single-thread and final set of bars are our current self-tuned decoupled look-ahead system. Applications that did not evolved we use the baseline decoupled look-ahead performance. Close inspection of performance gains reveals many insights.

First of all, when we target only handful of instructions then there is not enough potential to exploit. A scheme like speculative slice has spawning and managing overhead of slices on main thread that limits its speedup to about 10%. Second, DCE performs relatively well for the floating point applications compared to integer programs. This is due to good prefetching capability of DCE and floating point applications benefit more from prefetching. However, in integer applications it performs rather poorly and set of integer applications reported in the original proposal achieve only 9% speedup over single-thread. Our reproduction of their system achieves nearly the same performance in integer programs. In comparison, our optimized decoupled look-ahead achieves 1.44x (geomean) speedup across all 25 applications.

Compared to the micro helper thread approach, our design is not limited by a small number of targets. In many applications, especially newer codes, “problematic” instructions are numerous and widely spread out. Compared to dual-core execution (or other similar approaches [5, 42, 44]), our approach is much more effective in allowing the look-ahead thread to run faster and stay ahead of the main thread to be useful.

Program demultiplexing (PD) [34] tries to execute the program subroutines ahead of time in speculative manner on auxiliary processors to extract the implicit parallelism from sequential programs. PD was found to be applicable over a subset of the applications whereas our technique was tested over all applications that didn’t already saturate the pipeline. Among those subset presented in the paper on which PD worked well, using two cores, (1 auxiliary core), PD achieves a geometric mean speedup of 1.37x for those apps, our (decoupled look-ahead with genetic evolution) achieves 1.43x speedup (geometric mean).

To summarize, the current decoupled look-ahead system achieves significant performance compared to previous proposals across diverse set of applications that range from control-intensive integer codes to large loop-based floating point applications.
4.12 Future Work

In this section, we present a brief summary of future exploration work in weak dependences and genetic algorithm based self-tuning look-ahead design. We also discuss the potential use of such a framework in refining various heuristics and demonstrate one case of value prediction refinement in the context of look-ahead thread.

4.12.1 Heuristic Refinements

Genetic algorithm based framework potentially can be used to refine some of the general heuristics as well. As a proof-of-concept, we demonstrate the use of GA based framework to refine zero-value substitution (ZVS) heuristics which are used in the baseline decoupled look-ahead system. In the baseline look-ahead, whenever look-ahead thread stalls on L2 cache miss and the look-ahead distance reduces significantly, we perform a ZVS to the load in the look-ahead thread. Intuition behind doing zero substitution is the simplicity of the logic and also the fact that zero is most common value seen in the system. We tried two baseline ZVS strategies: ZVS for all L2 miss (\textit{AllLoadZVS}) and ZVS for non prefetching L2 load misses (\textit{NonprefetchLoadZVS}).

In this study, we only consider the applications which preform at least 10,000 ZVS during the whole execution. While \textit{AllLoadZVS} beats \textit{NonprefetchLoadZVS} it looses out significantly in gap. Our \textit{GAbasedZVS} outperforms both heuristics in all applications as shown in Figure 4.19. For the 11 applications, \textit{GAbasedZVS} achieves 1.84x speedup over single thread baseline, whereas the speedup of \textit{NonprefetchLoadZVS} and \textit{AllLoadZVS} is 1.66x and 1.77x respectively. If we just look at the integer programs \textit{GAbasedZVS} achieves 1.39x speedup over the single-thread baseline compared to \textit{NonprefetchLoadZVS} and \textit{AllLoadZVS} speedup of 1.32x and 1.35x. This suggests that GA based framework can be effectively used to refine sub-optimal baseline heuristics. As future exploration, we plan to refine various prefetching and other heuristics which have been employed in the baseline decoupled look-ahead system.
4.12.2 Efficient Payload using Genetic Algorithm

In this chapter, we explored only removal of weak instructions. Similarly, the instructions which were not part of the baseline skeleton, due to static compiler analysis, can play a role in altering the microarchitectural states by prefetching. In fact, when we tried them with our GA based framework in isolation they showed significant speedup over the baseline decoupled look-ahead. Single static payload instruction in mcf showed 6% speedup. As we remove weaker instructions, it creates room for adding more meaningful payload in the look-ahead thread which helps the main thread. As an ongoing work, we will explore the possibility of adding efficient payload to the look-ahead thread along with removing weak dependences.

4.12.3 Merging of Good Superposition Regions

From the superposition chromosomes’ fitness, shown in Figure 4.3, it is clear that there are regions in which superposition based combining works well. This can be known by inspecting the fitness scores as they keep increasing. After 50 or so genes there comes a bad gene which degrades the performance significantly. In static time, we can identify such genes and mask them out from the superposition chromosomes. This exploration can help us to create better heuristic based genes. In other words, we identify the constructive merging regions and destructive
genes along the line. In a static heuristic, all the constructive regions can be merged together and destructive genes can be eliminated. This would help evolution process to converge to the optimal solution in fewer generations.

### 4.12.4 Dynamically Weak and Skippable Loads

According to our study, similar to data-triggered threads [82, 83], we find that there are many loads which are redundant. A load which gets the same value from the same address as in previous instance is redundant in their terminology but can not be completely useless from execution point of view. For example, read only data always loads the same value and according to DTT terminology they can be called redundant but certainly they are not useless. Also, we might need to reload the data because the value from previous instance got corrupted due to intermediate computations. In general, only a subset of redundant loads are useless or skippable and we call them *dynamically weak loads* which can be skipped in our system.

Redundant loads are about 70%, as shown in Figure 4.20, and we need to first figure out how much of those are actually dynamically weak loads. Once we know then we can skip those loads in the look-ahead thread in order to make it faster. We are exploring the possibility to identify such loads using classification and prediction techniques in runtime.
4.13 Summary

Weak dependences interact in a non-linear fashion that makes it hard, if not impossible, to correctly judge whether a particular weak dependence should be removed simply based on the its local properties in isolation. Once identified by our proposed framework, the removed instructions often appear to fit the bill of weak instructions, but their weakness is a result of context and these weak instructions interact in a non-linear fashion that makes it even harder to generalize in conventional ad-hoc heuristics. So far, we have failed to form heuristics-based analysis that can easily identify them with any satisfactory consistency. We have observed that even one false positive in labeling a weak instruction often causes significant overall slowdown.

In this chapter, we have presented a metaheuristic based approach that incrementally removes weak dependences from the look-ahead thread. Our study has shown that there are plenty of instructions that contribute marginally to the purpose of look-ahead, which traditional, heuristics-based analysis can not easily identify with any satisfactory consistency. Our proposed algorithm provides a straightforward framework to empirically explore the space of interacting opportunities. The result of this search is encouraging. Dozens of weak instructions are removed from thousands of static instructions, resulting in speedup as much as 1.48x and a geometric mean of 1.14x, while reducing the energy consumption by 11% of the overall system. Our study also shows that the mechanism is rather robust in that (a) the performance boost is virtually the same across different L2 sizes; and (b) the system is rather tolerant to measurement noise due to sampling and approximations in fitness tests. An online evolution system incurs around 17 million instructions of control software overhead for the entire period of evolution.

With the help of the evolution framework, the speedup of a decoupled look-ahead system increases from 1.39x to 1.58x (geometric mean) using two cores compared to the baseline single-threaded execution. We estimate the time for evolution to be on the orders of seconds to minutes, and during the process the extra control overhead is negligible. In our opinion, this technique further strengthens decoupled look-ahead as at least a compelling mechanism, at least for turbo boosting. Moreover, fast and deep look-ahead will be an essential element in uncovering more of the significant implicit parallelism in general-purpose codes.
Do-It-Yourself (DIY) Branches and Skeleton Tuning

The look-ahead skeleton (even after removing weak dependences) still has all branches as default members. However, this is an arbitrary design choice to avoid the branch misalignment among look-ahead and main thread which simplified the branch outcome passing. Skeletons with all branches (and their backward dependence chain) are inherently slow (due to large size), and therefore are unable to perform efficient deep look-ahead. Further distillation of skeleton without compromising the quality of look-ahead has potential to speed up the overall system.

From our explorations, so far, we observed that fixed set of heuristics to generate skeleton are often suboptimal across various program phases. A suboptimal skeleton leads to an imbalanced look-ahead thread and as a result either the look-ahead agent continues to be the speed limit or is incapable of targeting sufficient bottlenecks in timely manner.

In this chapter, we propose two orthogonal techniques to balance look-ahead skeleton, thereby improving overall performance. First, we use a static, profile-driven technique to tune skeleton for various code regions. Second, we accelerate sluggish look-ahead by skipping branch based, non-critical, side-effect free code modules that do not contribute to the effectiveness of look-ahead. We call them Do-It-Yourself or DIY branches for which the look-ahead thread does not provide any assistance to the main thread and main thread relies on its own
branch predictor and data prefetcher. As a result, look-ahead thread propels ahead and assists in performance-critical code regions to improve the overall performance of decoupled look-ahead system. Alternatively, distilled skeleton also enables simplification of look-ahead core pipeline to reduce power and energy overhead with virtually no degradation in performance.

We present the motivation for skeleton tuning and DIY branches in Section 5.1. With the help of simple experiments, we present the evidence of significant potential in Section 5.2. We discuss architectural support needed and our framework in Sections 5.3 and 5.4, respectively. Experimental setup is described in Section 5.5. We present overall performance results and additional analysis of the system in Sections 5.6 and 5.7. Finally, we summarize our insights in Section 5.8.

5.1 Motivation

Not all branches in a program are equally critical and “hard-to-predict”. To maintain a good look-ahead distance, “easy-to-predict” branches can be delegated to the main thread. Main thread executes these branches (by its own) and does not receive any hints from the look-ahead thread. Execution resources saved through skipping of the DIY branches can be spent more meaningfully to execute performance-critical code regions.

Look-ahead thread and various look-ahead tasks (along with associated housekeeping) are analogous to a locomotive engine and cargo loads it carries. A heavier look-ahead load often makes look-ahead thread sluggish, whereas a lightweight look-ahead load under utilizes the look-ahead thread’s capabilities. Therefore, the key to an efficient look-ahead design is to have a flexible mechanism which can alter the look-ahead skeleton on-the-fly depending upon how much load look-ahead engine can pull through without slowing it down. In current incarnation, the decoupled look-ahead is entirely limited by the slow look-ahead thread and speeding it up will further speed up the entire system.
static void mark_modified_reg (dest, x)
rtx dest; rtx x;
{
   int regno, i;
   if (GET_CODE (dest) == SUBREG) dest = SUBREG_REG (dest);
   if (GET_CODE (dest) == MEM) modified_mem = 1;
   if (GET_CODE (dest) != REG) return;

   regno = REGNO (dest);
   if (regno >= FIRST_PSEUDO_REGISTER) modified_regs[regno] = 1;
   else
      for (i = 0; i < HARD_REGNO_NREGS (regno, GET_MODE (dest)); i++)
         modified_regs[regno + i] = 1;
}

Figure 5.1: A DIY branch based code example from 176.gcc.

5.1.1 Idea of Do-It-Yourself Branches

Naturally the benefits of looking ahead are maximized when look-ahead agent consistently runs ahead and maintains a sustainable distance from the main thread. Therefore, an intelligent look-ahead design must be selective in targeting the bottlenecks. We observed that not all branches in a program are equally critical and certain code modules do not contribute to the effectiveness of look-ahead. These frequent code modules are devoid of “hard-to-predict” data-dependent branches and cache misses. Case in point, mark_modified_reg() function in 176.gcc (shown in Figure 5.1): gets invoked after execution of every instruction of input program and constantly updates a small array (modified_regs[] in the last line) that hardly misses in the cache. If skipped completely in the skeleton then saves about 3% of the skeleton instructions and improves the decoupled look-ahead performance by 10%!

Similarly, all iterations of a loop may not be needed for prefetching, if memory accesses from consecutive iterations happen to the same cache line. In other words, executing only 1 out of $N$ iterations ($N$ is a loop access-pattern property) can accomplish perfect prefetching for the loop. A wide variety of library calls including printf, _OtsMove\(^1\), _OtsFill and reduction operations are often useless for look-ahead purpose as well. Combined these together constitute a significant fraction of skeleton that can be avoided without diluting the quality of the look-ahead performance.

\(^1\)Language support routine for character operations.
ahead. We observed numerous instances of such branch based code modules which we call “Do-It-Yourself” modules or DIY branches.\(^2\)

DIY branches can be thought as a “contract” between the look-ahead and the main thread which allows look-ahead thread to skip the easy-to-predict, side-effect free code modules. As a result, main thread does not receive any branch prediction and cache prefetching hints for such modules. Later, from a convenient point in the code, main thread starts receiving the look-ahead hints again. This allows the look-ahead thread to skip over the less critical code in order to ensure and maintain a consistence look-ahead distance.

5.1.2 Customized Skeletons

Skeleton tuning can be further extended and applied to any arbitrary code regions as opposed to just DIY branches and modules. We hypothesize that various programs and code regions within the same program have different “critical” bottlenecks, therefore, will benefit more from customized skeleton. To validate our hypothesis, we tested the performance of SPEC applications with different kind of skeletons. We created these skeletons by either including or excluding various types of cache and software prefetches (details in the Section 5.3.4). We also manipulated skeleton size by transforming biased branches\(^3\) to be unconditional “taken” or “not taken” – based on dominant direction. Combinations of prefetches and biased branches provide sufficient flexibility to create a wide gamut of skeletons – ranging from very fine (good for accuracy) to coarse (good for speed) grain.

Relative performance of decoupled look-ahead system for various customized skeletons reveals following insights:

- **INT applications favor biased branches to turn unconditional**: INT codes have significant number of data-dependent branches (depend on a chain of store-load pair [68]) and unmodified inclusion increases the skeleton size. Turning a few of these branches into unconditional branches reduces skeleton size significantly at a minimal price of “occasional” recoveries.

\(^2\)In this chapter, we use DIY modules and DIY branches interchangeably.

\(^3\)Conditional branches with $>99.5\%$ “taken” or “not taken” outcome.
• **FP applications favor biased branches as is:** FP codes tend to have relatively smaller skeletons due to simpler loop branches with shorter index computation chain. This provides enough room even for biased branches to keep all their backward dependences and avoid incurring one “extra” recovery on loop termination.

• **L2 prefetches should be prioritize over L1:** Applications with significant L2 misses (e.g., mcf, art, gap) are better off by targeting L2 misses first and should exclude L1 prefetches from the skeleton if the skeleton size hinders the speed of the look-ahead thread. Because individual L2 miss has greater performance impact, it should be the primary target in skeleton tuning.

• **L1 prefetch inclusion in skeleton:** L1 prefetches should only be included in the skeleton if after targeting L2 misses we still have a small skeleton (half of original program). Applications that do not have significant L2 misses (e.g., gzip, applu, crafty) should target L1 misses to reap the additional benefits of look-ahead.

Customizing skeletons at program level (best overall per program) achieves 5.6% performance gain over baseline decoupled look-ahead. “One-skeleton-suits-all” is hardly the case which motivates us to find suitable skeletons for individual programs and code regions. To recap, look-ahead acts similar to a locomotive engine which pulls the main thread by solving its bottlenecks, therefore customizing skeletons for speed and accuracy is crucial for the overall performance.

