ON THE ROLE OF SEARCH IN GENERATING
HIGH-PERFORMANCE BLAS LIBRARIES

A Dissertation
Presented to the Faculty of the Graduate School
of Cornell University
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

by
Kamen Yotov Yotov
January 2006
A key step in program optimization is the estimation of optimal values for parameters such as tile sizes and loop unrolling factors. Traditional compilers use simple analytical models to compute these values. In contrast, library generators like ATLAS use global search over the space of parameter values by generating programs with many different combinations of parameter values, and running them on the actual hardware to determine which values give the best performance. It is widely believed that traditional model-driven optimization cannot compete with search-based empirical optimization because tractable analytical models cannot capture all the complexities of modern high-performance architectures. This thesis disproves this belief.

In this work we replaced the global search engine in ATLAS with a model-driven optimization engine, and measured the relative performance of the code produced by the two systems on a variety of architectures. Our experiments show that model-driven optimization can be surprisingly effective, and can generate code with performance comparable to that of code generated by ATLAS using global search. For improving the code further, we advocate complementing modelling with local search and model refinement.

Model-driven optimization needs accurate values of hardware parameters. In
this thesis we also describe X-Ray, a robust framework of micro-benchmark for measuring such hardware parameters. X-Ray is designed to be extensible to make it easy to implement new micro-benchmarks. We have developed novel algorithms for measuring hardware parameters commonly used in optimizing software performance, and we have implemented them in X-Ray. We evaluate X-Ray experimentally on traditional workstations and servers as well as on embedded architectures, and show that it produces more accurate and complete results than existing tools.
Kamen Yotov was born on 06/03/1977 in Sofia, Bulgaria. He spent his early childhood years in Sopot, Bulgaria. He returned to Sofia in 1982 to attend elementary and secondary school. In 1995 he graduated from the National High-School for Mathematics and Natural Sciences and joined Sofia University, where he received an M.Sc. degree in Computer Science degree in 1999.

Kamen joined the Ph.D. program in Computer Science at Cornell University in 2000 and received an M.S. degree in 2003, and a Ph.D. degree in 2006. For the time being, he does not plan to study more... ever.

Kamen enjoys hiking, cooking, and achieving high performance and productivity in all aspects of science and life, but even more so in driving cars.
In memory of my late grandmother Maria Ungarova
ACKNOWLEDGEMENTS

I am especially grateful to my advisor Keshav Pingali for encouraging me and standing by me through the long journey. He contaminated me with his scientific views of fundamental simplicity, and at the same time, need of great depth in understanding the issues at hand. Over the years, these views have become my own, but even today, I cannot quite manage to express the essence of my work so succinctly and beautifully, as he would. I can only hope that this talent will develop in me later in my professional carrier. This thesis would not have been possible without all the hard work we did together. I learned a lot from you in the process and I continue learning from you to this day!

I would like to extend special thanks to Paul Stodghill for standing beside me throughout my graduate studies and providing not only academic advice but also friendly support, encouragement, and strong belief in my abilities. One of the many insights, which we came across in our Friday afternoon meetings, was central to the design of the X-Ray system, described in Chapter 4.

I thank the other two members of my committee – Ken Birman and Martin Burtscher – for being open and ready to give advice whenever I needed it, for reviewing the draft of this thesis and making constructive suggestions on improving its content, and for the wonderful classes I took with them (CS514, CS614 and ECE475, ECE575 respectively).

I thank all other professors in the Computer Science and Electrical and Computing Engineering Departments for showing me computer science in a much brighter light through their classes. These include (in alphabetical order) Joe Halpern (CS676), Mark Heinrich (ECE572), Andrew Myers (CS611), and Eva Tardos (CS681).
I thank David Padua from UIUC for the great time working with his group and for the willingness to give honest advice and sense of direction when I needed it.

I thank Fred Gustavson and John Gunnels from IBM T. J. Watson Research Center for all the interesting conversations and for the motivation and encouragement. I am sure we will do great things together during my post-doctorate work at IBM.

There are too many other people to thank, who have positively influenced me in the last six years, and whom I would be eternally grateful for that. I though I would mention the following few (in alphabetical order), whose comments and remarks helped me to stay on track and complete this big step of my life: Gianfranco Bilardi, Jack Dongarra, Ken Kennedy, Pratap Pattnaik, Marcus Peuschel, and Clint Whaley.

I thank Becky Stewart, Stephanie Meik, Helene Croft, Rebecca Rich-Goldweber, Tammy Howe, Cindy Robinson, and Bonnie Maine from the department of Computer Science at Cornell for always being there for me, and never getting tired of my “special” requests, no matter how different or unseen they were.

I thank my colleagues Dan Marques, Greg Bronevetsky, and Tom Roeder for the interesting conversations, support, and an overall unforgettable experience we shared as graduate students.

I am especially grateful to my parents, for bringing me to this world, for putting high hopes in me, and for influencing my personal and professional development, so that I can be the person who I am today. Without you, all this could never happen – even in theory ;)

Finally, I want to express my deepest appreciation and gratitude to my wife Borislava. She was the one who was on my side for the last eight years, helping
me and supporting me through the hardest times. She was the one who followed me in this new world, who gave me my daughter Julianne, and because of whom I believe family is more important than anything. Thank you!
TABLE OF CONTENTS

1 Introduction 1

2 Background 7
   2.1 Cache-aware programs and restructuring compilers . . . . . . . . . . 7
      2.1.1 Legality of Restructuring Transformations . . . . . . . . . . . 9
      2.1.2 Linear Loop Transformations . . . . . . . . . . . . . . . . . 13
      2.1.3 Other Restructuring Transformations . . . . . . . . . . . . .19
      2.1.4 Conventional Compiler Transformations . . . . . . . . . . . . 23
   2.2 Cache Oblivious Algorithms . . . . . . . . . . . . . . . . . . . . . 27

3 High-performance Code Generation 32
   3.1 Tiling for the L1 data cache . . . . . . . . . . . . . . . . . . . . . 32
   3.2 Tiling for the register file . . . . . . . . . . . . . . . . . . . . . . 33
   3.3 Scalarization of array elements . . . . . . . . . . . . . . . . . . . 35
   3.4 Pipeline scheduling . . . . . . . . . . . . . . . . . . . . . . . . . . 36
   3.5 Additional details . . . . . . . . . . . . . . . . . . . . . . . . . . 38
   3.6 Discussion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40

4 Measuring Hardware Parameters 42
   4.1 Introduction to X-Ray . . . . . . . . . . . . . . . . . . . . . . . . 42
   4.2 X-Ray Framework Overview . . . . . . . . . . . . . . . . . . . . . 44
      4.2.1 Nano-benchmarks . . . . . . . . . . . . . . . . . . . . . . . . 46
      4.2.2 Nano-benchmark Generator . . . . . . . . . . . . . . . . . . 48
      4.2.3 Implementing a new micro-benchmark . . . . . . . . . . . . 49
   4.3 CPU Micro-benchmarks . . . . . . . . . . . . . . . . . . . . . . . . 50
      4.3.1 CPU Frequency . . . . . . . . . . . . . . . . . . . . . . . . . 50
      4.3.2 Instruction Latency . . . . . . . . . . . . . . . . . . . . . . 50
      4.3.3 Instruction Throughput . . . . . . . . . . . . . . . . . . . . 51
      4.3.4 Instruction Existence . . . . . . . . . . . . . . . . . . . . . 53
      4.3.5 Number of Registers . . . . . . . . . . . . . . . . . . . . . . 54
      4.3.6 SMP and SMT parallelism . . . . . . . . . . . . . . . . . . . 55
   4.4 Memory Hierarchy Micro-benchmarks . . . . . . . . . . . . . . . . . 56
      4.4.1 Previous Approaches . . . . . . . . . . . . . . . . . . . . . . 57
      4.4.2 Compactness of Sequences . . . . . . . . . . . . . . . . . . . 61
      4.4.3 L1 Data Cache . . . . . . . . . . . . . . . . . . . . . . . . . 66
      4.4.4 Lower Levels of the Memory Hierarchy . . . . . . . . . . . . 74
      4.4.5 Measuring TLB Parameters . . . . . . . . . . . . . . . . . . . 81
   4.5 Future Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 84
5 Global Search in ATLAS

5.1 Estimating hardware parameters ........................................... 85
5.2 Search strategy ................................................................. 86
5.3 Finding $N_B$ ................................................................. 87
5.4 Finding $M_U$ and $N_U$ ....................................................... 88
5.5 Finding $K_U$ ................................................................. 88
5.6 Finding $L_s$ ................................................................. 88
5.7 Finding $F_F$, $I_F$, and $N_F$ ................................................. 89
5.8 Finding $N_{CN_B}$ ............................................................. 89
5.9 Finding parameters for clean-up codes .................................... 89
5.10 Discussion ................................................................. 90

6 Model-Driven Optimization ................................................... 91

6.1 Estimating hardware parameters ........................................... 91
6.2 Modeling strategy ............................................................... 92
6.3 Estimating $N_B$ ............................................................. 92
6.3.1 Fully associative, optimal replacement, unit line size .......... 92
6.3.2 Correcting for non-unit line size ....................................... 93
6.3.3 Correcting for LRU replacement policy ............................... 94
6.3.4 Correcting to avoid micro-MMM clean-up code ................. 96
6.3.5 Other cache organizations ............................................. 97
6.4 Estimating $M_U$ and $N_U$ ................................................ 97
6.5 Estimating $K_U$ ............................................................. 99
6.6 Estimating $L_s$ ............................................................. 99
6.7 Estimating other parameters ............................................. 100
6.8 Discussion ................................................................. 101