### 5.2 Performance Potentials

Before we describe our methodology and framework to systematically exploit DIY branches in the look-ahead thread, we further motivate by presenting a rough estimate of performance potentials for respective techniques. Our experimental setup is described in Section 5.5 but for now assume that we conduct our studies on a 4-wide, out-of-order machine.

We further motivate the need for skeleton tuning by presenting a rough estimate of performance potential. After measuring performance for all kind of skeletons at regular intervals (known as *epoch*) we pick the best performing skeleton from every *epoch* and combine their
individual gains together. On an average, for epoch size of 1000 instructions, we get about 1.21x performance improvement over baseline decoupled look-ahead skeleton (as shown in Figure 5.2). As we increase the epoch size, performance potential goes down a bit but there is still significant potential even at coarse grain epoch size of 1 million instructions.

In this study, we considered a total of 18 types of skeletons. We also experimented to reduce the number of skeleton types by considering only the top 5 best performing skeletons. With top 5 best performing skeletons, the potential goes down to 14% for epoch size of 1,000 instructions which suggests that we have to consider a wide range of skeletons to achieve significant performance improvement.

To gauge the potential of DIY branches, we measure the performance gain of individual DIY branch in isolation and call them DIY seed. If two DIY branches are non overlapping then assuming that their individual performance will be additive is not unrealistic and represents DIY potential. In reality, sum of individual performance gains gives an upper bound of potential. Any DIY seed which has better performance compared to baseline look-ahead is considered as a “good” DIY seed. We list total number of DIY seeds, number of good seeds and potential in Table 5.1. A performance potential of 0.5 means that we can achieve 1.5x speedup in the
Table 5.1: Total DIY branches (Seeds), good DIY seeds (Good) and upper bound of performance potential per application.

best case scenario. In reality, due to overlap between various DIY modules, we achieve only a fraction of this potential. We noticed that if an application has too many good DIY seeds (e.g., gcc, mgrid), they tend to overlap more often and the potential represents a very loose upper bound because individual gains are not exclusive.

5.3 Methodology and Tuning Framework

In this section, we describe our detailed methodology and overall framework to optimize look-ahead code via DIY branches and skeleton tuning. First, we describe various code transformations that we apply on candidate branches, subroutines and loops to turn them into DIY without creating control flow and execution inconsistencies in the skeleton.

5.3.1 Static DIY Transformations

To speed up the look-ahead thread, we propose a few code transformations to forward conditional blocks and backward branch based loops (depicted in Figure 5.3) as follows:

- **DIY$_{zap}$**: Simplest of all DIY transformations converts a candidate branch and control dependent instructions into NOPs (Figure 5.3-(1)). We call this transformation DIY$_{zap}$ because it creates the effect as if the branch never existed. In a loop, all instructions inside
the loop body are converted into NOPs – except for the exit branches and loop-carried dependences (Figure 5.3-(4)).

- \(DIY_{left}\): Second DIY transformation (Figure 5.3-(2)) converts branches into unconditional \(taken\) branch which we call \(DIY_{left}\). For loops without exit branches in the body, this transformation is illegal because it transforms a finite loop into a non-terminating loop.

- \(DIY_{right}\) or \(DIY_{fall}\): Next DIY transformation known as \(DIY_{right}\) (Figure 5.3-(3)) converts candidate branches into unconditional \(not\ taken\) branch. For a loop, this transformation is known as \(DIY_{fall}\) (Figure 5.3-(5)) in which only one iteration of the candidate loop per invocation is executed.

- \(DIY_{call}\): Final DIY transformation converts a subroutine call into a NOP instruction. \(DIY_{call}\) is applied to the call site instruction only and the effect is similar to the \(DIY_{zap}\) transformation for branches and loops.

These aforementioned static DIY transformations act as various “gears” to achieve delicate balance between the speed and the accuracy. The \(DIY_{zap}\) transformation clearly speeds up the look-ahead agent at the cost of reduced overall accuracy. In contrast, the \(DIY_{left}\) and \(DIY_{right}\) transformations achieve better accuracy compared to \(DIY_{zap}\) by executing the data dependence chains in the DIY region at the cost of speed. It is difficult to statically quantify the relative benefits of each transformation for a candidate branch. Thus, we apply all feasible transformations to each candidate and combine only the most beneficial transformations together.

### 5.3.2 Temporal Slicing of DIY Branches

Once a candidate branch goes through static DIY transformations, all dynamic instances are forced to behave in the same way. However, this \(all\ or\ nothing\) transformation is rather arbitrary and possibly harmful in general. To further improve the balance between speed and accuracy, we propose temporal slicing of DIY branches. When a DIY candidate branch is temporally sliced then only a fraction of dynamic instances of the branch are executed in DIY mode.
(A) Forward conditional branch (If-Then, If-Then-Else) transformations
(1) DIYzap [C]
(2) DIYleft [BR -> A -> C]
(3) DIYright [BR -> B -> C]

(B) Backward conditional branch (Loop) transformations
(4) DIYzap [A -> C]
(5) DIYfall [A -> B -> BR -> C]

Figure 5.3: DIY branch code transformations of conditional branches (A), and loops (B) in the look-ahead thread. Broken lines indicate NOP instructions or unreachable control paths.

While it may appear hard to pinpoint the right amount of temporal slicing of DIY candidates, we rely on an empirical method to determine that. We start by removing a few dynamic instances of a branch based code module (e.g., iteration of a loop, dynamic instance of an if-then-else block and subroutine) from the skeleton and measure the performance impact. If removal of a few dynamic instances of a candidate module improved the overall performance, we call the static module and temporal slice tuple a good DIY seed. To limit the number of DIY seeds, we only considered the branches with at least 100 dynamic instances in our simulation window. We represent the temporal slicing with a duty cycle value which represents the fraction of dynamic instances/iterations skipped. We experimented with a few different duty cycles (in the range of 5–100%) for each candidate.
5.3.3 Combining DIY Branches

After testing individual DIY seed for various duty cycles, we sort them based on their performance. Next, we pick the top \( n \) (25 in this study) DIY seeds and combine them in incremental manner. At this point, we could have used sophisticated metaheuristics e.g., genetic algorithm or simulated annealing – as proposed by [10]. Instead, we rely on one-pass incremental combining to create \( \sum n \) different configurations starting from each DIY seed and go all the way to the end of the list adding one good DIY seed at a time. After combining, we measure the performance of these composite DIY configurations. Finally, we pick the configuration with the best performance. Due to interaction between individual DIY seeds, this is only a suboptimal combining and we plan to experiment with better heuristics in future explorations.

5.3.4 Skeleton Tuning and Customization

To tune skeletons further, we employ a static, profile driven mechanism which carves out various regions in a program execution. For simplicity, we use instruction count based, fixed-size regions which we call epoch. Epoch size (an empirical parameter) can be anywhere from 100s to a few millions of instructions. In each epoch, we measure the performance of various types of skeletons – ranging from simple branch based coarse-grain to very fine-grain skeletons which may even include all the instructions. To distinguish various skeletons, we represent them using skeleton identifier (or SktID) that summarizes skeleton compositions. Table 5.2 lists various SktIDs and transformations they represent.

<table>
<thead>
<tr>
<th>SktID</th>
<th>Associated skeleton transformations</th>
</tr>
</thead>
<tbody>
<tr>
<td>( bB )</td>
<td>Biased branches converted to unconditional branches</td>
</tr>
<tr>
<td>( B )</td>
<td>Biased branches treated as regular branches</td>
</tr>
<tr>
<td>( L1 )</td>
<td>First level (L1) cache prefetches included</td>
</tr>
<tr>
<td>( L2 )</td>
<td>Last level (L2) cache prefetches included</td>
</tr>
<tr>
<td>( Sf )</td>
<td>Compiler generated software prefetches included</td>
</tr>
<tr>
<td>( DLA )</td>
<td>Baseline look-ahead skeleton – equivalent to ( bB+L1+L2+Sf )</td>
</tr>
<tr>
<td>( All )</td>
<td>Includes all insts – equivalent to dual-core execution [6]</td>
</tr>
</tbody>
</table>

Table 5.2: Skeleton identifiers and associated transformations.

A skeleton can be uniquely identified from its SktID. For example, SktID “\( bB+L2 \)” represents a skeleton in which all biased branches have been converted to unconditional “taken” or
“not taken” and has only L2 prefetches included. Based on the choices available for the skeleton components, various types of prefetches and biased branch transformation, we create all possible combinations – a total of 18 types of skeletons. Once we have these seed skeletons, we apply an offline analysis to construct the optimum skeleton. For epoch size we experimented with 1000 to 1 million instructions range with a few points in between. Our offline analysis process (depicted in Figure 5.4) works as described below.

- **Seed skeleton performance** – (A): First, we measure the performance and capture metadata information of initial seed skeletons for every epoch. More precisely, we record starting inst count, start PC and number of cycles.

- **Best skeleton per epoch** – (B): Next, we tabulate per epoch based information which consist of starting inst count, starting PC and the best SktID.

- **Per PC skeleton tuples** – (C): In theory, information from step (B) can be used to switch between various skeletons. However, it will require memorization of instruction count and $N$ different skeleton masks. Instead of directly using dynamic information, we construct static information without much loss of information content. Given the static PC
and best skeleton tuples, we create a multi-valued hash table in which the static PC is the key and values are pairs of SktID and their frequencies.

- **Best skeleton per PC – (D):** For every PC we only retain the SktID with highest frequency and drop the remaining SktIDs (pruning of multi-valued hash to create single valued hash table). We observe that there is a good affinity between a static PC and the best performing SktID. In other words, the most frequent SktID for a given static PC dominates the contributions from all other SktIDs.

- **Optimized skeleton – (E):** Finally, we pick the best SktID for a given PC and continue unless we see another PC where the SktID changes. We stitch parts of per PC based best skeletons to get the final optimized skeleton.

In hindsight, we believe that skeleton tuning should be the first method to be applied for look-ahead code distillation because of its simplicity and low complexity.

### 5.3.5 Putting It Together

Once we have tuned skeleton and the best combined DIY configuration, we can combine them in any order. One approach is to take the tuned skeleton mask and superimpose (logical AND operations on mask bits) best DIY skeleton mask on top of that. However, there may be some issues with this scheme. Because we evolved our DIY mask configuration on top of baseline skeleton, it may not be optimal if the underlying skeleton mask changed. Another approach is to treat the best DIY configuration as new kind of seed skeleton in the skeleton tuning process. If the DIY optimized skeleton is the best in any epoch, it will be picked up by skeleton tuning process automatically. Surprisingly, we noticed very similar performance from either approach which makes us believe that the ordering of these explorations is immaterial.

### 5.4 Architectural Design and Support

In this section, we describe the software, hardware and runtime support needed for DIY branches and skeleton tuning.
5.4.1 Software Analysis and Support

For skeleton tuning, an application needs to be profiled only $N$ times where $N$ is the number of distinct seed skeletons (18 in this study). Analysis to generate optimized skeleton is simple and done only once in the end after we have epoch based performance information for all seed skeletons. We implemented our offline analysis flow in the skeleton parser that takes various seed skeletons and their profiles to generate the final optimized skeleton as described in the Section 5.3.4.

Skeleton is represented by a bit vector which masks off instructions not needed for the look-ahead. In addition to regular bit mask (for baseline skeleton), new skeletons also have a secondary “DIY” bit mask which keeps only the instructions needed for DIY mode. All instructions inside the DIY branch scope for the $DIY_{zap}$ case are converted into NOPs. For the other two transformations, $DIY_{left}$ and $DIY_{right}$, all branches in candidate branch scope are marked as DIY and not passed via BOQ. In $DIY_{call}$ transformation, skeleton marks the call site instruction as special NOP. Skeleton also marks the end of DIY regions and incorporates a duty cycle value with every DIY instruction.

To support DIY loops, DIY mask skips all but loop-carried dependences (e.g., loop index computations) and exit branches. Backward dependence analysis of loop index variables and exit branches terminate at the beginning of the loop. Any call inside DIY loop is treated as DIY call because – even though it takes the control out of the loop – eventually the control will come back unlike exit branches. Similar to DIY loops, DIY branches also have a secondary “DIY” mask which only keeps exit branches and jumps that may take the control outside the scope.

5.4.2 Additional Hardware Support

For skeleton tuning, no modifications in the baseline decoupled look-ahead hardware is required. Our software framework creates tuned skeletons (after applying skeleton tuning flow) which are backward compatible. However, to handle DIY branches we need a few minor modifications in the existing hardware to maintain the execution alignment with the look-ahead thread after the DIY modules. In the new system (Figure 5.5), main thread maintains its own branch predictor for the DIY region so it has to be trained all the time. Main thread also stores
“DIY” skeleton bit mask (to precisely determine the end of a DIY region) and maintains a mode bit to indicate whether the execution is in DIY mode. Modified BOQ entries have one bit for the branch direction outcome and one bit to pass the DIY region start information.

To handle nested subroutines and recursive calls, main thread maintains a *DIY call depth register* which it uses to keep track of call graph depth in DIY mode. Every call in DIY mode increments and every return decrements the call depth register in the fetch stage of main thread. Call depth register is saved with every instruction and restored similar to a global history register in the case of a misprediction. When the call depth register becomes zero and main thread detects the end of DIY region (by reading the DIY end marker from the “DIY” bit mask) then only it starts reading from BOQ.

In our current setup, we use a 4-bit register (a maximum of 15 deep call graph tracking) to maintain the call graph depth information. If we exceed the call depth then main thread will start using BOQ entries prematurely and will eventually perform a recovery. At this point, main and look-ahead thread will be re-synchronized. We hardly noticed any performance difference between a 4-bit and a 64-bit call depth register which suggests that 4-bit call depth register is sufficient for all practical purposes. Similarly, if a code does not have matching number of calls and returns (due to the presence of tail jumps) then an eventual recovery will re-synchronize main and look-ahead threads.
5.4.3 Runtime Management Support

DIY Calls and Subroutines:

If a dynamic instance of a call is marked to be DIY then look-ahead thread treats the call instruction as a spacial NOP and proceeds sequentially. DIY call instance, through BOQ, indicates to the main thread that branches from this point onward are not passed until the matching return. Note that DIY property is associated with the call site as opposed to function itself. Every branch in the function body of a DIY call also becomes a DIY branch. Main thread, upon reading “DIY” marked BOQ entry, sets itself in DIY mode and ignores BOQ until it gets to the matching return after the subroutine which has DIY end marker.

DIY Loops and Conditional Branches:

Ideally, we prefer to make a loop iteration DIY if it accesses the same cache line as previous iteration. However, we can force any iteration to be a DIY iteration. For DIY iterations, we simply switch to secondary “DIY” mask in the look-ahead thread. If a forward conditional branch (e.g. if-then-else) is DIY then the look-ahead thread will simply treat all the instruction as NOP and fall through. It will also mark the corresponding BOQ entry to indicate the start of a DIY region. Main thread reaching upon to this branch will start using its own branch predictor. DIY start is also marked in the skeleton but it is up to the look-ahead thread to whether really start a DIY execution or not. If look-ahead thread decides so, the corresponding BOQ entry will be marked to inform the main thread.

5.4.4 In a Nutshell

DIY code modules are a “contract” between the look-ahead thread and the main thread. Their boundaries are marked with “DIY-start” and “DIY-end” markers in the skeleton statically. In runtime, the look-ahead thread decides about the DIY execution and “DIY-start” information is passed to the main thread via a special bit in BOQ. Main thread determines the “DIY-end” from the “DIY” bit mask. In some cases, look-ahead thread may execute DIY branches for its own data and control flow accuracy but never passes any outcome hints to the main thread. Main
thread from instruction “DIY-start” onward skips reading branch outcomes from BOQ and uses its own branch predictor. It again starts reading BOQ entries only after the “DIY-end” marker.

5.5 Experimental Setup

We perform our experiments using a cycle-level, execution-driven in-house simulator. We faithfully model support for a decoupled look-ahead system, including when the lead thread diverges from the actual program’s control flow. The simulator also faithfully models a number of details in advanced designs such as load-hit speculation (and scheduling replay), load-store replays, keeping a store miss in the SQ while retiring it from ROB [77]. Our baseline core is a generic out-of-order microarchitecture with parameters loosely modeled after next generation MIPS P-class Warrior core from Imagination Technology. An advanced hardware global stream prefetcher based on [79, 80] is also implemented [109]. We assume a 3 GHz clock frequency which translates a typical 80-85ns memory latency to about 250 clock cycles.

Our baseline system has state-of-the-art branch prediction mechanism. In addition to Gshare, we also modeled an 8-component based TAGE branch predictor with 130-bit maximum history length and 64K bits total storage budget [110]. We achieve spec-int MPKI of 8.9 using the TAGE predictor which is slightly better compared to 9.8 MPKI reported in [68]. Our Gshare implementation achieves 8.7 MPKI so we decided to choose Gshare for our further experiments. The configuration parameters are shown in Table 6.1.

5.5.1 Applications and Inputs

We use applications from SPEC CPU2000 benchmark suite compiled with optimization flag -O3 for Alpha using a cross-compiler built on gcc-4.2.1. We use the train input for profiling, and run the applications for 500 million instructions, which generally cover the exercised code region in later non-profiling runs. After the profiling, we use ref input and simulate 200 million instructions after skipping over the initialization portion as indicated in [81].

\footnote{Instruction interpretation and Linux system call emulation are partially borrowed from [76].}
5.6 Performance Analysis

5.6.1 Overall Performance

DIY branches and skeleton tuning techniques combined together boost the baseline decoupled look-ahead to achieve 1.53x speedup (right most dark bars in Figure 5.6) over single-thread for the applications where look-ahead thread is the bottleneck. This is a significant improvement of 15% from the baseline decoupled look-ahead speedup of 1.33x (left most bars in Figure 5.6) over single-thread. Best speedup comes for equake where decoupled look-ahead performance improves by 1.93x. If we look at the INT applications only, our techniques improve the performance of decoupled look-ahead by 13% whereas for FP applications the improvement is 16.3%. We list IPC of single-thread baseline (ST), baseline decoupled look-ahead (LA) and the skeleton tuning based decoupled look-ahead (LB) configurations in Table 5.4.

<table>
<thead>
<tr>
<th></th>
<th>gcc</th>
<th>mcf</th>
<th>eon</th>
<th>pbk</th>
<th>hzsp</th>
<th>twf</th>
<th>wup</th>
<th>mgd</th>
<th>art</th>
<th>eqk</th>
<th>fac</th>
<th>amp</th>
<th>luc</th>
<th>fma</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST</td>
<td>1.64</td>
<td>0.54</td>
<td>1.67</td>
<td>0.86</td>
<td>1.01</td>
<td>0.65</td>
<td>2.82</td>
<td>2.73</td>
<td>1.23</td>
<td>1.05</td>
<td>2.82</td>
<td>1.05</td>
<td>1.01</td>
<td>2.23</td>
</tr>
<tr>
<td>LA</td>
<td>3.20</td>
<td>0.80</td>
<td>1.79</td>
<td>1.00</td>
<td>1.26</td>
<td>0.78</td>
<td>3.21</td>
<td>3.17</td>
<td>1.78</td>
<td>1.11</td>
<td>3.20</td>
<td>2.23</td>
<td>1.98</td>
<td>2.43</td>
</tr>
<tr>
<td>LB</td>
<td>3.68</td>
<td>0.93</td>
<td>2.13</td>
<td>1.04</td>
<td>1.35</td>
<td>1.05</td>
<td>3.42</td>
<td>3.49</td>
<td>2.03</td>
<td>2.12</td>
<td>3.50</td>
<td>2.34</td>
<td>2.13</td>
<td>2.65</td>
</tr>
</tbody>
</table>

Table 5.4: IPC of single-thread (ST), baseline look-ahead (LA), and DIY+skeleton tuned decoupled look-ahead (LB).
Figure 5.6: Performance speedup of baseline decoupled look-ahead, DIY branch enabled look-ahead, skeleton tuned look-ahead and combination of DIY branches + skeleton tuned look-ahead over single-thread baseline.

**Breakdown of Performance Gains:**

To appraise individual techniques, we measured speedups just from DIY branches and skeleton tuning in isolation. DIY branches improve the performance of baseline look-ahead by 8% (second set of light grey bars in Figure 5.6), whereas from skeleton tuning (third set of darker gray bars in Figure 5.6), the improvement is 11%. Combination of these two techniques should have achieved close to 19% performance improvement, if the two techniques were completely independent.

**Robustness:**

Using our framework, we first discover best DIY branch configuration and on top of it we explore skeleton tuning. Due to some overlap between DIY branches and skeleton tuning, we achieve only 15% improvement when we combine them. We also experimented with starting from the skeleton tuning and on top of it we superimposed the best DIY branch configuration. We see almost similar results and the order in which these two techniques are explored does not have any consequence. To recap, the two solutions can be evolved almost independently and applied on top of each other in order oblivious manner.

**Additional Recoveries:**

DIY branch and skeleton tuning optimizations make the look-ahead thread to follow non-correct execution path more often than the baseline skeleton. However, the additional recoveries in dis-
tiled skeletons are still comparable to baseline decoupled look-ahead. Compared to 3.12 recoveries per 10,000 instructions in baseline decoupled look-ahead, we observed slightly higher 3.19 recoveries per 10,000 instructions in the DIY and skeleton tuning based decoupled look-ahead. Performance gain from the distillation of skeleton is far more significant than the performance degradation due to additional recoveries.

**Skeleton Size and Energy Saving:**

Baseline look-ahead skeleton is 71.8% of the original program. After skeleton tuning and DIY branches, we execute only 57.2% instructions in the look-ahead thread and reduce the skeleton by 14.6%. This significant saving in skeleton size, coupled with faster execution, also results into a reduction of 18% overall energy consumption from the baseline look-ahead.

### 5.6.2 Comparison with Weak Dependence

In this section, we compare and quantify the performance gain from our proposed techniques on top of a weak dependence removed look-ahead [10]. While similar in objective, weak dependence proposal targets individual instructions that are weak enough (do not affect the nature of a computation that much) and removing them in the look-ahead thread does not materially affect the quality of look-ahead but improves the speed. In our setup, performance improvement due to just from weak dependence look-ahead is 11.3% over the baseline look-ahead – slightly lower but close to originally reported. If we apply our skeleton tuning and DIY branch techniques on top of this system, we get the overall speedup to 19.8% (as shown in Figure 5.7).

Our techniques on top of baseline look-ahead achieved 15% performance, whereas on top of weak dependence removed look-ahead we only got 8.5% improvement. This indicates a significant overlapping between the weak dependence and our skeleton tuning techniques which is not hard to understand. A perfectly good weak instruction (already removed) from a candidate DIY module lightens the overall load, therefore, leaves little room for DIY branches and skeleton tuning. In contrast, we observed much lower gain for the weak dependence removal technique on top of skeleton tuning (about 1.6%). Because weak dependence removal eliminates all dynamic instances of an instruction, it has less chances of improving performance in case where
we are already dithering the enclosing code modules. Later we present a few cases of code modules in which eliminating only a fraction of dynamic instances is a better option than completely removing them. On the other hand, skeleton tuning and DIY branches still achieve relatively good success in further improving a skeleton with removed weak instructions.

Recap:

In hindsight, it appears that the order in which weak dependence and skeleton tuning techniques are applied makes a difference. Based on experimental insights, we believe that look-ahead acceleration techniques should be applied in this order: ① skeleton tuning, ② weak dependence removal and DIY branches should be evolved together due to interactions between them.

5.6.3 Comparison with Turbo Boost Technology

We compare the performance of our decoupled look-ahead system with a couple of Turbo Boost single core systems (Figure 5.8). The first system is a conventional Intel i7 based Turbo Boosting where we scale up the operating frequency to match the power overhead with our decoupled look-ahead system. Intel processors require 50 mV additional voltage to operate at 133 MHz faster from the highest rated base frequency [112]. For this particular configuration, we assume a base operating frequency of 3 GHz and a supply voltage of 1.38 V [113]. Due to a 13% boost in frequency and 11% higher supply voltage, the dynamic power overhead of this Turbo boost
### Operating conditions

<table>
<thead>
<tr>
<th></th>
<th>Baseline</th>
<th>Turbo Boost</th>
<th>Max Turbo Boost</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Freq (GHz)</td>
<td>3.0</td>
<td>3.4</td>
<td>3.75</td>
</tr>
<tr>
<td>Supply voltage (V)</td>
<td>1.38</td>
<td>1.53</td>
<td>1.63</td>
</tr>
<tr>
<td>Memory lat (in CPU cy)</td>
<td>250</td>
<td>283</td>
<td>313</td>
</tr>
<tr>
<td>Performance speedup</td>
<td>1.00</td>
<td>1.09</td>
<td>1.17</td>
</tr>
<tr>
<td>Power overhead</td>
<td>1.00</td>
<td>1.39</td>
<td>1.74</td>
</tr>
</tbody>
</table>

Table 5.5: Performance and power overhead comparison of Turbo Boost systems over baseline.

![Graph showing performance comparison](image)

Figure 5.8: Performance comparison of load balancing (DIY + skeleton tuned) DLA with two flavors of turbo boosted single-thread systems.

...system is about 39% over the single-thread baseline (as shown in Table 5.5). Power overhead of our decoupled look-ahead system is 1.37x over the single-thread baseline.

We also simulate the performance of another turbo boost system which we call Max Turbo Boost system (second set of bars in Figure 5.8). Max Turbo Boost operates at 25% higher clock frequency and the power overhead is about 1.74x – almost double compared to our decoupled look-ahead system. Max Turbo Boost system is the upper limit of performance that frequency scaling technique can achieve (to best of our knowledge) without violating the thermal design power (TDP) budget.

In our setup, Turbo Boost shows 9% (first set of bars in Figure 5.8) and 17% (second set of bars in Figure 5.8) performance speedup for a frequency scaling of 1.13x and 1.25x respectively. These are similar to recently reported results (measured from real systems) for Intel i7 kind of cores [111–113]. In comparison, decoupled look-ahead achieves significantly better performance (1.45x speedup across the board) with much lower TDP budget.
5.7 Diagnostics Analysis

To shed more light on the current look-ahead system and the techniques that we explored in this work, we present insights from numerous experiments that we carried out.

5.7.1 Look-ahead Core Simplification

One of the benefits of distilled skeleton is that it allows a simpler core for the look-ahead thread to improve the power, energy and area of the overall system. In our current system, we manage skeleton as a bit-mask stored in I-cache. It is not until the decode stage, when the look-ahead thread determines whether to execute an instruction or discard it. However, this is clearly an artifact and going forward we envision that finished product should maintain separate and distilled look-ahead binary. Unfortunately, a bit-mask based skeleton does not reduce the fetch bandwidth requirement even for distilled skeletons and hinders us from achieving the best power and energy efficiency that could be achieved.

Performance Impact:

Further distillation of skeleton should allow look-ahead core to have simpler pipeline (at least the back-end in current setup). Wide front-end is still needed because look-ahead thread is still fetching all the instructions. We experimented with reduced decode, rename, issue and commit width for look-ahead core. Average performance speedups of decoupled look-ahead systems with a 3- and a 2-wide look-ahead core (along with a 4-wide baseline) are shown in Figure 5.9. As we can see, with the new skeleton a 2-wide look-ahead core achieves roughly \textit{similar} performance what a 4-wide look-ahead core used to achieve with baseline skeleton. In INT applications, a 2-wide look-ahead core with tuned skeleton is slightly lower compared to a 4-wide look-ahead core with baseline skeleton. In FP codes, a 2-wide look-ahead with tuned skeleton is significantly better compared to a 4-wide look-ahead core with baseline skeleton. This is mainly because FP applications benefit more from prefetching than the perfect branch prediction.

Another key observation is that we only lose 3-4\% performance if the look-ahead core
width is reduced to 3 from the baseline width of 4. This is mainly due to two reasons: First, the baseline skeleton is only about 72% of the total program so for a 4-wide main core, a 3-wide look-ahead core should be sufficient to sustain the execution bandwidth. Second, the overall IPC (∼2 for INT and ∼2.5 for FP apps) is still much lower than the pipeline width of 3\(^5\).

**Power Savings:**

In Figure 5.10, we present the power reduction of various components of simpler look-ahead cores. We noticed, that a 2-wide renamer and decode logic has almost half the power consumption compared to a 4-wide which is consistent with previously reported results. We also observe almost 2x power saving in various components of RAT. These are some of the traditional hotspots and power hungry structures in the CPU pipeline and lower power of these component will allow look-ahead thread to be a very power efficient technique for turbo boosting.

Finally, with a 2-wide pipeline we are able to reduce the overall power of system by close to 11% (this only accounts the back-end pipeline width scaling of look-ahead core). 2-wide look-ahead core with new skeleton reduces the power overhead of baseline decoupled look-ahead system from 1.53x to 1.37x over the single thread baseline while delivering similar performance as baseline decoupled look-ahead. Our current system still has a wider fetch for look-ahead core,

---

\(^5\)In another unrelated single-thread study, we observed that a 3-wide core achieves within 3% of IPC compared to 4-wide core for SPEC INT apps.
so the power savings (presented here) from the simplification of look-ahead core are absolute lower bound of what we should have achieved.

### 5.7.2 Optimal Skeleton Distribution

To understand the relative performance of individual skeletons, we compare them across various intervals. We collect performance of every seed skeleton at fixed intervals (10,000 instructions in this study) and pick the best performing skeleton from each interval. After normalizing the number of intervals per skeleton we plot them in Figure 5.11. In X-axis, we have various types of skeletons that we explored and Y-axis is the % of intervals in which a specific type of skeleton was best among all other skeletons.

Skeleton identifier (or SktID) represents various components of skeleton that have been included. For example, skeleton “B+L2” (regular branches + L2 misses) is the best for *equake* for 35% of the time. From Figure 5.11, we do see that baseline decoupled look-ahead skeleton (DLA) is the best overall (about 30% of the time). However, for the remaining 70% of the time there are other kind of skeletons that perform better than baseline look-ahead skeleton (DLA).
5.7.3 Static Code and Skeleton Affinity

In order to tune skeletons, in theory, we need the dynamic information which ties a particular dynamic trace (or code interval) to a specific type of skeleton. While this information has high accuracy, it has several drawbacks in terms of implementation. We will be required to maintain $N$ different skeleton bit-masks (one for every seed skeleton) and additional storage for interval switching information.