7 Model Refinement and Local Search ...................................... 102

8 Experimental Results ......................................................... 105

8.1 Global search vs. model-driven optimization .......................... 105
8.1.1 DEC Alpha 21264 ......................................................... 109
8.1.2 IBM Power 3 ............................................................ 115
8.1.3 IBM Power 4 ............................................................ 119
8.1.4 SGI R12K ............................................................. 124
8.1.5 Sun UltraSPARC IIIi ................................................... 129
8.1.6 Intel Itanium 2 .......................................................... 134
8.1.7 AMD Opteron 240 ......................................................... 141
8.1.8 AMD Athlon MP ........................................................ 150
8.1.9 Intel Pentium III .......................................................... 155
8.1.10 Intel Pentium 4 .......................................................... 160
8.1.11 Discussion .............................................................. 164
8.2 Model refinement and local search ..................................... 167
8.2.1 x86-based architectures ............................................. 167
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Summary of optimization parameters</td>
<td>40</td>
</tr>
<tr>
<td>8.1</td>
<td>DEC Alpha 21264: Platform Specification</td>
<td>109</td>
</tr>
<tr>
<td>8.2</td>
<td>DEC Alpha 21264: Optimization Parameters</td>
<td>109</td>
</tr>
<tr>
<td>8.3</td>
<td>DEC Alpha 21264: Timings</td>
<td>109</td>
</tr>
<tr>
<td>8.4</td>
<td>IBM Power 3: Platform Specification</td>
<td>115</td>
</tr>
<tr>
<td>8.5</td>
<td>IBM Power 3: Optimization Parameters</td>
<td>115</td>
</tr>
<tr>
<td>8.6</td>
<td>IBM Power 3: Timings</td>
<td>115</td>
</tr>
<tr>
<td>8.7</td>
<td>IBM Power 4: Platform Specification</td>
<td>119</td>
</tr>
<tr>
<td>8.8</td>
<td>IBM Power 4: Optimization Parameters</td>
<td>119</td>
</tr>
<tr>
<td>8.9</td>
<td>IBM Power 4: Timings</td>
<td>119</td>
</tr>
<tr>
<td>8.10</td>
<td>SGI R12K: Platform Specification</td>
<td>124</td>
</tr>
<tr>
<td>8.11</td>
<td>SGI R12K: Optimization Parameters</td>
<td>124</td>
</tr>
<tr>
<td>8.12</td>
<td>SGI R12K: Timings</td>
<td>124</td>
</tr>
<tr>
<td>8.13</td>
<td>Sun UltraSPARC IIIi: Platform Specification</td>
<td>129</td>
</tr>
<tr>
<td>8.14</td>
<td>Sun UltraSPARC IIIi: Optimization Parameters</td>
<td>129</td>
</tr>
<tr>
<td>8.15</td>
<td>Sun UltraSPARC IIIi: Timings</td>
<td>129</td>
</tr>
<tr>
<td>8.16</td>
<td>Intel Itanium 2: Platform Specification</td>
<td>134</td>
</tr>
<tr>
<td>8.17</td>
<td>Intel Itanium 2: Optimization Parameters</td>
<td>134</td>
</tr>
<tr>
<td>8.18</td>
<td>Intel Itanium 2: Timings</td>
<td>134</td>
</tr>
<tr>
<td>8.19</td>
<td>AMD Opteron 240: Platform Specification</td>
<td>141</td>
</tr>
<tr>
<td>8.20</td>
<td>AMD Opteron 240: Optimization Parameters</td>
<td>141</td>
</tr>
<tr>
<td>8.21</td>
<td>AMD Opteron 240: Timings</td>
<td>141</td>
</tr>
<tr>
<td>8.22</td>
<td>AMD Athlon MP: Platform Specification</td>
<td>150</td>
</tr>
<tr>
<td>8.23</td>
<td>AMD Athlon MP: Optimization Parameters</td>
<td>150</td>
</tr>
<tr>
<td>8.24</td>
<td>AMD Athlon MP: Timings</td>
<td>150</td>
</tr>
<tr>
<td>8.25</td>
<td>Intel Pentium III: Platform Specification</td>
<td>155</td>
</tr>
<tr>
<td>8.26</td>
<td>Intel Pentium III: Optimization Parameters</td>
<td>155</td>
</tr>
<tr>
<td>8.27</td>
<td>Intel Pentium III: Timings</td>
<td>155</td>
</tr>
<tr>
<td>8.28</td>
<td>Intel Pentium 4: Platform Specification</td>
<td>160</td>
</tr>
<tr>
<td>8.29</td>
<td>Intel Pentium 4: Optimization Parameters</td>
<td>160</td>
</tr>
<tr>
<td>8.30</td>
<td>Intel Pentium 4: Timings</td>
<td>160</td>
</tr>
<tr>
<td>8.31</td>
<td>Optimization Paramameters for Intel Itanium 2</td>
<td>169</td>
</tr>
<tr>
<td>8.32</td>
<td>Timings for Intel Itanium 2</td>
<td>169</td>
</tr>
<tr>
<td>8.33</td>
<td>Optimization Paramameters for SGI R12000</td>
<td>170</td>
</tr>
<tr>
<td>8.34</td>
<td>Timings for SGI R12000</td>
<td>171</td>
</tr>
<tr>
<td>A.1</td>
<td>Summary of Experimental Results: CPU Features</td>
<td>178</td>
</tr>
<tr>
<td>A.2</td>
<td>Experimental results for registers</td>
<td>182</td>
</tr>
<tr>
<td>A.3</td>
<td>Summary of experimental results</td>
<td>187</td>
</tr>
<tr>
<td>A.4</td>
<td>Summary of Itanium 2 $C_3$ parameters</td>
<td>188</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

1.1 Architecture of ATLAS and of Model-driven ATLAS  . . . . . . . . 2
2.1 Structure of a simple compiler  . . . . . . . . . . . . . . . . . . . 8
2.2 Loop nest  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
2.3 Loop permutation  . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
2.4 Loop skewing  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
2.5 Loop reversal  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
2.6 Naïve MMM Code  . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
2.7 Tiled MMM Code  . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
2.8 Simple Loop  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
2.9 Strip-mined Loop  . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
2.10 Simple Loop  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
2.11 Completely Unrolled Loop  . . . . . . . . . . . . . . . . . . . . . . 23
2.12 Non-pipelined loop  . . . . . . . . . . . . . . . . . . . . . . . . . . 26
2.13 Pipelined loop  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
3.1 Naïve MMM Code  . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
3.2 MMM tiled for L1 data cache and registers  . . . . . . . . . . . . . 34
3.3 mini-MMM and micro-MMM  . . . . . . . . . . . . . . . . . . . . . . 35
4.1 A micro-benchmark in X-Ray  . . . . . . . . . . . . . . . . . . . . . . 45
4.2 High-level structure of a nano-benchmark  . . . . . . . . . . . . . . 46
4.3 Approach for timing nano-benchmarks  . . . . . . . . . . . . . . . . 47
4.4 Standard memory hierarchy benchmark  . . . . . . . . . . . . . . . 58
4.5 Memory address decomposition on Pentium III  . . . . . . . . . . . . 61
4.6 Sequences of sequences  . . . . . . . . . . . . . . . . . . . . . . . . . 63
4.7 Example of (semi-/non-)compact  . . . . . . . . . . . . . . . . . . . . 66
4.8 Decomposition of $W = \langle m_0, S, N \rangle$  . . . . . . . . . . . 68
4.9 Measuring $C$ and $A$ of L1 Data Cache  . . . . . . . . . . . . . . 70
4.10 Modified address sequence for measuring $B$  . . . . . . . . . . . . 71
4.11 Algorithm for measuring $B$  . . . . . . . . . . . . . . . . . . . . . . 72
4.12 Improved timing of memory accesses  . . . . . . . . . . . . . . . . 73
4.13 Memory address decomposition on Pentium III  . . . . . . . . . . . . 82
6.1 Schematic Pseudo-Code for mini-MMM  . . . . . . . . . . . . . . . . . 92
8.1 DEC Alpha 21264: MMM Performance  . . . . . . . . . . . . . . . . . 110
8.2 DEC Alpha 21264: Sensitivity of performance to $N_B$  . . . . . . . . 110
8.3 DEC Alpha 21264: Sensitivity of performance to $M_U$ and $N_U$  . 111
8.4 DEC Alpha 21264: Sensitivity of performance to $K_U$  . . . . . . . . 111
8.5 DEC Alpha 21264: Sensitivity of performance to $L_s$  . . . . . . . . 111
8.6 IBM Power 3: MMM Performance  . . . . . . . . . . . . . . . . . . . . 116
8.7 IBM Power 3: Sensitivity of performance to $N_B$  . . . . . . . . . . 116
8.52 Intel Pentium 4: Sensitivity of performance to $M_U$ and $N_U$ . . . . . . 162
8.53 Intel Pentium 4: Sensitivity of performance to $K_U$ . . . . . . . . 162
8.54 Intel Pentium 4: Sensitivity of performance to $L_s$ . . . . . . . 162
8.55 Summary of Model . . . . . . . . . . . . . . . . . . . . . . . . . . . 165
8.56 Summary of mini-MMM Performance. Performance numbers are normalized to that of ATLAS CGw/S, which is presented as 100% 166
8.57 Intel Itanium 2: MMM Performance . . . . . . . . . . . . . . . . . . 169
8.58 SGI R12000: MMM Performance . . . . . . . . . . . . . . . . . . . . 171
PREFACE

Every once in a while, a truly outstanding paper\textsuperscript{1} comes along that serves as a model for other papers in the area. This is such a paper. The work provides a thorough analysis of the potential for compiler models to select the parameters for optimizing transformations in a way that is competitive with the well-known ATLAS system, which uses an experimental search mechanism with trial runs to select these same parameters.

In addition to its lucid description of the experimental method, the paper provides an excellent description of the ATLAS system and its methodology for selection of optimization parameters. This is followed by a discussion of the model-based approach to selecting parameters. Finally, the paper presents its experimental results. I must admit that I was amazed by the depth and breadth of the experimental evaluation: it covers every interesting platform and evaluates the performance on a variety of different dimensions and for a variety of matrix sizes. Although the experiment focuses on matrix multiplication only, it is nevertheless a fascinating study, filled with rich insights on the problems of building useful models into compilers for optimizing performance on different machines. I also found the discussion of the results and conclusions in the final section to be balanced and insightful.

Prof. Ken Kennedy
Rice University

\textsuperscript{1}This preface is extracted from a review, written by Prof. Ken Kennedy for [65], which is at the heart of this dissertation. It is included here by permission of the author.
Chapter 1

Introduction

The sciences do not try to explain, they hardly even try to interpret, they mainly make models. By a model is meant a mathematical construct which, with the addition of certain verbal interpretations, describes observed phenomena. The justification of such a mathematical construct is solely and precisely that it is expected to work.

John Von Neumann

It is a fact universally recognized that current restructuring compilers do not generate code that can compete with hand-tuned code in efficiency, even for a simple kernel like matrix multiplication. This inadequacy of current compilers does not stem from a lack of technology for transforming high-level programs into programs that run efficiently on modern high-performance architectures; over the years, the compiler community has invented innumerable techniques for enhancing locality and parallelism, described in more detail in Chapter 2.

The simplest manual approach to tuning a program for a given platform is to write different versions of that program, evaluate the performance of these versions on the target platform, and select the one that gives the best performance. These different versions usually implement the same algorithm, but differ in the values they use for parameters such as tile sizes and loop unroll factors. The architectural insights and domain knowledge of the programmer are used to limit the number of versions that are evaluated. In effect, the analytical techniques used in current compilers to derive optimal values for such parameters are replaced by an empirical search over a suitably restricted space of parameter values (by empirical search, we
mean a three step process: (1) generating a version of the program corresponding to each combination of the parameters under consideration, (2) executing each version on the target machine and measuring its performance, and (3) selecting the version that performs best). This approach has been advocated most forcefully by Fred Gustavson and his co-workers at IBM, who have used it for many years to generate the highly optimized ESSL and PESSL libraries for IBM machines [51]. Recently, a number of projects such as FFTW [27, 28], PhiPAC [8, 3], ATLAS [60, 1], and SPIRAL [36, 50] have automated the generation of the different program versions whose performance must be evaluated. Experience shows that these library generators produce much better code than native compilers do on modern high-performance architectures.

Figure 1.1: Architecture of ATLAS and of Model-driven ATLAS

This thesis was motivated by a desire to understand the performance gap between the BLAS codes produced by ATLAS and by restructuring compilers, with the ultimate goal of improving the state of the art of current compilers. One reason why compilers might be at a disadvantage is that they are general-purpose and must be able to optimize any program, whereas a library generator like ATLAS can focus on a particular problem domain. However, this is somewhat implausible
because dense numerical linear algebra, the particular problem domain of ATLAS, is precisely the area that has been studied most intensely by the compiler community, and there is an extensive collection of well-understood transformations for optimizing dense linear algebra programs. Another reason for the inadequacy of current compilers might be that new transformations, unknown to the compiler community, are required to produce code of the same quality as the code produced by ATLAS. Finally, it is possible that the analytical models used by compilers to estimate optimal values for transformation parameters are overly simplistic, given the complex hardware of modern computers, so they are not able to produce good values for program optimization parameters.