To implement our static skeleton tuning from the dynamic trace information, we extracted static PC based SktID information (as described in Section 5.3.4) which incurs some loss of accuracy. We found that the top performing skeleton and PC pair contributes to more than 58% dynamic instances overall. For the remaining 42%, our PC based information will not be optimal. However, if the most prominent SktID for a given PC is better than the baseline skeleton in all intervals, we will still see some performance gain for remaining 42% instances.

To confirm that this is indeed the case we also modeled the dynamic instruction count based skeleton tuning. The performance difference between using dynamic instruction count based skeleton tuning and static PC based skeleton tuning is less than 1% overall.

5.7.4 Duty Cycle Sensitivity

Unlike weak dependence removal, we do not completely skip the DIY branches instead tune them to execute only a fraction of their dynamic instances. Depending upon data access pat-
terns, duty cycle plays a key role in achieving good performance for a given DIY module. We observed that majority of loops tend to have better performance for low duty cycles (5–20%), while functions that do minor adjustments are better off when skipped completely (90–100%).

For brevity, we present one case of DIY call from 179.art that has varying performance for different duty cycles (Figure 5.12). In this case, executing only a few instances (1 out of 10) is sufficient for the look-ahead and results into best performance. This shows that duty cycle for a given module is a crucial parameter and must be chosen diligently after experimentally verifying the performance impacts. We found many DIY seeds that improve performance significantly (by >5%). Here we list a top few DIY modules and their individual gains in isolation (Table 5.6). We noticed that subroutines with high number of recoveries, should be skipped as well. Because look-ahead was not doing a good enough job there anyways so by skipping such subroutines we can improve performance and save energy.

### 5.7.5 Profiling Overhead

Profiling overhead for our skeleton tuning and DIY branches is quite small. For skeleton tuning, we profile about 18 different skeletons. The manipulation overhead of these 18 seed skeleton to construct the optimum skeleton is on the order of control software overhead in weak dependence removal. For DIY exploration, we profile non-overlapping modules in single run. Due to this, the performance of all subroutine based DIY modules (recall that the DIY functions are call
site based) can be collected in a single run. The total number of DIY seeds are an order lower compared to weak instructions genes. We do have to profile each DIY seed for different duty cycles. In the end, for DIY and skeleton tuning we end up profiling between 4.2 billion (ammp) to 15 billion (perlbmk) instructions with an average of 6.7 billion instructions. This is a 5x lower profiling overhead compared to weak instruction removal (last row in Table 5.7).

<table>
<thead>
<tr>
<th>Apps</th>
<th>Module name</th>
<th>Duty cy(%)</th>
<th>Perf gain(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>_OtsMove</td>
<td>gcc</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>sortlt</td>
<td>bzip2</td>
<td>90</td>
<td>6</td>
</tr>
<tr>
<td>memory_address_p</td>
<td>mcf</td>
<td>90</td>
<td>12</td>
</tr>
<tr>
<td>primal_net_simplex</td>
<td>mcf</td>
<td>20</td>
<td>10</td>
</tr>
<tr>
<td>WeightAdj</td>
<td>art</td>
<td>90</td>
<td>7</td>
</tr>
<tr>
<td>mark_modified_reg</td>
<td>gcc</td>
<td>90</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 5.6: A few examples of DIY code modules.

<table>
<thead>
<tr>
<th>DLA configs</th>
<th>Max</th>
<th>Avg</th>
<th>Min</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weak dependence removal</td>
<td>61.5</td>
<td>32.1</td>
<td>16.9</td>
</tr>
<tr>
<td>Skeleton tuning</td>
<td>1.8</td>
<td>1.8</td>
<td>1.8</td>
</tr>
<tr>
<td>DIY module exploration</td>
<td>13.2</td>
<td>4.9</td>
<td>2.4</td>
</tr>
<tr>
<td>DIY module + skeleton tuning</td>
<td>15.0</td>
<td>6.7</td>
<td>4.2</td>
</tr>
</tbody>
</table>

Table 5.7: Profile overhead comparison of DIY branches and skeleton tuning with weak dependence removal.

### 5.7.6 Comparison with Idealized Systems

Finally, to understand the effectiveness of tuned decoupled look-ahead and remaining potential, we conduct additional experiments. We compare the performance of baseline and tuned decoupled look-ahead with a set of idealized systems and summarize the results in the Table 5.8 and in Figure 5.13 we show the comparison for SPEC apps.
### Speedup over Single-Thread

<table>
<thead>
<tr>
<th>Application</th>
<th>Perfect BR</th>
<th>DLA: BR only</th>
<th>Perfect L2</th>
<th>Perfect L2+BR</th>
<th>DLA: L2+BR</th>
<th>Perfect L1</th>
<th>Perfect L1+BR</th>
<th>DLA: Baseline</th>
<th>DLA: Final</th>
</tr>
</thead>
<tbody>
<tr>
<td>gzip</td>
<td>3.56</td>
<td>3.72</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vpr</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>gcc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mcf</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>crafty</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>eon</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pbmk</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>gap</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vortex</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bzip2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>twolf</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gmean</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Perfect Branches:

A bare bone skeleton that targets only branches, brings in a bunch of loads and other instructions (backward dependence chain of branches). In INT applications, a good chunk of L2 misses originate from the loads that feed these data-dependent branches. This makes it harder to isolate branches and L2 prefetching in look-ahead. As a result, we see almost 2x performance for look-ahead over the perfect branches because targeting branches has substantial L2 prefetching side-effects. In FP applications, we are able to target only the branches and achieve close to perfect branches performance which is not whole lot because of simpler predictable loop branches.

This distinct branch behavior in INT and FP applications confirms a very fundamental fact about branches. In INT applications, majority of the branches are data-dependent and a good chunk of these data miss in the caches (e.g., pointer-chase in mcf). On the other hand, in FP codes, a good chunk of branches are from loops which have simple index-computation chain dependent upon hot variables that are often cached in L1 caches.

---

**Figure 5.13:** Performance potential of idealized system and comparison with various decoupled look-ahead systems (for SPEC apps).
<table>
<thead>
<tr>
<th>Idealized systems/ DLA configs</th>
<th>INT</th>
<th>FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perfect branches</td>
<td>1.12</td>
<td>1.03</td>
</tr>
<tr>
<td>$DLA_a$ (target branches only)</td>
<td>1.26</td>
<td>1.02</td>
</tr>
<tr>
<td>Ideal L2 cache</td>
<td>1.24</td>
<td>1.35</td>
</tr>
<tr>
<td>Ideal L2 cache + Perfect branches</td>
<td>1.35</td>
<td>1.37</td>
</tr>
<tr>
<td>$DLA_b$ (target branches + L2)</td>
<td>1.27</td>
<td>1.27</td>
</tr>
<tr>
<td>Ideal L1 caches</td>
<td>1.53</td>
<td>1.49</td>
</tr>
<tr>
<td>Ideal L1 caches + Perfect branches</td>
<td>1.78</td>
<td>1.52</td>
</tr>
<tr>
<td>Baseline DLA (target branches + L2 + L1)</td>
<td>1.29</td>
<td>1.34</td>
</tr>
<tr>
<td>DIY-skeleton tuned DLA (target branches + L2 + L1)</td>
<td>1.40</td>
<td>1.49</td>
</tr>
<tr>
<td>Final DLA (weak dependences + DIY + skeleton tuned)</td>
<td>1.53</td>
<td>1.50</td>
</tr>
</tbody>
</table>

Table 5.8: Comparison of DLA configs with idealized systems.

**Ideal L2 Caches and Branches:**

A skeleton without branches will require an alternate control path mechanism. That’s why we combine the potential of ideal L2 cache with perfect branches. We are able to achieve about 77% of potential for INT and about 73% for FP applications. This is a good conversion of potential into real performance despite the contention created in shared caches by look-ahead thread.

**Ideal Caches and Branches:**

Finally, we show the performance of ideal L1 caches combined with perfect branches. IPC for this configuration is close to pipeline width (>3) and represents very “optimistic” upper bound of ultimate performance. In FP applications, baseline look-ahead used to achieve only 65% of the performance potential. Final tuned look-ahead achieves close to 96% of the ideal L1 and perfect branches potential. However, for INT applications, we achieve about 68% of the potential. We plan to investigate this performance gap as a part of our future explorations.

**5.7.7 Future Directions**

In the current setup, we could not convert all the potential of L1 prefetching to real performance. First of all, by the time we include L1 prefetches we already have a large skeleton. Second, L1 prefetches are more time (when to launch) and space (where to place) sensitive compared to L2.
prefetches and require additional tuning. We reckon a better solution will be to have multiple look-ahead threads that target different bottlenecks. The idea is to not over-burden one look-ahead thread but to specialize them for different tasks. In this work, we explored only a handful of seed skeletons as a first step. In future, we will consider more variations of skeletons e.g., tackling misses that matter more. We also plan to identify DIY code modules and classify them at source code level so that a helper thread compiler framework can exploit these insights to generate better codes for helper threads.

5.8 Summary

Due to numerous advantages, the task of making a program run faster could and should be taken over by automated mechanisms rather than left to individual programmers. In this paper, to exploit implicit parallelism, we present a compelling case of further accelerating look-ahead thread and a fully automated mechanism to accomplish that via selectively skipping DIY branches and skeleton tuning. Unlike previous approaches, we employ simple heuristics to optimize skeleton with individually tuned DIY branches and seed skeletons. While it is hard to reason about what code should be skipped (based on static analysis) in the look-ahead thread, it is straightforward to verify the impact empirically.

DIY branches and skeleton tuning, combined with weak dependence removal, improve the performance of baseline decoupled look-ahead by upto 2.12x (equake) with a geomean of 1.20x. This brings up the speedup of decoupled look-ahead to 1.61x over single-thread baseline. Alternatively, distilled skeleton enables simplification of look-ahead core to reduce the power overhead of baseline look-ahead system from 1.53x to 1.37x. Finally, a distilled skeleton paves the way for single-core SMT based decoupled look-ahead system – even more cost-effective implementation.
Self Tuning (Rationing) in Shared Caches

Shared caches are generally optimized to maximize the overall throughput, fairness, or both, among multiple competing programs. In shared environments and compute clouds, users are often unrelated to each other. In such circumstances, an overall gain in throughput does not justify an individual loss. We explore cache management policies that allow conservative sharing to protect the cache occupancy for individual programs, yet enable full cache utilization whenever there is an opportunity to do so.

We propose a hardware-based mechanism called cache rationing. Each program is assigned a portion of the shared cache as its ration. The hardware support protects the ration so it cannot be taken away by peer programs while in use. However, a program can exceed its pre-allocated ration, but only if another program has unused space in its allocated portion of ration. We show that rationing provides good resource protection and full cache utilization of the shared cache for a variety of co-runs. In addition, the same underlying hardware support can enable energy-efficient and hardware-software collaborative caching.

The rest of the chapter is organized as follows: Section 6.1 presents motivation for the self tuning in shared caches. Section 6.2 describes hardware support for cache rationing and Section 6.3 describes the uses of this hardware support for collaborative caching. Section 6.4 compares rationing with other methods. Section 6.5 details the experimental setup and 6.6 presents the overall performance results. Section 6.7 overviews related work, and finally Section 6.8 summarizes our findings.
6.1 Motivation for Shared Cache Management

As multi-core clusters with shared caches become commonplace and cloud computing gains acceptance, more applications share the same cache hierarchy. Managing cache sharing is crucial not just for achieving good performance, but also for ensuring stable performance in a dynamic environment; and not just for parallel programs but also for co-running sequential programs.

The basic task of cache management is to efficiently allocate cache blocks to co-running programs. There are two extremes: hard equal cache partitioning and “free-for-all” cache sharing (also called no-partitioning). Neither strategy is perfect and numerous other proposals have devised more effective sharing strategies. The goal has always been to optimize throughput, fairness, QoS, or some combination of the three. Previously proposed mechanisms evolved around adaptive sharing – allocating cache resources in some proportion of the demand or expected benefit. These solutions are optimistic since the allocation is based on the prediction of the favorable aggregate performance.

One problem with optimistic sharing is that it is intrusive; that is, it re-allocates space based on the collective needs of all programs, even if certain individual programs are hurt. In this paper, we explore the approach of conservative sharing, in which cache re-allocation is only done if no program is hurt by it. Increasingly in shared environments, especially compute clouds, users are unrelated to one another. A collective gain does not justify an individual loss; thus, a provision for conservative sharing is needed to prevent resource hogging by non-cooperative users yet still enable resource sharing whenever possible. As opposed to yielding more cache resources to aggressive and demanding programs, we prefer to re-allocate resources only when no program is hurt.

We present a hardware technique called cache rationing. Each program is allocated a ration which can be shared, but only under a strict condition. In particular, a program can share the cache ration of a peer program only if that peer program does not use its full ration. If a program uses all of its ration, no other program can take its promised allocation away. In this way, rationing provides both good resource protection and full utilization. To implement cache rationing, two forms of hardware support are needed. The first is accounting, i.e., keeping track of how much cache space is used by each program. This requires maintaining a set of counters...
for every program that is sharing the cache. The second is usage tracking, \textit{i.e.}, monitoring whether a program needs all its ration or can share it with others. To track usage, we use a solution similar to OS memory allocation: an “access-bit” per cache line to indicate whether the cache line has been actively used in the recent past.

Traditionally, caches are managed by temporal and spatial usage requirements. For protection, we must allocate cache blocks based on who accesses the data rather than when the accesses happen. With per cache line access bit, we will show that cache rationing can combine entitlement and usage-based cache allocation, and hence obtain the benefits of partitioning and sharing while avoiding their disadvantages. The rationing support also allows hardware-software collaborative caching [115]. In particular, we add a single bit to indicate a special memory load and store. A special operation marks the cached line as “evict-me” by clearing the access bit for the special access. We will show that collaborative caching can further improve the performance of a rationed cache. Cache rationing aims to achieve Pareto optimality\footnote{A resource allocation is Pareto optimal if \textit{one cannot make any one individual better off without making at least one individual worse off.}} for cache allocation. Sharing of an unused ration is a Pareto improvement, since it helps a program without making anybody else worse off. Hardware-software collaboration is also a Pareto improvement, since one program can optimize the use of its ration without affecting others.

### 6.2 Cache Rationing

A ration for a program is the \textit{guaranteed} effective size of space in the shared cache that is allocated to the program. The ration can be specified by software, \textit{e.g.}, through privileged instructions. First, we define two new objectives in cache sharing, then we show how cache rationing is designed to achieve these objectives.

#### 6.2.1 Objectives of Rationed Cache

Traditionally, in engineering computing, we strive to maximize performance. In a shared facility, \textit{e.g.}, a university computing center, we also strive to ensure fairness. In such cases, the hardware is fully owned by the user or organization running the applications.
Cloud computing operates on a retail model where cloud processors are rented on the open market. A user temporarily takes ownership of part of a machine. For example Amazon Elastic Compute Cloud (EC2) lets a user rent a virtual computer. The size of the computer is elastic and measured by how fast it is. The speed is measured by *elastic compute units* (*ECU*). Amazon defines the ECU by the average performance of a standard CPU configuration. In the cloud context, the fairness and efficiency are different than in the workstation and computing center. Thus we define two new objectives and compare them with throughput and fairness.

**Resource Protection vs. Fairness:**

The rented computing power is measured by the dedicated use of hardware, including the size of cache. When sharing a processor among multiple users, the goal is not to evenly divide the shared cache but to guarantee that a rented core has cache at or above the specified size. We call this guarantee *resource protection*. Resource protection is not the same as fairness. In fairness, a group of tasks have an equal partition of the cache (resource fairness), or an unequal partition so they have a similar performance loss (performance fairness). In resource protection, we reserve a cache partition at least the size of the ration, regardless of the demand of peer programs.

Rationing is designed to guard against unexpected performance loss. Based on total demand, a user may reduce the ration given to a program. There is a performance loss due to reduced ration, as happens when more programs are added to share the same machine. But unlike traditional cache sharing, cache rationing bounds the worst-case resource allocation and losses. It provides a “safety net” for performance of individual programs and supports QoS.

**Utilization vs. Throughput:**

A program may not use all of its ration, depending on the cache demand of the program and the ration it is given. We want to utilize unused ration it if there is a co-run program that can benefit from more cache. We call the amount of cache used by a program its *cache utilization*.

---

2 According to Amazon, “We use several benchmarks and tests to manage the consistency and predictability of the performance of an EC2 Compute Unit. One EC2 Compute Unit provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor. This is also the equivalent to an early-2006 1.7 GHz Xeon processor referenced in our original documentation.” (source: EC2 wiki page)
Maximal utilization is different from maximal throughput. To maximize overall performance, we allocate each chunk of cache space to the program that can gain the most by having the additional space at the potential danger of some program being hurt. To maximize utilization, we detect “unused” space and make it available for sharing. For throughput we need global optimization, while for utilization we need just local detection.

6.2.2 Support for Ration Accounting: Ration Tracker

To implement ration accounting, we propose two sets of additional storage to existing cache hardware: a \textit{ration tracker} for each program/core\footnote{In this paper, we use program (in the context of software) and core (in the context of hardware) interchangeably.} and a \textit{block owner} for each cache block (as shown in Figure 6.1). A ration tracker is a register-counter pair: the number of cache blocks rationed for a specific program are stored in the “owner allocation” register and the number of resident cache blocks of the same program are tracked by “ration counter”. Each cache block stores its “block owner”, which is a reference to one of the ration tracker. The storage overhead of “block owner” is very low. For example, in a 4-core sharing, the block owner record needs 2 bits per cache block to indicate which core owns the block as part of its ration. It also needs a way to indicate if a valid block is unowned. For this, we can use the access bit described in Section 6.2.3. If the block is unused, then it also has no owner.

Rationing can be enforced at multiple levels – ranging from the whole cache (global) to individual set (local) level. In the coarse-grain rationing, each core has only one set of ration tracker to keep track of total allocation. On the other extreme, the fine-grain rationing is enforced at each set level and each core has an allocation in every set. Cache level allocation allows more flexible partitioning since a ration can be any integer between 1 and the cache size. However, it has weaker protection against interference since local hotspots can happen. Per set allocation provides fine-grained protection but requires a set of ration trackers for every cache set which has relatively higher storage overhead. We will compare these two flavors of rationing in the evaluation section.

Maintenance of the ration trackers requires knowing the identity of the core/program responsible for individual memory accesses. The maintenance logic is shown in the pseudocode.
as follows. There are three cases described here: a cache block *fetch* after a miss, an *eviction* and a normal *access* that hits in the cache.

```c
# Cache "blk" is fetched by core "p" after a miss
FETCH( blk, p )
    blk.block_owner = &(p.ration_tracker)
    blk.block_owner->ration_counter ++

# Cache "blk" is evicted
EVICT( blk )
    blk.block_owner->ration_counter --
    blk.block_owner = &(default.ration_tracker)
    blk.block_owner->ration_counter ++

# Cache "blk" is accessed by core "p" and hits in the cache
ACCESS( blk, p )
    if (blk.block_owner != p.ration_tracker)
        blk.block_owner->ration_counter --
        blk.block_owner = &(p.ration_tracker)
        blk.block_owner->ration_counter ++
```

At a cache block fetch after a miss, we set the block owner record to point to the ration tracker of the loader and increment its ration counter. At an eviction, we follow the block owner record to decrement the ration counter. The evicted block becomes unowned unless another program claims it. There are two cases when the ration counters become inconsistent: (a) in a per-core rationing when a task migrates and (b) in multi-threaded code with data sharing. In both cases, a block is loaded by one core but evicted by another core.
The problem of inconsistent ration counters can be solved in two ways. The first is counter association. If we assign a ration counter per task, then migration does not cause inconsistency. If we assign the same ration counter for all threads of a program, then the problem is solved for data sharing. We can design more elaborate schemes where the ration counters are dynamically coalesced and split. The second solution is ownership update. At a cache (hit) access, if there is a mismatch we update the owner record accordingly. The preceding pseudocode shows the second solution.

6.2.3 Support for Usage Tracking: Access Bit

The hardware support needed to detect an unused ration is the access bit which is maintained per cache block, as shown in Figure 6.1. The access bit is set whenever the block is referenced. Periodically, all access bits are reset simultaneously. The time between consecutive resets is called reset interval. Simultaneous reset may require maintaining access bits in a register based storage\(^4\) as opposed to SRAM storage. As an alternative, we can use a cyclic resetting of access bits – similar to DRAM refresh mechanism. Slow cyclic resetting should not result into any performance degradation because reset points are quite sparse. A rationed cache block is deemed “unused” if either it is not owned or the access bit is not set.

6.2.4 Rationing Control

We augment the cache management logic to implement rationing. The description is based on a set-associative LRU cache. Adaptations for other types, e.g., pseudo LRU, can also be made. The description here focuses on how to use the ration counter and access bit. For each ration, we keep track of the least recently used block whose access bit is set. For each cache hit, the stack is adjusted as in a traditional cache. In addition, the rationed cache records the LRU block for each ration (whose access bit is 1).

The following algorithm shows the replacement logic at a miss. Assume a total of \(n\) programs, \(p_1\) to \(p_n\), share the cache. Without loss of generality, let the miss be caused by program \(p_m\).

---

\(^4\)Register or flip-flop based storage allow parallel resets.
The problem is finding the victim at a miss. The preceding algorithm first checks whether there is a cache block that is invalid or valid but unused (access bit is 0). If there is none, it checks if the program that loaded the cache block (pm in this case) is at or over its ration. If both checks are false, then one of the programs pi (where i = 1 : n and i ≠ m) is over its ration. In the general case, the else clause needs to find, in a set, a core that is over its ration. In hardware, this can be done by an associative search and picking the counter with highest overuse of its ration.

6.2.5 Additional Hardware Optimization

In the “1-bit” access bit design, a still-in-use line becomes vulnerable right after the reset. To guard against this vulnerability, we propose to use “multi-bit” counter based access-bits – similar to UNIX page management\(^5\). This will allow the access bit reset algorithm to support a more graceful degradation. At the reference, we set all the bits in the access-bit counter and at a reset point, we decrement it. Once the access-bit counter value becomes zero, only then we indicate that line is currently not being used. This will prevent the immediately accessed lines from becoming victims just after the reset point. As a first step, with a simple 1-bit access bit we are able to protect the ration of most of the application pairs that we ran – except for 1 case in symmetric run and 2 cases in co-run with mcf (out of a total of 48 co-runs). Our

\(^5\)UNIX maintains a 10-bit counter per page. After every non-reference, the value is right shifted; once it becomes zero the page is chosen as victim.
future exploration will consider more robust implementation such as “multi-bit” counter based access-bit and evicted address filters [116].

To reduce the accounting overhead, we can skip the ration counters and the block owner records altogether. In this case, current ownership of a program can be directly computed from the “thread id” info of the tag and count of non-zero access bits, in a given set, for a given thread. This will require an additional adder per cache to compute the ration counter values on-the-fly at the time of new cache block allocation to select the right victim.

6.2.6 Support for Multi-threaded Workloads

In the current setup, we consider only multi-programmed workloads and a simplified approach (with low overhead) to demonstrate the effectiveness of baseline rationing. The ultimate goal of rationing is to evict low reuse data and keep the high reuse data in the caches. Ownership of shared data can be maintained for a group of threads as opposed to individual threads. In a straightforward design, if we have a total of \( n \) threads, the number of ration trackers will increase from current \( C(n, 1) \) to \( \sum C(n, k) \) where \( k \) ranges from 1 to \( n \). For 4 threads, we will require 11 ration trackers as opposed to 4 needed w/o sharing. This will definitely increase the overhead of rationing but the current overhead is only about a percent of total storage. To reduce the overhead, we can always rely on compaction of owner counters and allow false sharing. False sharing should only impact the performance slightly in short bursts.

An optimization to maintaining all sharing combinations, is to have an \( n \)-bit field (similar to a directory bit-vector) for block owner which encodes the identities of owners in a positional vector. Currently we have \( \log n \) bits to remember the block owner’s identity. We can limit the number of ration trackers per set to reduce the overhead. Rationing can be implemented as a simple extension of a baseline directory with modification to replacement policies and we plan to explore it in our future work. going forward thats what we will be exploring. We do model detailed cache coherence protocol in our simulation framework to faithfully simulate multi-threaded workloads which we intend to leverage for future explorations.
6.2.7 Energy Efficient Rationed Caching

Kaxiras et al. have shown that cache accesses exhibit generational behavior [117]. When we access a cache line, typically the first access is a miss, followed by a series of hits. After the last re-use the line is dead and it relies on LRU to evict it even though it can be evicted right after the last use. We corroborate that about 80% of the time a cache line is dead and waits for eviction. A few hardware designs [117–119] have proposed to predict the last reuse of a cache line and then either evict the line immediately or shut off the ways to save leakage power.

We can leverage the cache rationing hardware support to identify dead blocks to save energy. To enable energy efficient cache rationing, we propose to use the ration counter for two purposes. As shown in Figure 6.2, when resetting the access bits, all the ration counters are also reset. During the execution, an access increments the respective ration counter. At the next reset point, if a ration counter is still zero then it indicates that no access was made to this particular set by a given core. This information can be used to evict all the lines that belong to the core with zero ration counter value. With our preliminary analysis we have found that it is safe to use the access pattern of previous reset interval to predict the access behavior of the next reset interval with good accuracy.
6.3 Hardware-Software Collaborative Caching

A number of processors provide special load/store instructions that a program can use to influence cache management. These include the placement hint on Intel Itanium [120], bypassing access on IBM Power series [121], the evict-me bit [115], and non-temporal instructions on IBM and Intel processors [122–124]. Wang et al. called a combined software-hardware solution collaborative caching [115]. We call a special memory instruction a cache hint.

There are two common uses of cache hints. The first is to mark accesses whose data will have no chance of reuse before eviction. This can be done using compiler analysis [115, 120] or reuse-distance profiling [120]. Taking the hints, the hardware would choose not to cache those data and hence save the space for data that may be reused. A second solution was developed recently to use the OPT stack distance (i.e., the OPT stack position) instead of the reuse distance (LRU stack distance) as an indicator of whether a block should be marked for eviction [124]. The benefit of this approach is that it can select a part of the working set to cache if the whole working set is too large. Regardless of what the software does, it needs the hardware instruction in order to mark a data block and tell the hardware to replace it before replacing other blocks. Such an instruction can be readily supported by cache rationing.

We add an MRU-hint bit to load/store instructions. Gu and Ding (2011) defined the bipartite cache, in which each access may be placed in either the highest (LRU) or lowest (MRU) priority stack position [125]. One way to implement this in hardware is with an MRU-hint bit, which instructs the policy to place the data at the bottom of the LRU-MRU stack. In the context of the rationed cache, a set MRU-hint bit should clear the item’s access bit (which is normally set on an access). Software can use this bit to inform the hardware if it knows that the block will have no more cache reuse, or if its eviction would free cache space for other blocks. The block then becomes unused ration and will be favored for immediate eviction (before every block whose access bit is 1).

As an example, consider two cores sharing a four-block cache. Let the access traces be “xyzxyz...” for one core and “abcabc...” for the other. With equal rationing, neither core has enough cache to obtain any reuse. However, with the MRU-hints, the software can free up

---

6 A secondary impact of this policy is energy saving.
Thread 1 | a b c a b c a b c
MRU-Hint Bit | 0 1 0 1 0 1 0 1 0
Access Bit | 1 0 1 0 1 0 1 0 1
Misses | M M M M M M M
--------- | -------------------
Thread 2 | x y z x y z x y z
MRU-Hint Bit | 0 1 0 1 0 1 0 1 0
Access Bit | 1 0 1 0 1 0 1 0 1
Misses | M M M M M M M
--------- | -------------------
Post-Access | x y z x y z x y z
Size 4 Cache | a b c a b c a b c
Stack | x x z z y y x x
      | a a c c b b a a

Figure 6.3: An example of cache rationing with an MRU-hint bit. If the hint bit is set, the access bit is not set so that the accessed blocks will not be kept in the cache. The contents of a size 4 cache are shown after each pair of accesses, and blocks with their access bit zeroed are shown as outlined letters.

cache space by zeroing some access bits. In Figure 6.3, every other access has its MRU-hint bit set, so the access bit is zeroed. In this case, the non-compulsory miss ratio is reduced from 1 to 1/2. In [126], it is shown that a hint-based solution can achieve optimal caching if the hints are given based on whether an optimal (OPT) cache would keep an item between successive accesses. Program-assisted cache management (Pacman) adapts this idea for pre-profiled, solo-run programs [124].

6.4 Comparison with Other Proposals

With the help of simple examples we compare rationing with a few well known policies to point out the relative strength over others.

6.4.1 “Equal Partitioning” and “No Partitioning” Policies

Rationing has two goals: resource protection and cache capacity utilization. We show an example of each case in Figure 6.4. Assume we have two cores, C1 and C2, each accessing a separate set of data, and an evenly rationed cache with two blocks for each core. The figure shows the access trace for each core on the left side. It also shows the content of access bits (1
for each block, 4 total) and ration counters (1 for each core, 2 total) on right side. Not shown is the block owner pointer pointing to one of the two ration counters and owner allocation register. In the interleaved execution, if requests from two cores come at the same time, we assume that the cache prioritizes requests from C1.

The first example, Figure 6.4(a), demonstrates resource protection. In this case, C1 uses 2 blocks, which it can hold entirely within its ration. However, in the no-partitioning policy data from C2 can evict data used by C1. In contrast, the equal partitioned cache and the rationed cache do not permit C2 to intrude on the ration of C1. Due to the lack of protection, the no-partitioning policy causes the most misses (shaded in red). Equal partitioning and rationing perform equally well by providing resource protection. However, the mechanisms to achieve resource protection are different. Rationing permits sharing whereas equal partitioning does not, as the next example shows.

The second example, Figure 6.4(b), illustrates cache utilization. If C1 uses just 1 block, and C2 uses 3, the equal partitioning would under-utilize the partitioned space for C1. The no partitioning and the cache rationing, in contrast, can fully utilize the 4 cache blocks for the 4 program blocks. These two examples demonstrate that cache rationing can indeed combine the advantages of cache partitioning and sharing while avoiding their shortcomings.