No definitive studies exist to settle these matters. This dissertation is the first quantitative study of these issues.

Figure 1.1 shows our experimental set-up, which makes use of the original ATLAS system (top of the figure) and a modified version (bottom of the figure) that uses analytical models instead of empirical search. Like any system that uses empirical search, ATLAS has (i) a module that controls the search, which is used to determine optimal values for code optimization parameters (mmsearch), and (ii) a module that generates code, given these values (mmcase). The parameters used by ATLAS are described in more detail in Chapter 3; for example, \( N_B \) is the tile size to be used when optimizing code for the L1 data cache.

In general, there is an unbounded number of possible values for a parameter like \( N_B \) so it is necessary to bound the size of the search space. When ATLAS is installed, it first runs a set of micro-benchmarks to determine hardware parameters such as the capacity of the L1 data cache and the number of registers. The values of these hardware parameters are used to bound the search space. Since
ATLAS relies on search, it can afford to use approximate values for these hardware parameters. In Chapter 4, we describe X-Ray, a tool that we have implemented for more accurate measurement of such hardware parameters.

The mmsearch module within ATLAS enumerates points within the bounded search space, invokes the mmcase module to generate the appropriate code (denoted by mini-MMM in the figure), runs this code on the actual machine for a long enough time to ensure accurate measurement, and records its execution time. At the end of the search, the parameter values that gave the best performance are used to generate the library code. This library is coded in a simple subset of C, which can be viewed as portable assembly code, and it is compiled to produce the final executable.

We first studied the code generation module\footnote{Our description of ATLAS was arrived at by studying the ATLAS source code. In case of any discrepancy between this description and how the ATLAS system is actually implemented, the documentation of the ATLAS project should be considered to be authoritative [60, 59, 58].}, and determined that the code it produces can be viewed as the end result of applying standard compiler transformations to high-level BLAS codes. As we describe in Chapter 3, the code produced by ATLAS is similar to what we would get if we applied cache tiling, register tiling, and operation scheduling to the standard three-loop matrix multiplication code. This exercise ruled out the possibility that ATLAS incorporated some transformation, unknown to the compiler community, that was critical for obtaining good performance. We then modified ATLAS by replacing the search module, described in more detail in Chapter 5, with a module (mmmodel) that uses standard analytical models to estimate optimal values for the optimization parameters, as described in Chapter 6. Since both ATLAS and the modified ATLAS use the same code gen-
erator, we are assured that any difference in the performance of the generated code results solely from different choices for optimization parameter values. Of course, the optimization parameter values produced by using models can be refined by using local search, as described in Chapter 7.

In Chapter 8, we present experimental results on ten different platforms, comparing

- the time spent to determine the parameter values,
- the values of the parameters, and
- the relative performance of generated code.

Our results show that on all ten platforms, a relatively simple and very intuitive model is able to estimate near-optimal values for the optimization parameters used by the ATLAS Code Generator. We conclude in Chapter 9 with a discussion of our main findings, and suggest future directions for research.

One feature of ATLAS is that it can make use of hand-tuned BLAS routines, many of which are included in the ATLAS distribution. When ATLAS is installed on a machine, these hand-coded routines are executed and evaluated. If the performance of one of these hand-coded routines surpasses the performance of the code generated by the ATLAS Code Generator, the hand-coded routine is used to produce the library. For example, neither the ATLAS Code Generator nor the C compilers on the Pentium 4 exploit the SSE2 vector extensions to the x86 instruction set, so ATLAS-generated matrix multiplication code runs at around 1.5 GFLOPS on this architecture. However, the matrix multiplication routine in the library produced by ATLAS runs at 3.3 GFLOPS because it uses SSE2 kernels carefully hand-coded by expert programmers.
Our concern is not with handwritten code, but with the code produced by the ATLAS Code Generator and with the estimation of optimal values for the parameters that are inputs to the code generator. To make clear distinctions, we use the following terminology.

- *ATLAS CGw/S*: This refers to the ATLAS system in which all code is produced by the *ATLAS Code Generator with Search* to determine parameter values. No hand-written, contributed code is allowed.

- *ATLAS Model*: This refers to the modified ATLAS system we built in which all code is produced by the ATLAS Code Generator, using parameter values produced from analytical models.

- *ATLAS Unleashed*: This refers to the complete ATLAS distribution, which may use hand-written codes and predefined parameter values (*architectural defaults*) to produce the library. Where appropriate, we include, for completeness, the performance graphs for the libraries produced by ATLAS Unleashed.
Chapter 2

Background

In this chapter, we discuss two different approaches to optimizing the cache performance of programs. The first approach is cache-aware, and it exploits detailed knowledge of the memory hierarchy of a machine to improve cache performance. In particular, there is a large body of research on transforming affine loop nests to improve locality in a cache-aware manner, and we summarize this work in this chapter. The second approach we discuss is a programming methodology called the cache-oblivious approach. The key argument for the cache-oblivious approach is that programs that implement divide-and-conquer algorithms perform well on any memory hierarchy without the need for tuning. The argument is that each division step usually reduces the problem data size, so the data for entire sub-problems fit in different levels of the memory hierarchy at some point during the divide-and-conquer process. Once the data for a sub-problem is in cache, that sub-problem can be solved completely without cache misses other than conflict misses.

2.1 Cache-aware programs and restructuring compilers

Compiler design has been an active area of research since the fifties. The structure of a simple compiler is presented in Figure 2.1 [46].

Optimizing program execution was a concern even for early compilers. Program optimizations can be either machine-independent or machine-dependent. Machine-independent optimizations usually try to reduce the amount of computation by eliminating repeated or redundant computations; for example, loop invariant removal [5] moves invariant computations out of loop bodies. Other
machine-independent optimizations such as strength reduction [19] replace expensive operations with simpler ones. Machine-dependent optimizations on the other hand attempt to tailor the code to take advantage of particular machine features; for example, register allocation [14, 16, 13] attempts to assign variables to registers because it is faster to access a value that is in a register than one that is stored in memory. One complication with machine-dependent optimizations is that taking full advantage of machine-specific features usually conflicts with portability of the compiler.

In general, optimizations are performed on some form of intermediate program
representation. When this intermediate form is an abstract syntax tree, the optimization is sometimes referred to as a source-to-source transformation because it is possible to generate a high-level program from the transformed abstract syntax tree. Compilers that perform optimizations at this level are known as restructuring compilers. Current compilers implement several levels of intermediate code, gradually converting the program from more abstract to more machine dependent representation. There are usually at least two levels of intermediate representation:

- **High-Level Intermediate Representation (HIR):** The structure of loops and array references are preserved in this representation so that high-level analyses such as dependence analysis and restructuring transformations such as loop transformations can be performed.

- **Low-Level Intermediate Representation (LIR):** Once high-level transformations have been performed, it is necessary to optimize address arithmetic and perform traditional "Dragon-book" style optimizations. To facilitate this, the level of the code is lowered by replacing loops and array references with conditional transfer of control constructs and pointer arithmetic respectively.

### 2.1.1 Legality of Restructuring Transformations

Before a compiler can restructure a program, it is important to prove that the transformation is legal, i.e., that the transformation does not change the semantics of the program. To prove legality, current compilers use a conservative approach like dependence analysis.

Intuitively, a data dependence exists from statement $S_1$ to statement $S_2$ if (i) they both touch the same location with at least one of them writing to it, and (ii)
there is a possible control flow from $S_1$ to $S_2$. For such statements, it is clear that program semantics will change if the two statements are interchanged. On the other hand, transformations which preserve every dependence in a source program are legal [6].

Dependences can be classified in the following four categories.

1. *Flow dependence*: A flow dependence is said to exist from statement $S_1$ to statement $S_2$ if $S_1$ writes to a location which may later be read by $S_2$, e.g.:

   $S_1: \quad X = \ldots$

   $\ldots$

   $S_2: \quad \ldots = X$

   This dependence is also known as a *read-after-write (RAW)* dependence.

2. *Anti-dependence*: An anti-dependence is said to exist from statement $S_1$ to statement $S_2$ if $S_1$ reads from a location that is later written to by statement $S_2$, e.g.:

   $S_1: \quad \ldots = X$

   $\ldots$

   $S_2: \quad X = \ldots$

   This dependence is also known as a *write-after-read (WAR)* dependence.

3. *Output dependence*: An output dependence is said to exist from statement $S_1$ to statement $S_2$ if $S_1$ and $S_2$ write to the same location, e.g.:

   $S_1: \quad X = \ldots$

   $\ldots$

   $S_2: \quad X = \ldots$
This dependence is also known as a *write-after-write (WAW) dependence.*

4. *Input dependence* is a dependence in which both \( S_1 \) and \( S_2 \) read the same location, e.g.:

\[
S_1: \ldots = x \\
\ldots \\
S_2: \ldots = x
\]

Unlike other dependences, input dependences can be reordered without violating the semantics of the program, and therefore they are not considered when proving the legality of optimizing transformations. On the other hand, they are important to consider for optimizations for improving data locality.

This dependence is also known as a *read-after-read (RAR) dependence.*

Restructuring compilers have focused mainly on optimizing affine loop nests, i.e., loop nests where loop boundaries and array indices are affine functions of loop control variables. Loop nests are usually divided into *perfectly nested* loop nests and *imperfectly nested* loop nests. In a perfectly nested loop nest, all assignment statements are contained in the innermost loop of the loop nest. Figure 2.2 shows a perfectly nested loop nest (for loop index sets we use MATLAB notation \([\text{First} : \text{Step} : \text{Last}]\), which represents the set of all integers between \( \text{First} \) and \( \text{Last} \) in steps of \( \text{Step} \)). If there are assignment statements contained within some but not all of the loops in a loop nest, that loop nest is called an imperfectly nested loop nest. For example, if there was an assignment statement between the \( i_1 \) and \( i_2 \) loops in Figure 2.2, the loop nest would be imperfectly nested.

An iteration is typically denoted by an iteration vector \( \bar{v} = \langle v_1, v_2, \ldots, v_n \rangle \), where \( v_k \) is the value of the control variable \( i_k \) for this iteration. The space of
for $i_1 \in [L_1 : 1 : U_1]$

for $i_2 \in [L_2 : 1 : U_2]$

\ldots

for $i_n \in [L_n : 1 : U_n]$

$S(i_1, i_2, \ldots, i_n)$

Figure 2.2: Loop nest

iteration vectors is known as iteration space.

If there is a dependence from the instance of the loop body in an iteration $\bar{v} = \langle v_1, v_2, \ldots, v_n \rangle$ to the instance of the loop body in iteration $\bar{w} = \langle w_1, w_2, \ldots, w_n \rangle$, we can compute the dependence distance vector $\bar{\delta} = \bar{w} - \bar{v} = \langle \delta_1, \delta_2, \ldots, \delta_n \rangle$. As with basic dependences, each dependence vector can be classified as RAW, WAR, WAW, or RAR dependence.

Different approaches have been developed for computing dependence distance and direction vectors for affine loop nests. These include simple conservative tests like Zero Induction Variable (ZIV), Single Induction Variable (SIV), Multiple Induction Variable (MIV) [6], and more complex and powerful tests based on integer linear programming techniques, such as the Omega test [49].