### 6.4.2 Promotion/Insertion Pseudo Partitioning (PIPP)

We contrast rationing with recently proposed PIPP design [127] which tries to achieve partitioning with the help of intelligent insertion and promotion policies. Because PIPP does not explicitly and pro-actively partition the cache, it is pseudo-partitioning as the name suggests. The baseline PIPP design works as follows: For n cores, it assumes that there exists a set of target partitions $P = \{p_1, p_2, ..., p_n\}$ such that $\sum p_i = w$, where $w$ is the set associativity of the cache. Baseline PIPP implements three policies. On insertion, core_i simply installs all new incoming lines at priority position $p_i$. On a cache hit, the promotion policy promotes the cache line by a single priority position. Finally, the victim selection always chooses the line from the lowest-priority position – similar to conventional LRU. PIPP does not strictly enforce the target partitioning and does not guarantee resource protection, but the combination of targeted
collaborative caching, as described in Section 6.3, while the other policies cannot. Programs and ensures resource protection. Finally, rationing can support hardware-software collaboration more effectively than PIPP. Most of the cache partitioning proposals do not guarantee resource protection, and the overall speedup may come at the cost of slowing down the less aggressive programs. In contrast, rationing does not slow down less aggressive programs and ensures resource protection. Finally, rationing can support hardware-software collaborative caching, as described in Section 6.3, while the other policies cannot.

In a symmetric co-run, PIPP can effectively reduce the cache capacity and may allow certain dead blocks to occupy the first half of a cache set forever while the other half of the cache experiences more conflict misses. Assume two cores are sharing a 4-way cache. Because C1 and core insert at the same location in PIPP policy, it never gets a chance to remove the stale data before it kicks out the recently accessed data. On the other hand, rationing based policy evicts the dead blocks more effectively than PIPP. Most of the cache partitioning proposals do not guarantee resource protection, and the overall speedup may come at the cost of slowing down the less aggressive programs. In contrast, rationing does not slow down less aggressive programs and ensures resource protection. Finally, rationing can support hardware-software collaborative caching, as described in Section 6.3, while the other policies cannot.

(b) Cache capacity utilization illustration.

Figure 6.4: Illustration of resource protection (a) and capacity utilization (b) in evenly rationed cache in comparison with equal partitioning and no partitioning policies for a 4-way cache shared among 2 cores. Core access patterns are shown in the left and the accesses that experienced misses are shaded in the red.
Figure 6.5: Comparison of cache rationing with baseline PIPP policy. PIPP-equal (PIPP with equal partitioning) for symmetric co-run incurs more misses (shaded in red), and allows a dead line to stay, increasing contention on the second half of the set.

6.4.3 Adaptive Spill-Receive Caches in CMP

Adaptive Spill-Receive (ASR) work uses a global 1-bit per cache to indicate whether a cache is either spiller or receiver (but not both) [130]. A spiller cache requires extra space and a receiver cache can provide extra space in a CMP setup. Proposed technique does not take care of the scenario where certain sets can be spiller and other sets can be receiver within the same cache. In that sense, rationing allows more fine-grain control at the set level and avoids local hotspots. Second, set dueling (a heuristic based on limited sampling in previous interval to predict the behavior in next interval) to detect whether a cache is truly a spiller or receiver does not guarantee the worst case performance loss for receiver cache. Finally, in ASR technique, certain sets may experience double contention – inter-caches and intra-cache. In ASR, 20 out of 1980 applications had an IPC degradation of more than 5% and required additional support to maintain QoS. In contrast, QoS goals are baked into the baseline rationing and does not allow severe performance degradation to begin with. In other words, rationing is more aware of QoS issues and achieves better resource protection.
6.5 Experimental Setup

We perform our experiments using an in-house simulator with true execution-driven, cycle-level simulation. We also model support for multi-program workloads on chip multiprocessors with extensive details in coherence including transient states.

Microarchitecture and Configuration:

The simulator models major microarchitectural components of processor pipeline such as issue queues, register renaming, ROB, and LSQ. Features like load-hit speculation (and scheduling replay), load-store replays, keeping a store miss in the SQ while retiring it from the ROB are all faithfully modeled. Our baseline core is a generic out-of-order microarchitecture loosely modeled after POWER5 [121]. Other details of the configurations are listed in the Table 6.1.

<table>
<thead>
<tr>
<th>Core and Cache Configurations</th>
<th>8 / 4 / 6 / 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch/Decode/Issue/Commit</td>
<td>128</td>
</tr>
<tr>
<td>ROB</td>
<td>INT: 2 + 1 mul + 1 div</td>
</tr>
<tr>
<td>Functional units</td>
<td>FP: 2 + 1 mul + 1 div</td>
</tr>
<tr>
<td>Fetch Q/ Issue Q / Reg. (int,fp)</td>
<td>(32, 32) / (32, 32) / (80, 80)</td>
</tr>
<tr>
<td>LSQ(LQ,SQ)</td>
<td>64 (32,32) 2 search ports</td>
</tr>
<tr>
<td>Branch predictor</td>
<td>Gshare – 8K entries, 13 bit history</td>
</tr>
<tr>
<td>Br. mispred. penalty</td>
<td>at least 7 cycles</td>
</tr>
<tr>
<td>L1 data cache (private)</td>
<td>32KB, 2-way, 64B line, 2 cycles</td>
</tr>
<tr>
<td>L1 inst cache (private)</td>
<td>32KB, 2-way, 64B, 2 cycles</td>
</tr>
<tr>
<td>L2 cache (shared)</td>
<td>512KB/core, 8-way, 64B, 15 cycles</td>
</tr>
<tr>
<td>Memory access latency</td>
<td>150 cycles</td>
</tr>
</tbody>
</table>

Table 6.1: Microarchitectural configurations.

Applications and inputs:

We use applications from SPEC 2000 benchmark suites compiled for Alpha using a cross-compiler based on gcc-4.2.1. We use ref inputs and simulate at least 200 million instructions per program after skipping over the initialization portion as indicated in [131]. If an application completes 200 million instructions it continues to run until the last application completes 200 million instructions to ensure that other co-running applications continue to interfere for the whole run. To compare performance of various cache management policies, we only account
Table 6.2: SPEC 2000 applications used in experiments and their index used in the result figures.

for first 200 million instructions. In our experiments, many applications often go well beyond
one billion instructions. Due to space constraints, in the figures in Section 6.6.2, we use numbers
to represent applications. Corresponding applications are listed in Table 6.2.

Co-run Test Suites:

For $k$ programs, there are $C(m + k − 1, m)$ choices for $m$-program co-runs. We cannot test
them exhaustively so we choose the symmetric tests plus a few groups that require protection
(co-runs with equake and mcf) and offer opportunities for utilization (co-runs with eon). In
particular, we present results for four pair-run test suites:

- **SPEC 2000 symmetric**: each program co-runs with itself.
- **SPEC 2000 with eon/equake/mcf**: either eon or equake or mcf co-runs with each of SPEC
  2000 programs including itself.

We also create 2 four-program co-run test suites:

- **SPEC 2000, symmetric**: each of the 26 programs co-runs with three clones of itself.
- **SPEC 2000 with 2 equake**: for each of the SPEC 2000 programs, we run two clones of
  the program and two of equake.

Comparisons:

We compare rationing with equal partitioning, no partitioning, and a flavor of pseudo-
partitioning (PIPP). For PIPP policy, we assign the partition equally in a 2 program co-run
and call it **PIPP-equal**. This is a reasonable design choice for symmetric co-run because the

<table>
<thead>
<tr>
<th>SPEC 2000 Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>INT</strong></td>
</tr>
</tbody>
</table>
two copies of program are identical. For asymmetric co-runs, PIPP will require additional pro-
filig to estimate the right partition that is typically based on working set information of each
program. In asymmetric co-runs, because for rationing we use equal rations, we use equal parti-
tions for PIPP as well. Our primary goal is equal resource allocation, so it is intuitive to allocate
equal partitions and rations for PIPP and rationing policies, respectively. As we will see later,
the correct “initial” partition is not critical for the success of rationing whereas for PIPP it is
quite crucial to allocate the right partition to begin with.

There are many, often more elaborate techniques. We pick these three policies to compare
with rationing policy because they are simple and representative. On average, PIPP has been
shown to perform better than other proposals (e.g., UCP and TADIP [127]). The equal par-
titioning and no partitioning policies suit our purposes because they are the extreme cases of
protection and utilization. If rationing can have similar protection as equal partitioning and uti-
lization as no partitioning, then there is little room for further improvement. Furthermore, past
techniques target throughput and fairness, which are different from protection and utilization.
To recap, we compare with the best possible method for each single purpose, as represented by
the combined strength of these two extreme policies along with the PIPP technique.

6.5.1 Rationing Overhead

Let \( p \) be the number of cores sharing the cache, \( s \) be the number of sets in the cache, \( w \) be the
number of ways in each set, and \( b \) be the cache block size in bytes. The total number of bits
required for the data array can be computed from the expression (1):

\[
\text{size(data\_array)} = s * w * b * 8
\]  

(6.1)

Assuming a 40-bit physical address, the tag storage in a physically indexed cache will be:

\[
\text{size(tag\_array)} = s * w * (40 - \log_2 s - \log_2 b + 2)
\]  

(6.2)

2-bit (last factor in expression (2)) storage is required to maintain the valid and dirty bits for
each block assuming a writeback cache.
To implement cache rationing on top of baseline cache architecture, we require 1-bit of storage for the access-bit per cache block, $p$ ration counters per set, $p$ owner allocation registers per set and $w$ block owner per set. Each ration counter needs to count only up to the associativity of the cache whereas block owner has to store upto $\log_2 p$ bits. The total storage overhead of the rationed cache is:

$$\text{size}(\text{ration\_array}) = s \ast (w \ast (1 + \log_2 p) + p \ast 2\log_2 w)$$  \hspace{1cm} (6.3)

After computing the storage overhead (from expression (3)) for various configurations of caches and numbers of cores, three key observations about the storage overhead are as follows:

- Storage overhead is *almost constant* for any cache size if the associativity and number of cores sharing the cache are fixed. For 2 cores sharing an 8-way associative cache of any size, the storage overhead is 0.3% of the total cache storage.

- Storage overhead *increases slightly with the number of cores* for a given cache. For a 16-way, 1 MB cache, storage overhead for 2 cores is 0.26% and increases slightly to 0.94% for 16 cores.

- Storage overhead *decreases slightly with the associativity* for a fixed number of cores and fixed size cache. 1 MB cache shared among 2 cores has an overhead of 0.32% when it is 2-way associative. The same cache with 16-way associativity has 0.26% overhead.

The last property of rationing overhead is important because last level caches tend to be highly associative and are often shared among multiple programs/cores. *A highly associative cache amortizes the storage overhead of rationing.* To conclude, the overall storage overhead of rationing is $< 1\%$ of the total cache storage.
6.6 Experimental Analysis

6.6.1 Measure of Protection and Utilization

In this section, we identify a set of performance markers to quantify the level of resource protection and utilization, the two goals defined in Section 6.2.1. We will use them here to evaluate rationing and other cache-sharing techniques.

The following defines the solo-run performance. In this paper, performance is quantified by instructions per cycle (IPC).

- **Baseline**: The baseline is the performance when running solo on a single-core with the size of cache equal to the ration. It is the performance without sharing and the desired lower bound with sharing.

- **Maximum gain**: Maximum gain is the performance of an application running solo using 100% of the (shared) cache.

Next we define the performance in a co-run group.

- **Unhealthy co-run**: A co-run group is unhealthy if one or more of its members loses performance beyond a certain threshold. In this sense, we consider slowdown to be “damage”. We set the damage threshold to be 1%. The damage of an unhealthy co-run is the worst slowdown among its group members.

For co-run groups that are not unhealthy, we further divide them into two groups.

- **Healthy co-run**: A co-run group is healthy if no member sees a degradation and if one or more members see a significant benefit from cache sharing. We set the benefit threshold to be 1% over the baseline. The benefit of a healthy co-run is the highest gain by a group member.

- **Neutral co-run**: If a co-run is neither healthy nor unhealthy, it is neutral, meaning that running in a group does not have a significant impact one way or the other.
Finally we have the metrics for protection and utilization. Given a test suite, which is a set of co-run groups, we evaluate a cache-sharing technique as follows:

- **Level of protection**: A low number of unhealthy co-runs and low average damage inflicted in these co-runs indicates a high level of protection.

- **Level of utilization**: Indicated by the number of healthy co-runs and the average benefit enjoyed in these co-runs.

### 6.6.2 Overall Performance

We first summarize the overall resource protection and cache utilization results of various co-runs in Table 6.3 before analyzing individual performance gains and slowdowns. First observation we make is that even though no partitioning achieves good performance gains but these gains come at the cost of slowing down a large number of applications (forth column in the upper half of Table 6.3). In contrast, rationing performs very similar to hard partitioning from resource protection point of view (second and sixth column in the upper half of Table 6.3). Finally, whenever there is an opportunity for sharing, rationing beats no-partitioning consistently due to its non-intrusive nature (third and fifth column in the lower half of Table 6.3). PIPP performs slightly better compared to no partitioning but does not guarantee resource protection and slows down a good number of co-running applications (second last column in the Table 6.3).

**Symmetric co-run suites:**

Figure 6.6 (upper) shows the performance for the four symmetrical co-run suites. As shown (and numbered) in Table 6.2, the left side applications are integer applications and the right ones are floating point applications. Identical programs tend to cause greater contention for the same part of the shared cache due to similar access pattern. We first discuss symmetric co-run results and then elaborate with the performance improvement after adding per-program address offsets (to avoid hotspots).

Due to strong interference, the no-partitioning policy sees 15 damaged pairs out of 26 for 2-core co-run system, and 13 damaged quartets out of 26 for 4-core co-run system, shown
Table 6.3: Overall protection and utilization for all test suites.

<table>
<thead>
<tr>
<th>Co-runs</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Equal-Partitioning</td>
<td>No-Partitioning</td>
<td>Rationing</td>
<td>PIPP-equal</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Damaged pairs</td>
<td>Average slowdown</td>
<td>Damaged pairs</td>
<td>Average slowdown</td>
<td>Damaged pairs</td>
</tr>
<tr>
<td>spec2k - symmetric</td>
<td>1</td>
<td>1.03%</td>
<td>15</td>
<td>3.34%</td>
<td>1</td>
</tr>
<tr>
<td>spec2k - w/ eon</td>
<td>0</td>
<td>0.0 %</td>
<td>9</td>
<td>5.65 %</td>
<td>0</td>
</tr>
<tr>
<td>spec2k - w/ eqk</td>
<td>1</td>
<td>1.05%</td>
<td>17</td>
<td>9.76%</td>
<td>3</td>
</tr>
<tr>
<td>spec2k - w/ mcf</td>
<td>0</td>
<td>0.0 %</td>
<td>24</td>
<td>10.54 %</td>
<td>2</td>
</tr>
</tbody>
</table>

Rationing protects most pairs from damage. It has 1 damaged pair out of 26, and 1 damaged quartets out of 26. The protection of equal partitioning is similar, which has 1 each in 2-core and 4-core system. The reason for damage in the equally partitioned cache is the number of memory ports. In both the (solo) baseline and the partitioned co-run, the number of memory ports are same, which is 2. In a few cases, the contention causes slight damage.

The protection of rationing is almost as absolute as that of equal partitioning. The average damage of rationing is similar, 1.01% vs 1.03% (2-core), and 1.14% vs 1.07% (4-core). In comparison, the average damages by the no partitioning for the two co-runs are 3.34% (2-core) and 4.52% (4-core) respectively.

For lack of space, Figure 6.6 (lower) shows only the individual results for the no partitioning and rationed caches. Here the performance is normalized to the equally partitioned cache performance. We see 3 (no partitioning), and 2 (rationing) cases of benefits, including a large gain. In 4-program co-runs, sharing and rationing improve the pair of apsi sharply by 1.35x and 1.45x. This is because its working set is around 550 KB, close to a quarter of the cache size and extremely sensitive to any slight increase in space (from sharing of unused ration).
Figure 6.6: SPEC 2000 symmetric co-runs of 2- (upper) and 4- (lower) programs. In the 2-program co-runs, IPCs are normalized to respective solo runs with half the cache. In the 4-program co-runs, IPCs are normalized to equal partitioning. Co-running applications (index to name mapping from Table 6.2) are adjacent to each other (shown in the same shade) and different co-runs are separated by alternating shades.

When two copy of same program share a cache their access patterns are almost similar – creating hotspots in certain sets. The V-way cache [132] is a hardware solution to relieve the contention (in a solo run) by assigning data blocks away from the contended area. Since our problem is specific, we use a simple solution. The idea is to add a clone-specific offset to the set mapping. We use a prime number series and call them offsets, one for each of the $p$ cores. The modified set mapping is:

\[
\text{set} = (\text{set} + \text{offset}[\text{pid}]) \mod \text{num\_sets}
\]

After the above modification, we see significant improvements in three application pairs for rationed cache. \texttt{apsi} improves by 80\%, \texttt{perlbmk} improves by 4.3\% and \texttt{ammp} improves by 2.8\%. However \texttt{art} and \texttt{lucas} slightly degrade (1\%). The no-partitioning policy sees a similar improvement for \texttt{apsi}, 3\% for \texttt{perlbmk} and a slight slowdown (-0.7\%) for \texttt{ammp}. 

\[
\text{(set + offset[pid])} \mod \text{num\_sets}
\]
Co-run with low cache demand peer (eon):

eon’s working set is smaller than 128KB. Co-run programs can benefit from unused cache if the cache is rationed or shared. Figure 6.7 shows the performance for the eon pair-run test suite. On average, the no-partitioning policy speeds up the peer by 9.38% but slows down eon by 5.65%. In comparison, rationing does not slow down eon while it achieves 14.46% speedup for other applications.

The damage to eon shows that sharing can lead to interference even for programs with very small working sets. By removing the damage, rationing shows the benefit of donating the otherwise unused portion of a ration while protecting what is being used by the ration’s owner.

Sharing with highly disruptive peer (mcf):

When an application shares cache with a highly disruptive peer such as mcf, the best policy is to confine the disruptive application using hard partition. From Figure 6.8, it seems that rationing improves over no partitioning but still is not as good as hard equal partitioning. The inability to fully protect happens only for mcf among the total of 46 programs (out of all the SPEC 2000 and 2006 programs we studied).
6.6.3 Diagnostic Analysis

To understand rationing better, we conduct additional experiments that shed more light on the inner working of the rationing mechanism. Cache rationing supports other optimizations too with little additional cost. We briefly discuss some of them here.

Access-bit reset interval:

We experimented with a range of resetting intervals for the access bit. We observed that if the resets are too frequent, sharing dominates protection. For example, if we reset access-bits at every cache access, rationing becomes no partitioning. On the other hand, if resets are too infrequent, protection dominates sharing. If we never reset, rationing becomes a type of hard partitioning. Figure 6.9 shows reset intervals from 1 instruction to infinite (never reset). For each case, the performance is measured by the number of healthy and unhealthy pairs (out of 26). The change in these numbers confirms the reasoning above. The general trend shows that keeping the reset interval around 50K to 200K instructions is a good choice because it minimizes the number of damaged pairs and maximizes the pairs that see damage free gains.
Figure 6.9: Access-bit reset interval sensitivity for *equake* pair-run.

**Coarse-grain rationing:**

So far we have considered only fine-grain rationing enforced at each cache set level. Per-set rationing enables fine-grained protection, but it incurs a higher storage overhead. A simpler and more storage efficient alternative is to maintain a coarse-grain rationing with a single set of ration counters at whole cache level. The counter consolidates all the usage of a particular core. Coarse-grained rationing can victimize a less aggressive application in the sets in which its data occupies all cache ways. The overall storage overhead for the coarse-grain rationing is 0.019% – assuming 1-bit access-bit per 64 bytes cache line and per-core 14-bit ration counters for 1MB L2 cache.

Figure 6.10: Comparison of fine-grained (per set) and coarse-grained (global) rationing for symmetric co-runs.

Figure 6.10 shows the performance of two rationing techniques for symmetric co-run. By
and large, the results are similar for the two versions of rationing. For the sake of brevity we only show symmetric co-runs. As we can see, the impact of relaxing the per-set precision is not that bad; with cache level rationing, we are able to achieve resource protection in nearly all cases – except one.

### 6.6.4 Hardware-software Collaborative Caching

We demonstrate two usages of software control collaborative caching enabled by the rationing hardware support.

**Improved protection:**

We make a hypothetical study and assume that software has complete knowledge of program locality, in particular, the forward reuse distance distribution for every memory reference. If a memory reference tends to make accesses with reuse distance greater than the cache size, we mark it with the evict-me bit, so the rationed cache would clear the access bit for the accessed data and force them to be removed first.

In preliminary testing, we found that a program pair benefits significantly from the software hint. Figure 6.11 compares the *equake-lucas* co-run performance under free-for-all sharing, rationing, and collaborative rationing (with software hints). *lucas* has a large working set of around 2MB, much of which has zero reuse in the 1MB cache. This means that even if *lucas* brings the data into the cache we should place it at the bottom of the LRU stack. When *lucas* is paired with *equake*, because of the aggressive nature of *lucas*, *equake* suffers badly under the unpartitioned policy. *lucas* slows down by only 1%, whereas *equake* slows down by 17%. Rationing reduces the slowdown of *equake* to around 5%. Collaborative rationing further improves by marking some *lucas* references with the software hint. With this, slowdown of *equake* becomes negligible (0.4%) and *lucas* still improves by 1%.

**Optimal caching:**

The Pacman system profiles programs’ memory access traces to formulate a prediction for each block as to whether it will be kept in an optimal (OPT) cache of a given size from one access
Figure 6.11: *equake* and *lucas* co-run. Collaborative rationing obtains full protection while the hardware-only rationing does not.

to the next. If it is predicted to be removed before reuse, the MRU-hint bit is set, so that the block is placed at the bottom of the LRU-MRU stack, and is first in line for eviction. In this way, Pacman emulates the LRU-MRU eviction policy that [126] showed could be optimal.

We have tested an extension of this policy for multiple cache-sharing programs. For each program, the determination of the hint bit is based on its ration size $r$. In other words, the MRU-hint bit is set for a cached item when profiling data indicate that the OPT policy would not keep it in a cache of size $r$.

Figure 6.12 compares Pacman *cache rationing* to Pacman cache partitioning with a streaming application and a successive over-relaxation (SOR) program, which has been used to solve linear systems of equations. The program traces were interleaved with 4 SOR references for each Streaming reference, but the results are nearly identical for 1:1 or 1:4 interleaving. The cache was rationed between SOR and Streaming in ratios from 10%:90% to 90%:10%.

In all tests, rationing shows only very modest improvement over partitioning for Stream. For SOR, however, miss ratios are nearly halved for some cache sizes, almost to the miss ratio for having the whole cache. In this range of cache sizes, Pacman with rationing is approximately as effective as LRU with partitioning, but in cases where Pacman outperforms LRU (see [124]) Pacman will likely outperform LRU with either partitioning or sharing.

This example demonstrates the two potential advantages of Pacman cache rationing to manage program co-run: safety and sharing. Co-run programs will not encroach on each other’s
Figure 6.12: Pacman miss ratios for cache rationed and cache partitioned simulations of Streaming and SOR. Partitioned-cache points are plotted for cache sizes of 0.125MB, 0.25MB, 0.5MB, and 1MB. For rationed-cache points, the horizontal axis represents the cache ration as a percentage of the total 1MB cache.

rations, but a program can still utilize the available cache space when it is not being used or rationed by other programs.

6.7 Related Work

6.7.1 Cache Partitioning

There has been a plethora of work to efficiently partition the cache among multiple threads and programs [128, 133–141]. Mainly there are two ways to partition the cache. The first is to divide the ways in each cache set [142]. The second is to place cache data with OS supported page coloring [143]. Page coloring requires software support, while set-based partitioning may be controlled in either software or hardware. In both cases, fixed cache partitioning cannot guarantee full utilization, since a program may not fully utilize its partition.

The problem of fixed partitioning can be ameliorated through dynamic partitioning. One
solution is to label each task with a quota, and let the OS partition the cache space based on the demand of all running programs [144]. If there is only one program running, it will have the entire cache regardless of the actual quota it declares. When a task arrives or leaves, or its quota is changed, the OS will dynamically adjust the cache partitions.

Even in dynamic partitioning, the actual demand of a task may not match the space assigned to it. A program can be given a large space (by declaring a large quota) but uses only a fraction of it. Incomplete utilization would still ensue.

Cache rationing is a type of dynamic allocation like quota partitioning. Unlike fixed partitioning, the ration is “soft” in that the unused ration of one program could be given to other programs. The support happens in hardware which is more responsive and may change at each cache access. In comparison, a quota is managed by the OS and does not change without OS intervention.

Two recent papers have proposed different forms of baseline partitioning. Natural and equal baseline partitioning minimize the miss count for a group of programs with the constraint that no program can have a higher predicted miss ratio than it would with free-for-all sharing (natural baseline), or with equal partitioning (equal baseline) [145]. The elastic miss-ratio baseline (EMB) and elastic cache-allocation baseline (ECB) each allow individual programs to suffer some percentage increase in predicted miss ratio (EMB) or decrease in cache allocation (ECB), over an equal partitioning [146].

Like rationing, baseline partitioning guarantees each cache sharer some minimum performance, and optimizes cache allocation under this constraint. Unlike hardware rationing, baseline partitioning is fixed; a program cannot use its full partition in one phase and donate some of it in the next.

Cook et al. propose a cache way partitioning to avoid degradation and maintain responsiveness of interactive foreground applications that share cache resources with simultaneously running background applications [147]. While similar in goals, there are important differences between these two approaches. First, rationing can be enforced and successfully achieves resource protection at much smaller shared cache (1MB) whereas Cook et al. use LLC of 6MB size. Most of the applications that we studied have bigger footprint from 1MB and yet rationing
limits slowdown to a very few co-runs whereas in way partitioning worst case slowdown is still 7% with an average slowdown of 2%. Unlike rationing, way partitioning does not allow one process to use the part of cache that is currently not being used by another process.

6.7.2 Cache Management

Many techniques have been proposed to improve caching effectiveness by choosing victims more carefully, for example by detecting streaming accesses. In particular, many studies focus on shared caches under multi-programming workloads. For instance, restricting memory-intensive threads [148] or giving more space to programs that can best use the additional space to reduce misses [128]. Other factors such as memory-level parallelism can also be factored in so that the heuristics can directly target performance gain [133–135]. Such management can also take on a temporal dimension [136, 137] or try to improve fairness [138]. Adaptive insertion policies have also been studied to share the cache more effectively among multiple threads/applications [129, 149]. Jaleel et al. proposed Re-Reference Interval Prediction (RRIP), an extension of the NRU policy that stores M bits for each block, indicating the expected nearness of its next reuse. Thread-Aware Dynamic RRIP (TA-DRRIP) employs Set Dueling [129] for each thread to select either RRIP or a variant of it which gives most blocks a distant reuse interval prediction and only a few blocks a slightly lower reuse interval prediction [150]. Jimenez et al. proposed an algorithm that is based on PseudoLRU but uses set-dueling to dynamically adapt its insertion and promotion policy [151].

Some proposals try to manage cache by set-pinning [152]. In the contexts of CMP there have been a few proposals [153–156]. A few techniques for improving QoS in CMP environments with large last level cache have also been proposed [157–159]. Zhang et al. proposed a software framework that manages multicore resources via controlled hardware execution throttling on selected CPU cores [160].

The aforementioned techniques are adaptive but all use heuristics. Because of the nature of heuristics, e.g., predicting low-locality data, they may be wrong and hence counterproductive in some cases. The occasional loss is permitted as long as the overall throughput or eventual fairness is improved. For throughput, performance loss in a minority of tasks is permitted if it is
compensated by a higher gain by others. For fairness, a task can lose performance temporarily, even frequently, if the loss is canceled by sufficient gains in other periods of execution. These solutions emphasize dynamic feedback and control.

Cache rationing has a different goal, which is to ensure resource protection in all cases at all times. It does not try to discern between high and low-locality data. Instead, its hardware support protects against interference by all peer cache accesses. Rationing is dynamic in the detection of an unused cache block. For this purpose, it adds an access bit and revises the replacement logic based on that access bit. The goal is not overall throughput but the so-called Pareto optimality where we cannot make one program better off without making another program worse off.

### 6.7.3 Collaborative Caching

Previous hints included placement, bypassing, or evict-me and were designed for a solo-use or unpartitioned shared cache [115, 120, 121]. They may be used in a partitioned cache, partitioned either by cache or page coloring. However, this possibility has not been studied or evaluated. Moreover, partitioning and collaboration use separate mechanisms. In cache rationing, we show an integrated design – the access bit provides both the rationing mechanism and the hint mechanism. We believe that this is the first design to combine the two mechanisms. The integration expands the scope of software-hardware collaboration to include not just ration utilization of individual programs but also better ration protection between programs, as shown in the evaluation section by the _equake-lucas_ pair run (Figure 6.11).

### 6.7.4 Resource Allocation

McCann et al. proposed _equipartitioning_, which offers an equal number of processors to each job, but does not give any job more than it has use for (that is, a job is given a number of processors up to the number of threads it has). Dynamic equipartitioning allows online reassignment of processors to different jobs [161]. Similarly, cache rationing allocates cache to programs, but not more to a single program than it needs (when another program could use it).
6.8 Summary

In this chapter we presented a self tuning based rationing technique to manage shared caches. The new cache hardware protects the cache ration of each program and at the same time finds unused ration to share among the co-run programs. A core is allowed to use part of another core’s ration only if/when the ration is not being fully used by the designated owner. We show that the new support can be added on top of existing cache architecture with minimal additional hardware and scales well with the cache size, number of cores and the cache associativity. When a program does not use all its ration, rationing achieves good utilization similar to no partitioning. When a program exerts strong interference, rationing provides good protection similar to hard partitioned cache.

In a co-run with high cache demand application mcf, “no partitioning” slows down 24 programs (out of 26 co-runs) with an average slowdown of 10.5% compared to solo run with half the cache. In contrast, rationing restricts any performance loss to 2 applications only with an average of 1.8%. With a low cache demand application eon, rationing speeds up 14 applications by an average of 14.5%. On the other hand, “no partitioning” speeds up only 7 programs with an average of 9.4% and “equal partitioning” speeds up no applications. In addition, rationing provides an integrated design for cache sharing, software-hardware collaboration and energy-efficient caching.
Chapter 7

Summary and Future Work

In this chapter, we summarize the contributions of this thesis along with potential impact of our explorations of intelligent look-ahead schemes and also present directions for future work.