In some programs, it may not be possible to represent a dependence by a single distance vector. In that case, an abstraction of distance vectors called direction vectors is often used. A direction vector $\bar{d} = \langle d_1, d_2, \ldots, d_n \rangle$ which corresponds to the distance vector $\bar{\delta} = \langle \delta_1, \delta_2, \ldots, \delta_n \rangle$ is defined as follows:

$$d_i = \begin{cases} +, & \delta_i > 0 \\ - , & \delta_i < 0 \\ 0, & \delta_i = 0 \end{cases}$$

Sometimes an asterisk (*) is used as an element of a direction vector, when
that element can potentially fall in more than one of the disjoint groups above.

Note that a distance (or direction) vector $\bar{\delta}$ is always lexicographically positive (denoted $\bar{\delta} \succ \bar{0}$), which means that the first non-zero element of it is positive (or +). For a reordering transformation to be legal, it needs to preserve the lexicographical positivity of dependence vectors (otherwise it violates that particular dependence).

### 2.1.2 Linear Loop Transformations

Although one can imagine many transformations of perfectly nested loops, there are only a handful that are important in practice. Of these, permutation is the most useful one, and it is described in more detail below. In some loop nests, it may be necessary to first transform the loop nest using other transformations such as loop skewing and reversal [7, 62, 40] before permutation can be applied legally.

A powerful way of thinking about these loop transformations is to consider them to be linear transformations that change the basis of the iteration space of the loop nest $I = \{\langle i_1, i_2, \ldots, i_n \rangle : i_k \in [L_k, U_k], k \in [1, n]\}$. Any linear transformation can be described by means of a transformation matrix $T$. If $T$ is an integer, $n \times n$ unimodular matrix (that is, it has a determinant of $+1$ or $-1$), we call this transformation a unimodular loop transformation. Unimodular linear transformations map dense sets of integer points to dense sets of integer points.

For affine loop nests, loop bounds can be described by an expression of the form $\bar{a}^T \times \bar{i} \leq \bar{b}$, where $\bar{a}$ is the vector of coefficients for each loop control variable in the iteration vector $\bar{i}$. Therefore all loop bounds can be expressed in matrix form as:

$$A \times \bar{i} \leq \bar{b}.$$  

Note that this notation allows us to express complex loop bounds, such as
conjunctions of several affine inequalities, just by including each element of the conjunction as a separate row of matrix $A$. Such conjunctions often arise from inequalities of the form $\ldots \geq \max(\ldots)$ and $\min(\ldots) \leq \ldots$, which are common for loop nests describing convex iteration spaces.

A unimodular loop transformation with matrix $T$ on the iteration space $I$ produces a new iteration space $J$ as follows:

$$J = T \times I.$$ 

A specific iteration $\bar{i}$ is mapped to the iteration $\bar{j} = T \times \bar{i}$, and therefore $\bar{i} = T^{-1} \times \bar{j}$. Therefore we can construct the body of the transformed loop nest from the body of the original loop nest by substituting each occurrence of a loop control variable from $\bar{i}$ with the corresponding expression from $T^{-1} \times \bar{j}$. The loop bound conditions of the new iteration space can be computed using integer-linear programming from the inequality:

$$\left( A \times T^{-1} \right) \times \bar{j} \leq b.$$ 

One of the main advantages of unimodular loop transformations lies in the easiness of checking their legality. Because we have a unimodular transformation $T$ of the iteration space we can easily compute how an existing dependence vector $\delta$ is transformed by this particular transformation:

$$\delta_{\text{new}} = T \times \delta.$$ 

To ensure dependences are respected, all transformed dependence vectors should be lexicographically positive. If we express the dependence vectors as columns in a dependence matrix $D$, this condition can be expressed as $D \times T \succ 0$.

As discussed below, unimodular loop transformations [7, 17, 23, 40, 62, 37] include loop permutation (interchange), loop skewing, and loop reversal. In ad-
dition, because a product of two unimodular matrices is a unimodular matrix, any sequence of these transformations is also a unimodular transformation. Moreover, any unimodular loop transformation can be represented as a sequence of loop permutation, loop skewing, and loop reversal.

Loop scaling [40] is also a linear loop transformation but it is not unimodular, because it stretches the transformed space, transforming dense sets of integer points into sparse sets of integer points. Including loop scaling in our repertoire of transformations simplifies the synthesis of appropriate sequences of loop transformations for enhancing locality of reference or for parallelization.

**Loop permutation**

Loop permutation is a transformation with a two-fold effect. By changing the loop order, one can change the stride with which a certain array is accessed in memory. Achieving stride one access is crucial for improving spacial locality and in some cases triggering hardware data prefetchers. On another level, loop permutation (together with strip-mining) allows loop tiling, which can increase temporal data locality.

Loop permutation can be represented by a unimodular loop transformation, whose corresponding matrix $T$ is a permutation matrix. Consider the following loop nest:

$$\text{for } i_1 \in [1 : 1 : N]$$
$$\text{for } i_2 \in [1 : 1 : M]$$
$$S(i_1, i_2);$$
If we apply the unimodular transformation:

\[ T = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \]

The result is:

for \( j_1 \in [1 : 1 : M] \)

for \( j_2 \in [1 : 1 : N] \)

\( S(j_2, j_1) \);

The transformation can be presented pictorially as shown in Figure 2.3.

Figure 2.3: Loop permutation

Note that, because of the generality of the approach, loop permutation expressed as an unimodular transformation can more complex iteration spaces like the triangular ones in linear algebra codes like triangular solve and Cholesky factorization.

It is obvious that, since loop permutation can be expressed by a unimodular permutation matrix, the elements of dependence vectors will also be permuted in the resulting loop nest. Therefore loop permutation is not always legal. One special case is having a fully permutable loop nest, which as we will see later, is requirement for legality for loop tiling. Since in this case any permutation should be legal, to guarantee that the resulting dependence vectors are lexicographically
positive, it is necessary that all elements of the dependence matrix $D$ are non-negative.

In cases when a loop nest is not fully-permutable, there might be a chance to transform it to a fully-permutable loop nest by employing other unimodular transformations like loop skewing and loop reversal. We discuss these next.

**Loop skewing and Loop reversal**

Consider the following loop nest:

\[
\begin{align*}
\text{for } i_1 & \in [1 : 1 : N] \\
\text{for } i_2 & \in [1 : 1 : N] \\
X[i_1, i_2] & = X[i_1 - 1, i_2 + 1];
\end{align*}
\]

It is clear that the only dependence has a distance vector $\bar{\delta} = \langle 1, -1 \rangle$, and therefore the dependence matrix is:

\[
D = \begin{pmatrix}
1 \\
-1
\end{pmatrix}.
\]

Therefore loop interchange is not legal. We will correct this problem by applying the loop skewing and loop reversal unimodular transformations. First we can apply loop skewing with transformation matrix

\[
T_S = \begin{pmatrix}
1 & 1 \\
0 & 1
\end{pmatrix},
\]

which transforms the loop nest as shown in Figure 2.4.

Unfortunately, $T_S \times \bar{\delta} = \langle 0, -1 \rangle$ and therefore the resulting loop nest is neither fully-permutable, nor legal. We will fix this problem by applying loop reversal
Figure 2.4: Loop skewing

using the transformation matrix

\[ T_R = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}, \]

which transforms the loop nest as shown in Figure 2.5.

Figure 2.5: Loop reversal

The resulting transformation can be represented by a single unimodular matrix

\[ T = T_R \times T_S = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 0 & -1 \end{pmatrix}, \]

and the resulting dependence vector is \( T \times \bar{\delta} = (0, 1) \). Therefore the unimodular transformation with matrix \( T \) is legal, even though one of its component \( T_R \) is not legal. Furthermore the resulting loop nest is fully permutable, and therefore can be tiled.
2.1.3 Other Restructuring Transformations

Loop tiling

Modern hardware architectures have complex memory hierarchies, and optimizing programs to execute efficiently requires taking this design feature into account. Loop tiling \cite{38, 39} is a restructuring technique, which aims to improve data locality by reducing the working set size of a piece of code, so that it fits into some level of the memory hierarchy, and therefore executes faster. We discuss loop tiling in more detail in terms of matrix multiplication.

Blocking is an algorithmic transformation that converts the matrix multiplication into a sequence of small matrix multiplications, each of which multiplies small blocks of the original matrices. Blocking matrix multiplication for memory hierarchies was discussed by McKellar and Coffman as early as 1969 \cite{43}. The effect of blocking can be accomplished by the loop transformation called tiling, which was introduced by Wolfe in 1987 \cite{63}.

In Figure 2.6 we show the simple triply-nested version of Matrix-Matrix Multiply. It is a well-known fact that if this code is executed as is, exhibits very poor data locality and has very low performance.

\[
\begin{align*}
\text{for } i & \in [0 : 1 : N - 1] \\
\text{for } j & \in [0 : 1 : M - 1] \\
\text{for } k & \in [0 : 1 : K - 1] \\
C_{ij} & = C_{ij} + A_{ik} \times B_{kj}
\end{align*}
\]

Figure 2.6: Naive MMM Code

To improve performance the three loops in Figure 2.6 can be tiled by a factor \(N_B\), producing the code in Figure 2.7. For simplicity we have assumed that the
tile size $N_B$ divides the matrix dimensions $M$, $N$, and $K$.

\[
\text{for } i \in [0 : N_B : N - 1] \\
\text{for } j \in [0 : N_B : M - 1] \\
\text{for } k \in [0 : N_B : K - 1] \\
\text{for } i' \in [i : 1 : N_B - 1] \\
\text{for } j' \in [j : 1 : N_B - 1] \\
\text{for } k' \in [k : 1 : N_B - 1] \\
C_{i'j'} = C_{i'j'} + A_{i'k'} \times B_{k'j'}
\]

Figure 2.7: Tiled MMM Code

Intuitively, the code in Figure 2.7 performs better because the inner loop nest \(\langle i', j', k' \rangle\) only operates on relatively small $N_B \times N_B$ tiles of the full matrices, and does not touch other tiles before it is finished with the ones it has started working on.

It is not immediately obvious that such drastic code transformation is legal, and the fact is that, although it preserves the semantics of the computation for this simple case (Matrix-Matrix Multiply), there are many other codes for which it does not. We formalize the legality conditions further below.

One way to look at tiling is as a combination of strip-mining and loop permutation. Strip-mining is a simple procedure in which a loop is chopped into smaller loops of specific size (say $N_B$). For example the loop in Figure 2.8, when strip-mined with size of $N_B$, transforms into the loop in Figure 2.9.

\[
\text{for } i \in [0 : 1 : N - 1] \\
S(i);
\]

Figure 2.8: Simple Loop
for $i \in [0 : N_B : N - 1]$
\begin{align*}
  \text{for } i' \in [i : 1 : N_B - 1] \\
  S(i');
\end{align*}

Figure 2.9: Strip-mined Loop

Strip-mining is always legal, as it does not change the relative order of the original iterations. If we perform strip-mining on all the loops in Figure 2.6 in the same way, we would get the loop nest $\langle i, i', j, j', k, k' \rangle$. To obtain the same result as tiling, as presented in Figure 2.7, we need to permute the loops. Because each loop needs to be interchangeable with any other loop, it is necessary to have a fully permutable loop nest. As we discussed in Section 2.1.2, the necessary and sufficient condition for legality of such transformation is that all element of the dependence matrix should be non-negative.

Loop tiling can be performed several times to accommodate arbitrary complex memory hierarchies. Tiles do not have to be the same size in all dimensions (like $N_B$ in our example).

**Tiling for the register file and loop unrolling**

The register file is the highest, smallest, and fastest level of the memory hierarchy. As such, it should be given major consideration during program optimization. In a sense it is not very different from a cache, which is fully associative with optimal replacement and unit line size. In case of SIMD vector registers it even has non-unit line size.

To make effective use of hardware registers it is necessary to store frequently used variables in them. The procedure of assigning which variable is stored in which register at a certain point in time is known as register allocation. Current
compilers typically cannot allocate elements of aggregate data types like arrays and structures to registers directly. Therefore it is necessary to copy these elements to scalar variables, a technique known as scalar replacement or scalarization. We discuss register allocation and scalar replacement techniques in more detail in Section 2.1.4.

Since there is no indexed access to registers, tiling for the register file requires an extra step compared to tiling for the regular cache. This step, known as loop unrolling [6, 48], not only facilitates register allocation, but also decreases loop overhead.

Since the register file is at the top of the memory hierarchy, its corresponding loop nest is the innermost loop nest of the tiled code. This loop nest needs to be completely unrolled. Complete loop unrolling is when the looping construct is removed and its different iterations are explicitly inlined in its place.

For example the result of completely unrolling the loop in Figure 2.10 is presented in Figure 2.11.

\[
\text{for } i \in [i_{\text{min}} : i_{\text{step}} : i_{\text{min}} + k \times i_{\text{step}}] \\
S(i); \\
\]

Figure 2.10: Simple Loop

Unrolling is always legal, as it does not change the relative order of any iterations, but it can have adverse effect on performance if the total size of the unrolled code is larger than the instruction cache capacity.
...  
\[ S(i_{\text{min}}); \]
\[ S(i_{\text{min}} + i_{\text{step}}); \]
\[ S(i_{\text{min}} + 2 \times i_{\text{step}}); \]
...
\[ S(i_{\text{min}} + k \times i_{\text{step}}); \]
...

Figure 2.11: Completely Unrolled Loop

2.1.4 Conventional Compiler Transformations

There are numerous conventional compiler transformations which are generally useful for many different types of code. Some examples are dead code elimination, constant propagation, induction variable substitution, etc. In this section we will concentrate on some which are of greater importance to dense numerical linear algebra codes, namely:

- register allocation and scalar replacement, and

- instruction scheduling and software pipelining.

Register allocation and scalar replacement

Register allocation assigns variables to registers. It is of paramount importance because registers are almost always a scarce resource.

Register allocation is perhaps simplest to perform using local methods, which estimate the benefit from allocating a specific variable to a register inside a specific block of code, by some quantitative measure. One can then locally allocate those variables to available registers which show highest benefit.
Modern compilers often implement more sophisticated, global register allocation schemes, primarily based on graph coloring. The idea can be summarized in the following four steps.

1. Allocate as many symbolic registers as are necessary to hold all objects that are candidates for allocation (variables, temporaries, constants, etc.)

2. Construct an interference graph, with one node for each symbolic and each real register, and undirected edges representing interferences as follows.
   - An edge between two symbolic registers means that the corresponding objects have overlapping lifespan, and therefore cannot be assigned to the same register.
   - An edge between a symbolic register and a real register means that the corresponding variable cannot be assigned to that particular real register.
   - An edge is also introduced between any two real registers.

3. Color the nodes of the interference graph with $R$ colors, where $R$ is the number of available registers, so that adjacent nodes have different colors.

4. Assign each object to the register with the same color.

Effective register allocation is even more important for scientific codes, like dense numerical linear algebra ones, because it is often the case that such codes require whole submatrices to be stored in registers to obtain good performance (in contrast to just scalar variables and temporaries). This poses a problem, because traditional register allocation is designed to work for scalars.
To enable register allocation for elements of aggregate objects compilers use a transformation called scalar replacement [12]. Scalar replacement is performed by declaring scalar variables of the type of the elements of the aggregate object, copying the actual data to these scalar variables, and finally copying the results back to the aggregate object if necessary. This procedure enables the use of standard register allocation of elements of aggregate types.

To determine legality of scalar replacement, compilers need to perform alias analysis on both the whole aggregate object and the corresponding element.

**Instruction scheduling and software pipelining**

Instruction scheduling is the process of breaking the computation of the original program into small pieces which the underlying architecture understands and then serializing them in the appropriate order to achieve high performance. To effectively schedule instructions, a code scheduler needs detailed description of the architecture issue width and latencies and throughputs of all instructions.

One place where instruction scheduling is crucial, at least for dense numerical linear algebra codes, is inside a single basic block. Intuitively it consist of building a dependence DAG from the instructions in the block and then producing a topological sort of the DAG which executes fastest on the desired architecture. Even in this simple formulation instruction scheduling is an $NP$-complete problem, so compilers traditionally employ heuristics, like list scheduling, to perform it in polynomial time.

It is worth noting that instruction scheduling is not independent from register allocation. Assigning several temporaries to the same register creates dependences in the code and restricts the set of possible schedules. Many compilers perform
instruction scheduling immediately before register allocation and then repeat instruction scheduling if any spill code is generated. This approach has proven to be more effective than a single scheduling pass either before or after register allocation.

An important optimization that can improve performance of executing loops is software pipelining. It can be very effective on architectures which benefit from instruction-level parallelism (ILP). It works by allowing parts of several iterations to be executed simultaneously.

Intuitively, each iteration of a non-pipelined loop may take a while to ramp up to a point where it is exploiting a lot of instruction-level parallelism; similarly, the instruction pipelines drain out as the execution of the iteration completes. This idea is presented graphically in Figure 2.12.

![Figure 2.12: Non-pipelined loop](image)

After software pipelining the loop, we allow iterations to overlap which can potentially avoid the drops of sustained performance as shown in Figure 2.13.

![Figure 2.13: Pipelined loop](image)

We conclude the discussion with a simple example of software pipelining. Consider the following loop, where the operations $S_1$ and $S_2$ have to be executed
sequentially:

\[
\begin{align*}
&\text{for } i \in [1 : 1 : N] \\
&\{ \\
&\quad S_1(i); \\
&\quad S_2(i); \\
&\}
\end{align*}
\]

One software pipelined version of this loop, where \( S_1 \) and \( S_2 \) could be executed in parallel is:

\[
\begin{align*}
&\quad S_1(1); \\
&\text{for } i \in [1 : 1 : N - 1] \\
&\quad \{ \\
&\quad \quad S_2(i); \\
&\quad \quad S_1(i + 1); \\
&\quad \}
\end{align*}
\]

\[
\begin{align*}
&\quad S_2(N);
\end{align*}
\]

Note that \( S_1 \) could be communicating results to \( S_2 \) which might be register allocated. Software pipelining executes two iterations simultaneously, which could increase register pressure, and ultimately decrease performance. Therefore this transformation is not always beneficial.

### 2.2 Cache Oblivious Algorithms

Compiler transformations like tiling, discussed in Section 2.1.3, aim to improve locality to achieve better performance. When implementing such transformations, compilers need to prove their legality before they can apply them. Many of these transformations also require parameters; for example, tiling requires a value for
$N_B$, the cache tile size. Such optimization parameters are very architecture specific and inherently non-portable [10, 20, 54]. For these reasons, performance-conscious programmers often optimize their programs manually [25, 29].

The values of optimization parameters can be determined by performing global search in a subspace of the space of possible values or using analytical models to compute their values based on the hardware parameters. Chapters 5 and 6 of this thesis discuss these two approaches in detail. We refer to such approaches as cache-aware, because they are tailored to the specific machine and therefore re-tuning is necessary when the code is ported to a different architecture.

Cache-oblivious algorithms and data structures are an alternative approach to solving the problem of program optimization for memory hierarchies. In this approach, programs are aware that modern processors have complex multi-level memory hierarchies, but are oblivious to the particular parameters of these hierarchies (e.g., cache size at each level, block size, etc.) In principle, cache-oblivious programs provides good performance on a range of architectures without tuning.

Most commonly cache oblivious algorithms are described as divide-and-conquer versions of conventional algorithms. The idea is that, by partitioning the problem enough times, the working set of the computation is reduced, and it will eventually fit in a particular level of the memory hierarchy. The fit is not guaranteed to be precise, but often the number of data transfers between the different levels of the memory hierarchy are provably optimal in an asymptotic sense.

However, our experiments show that there is significant performance penalty in using cache oblivious algorithms in this context, and this mainly stems from the existence of optimized codes which often achieve near-peak performance. For example, one way to implement a cache oblivious matrix multiplication is as fol-
lows. An \( M \times K \) matrix \( A \) is multiplied by an \( K \times N \) matrix \( B \), and the result is accumulated into a \( M \times N \) matrix \( C \). The main idea of the algorithm is to divide the largest of the dimensions \( M \), \( N \), and \( K \), and solve the larger problem by solving two smaller problems. The partitioning can continue until all dimensions are of size 1, at which point the problem can be solved directly as a single multiply-accumulate operation.

Such a cache-oblivious implementation has asymptotically optimal I/O complexity, but it may have a significant performance problem.

- **Recursion overhead** – recursing down to a single matrix element produces a great amount of function call overhead, which significantly impairs performance.

- **Ignoring Registers** – recursing down to a single matrix element does not give the compiler a chance to optimize for the register file, which is the highest level of the memory hierarchy, and therefore the one from which performance benefits the most. The problem here arises partially from the fact that cache-oblivious algorithms assume that all levels of the memory hierarchy above main memory are transparent to the program.

Both these problems can be alleviated by inlining the function calls at the bottom of the call tree for some number of levels. This eliminates some of the call overhead and produces large basic blocks, which compilers can schedule and perform register allocation on. Unfortunately such inlining is of limited use, because it can easily overflow the instruction cache. Moreover the recursive order of accessing the matrix elements, while beneficial at the high-level, is disastrous for spacial locality of the inlined code; moreover, the irregular access pattern is not suitable
for the LRU replacement most caches implement. After performing a number of optimizations to address these problems, we were not able to achieve performance higher than roughly 60% of peak on most architectures, whereas carefully tuned cache-aware implementations easily attain more than 90% of peak.

One way to get the best of both worlds is to be partially aware of the memory hierarchy, say only to registers and L1 data cache. In an implementation like the one above, this would mean performing the recursive partitioning until some problem size is reached, and then using a L1 and register optimized kernel to solve the problem of that size directly.

Not surprisingly, cache oblivious algorithms perform best if paired with cache oblivious data structures. The idea of implementing such data structures is similar to the divide-and-conquer approach used to implement the corresponding algorithms. For example, in dense numerical linear algebra, matrices can be stored in a non-conventional manner, in which a space filling curve (e.g. Morton Z-order) is used to recursively order their elements. Such approach is most beneficial when the control flow of the corresponding algorithm working on this matrix, follows precisely the way the matrix is partitioned, because otherwise indexing the individual matrix elements is very hard.

Previous work exists on implementing cache-oblivious variants of other important data structures, like binary trees, priority queues, B-Trees, etc.