7.1 Summary

With the significant slowdown in processor clock and microarchitectural improvement, single-thread performance is lacking two significant traditional drivers. Yet, it remains a key processor design goal as it offers across-the-board benefit without requiring programmers’ intervention. We believe that the look-ahead strategy still has significant potential and worth further researching. Due to the proliferation and ubiquity of the multi-core architectures, a more decoupled look-ahead architecture is a serious candidate for performance boosting.

Recent microarchitectures (e.g., Apple’s A9, ARM’s Ares) are already employing very large monolithic re-order buffers (~200-entry), wider pipelines to sustain high single-thread performance – beyond 1200 specint score. The next big jump in single-thread performance can easily be achieved by decoupling and smarter look-ahead techniques, as it becomes harder for a monolithic core to keep up the pace. Executing dedicated code to run ahead of the main program execution to help extract implicit parallelism better is a promising approach and largely been overlooked so far. While the decoupled look-ahead technique has shown significant performance benefits, the look-ahead thread has often become the new speed limit. Fortunately, without hard correctness constraints, there are more opportunities to solve the problem.
7.1.1 Speculative Parallelization in Decoupled Look-ahead

We have proposed a mechanism to apply speculative parallelization to the look-ahead thread. This approach is motivated by two intuitions: the look-ahead code contains fewer dependences, thus lends itself to (speculative) parallelization; without correctness constraints, hardware support for speculative parallelization in the look-ahead thread can be much less demanding. We have presented a software mechanism to probabilistically extract parallelism and shown that indeed the look-ahead code affords more opportunities. We have also presented a hardware design that does not contain all the support needed for conventional speculative parallelization (i.e., dependence tracking, complex versioning, etc.). For an array of 14 applications, where the speed of the look-ahead thread is the bottleneck, the proposed mechanism speeds up the baseline decoupled look-ahead system by up to 1.39x with a geometric mean of 1.13x. We also propose to extend the current framework with a credit based mechanism which intends to improve the coverage by considering more parallelization opportunities that were missed out. Experimental data suggest that there is further performance potential to be extracted.

7.1.2 Weak Dependence Removal in Decoupled Look-ahead

We have presented a case for exploiting weak dependences in the look-ahead thread. Instructions that contribute marginally to the purpose of the look-ahead can be removed to lighten the load and accelerate the look-ahead thread and, in turn, the overall system. Through a self-tuning framework based on the genetic algorithm, we can automatically tune the look-ahead thread to remove the weak dependences. We have shown that for a group of 14 applications, limited by look-ahead thread’s speed, this framework achieves up to 1.48x speedup with a geometric mean of 1.14x. Under the new environment, using two cores, a decoupled look-ahead system achieves 1.58x speedup over a single-core baseline, making it a very effective performance boosting technique. Beyond current results, fast and deep look-ahead will be an essential element in uncovering more of the significant implicit parallelism in general-purpose codes in the near future.
7.1.3 Look-ahead Skeleton Tuning and Do-It-Yourself Branches

Extending the idea of self-tuning further, we propose two orthogonal techniques to balance look-ahead thread’s workload thereby improving overall performance. First, we use a static, profile-driven technique to tune skeleton for various code regions. Second, we accelerate sluggish look-ahead by skipping branch based, side-effect free code modules that do not contribute to the effectiveness of look-ahead. We call them Do-It-Yourself or DIY branches for which the main thread does not get any help from the look-ahead thread, instead relies on its own predictor and prefetcher.

Assisted by these techniques, look-ahead thread propels ahead and provides performance-critical assistance in efficient manner to improve the overall performance of decoupled look-ahead by upto 2.12x with a geomean of 1.20x. As a result of 14% reduction in skeleton size and 20% performance improvement, we reduce the energy of decoupled look-ahead system by 17%. Alternatively, smaller skeleton also enables simplification of look-ahead core pipeline to reduce power overhead of baseline look-ahead system from 1.53x to 1.38x with virtually no degradation in performance. A distilled skeleton, by three aforementioned techniques, paves the way for a single-core SMT based implementation of decoupled look-ahead system.

7.1.4 Self Tuning (Rationing) in Shared Cache Management

Finally, we present one form of self-tuning (rationing) for shared cache management to achieve full protection and good cache utilization. The new cache hardware protects the cache ration of each program and at the same time finds unused ration to share among the co-run programs. A core is allowed to use another core’s ration only if/when the ration is not being used. Rationing support can be added on top of existing cache architecture with minimal additional hardware and scales well with the cache size, number of cores and the cache associativity. When an application does not use all its ration, rationing achieves good utilization similar to “free-for-all” shared cache. When a program exerts strong interference, rationing provides good protection similar to partitioned cache. In addition, rationing provides an integrated design for cache sharing, energy-efficient caching and software-hardware collaboration.
7.2 Future Explorations

Our current framework only uses the decoupled look-ahead technique to accelerate a compiled program binary, and mostly by reducing branch mispredictions and cache misses. However, the principle of look-ahead can have a much broader application. For instance, future information from look-ahead can be used to promote better speculative parallelization or can help accelerate the execution of interpreted programs.

There are many ideas to pursue in the look-ahead paradigm. We list some of the interesting avenues for future explorations in this section.

7.2.1 SMT based Single-core Decoupled Look-ahead

In hindsight, we realized that our current way of maintaining skeleton (as a mask on program binary) has prevented us from simplifying some aspects of look-ahead core pipeline to reduce the complexity. In modern high-end microarchitectures, running sequential codes (e.g., specint), front-end presents significant bottlenecks. Even though when we distilled the skeleton, we never benefited from reduced fetch bandwidth demand because masked instructions in skeleton still consumed fetch bandwidth. Recall, it was only in the decode stage when we dropped the skeleton instructions that were NOPs. Alternatively, if we can store skeleton as a separate binary then fetch bandwidth requirement for skeleton will reduce significantly. As shown in Section 5.7.2, distilled skeletons can sustain good look-ahead quality running on a 2-wide\(^1\) out-of-order core to feed a 4-wide out-of-order main core. This simplification of look-ahead core will not only reduce the cycle time (therefore, enable higher operating clock frequency for look-ahead core), but will bring significant savings in terms of power, energy and area.

The current version of decoupled look-ahead system uses a separate core to perform the look-ahead. If an SMT version (as opposed to multicore version) can deliver a significant fraction of the look-ahead effect, it will be a very attractive design paradigm for near-term products. In our exploration, the look-ahead thread is completely speculative and advisory. From the analysis of computations and values produced in the look-ahead thread we observed

\(^1\)For these experiments, we reduced only the back-end of the core pipeline and kept the front-end to 8-wide.
that about 90% of the values produced in the look-ahead thread are correct. So in theory, it may be possible to prove that a strand of computation in look-ahead is correct and can be directly committed to main thread. This kind of computational strand integration will save on the total throughput needed for the decoupled look-ahead. We envision that a skeleton with one-third size of original program can be executed in a SMT core to reap good benefits of look-ahead scheme. Almost all of spec-fp skeletons are in the range of 30-40% of original program binary after applying various distillation mechanisms that we explored in this thesis.

7.2.2 Machine Learning based Self-Tuning Look-ahead Process

Removing weak instructions from look-ahead thread speeds up overall decoupled look-ahead system significantly. However, identifying weak instructions from a pool of 10s of thousands of static instructions by conducting individual fitness tests is rather cumbersome and resource intensive. As an alternative, we propose to explore and employ a machine learning based prediction mechanism to identify “potential” weak instructions across programs. Once “potential” weak instructions are identified they can be tested and verified in a more controlled environment and combined together to make the overall look-ahead system faster and efficient.

Various classifiers and predictors have been proposed in machine learning which fall under two categories: Parametric based and non-parametric based. In parametric based techniques some of the parameters of classifier are already known thus they require very few training examples to make reasonable predictions. On the other hand, a non-parametric based technique requires relatively larger number of training examples to learn the parameters by trail-and-error. Non-parametric classifiers also tend to be more generic and avoid the pitfalls of overfitting. While Parametric based techniques are more intuitive to understand the underlying actions and mechanism they tend to be application specific.

\[ h \in H \] overfits training set \( S \) if there exists \( \tilde{h} \in H \) that has higher training set error but lower test error on new data points.
Problem Formulation:

The problem of inducing general functions from specific training examples is central of learning. In our case, finding weak instructions in a program fulfills the basic requirements of a machine learning problem very well. In this section, we formulate the problem of predicting weak instructions using standard notations and conventions.

Assume, set of all instructions among all applications is $X$. An individual instruction is denoted by $x$ such that $x \in X$. The concept or the function to be learned is called target concept which is denoted by $c$. In general $c$ can be any Boolean-valued functions defined over the instructions $X$ such that $c : X \rightarrow \{0, 1\}$. In our case, the target concept corresponds to the nature of instructions whether it is weak or strong. In other words, $c(x) = 1$ if instruction is weak and $c(x) = 0$ if instruction is not weak. Specific training examples (with positive and negative outcome) are available in our case which can be obtained by conducting individual fitness tests. Finally, given the training examples of the target concept $c$, the problem faced by learning system is to hypothesize, or estimate $c$. If we assume $H$ is the set of all possible hypotheses then each hypothesis $h$ in $H$ represents a Boolean-valued function defined over $X$; that is, $h : X \rightarrow \{0, 1\}$. The ultimate goal of the learning system is to find a hypothesis $h$ such that $h(x) = c(x)$ for all $x$ in $X$.

Choices for Learning Algorithm:

It is important to note that by selecting a hypothesis representation, we implicitly define the space of all hypotheses that algorithm can ever represent and therefore can ever learn. Decision tree based Random Forest methods have shown good accuracy in learning the complex hidden pattern in real problems, therefore, we propose them as initial candidates to build the classifier.

Decision tree based learning is one of the simplest and most intuitive method which is heavily used in real life applications [162]. The goal is to create a simple tree model in which the leaves are the final outcome and internal nodes are the input variables and edges are the possible values of each input variable. Initially, a few examples are used to train the classifiers and once it has recognized various parameters it can be used to predict the outcome of the inputs which it has not encountered in the past.
Figure 7.1: Random forest based learning framework to identify and predict weak instructions. Output of the model is a Boolean information whether the instruction is weak or strong. Input to the model is instruction under test, its attributes along with limited context information in which the instruction exists.

Random Forest based learning method is based on decision tree learning. Instead of using just one decision tree to make the final prediction, Random Forest uses set of trees (typically in the range of 10s to 100s) to make the decision [163, 164]. This classifier suits a system where we have large number of parallel execution resources such as a large cluster. The final prediction outcome is the mode (most commonly observed value of a variable in a given population) of the outcomes from individual trees. Random Forest based learning has shown to recognize very complex and hidden patterns in the data. It is also computationally efficient and can be employed in a large scale cluster because each tree is independent from each other.

**Challenges in Mapping:**

There are a few challenges in mapping the problem of weak instruction prediction to current learning methods. Decision tree and Random forest both require well defined set of input variables and their attributes. In decision tree method, we need to know the relative importance of the input attributes so that most important attribute can be mapped to the root of the tree. However, from the weak instruction study we don’t know the *exact* input variables and their relative importance. One thing we do know is that relative expected values of operands play an
important role in making an instruction either weak or strong. Other attributes such as number of operands per instructions, type of instructions are of less importance. Surrounding instructions and their operands’ expected values also play major role in making an instruction either weak or strong. Due to this limitation we would prefer to map our problem to an algorithm which tries to extract the interplay among numerous input variables and a context (in this case surrounding instructions or a computation strand itself) without relying on specific inputs and explicit information related to inputs.

7.2.3 Cost Effective Heuristics for Skeletons

In the baseline look-ahead, we observed that if we treat all the branches as default member, and do the backward dependence analysis, the average size of skeleton reaches to about 70% of the original program. Skeletons of this size are inherently slow to execute, thus we propose to explore a range of static, heuristic-based techniques to construct the skeletons of different sizes to achieve varying degree of coverage and speed for the look-ahead thread. In particular, we treat the problem as knapsack optimization problem.

We treat a branch and the branches on which it is control dependent upon as a group (known as knapsack item) and construct the skeletons for each group. We sort the groups according to their costs-benefit ratio. Cost can be approximated by the number of instructions to be executed and benefit the number of mispredictions covered by the group. Finally, we incrementally build the skeleton by picking the branch groups which have better cost-benefit ratio and keep picking until we achieve the pre-defined skeleton size.

7.2.4 Value Prediction Using Look-ahead Thread

We observed that about 90% of the register values produced in the look-ahead thread are correct. These values can be communicated to the main thread via a queue and can be used as value prediction hints which can avoid the computation duplication.
7.3 Closing Remarks: Potential Impact

For the past a dozen years or so, main-stream microprocessors have been increasing the number of cores packaged in a chip. The trend started earlier and is more obvious in server chips than those used in desktops or mobile devices. The increase directly benefits system throughput when there is enough workload. For a single program, the case is less straightforward. Some programs are easier to parallelize, others are more difficult. New tools and programming models constantly emerge to make it easier. Increased education efforts promise to produce engineers more experienced for developing parallel programs. But at the end of the day, parallel codes – especially efficient ones – take more effort to write, debug, and maintain.

Even when a program has already been parallelized, the parallel efficiency tends to drop as the number of threads increases. In other words, increasing the number of cores does not easily translate into performance improvement. Sometimes significant tuning effort is needed to realize the performance potential. At some point, that extra human effort is no longer worthwhile and could be better spent for more creative purposes. Given the world that relies so much on technology today and so many new things can be created, security enhanced, and reliability improved, it is hard to imagine tuning for performance would be worth the average programmer’s effort. We strongly believe that the task of making a common program run faster could and should be taken over by automated mechanisms rather than left to individual engineers.

Clearly, there is no silver bullet in improving program execution speed in an energy-efficient manner. Multithreaded execution, leveraging special-purpose accelerators and increasing the execution speed of a single thread are all part of the arsenal. For a long time, increasing single-thread performance has been the central focus of the microprocessor industry and, to a lesser extent, in the related research community. The two main drivers for single-thread performance faster clocks and advancements in microarchitecture have all but stopped in recent years. Without these traditional driving forces, improving single-thread performance for general-purpose applications is undoubtedly more challenging.

A primary factor of the impact in exploiting implicit parallelism is the broad applicability of the techniques discovered. The performance gain does not depend on specific algorithms or the structure of the code. It is applicable to sequential code as well as parallel code. In addition
to the potential impact in future processor design, our work makes important observations and contributions that can provide a foundation for subsequent work:

- We have demonstrated that there can be substantial improvement from self-tuning and *intelligent* look-ahead techniques.

- The proposed architecture fundamentally separates performance and correctness concerns and thus allows systems to openly import tuning results obtained by others. This in turn allows not only rapid deployment of optimizations but also more effective, collaborative self-tuning among multiple systems.

- While earlier work sometimes rely on manual effort to demonstrate the potential of certain direction of look-ahead, our proposal advocates for an *automated methodology* that demonstrates practically achievable gains.

With the proliferation of commercial multicore products, the research interest of increasing single-thread performance has reduced dramatically, if not disappearing completely. This may be the result of having more near-term opportunities in other avenues at the moment but is decidedly not due to the lack of potential. Indeed, there is a significant level of implicit parallelism both in old benchmarks used in classic studies and in their modern counterparts [10]. Current systems are far from exhausting this implicit parallelism. We believe that continued exploration of opportunities in this space now will be fruitful because large-scale industrial adoption is near-term reality.
Bibliography