The real challenge we are facing in dense numerical algebra is to show whether the semi-aware approach, discussed above, is enough to achieve compatible performance with completely cache-aware algorithms on all architectures. One reason performance is similar at this point, is that for current architectures, a single level of cache tiling is enough to achieve near-peak performance. Intuitively, this picture
might change when the constantly increasing performance gap between processor and memory speed reaches some threshold. This is an active area of research we are pursuing at the moment, and therefore it is not further discussed in this thesis.
Chapter 3

High-performance Code Generation

In this section, we use the framework of restructuring compilers, described in Chapter 2, to describe the structure of the code generated by the ATLAS Code Generator. While reading this description, it is important to keep in mind that ATLAS is not a compiler. Nevertheless, thinking in these terms helps clarify the significance of the code optimization parameters used in ATLAS.

We concentrate on matrix-matrix multiplication (MMM), which is the key routine in the BLAS. Naïve MMM code is shown in Figure 3.1. We use the MATLAB notation \([First : Step : Last]\) to represent the set of all integers between \(First\) and \(Last\) in steps of \(Step\).

\[
\begin{align*}
\text{for } i & \in [0 : 1 : N - 1] \\
\text{for } j & \in [0 : 1 : M - 1] \\
\text{for } k & \in [0 : 1 : K - 1] \\
C_{ij} & = C_{ij} + A_{ik} \times B_{kj}
\end{align*}
\]

Figure 3.1: Naïve MMM Code

The code shown in Figure 3.1 can be optimized for locality by blocking for the L1 data cache and registers.

3.1 Tiling for the L1 data cache

ATLAS implements an MMM as a sequence of \(mini\)-MMMs, where each \(mini\)-MMM multiplies sub-matrices of size \(N_B \times N_B\). \(N_B\) is an optimization parameter whose value must be chosen so that the working set of the \(mini\)-MMM fits in the
33

In the terminology of restructuring compilers, the triply-nested loop of Figure 3.1 is tiled with tiles of size $N_B \times N_B \times N_B$, producing an outer and an inner loop nest. For the outer loop nest, code for both the JIK and IJK loop orders are implemented. When the MMM library routine is called, it uses the shapes of the input arrays to decide which version to invoke, as described later in this section. For the inner loop nest, only the JIK loop order is used, with $(j', i', k')$ as control variables. This inner loop nest multiplies sub-matrices of size $N_B \times N_B$, and we call this computation a mini-MMM.

3.2 Tiling for the register file

ATLAS represents each mini-MMM into a sequence of micro-MMMs, where each micro-MMM multiplies an $M_U \times 1$ sub-matrix of $A$ by a $1 \times N_U$ sub-matrix of $B$ and accumulates the result into an $M_U \times N_U$ sub-matrix of $C$. $M_U$ and $N_U$ are optimization parameters that must be chosen so that a micro-MMM can be executed without floating-point register spills. For this to happen, it is necessary that $M_U + N_U + M_U \times N_U \leq N_R$, where $N_R$ is the number of floating-point registers.

In terms of restructuring compiler terminology, the $(j', i', k')$ loops of the mini-MMM from the previous step are tiled with tiles of size $N_U \times M_U \times K_U$, producing an extra (inner) loop nest. The JIK loop order is chosen for the outer loop nest after tiling, and the KJI loop order for the loop nest of the mini-MMM after tiling.

The resulting code after the two tiling steps is shown in Figure 3.2. To keep this code simple, we have assumed that all step sizes in these loops divide the appropriate loop bounds exactly (so $N_B$ divides $M$, $N$, and $K$, etc.). In reality, code should also be generated to handle the fractional tiles at the boundaries of
// MMM loop nest (j, i, k)
// copy full A here
for j ∈ [1 : N_B : M]
  // copy a panel of B here
  for i ∈ [1 : N_B : N]
    // possibly copy a tile of C here
    for k ∈ [1 : N_B : K]
      // mini-MMM loop nest (j', i', k')
      for j' ∈ [j : N_U : j + N_B - 1]
        for i' ∈ [i : M_U : i + N_B - 1]
          for k' ∈ [k : K_U : k + N_B - 1]
            for k'' ∈ [k' : 1 : k' + K_U - 1]
              // micro-MMM loop nest (j'', i'')
              for j'' ∈ [j' : 1 : j' + N_U - 1]
                for i'' ∈ [i' : 1 : i' + M_U - 1]
                  C_{i'', j''} = C_{i'', j''} + A_{i'', k''} × B_{k'', j''}

Figure 3.2: MMM tiled for L1 data cache and registers

the three arrays; we omit this clean-up code to avoid complicating the description.

The strategy used by ATLAS to copy blocks of the arrays into contiguous storage is discussed later in this section. Figure 3.3 is a pictorial view of a mini-MMM computation within which a micro-MMM is shown using shaded rectangles. In this figure, the values assigned to variable K are produced by executing the two for loops in Figure 3.2 corresponding to indices k' and k''.
To perform register allocation for the array variables referenced in the micro-MMM code, ATLAS uses techniques similar to those presented in [11]: the micro-MMM loop nest $(j'', i'')$ in Figure 3.2 is fully unrolled, producing $M_U \times N_U$ multiply and add statements in the body of the middle loop nest. In the unrolled loop body, each array element is accessed several times. To enable register allocation of these array elements, ATLAS uses scalar replacement to introduce a scalar temporary for each element of $A$, $B$, and $C$ that is referenced in the unrolled micro-MMM code, and replaces array references in the unrolled micro-MMM code with references to these scalars. Appropriate assignment statements are introduced to initialize the scalars corresponding to $A$ and $B$ elements. In addition, assignment statements are introduced before and after the $k'$ loop to initialize the scalars corresponding to $C$ elements, and to write the values back into the array respectively. It is expected that the back-end compiler will allocate floating-point registers for these scalars.
3.4 Pipeline scheduling

The resulting straight-line code in the body of the $k''$ loop is scheduled to exploit instruction-level parallelism. Note that the operations in the $k''$ loop are the $M_U + N_U$ loads of A and B elements required for the micro-MMM, and the corresponding $M_U \times N_U$ multiplications and additions. On hardware architectures that have a fused multiply-add instruction, the scheduling problem is much simpler because multiplies and adds are executed together. Therefore, we only discuss the more interesting case when a multiply-add instruction is not present. An optimization parameter $FMA$ tells the code generator whether to assume that a fused multiply-add exists. The scheduling of operations can be described as follows.

- Construct two sequences of length $(M_U \times N_U)$, one containing the multiply operations (we will denote them by $mul_1, mul_2, \ldots, mu_{M_U \times N_U}$) and the other containing the add operations (we will denote them by $add_1, add_2, \ldots, add_{M_U \times N_U}$).

- Interleave the two sequences as shown below to create a single sequence that is obtained by skewing the adds by a factor of $L_s$, where $L_s$ is an optimization parameter. Intuitively, this interleaving separates most dependent multiplies and adds by $2 \times L_s - 1$ independent instructions to avoid stalling the processor.
pipeline.

\[
\text{mul}_1 \\
\text{mul}_2 \\
\ldots \\
\text{mul}_{L_s} \\
\text{add}_1 \\
\text{mul}_{L_s+1} \\
\text{add}_2 \\
\text{mul}_{L_s+2} \\
\ldots \\
\text{mul}_{M_U \times N_U - 1} \\
\text{add}_{M_U \times N_U - L_s} \\
\text{mul}_{M_U \times N_U} \\
\text{add}_{M_U \times N_U - L_s + 1} \\
\text{add}_{M_U \times N_U - L_s + 2} \\
\ldots \\
\text{add}_{M_U \times N_U}
\]

- Inject the \(M_U + N_U\) loads of the elements of A and B into the resulting sequence of arithmetic operations by scheduling a block of \(I_F\) (Initial Fetch) loads in the beginning and blocks of \(N_F\) loads thereafter as needed. \(I_F\) and \(N_F\) are optimization parameters.

- Unroll the \(k''\) loop completely. The parameter \(K_U\) must be chosen to be large enough to reduce loop overhead, but not so large that the body of the \(k'\) loop overflows the L1 instruction cache.

- Reorganize the \(k'\) loop to enable the target machine to overlap the loads from
one iteration with arithmetic operations from previous iterations. Techniques for accomplishing this are known as software pipelining or modulo scheduling [52].

Note that skewing of dependent adds and multiplies increases register pressure; in particular, the following inequality must hold to avoid register spills (that is, saving in memory the value stored in a processor register):

\[ M_U \times N_U + M_U + N_U + L_s \leq N_R \]  

(3.1)

### 3.5 Additional details

There are several details we have not discussed so far.

- ATLAS considers a primitive form of L2 cache tiling, driven by a parameter called \( \text{CacheEdge} \). ATLAS empirically finds the best value of \( \text{CacheEdge} \) and uses it to compute \( K_P \), based on Inequality 3.2.

\[ 2 \times K_P \times N_B + N_B^2 \leq \text{CacheEdge} \]  

(3.2)

\( K_P \) is further trimmed to be a multiple of \( N_B \). The computed value of \( K_P \) is used to block the \( K \) dimension of the original problem for one additional level of the memory hierarchy.

- ATLAS chooses the outermost loop order (shown as JIK in Figure 3.2) during runtime. This technique is known as versioning, because it requires both versions of the code to be compiled in the library.
The decision of which loop order to choose is based on the size of matrices $A$ and $B$. If $A$ is smaller than $B$ ($N < M$), ATLAS chooses the JIK loop order. This guarantees that if $A$ fits completely into the L2 or a higher cache level, it is reused successfully by the loop nest. Similarly, if $B$ is the smaller matrix ($M < N$), ATLAS chooses the IJK loop order.

For brevity, we consider only the JIK loop order.

- Unless the matrices are too small or too large, ATLAS copies tiles of matrices $A$, $B$ and $C$ to sequential memory to reduce the number of conflict misses and TLB misses during the execution of a mini-MMM. Copying is performed in a manner that allows the copied tiles to be reused by different mini-MMMs. The comments in Figure 3.2 and the discussion below explain how this goal is achieved for the JIK loop order.

  - Copy all tiles of $A$ before the beginning of the outermost $j$ loop. This is necessary as these tiles are fully reused in each iteration of the $j$ loop.

  - Copy all tiles from the $j^{th}$ vertical panel of $B$ before the beginning of the $i$ loop. This is necessary as this panel is fully reused by each iteration of the $i$ loop.

  - The single $(i, j)$ tile of $C$ is copied before the beginning of the $k$ loop if $\frac{K_P}{N_B} \geq 12$. This may reduce TLB misses, which may be beneficial since this tile is reused by each iteration of the $k$ loop, provided that the cost of copying the tile of $C$ to a temporary buffer and back can be amortized by the computation (large enough $K_P$).

If the matrices are very small or if there is insufficient memory for copying tiles, the cost of copying might outweigh the benefits of reducing conflict
misses during the computation. Therefore, ATLAS generates non-copying versions of mini-MMM as well and decides at runtime which version to use. Without copying, the number of conflict misses and TLB misses may rise, so it makes sense to use a smaller tile size for the non-copying mini-MMM. In ATLAS, this tile size is another optimization parameter called $NCN_B$ (non-copying $N_B$). Roughly speaking, the non-copy version is used if (i) the amount of computation is less than some threshold ($M \times N \times K$ in Figure 3.1 is less than some threshold), and (ii) at least one dimension of one of the three matrices is smaller than $3 \times NCN_B$. The non-copy version is also used when there is insufficient memory to perform the copying.

3.6 Discussion

Table 3.1 lists all optimization parameters.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N_B$</td>
<td>L1 data cache tile size</td>
</tr>
<tr>
<td>$NCN_B$</td>
<td>L1 data cache tile size for non-copying version</td>
</tr>
<tr>
<td>$MU, NU$</td>
<td>Register tile size</td>
</tr>
<tr>
<td>$K_U$</td>
<td>Unroll factor for $k'$ loop</td>
</tr>
<tr>
<td>$L_s$</td>
<td>Latency for computation scheduling</td>
</tr>
<tr>
<td>$FMA$</td>
<td>1 if fused multiply-add available, 0 otherwise</td>
</tr>
<tr>
<td>$FF, IF, NF$</td>
<td>Scheduling of loads</td>
</tr>
</tbody>
</table>

It is intuitively obvious that the performance of the generated mini-MMM code suffers if the values of the optimization parameters in Table 3.1 are too small
or too large. For example, if $M_U$ and $N_U$ are too small, the $M_U \times N_U$ block of computation instructions might not be large enough to hide the latency of the $M_U + N_U$ loads. On the other hand, if these parameters are too large, register spills happen. Similarly, if the value of $K_U$ is too small, there is more loop overhead, but if this value is too big, the code in the body of the $k'$ loop will overflow the instruction cache. The goal is to determine optimal values of these parameters for obtaining the best mini-MMM code.
Chapter 4

Measuring Hardware Parameters

4.1 Introduction to X-Ray

There is growing interest in the design of self-optimizing computer systems that can improve their performance without human intervention. Some of these systems such as ATLAS [60] and FFTW [27] optimize themselves statically at installation time, while other systems such as those envisioned in IBM’s autonomic systems initiative [2] optimize their performance continuously during execution.

In many self-optimizing systems, the process of optimization involves choosing optimal values for certain performance-critical parameters. For example, when ATLAS, a portable system for generating numerical linear algebra libraries, is installed on a machine, it performs a series of experiments to determine optimal values for software parameters like loop tile sizes and loop unroll factors, and then generates appropriate code with these parameter values. Although the optimal values of the parameters depend on hardware resources like cache capacity, number of registers, etc. and therefore vary from machine to machine, ATLAS is able to adapt itself automatically in this way to each platform without human intervention.

In general, the space of possible values for optimization parameters is too large to be explored exhaustively, so it is necessary to limit the search for optimal parameter values. For example, to find the optimal cache tile size, it is sufficient to limit the search to tile sizes that are smaller than the capacity of the cache; larger tiles will not fit in the cache and cannot be optimal. Similarly the search space for register tiles can be bounded by the number of registers in the processor. In general therefore, bounding the search space of optimization parameters requires
knowledge of hardware parameters such as the capacity of the cache, and the num-
ber of registers. Furthermore, for some programs, analytical models can be used to 
compute optimal values for code optimization parameters; not surprisingly, these 
models may require more detailed and accurate information about the hardware, 
such as the line size and associativity of various levels of the memory hierarchy [64].

For software to be self-tuning and self-optimizing, it is necessary that these 
hardware parameters be determined automatically without human intervention. 
On some machines, it may be possible to determine some of this information by 
reading special hardware registers or records in the operating system [21]. However, 
most processors and operating systems do not support such mechanisms or provide 
very limited support; for example, we do not know of any platform on which the 
latency and throughput of instructions can be determined in this manner.

An alternative approach is to use carefully crafted micro-benchmarks to mea-
sure hardware parameters automatically. Perhaps the most well-known micro-
benchmark is the Hennessy-Patterson benchmark [33] that determines memory 
hierarchy parameters by measuring the time required to access a series of array 
elements with different strides. However, the timing results from this benchmark 
must be analyzed manually to determine memory hierarchy parameters, so it is 
not suitable for incorporation into an autonomic system. Furthermore, the bench-
mark makes fixed stride accesses to memory, so hardware pre-fetching on machines 
like the Power 3 can corrupt the timing results. Finally, this benchmark does not 
determine the latency and throughput of instructions, or whether instructions like 
fused multiply-add exist. Some of these problems were addressed in more recent 
tools like Calibrator [41], lmbench [45], and MOB [9], but our experience with 
these tools is somewhat mixed, as we describe in Appendix A.
This chapter makes the following contributions.

• We describe a robust, easily extensible framework for implementing micro-benchmarks to measure hardware parameters.

• We design a number of novel algorithms for measuring memory hierarchy parameters and other architectural features, and show how they can be implemented in this framework.

• We describe a tool called X-Ray, which is an implementation of these ideas, and evaluate its capabilities on seven modern architectures. We show that it produces more accurate and complete results than preexisting tools.

For portability, X-Ray is implemented completely in C [24]. One of the interesting challenges of this approach is to ensure that the C compiler does not perform any high-level optimizations on our benchmarks that might pollute the timing results.

4.2 X-Ray Framework Overview

Figure 4.1 shows the structure of a micro-benchmark in the X-Ray framework. Each hardware parameter is measured by one such micro-benchmark.

For example, consider measuring the number of available registers for a particular data type $T$. One way to determine this value is to execute a number of experiments, all of which perform the same computations, but on a different number of variables ($N$) of type $T$. When $N$ exceeds the number of available registers for type $T$, not all variables can be register allocated, and execution time should increase substantially. The number of available registers can be inferred from this cross-over point.
Some general conclusions can be drawn from this example. A single micro-benchmark may need to execute several timing experiments, which we call nano-benchmarks. Since there may be no \textit{a priori} bound of the number of required nano-benchmarks, we need a \textit{Nano-benchmark Generator}, which can produce \textit{Nano-benchmark C Code} from a high-level \textit{Nano-benchmark Specification}. Finally, generation should happen on-the-fly since the results of one nano-benchmark may determine the nano-benchmark to be executed next.

The execution of a micro-benchmark is orchestrated by its \textit{Control Engine}, which chooses which nano-benchmarks to execute, along with the appropriate parameters, and the order in which to execute them. It determines the value of the hardware parameter based on these timing results.

Some micro-benchmarks may also require input from the results of running other micro-benchmarks. For example, to determine the latency of an instruction in cycles rather than in nanoseconds, the control engine needs to know the cycle time of the processor. This can be specified by the user or it can be measured by another micro-benchmark as discussed in Section 4.3.
4.2.1 Nano-benchmarks

Even with access to a high-resolution timer, it is hard to accurately time operations that take only a few CPU cycles to execute. Suppose we want to measure the time required to execute a C statement $S$. If this time is small compared to the granularity of the timer, we must measure the time required to execute this statement some number of times $R_S$ (dependent on $S$), and divide that time by $R_S$. If $R_S$ is too small, the time for execution cannot be measured accurately, whereas if $R_S$ is too big, the experiment will take longer than it needs to.

$$R_S \leftarrow 1;$$

$$\text{while } (\text{measure}_S(R_S) < t_{\text{min}})$$

$$R_S \leftarrow R_S \times 2;$$

$$\text{return } (\text{measure}_S(R_S) \div R_S);$$

Figure 4.2: High-level structure of a nano-benchmark

Figure 4.2 shows the timing strategy used in X-Ray nano-benchmarks. In this code, $\text{measure}_S(R_S)$ measures the time required to execute $R_S$ repetitions of statement $S$. To determine a reasonable value for $R_S$, the code in Figure 4.2 starts by setting $R_S$ to 1, and then doubles it until the experiment runs for at least $t_{\text{min}}$ seconds. The value of $t_{\text{min}}$ can be specified by the user and defaults to 0.25 seconds in the current implementation.

A simplistic implementation of $\text{measure}_S$ is shown in Figure 4.3(a). The call to $\text{now}()$ is assumed to invoke the most accurate timer available on the platform. The code in Figure 4.3(a) incurs considerable loop overhead, which might even be greater than the time spent in executing $S$. To address this problem, we can unroll the loop $U$ times as shown in Figure 4.3(b).
Another problem is that restructuring compiler optimizations may corrupt the experiment. For example, consider the case when we want to measure the latency of a single addition. In our framework, we would measure the time taken to execute the C statement $p_0 = p_0 + p_1$. It is important to optimize $p_0$ and $p_1$ by allocating them in registers, but it is crucial not to optimize the $U$ statements in the loop body to $p_0 = p_0 + U \times p_1$, which would prevent from timing the original statement correctly.
To solve such problems, we need to generate programs which the compiler can aggressively optimize, without disrupting the sequence and type of operations, whose execution time we want to measure. We solve this problem using a switch statement on a volatile variable $v$ as shown in Figure 4.3(c). The semantics of C require that all accesses of $v$ go to the memory hierarchy, and therefore the compiler cannot assume anything about which case of the switch is selected. Because there is potential control flow to each of the case blocks, it is impossible to combine or reorder them in any way.

The final problem is that if the compiler is able to deduce that the result of the computations performed in $S$ is not used in the rest of the code, it might perform dead-code elimination and remove all instances of $S$ altogether. To prevent this unwanted optimization we assign the initial value of all variables that appear in $S$ from appropriately typed volatile variables in the initialize statement and copy their final values back to the same volatile variables in the use statement.

As we will see in Section 4.3, there are cases where we wish to measure the performance of a sequence of different statements $S_1, S_2, \ldots, S_n$. To prevent the compiler from optimizing this sequence, the code generator will give each $S_i$ a different case label, generating code of the form shown in Figure 4.3(d). In this figure, the number of case labels $W$ is the smallest multiple of $n$ greater than or equal to $U$.

### 4.2.2 Nano-benchmark Generator

The X-Ray nano-benchmark generator accepts as an input a nano-benchmark specification and produces nano-benchmark C code structured as shown in Figure 4.3(d).
The nano-benchmark specification is a tuple which contains a statement $S$ to be timed and type information for all variables in $S$. For example, to measure the latency of double-precision floating point ADD operation, we use the nano-benchmark specification $\langle p_1 = p_1 + p_2, (p_1, p_2 : \text{F64}) \rangle$, which means that we time the statement $p_1 = p_1 + p_2$, where $p_1$ and $p_2$ are variables of type double (defined as F64 in X-Ray). Given this specification, the nano-benchmark generator can produce code as shown in Figure 4.3(c). Generating code of the form shown in Figure 4.3(d) is more complex and requires the first element of the tuple to be a function $f : \text{integer} \rightarrow \text{string}$, which computes the code for statement $S_i$ from the case label $i$.

### 4.2.3 Implementing a new micro-benchmark

The following two steps describe the process of implementing a new micro-benchmark within the X-Ray framework.

1. Decide what timing experiments are needed and implement the corresponding nano-benchmarks. If the structure of these nano-benchmarks fits the output of the X-Ray nano-benchmark generator, as presented in Figure 4.3(d), one only needs to provide the corresponding nano-benchmark specifications.

2. Implement the micro-benchmark control engine, describing which nano-benchmarks to run, with what parameters, in what order, and how to produce a final result from the external parameters and the timings.

As we will see in Section 4.3, both these steps require very few lines code.
4.3 CPU Micro-benchmarks

We now describe how X-Ray measures a number of key CPU parameters.

4.3.1 CPU Frequency

CPU frequency \( F_{CPU} \) is an important hardware parameter because other parameters are measured relative to it (in clock cycles). X-Ray assumes that dependent integer additions can be executed at the rate of one per cycle, which is valid for most current processors. The assumption of dependence is important because modern architectures can often issue two or more independent integer addition operations in one cycle, so timing independent addition operations would be misleading.

For this micro-benchmark we use a single nano-benchmark with specification \( S = \langle p_0 = p_0 + p_1, \langle p_0, p_1 : \text{int} \rangle \rangle \). Given the time \( \text{time}(S) \) in nanoseconds required to execute the statement \( S \), we compute the CPU frequency in MHz as follows.

\[
F_{CPU} \leftarrow \frac{1000}{\text{time}(S)}
\]

4.3.2 Instruction Latency

The latency \( L_{O,T} \) of an operation (instruction) \( O \), with operands of type \( T \), is the number of cycles after one such instruction is dispatched until its result becomes available to subsequent dependent instructions.

For this micro-benchmark we use a nano-benchmark with specification \( S_{O,T} = \langle p_0 = O(p_0, p_1), \langle p_0, p_1 : T \rangle \rangle \). We then compute the instruction latency in clock cycles as follows.

\[
L_{O,T} \leftarrow \frac{\text{time}(S_{O,T})}{1000 \div F_{CPU}}
\]
4.3.3 Instruction Throughput

The throughput $\text{TP}_{O,T}$ of an operation (instruction) $O$, with operands of types $T$, is the rate in cycles at which the CPU can issue independent instructions of that type. On modern processors the throughput of an instruction is usually much smaller than its latency, because of pipelining and super-scalar execution.

To measure $\text{TP}_{O,T}$, we use a nano-benchmark specification

$$S_{O,T,N} = \langle p_i \% N = O(p_i \% N, p_N), \langle p_0, p_1, \ldots, p_N : T \rangle \rangle .$$

Note that this specification generates code of the form shown in Figure 4.3(d). It is further parameterized by $N$ – the number of independent instructions to generate. For example, the sequence of statements generated for $N = 3$ is the following.

```c
    case 0 : p_0 = O(p_0, p_3);
    case 1 : p_1 = O(p_1, p_3);
    case 2 : p_2 = O(p_2, p_3);
    case 3 : p_0 = O(p_0, p_3);
    ...
    case W : p_2 = O(p_0, p_3);
```

We then compute the instruction throughput in clock cycles as follows.

$$N \leftarrow 2;$$

$$\text{while } \left( \frac{\text{time}(S_{O,T,N})}{\text{time}(S_{O,T,N-1})} > 1 - \epsilon \right)$$

$$N \leftarrow N + 1;$$

$$\text{TP}_{O,T} \leftarrow \frac{\text{time}(S_{O,T,N-1})}{1000 \cdot F_{CPU}};$$

The nano-benchmark code for $S_{O,T,N}$ exhibits instruction-level parallelism (ILP) on the order of $N$. The control engine times the nano-benchmark for successively
growing values of $N$ while performance continues to increase due to the additional ILP. When the performance levels off for some $N$, the throughput $TP_{O,T}$ is computed as the last (fastest) timing divided by the clock cycle time.

In practice, the X-Ray implementation of the throughput micro-benchmark is somewhat more complex. The reason for this increased complexity is that the implementation we just described does not permit the exploitation of instruction-level parallelism in statically scheduled VLIW cores because the case labels prevent the compiler from scheduling VLIW bundles with more than one instruction. Therefore we use a nano-benchmark specification with a more complex statement generating function, that puts $B$ independent instructions at each case statement, as follows.

$$S_{O,T,N,B} =$$

$$<$$

$$\{$$

$$p_{(i\times B+0)\%N} = O(p_{(i\times B+0)\%N}, p_N);$$

$$p_{(i\times B+1)\%N} = O(p_{(i\times B+1)\%N}, p_N);$$

$$\ldots$$

$$p_{(i\times B+B-1)\%N} = O(p_{(i\times B+B-1)\%N}, p_N);$$

$$\};$$

$$\langle p_0, p_1, \ldots, p_N : T \rangle$$

$$>$$
The corresponding control engine algorithm is as follows.

\[ N \leftarrow 2; \]
\[ \text{while} \left( \frac{\text{time}(S_{O,T,N,1})}{\text{time}(S_{O,T,N-1,1})} > 1 - \epsilon \right) \]
\[ N \leftarrow N + 1; \]
\[ B \leftarrow 2; \]
\[ N \leftarrow N - 1; \]
\[ \text{while} \left( \frac{\text{time}(S_{O,T,N \times B,B})}{\text{time}(S_{O,T,N \times (B-1),B-1})} > 1 - \epsilon \right) \]
\[ B \leftarrow B + 1; \]
\[ TP_{O,T} \leftarrow \frac{\text{time}(S_{O,T,N \times (B-1),B-1})}{1000 \div F_{CPU}}; \]

### 4.3.4 Instruction Existence

Many embedded processors do not have dedicated floating-point hardware. Some have single-precision floating-point functional units, but not double-precision ones. In case dedicated floating-point hardware is not present, usually an emulation library is used. In X-Ray we can easily determine the presence of dedicated floating-point hardware by measuring the latency of a floating-point ADD operation of the appropriate type. If the latency is more than a few cycles (10 in our implementation), we conclude that the operation is emulated, otherwise we conclude that it is executed in hardware.

In certain cases, it is not obvious how a certain C statement is translated to instructions. One very common operation for numerical applications is \[ p_1 = p_1 + p_2 \times p_3 \]. On some platforms, this statement is compiled into a single fused multiply-add instruction (FMA), while on some it is compiled into a separate multiply and add instructions. If an FMA instruction does not exist, the compiler will need an extra register to store the intermediate value and schedule two instructions
instead of one. This has an impact on how such sequences of statements need to be scheduled. For example, the ATLAS [60, 61] library generator produces different code for the compiler depending upon the existence of an FMA instruction.

With an FMA instruction the CPU can execute an add instruction “for free” together with a multiply instruction. Therefore we determine the existence of FMA by comparing the throughput of a simple multiply with that of a fused multiply-add.

### 4.3.5 Number of Registers

To measure the number of registers $\mathbf{NR}_T$ of particular type $T$, we use the specification $S_{T,N} = \langle p_i \% N = p_i \% N + p_{(i+N-1)} \% N, (p_0, p_1, \ldots, p_N : T) \rangle$. For example, the sequence of statements generated for $N = 4$ is as follows.

```plaintext
    case 0 : p_0 = p_0 + p_3;
    case 1 : p_1 = p_1 + p_0;
    case 2 : p_2 = p_2 + p_1;
    case 3 : p_3 = p_3 + p_0;
    case 4 : p_0 = p_0 + p_3;
        \ldots
    case W : p_3 = p_3 + p_0;
```

If all of $p_i$ are allocated in registers, the time per operation is much smaller than when some are allocated in memory. The goal is to determine the maximum $N$, for which no variables are allocated to memory. The control engine doubles $N$ until it observes a drop in performance. After that it performs a binary search in
the interval \([N \div 2, N]\). The actual control engine algorithm is as follows.

\[
N \leftarrow 4; \\
\textbf{while} \left( \frac{\text{time}(S_{T,N})}{\text{time}(S_{T,2})} < 1 + \epsilon \right) \\
N \leftarrow N \times 2; \\
R \leftarrow N; \\
L \leftarrow \frac{N}{2} \\
\textbf{while} (R - L > 1) \\
P \leftarrow \frac{(R + L)}{2}; \\
\textbf{if} \left( \frac{\text{time}(S_{T,P})}{\text{time}(S_{T,2})} < 1 + \epsilon \right) \\
R \leftarrow P; \\
\textbf{else} \\
L \leftarrow P; \\
NR_T \leftarrow L;
\]

4.3.6 SMP and SMT parallelism

To measure the number of processors in a SMP architecture, X-Ray once again uses the “throughput” nano-benchmark. Its configuration is instantiated with \(\langle \text{ADD, I32, } n \rangle\), where \(n\) is the value at which the CPU is saturated as measured by the Initiation Interval micro-benchmark.

The number \(p\) of concurrent instances of this configuration, which exhibits no slowdown compared to running a single instance characterizes the number of CPUs in a SMP. Reading the number of CPUs with an OS call returns the number \(v\) of virtual SMT processors. The SMT per CPU of the system is computed as \(\frac{v}{p}\).

To find which two virtual processors share the same physical processor, X-Ray executes instances of the aforementioned configuration concurrently on both.
there is no slowdown, the two virtual processors do not share a physical processor.

### 4.4 Memory Hierarchy Micro-benchmarks

We present micro-benchmarks for measuring the parameters of the memory hierarchy of a platform, including all levels of cache, registers, and the TLB. Existing tools such as lmbench [44], Calibrator [41] and MOB [9] measure some of these memory hierarchy parameters, but our experiments show that none of them offer the same coverage of parameters or the accuracy of our micro-benchmarks. These tools implement variations of the micro-benchmark developed by Saavedra [53], which is reproduced in Hennessy and Patterson’s architecture book [33] and is discussed in Section 4.4.1. This benchmark, which is a C program, measures the time required to access a series of array elements with different strides. The timing results are fairly complex because the micro-benchmark considers all levels of the memory hierarchy simultaneously. Therefore, these results are usually interpreted manually to obtain the memory hierarchy parameters. Although tools like MOB, Calibrator and lmbench can determine some cache parameters automatically from these timing results, none of them are able to measure cache associativity for example. Moreover, optimizations performed by modern compilers when compiling the C code can confuse the timing measurements. Yet another problem is that hardware pre-fetching for fixed-stride accessed to memory on machines like the IBM Power architecture can compromise the timing measurements further.

The micro-benchmarks we present determine cache parameters for one level of cache at a time, rather than considering all levels simultaneously. In particular, when measuring the parameters of a cache level \( i \), the micro-benchmarks ensure that higher cache levels \( L_1, L_2, \ldots, L_{i+1} \) are “transparent” to the measurements
in the sense that the memory accesses are guaranteed to miss in those caches. Of course this is not a problem for the L1 cache, so our micro-benchmarks use fixed stride accesses there, as we discuss in Section 4.4.3. However, our algorithms are novel, and we take care to ensure that compiler optimizations and hardware prefetching do not compromise the timing measurements. In Section 4.4.4, we show that by using more complex memory access patterns (in particular, a sequence of sequences), we can measure the parameters of lower cache levels without interference from higher cache levels. In Sections 4.4.5 and A.1.3, we show how some TLB parameters and the number of registers can be measured automatically. We present experimental results in Appendix A that show that we provide better accuracy and coverage of memory hierarchy parameter measurement than preexisting tools. Finally, in Section 4.5, we discuss future work.

4.4.1 Previous Approaches

The general approach to measuring memory hierarchy parameters is to repeatedly access the elements of a large array in memory using different strides, and measure the average time per access. The results are then interpreted to deduce different memory hierarchy parameters. The most widely known micro-benchmark for such measurements is the benchmark of Saavedra [53], a stylized version of which is presented in Figure 4.4. We make the following observations.

1. The code performs series of experiments for pairs of the form \((csize, \text{stride})\), where the array size \(csize\) varies between \(\text{CACHE\_MIN}\) and \(\text{CACHE\_MAX}\) and the stride \(\text{stride}\) varies between 1 and \(csize\). Both are restricted to powers of 2.
```c
#define SAMPLE (5)
#define CACHE_MIN (1024)
#define CACHE_MAX (16*1024*1024)

int x[CACHE_MAX];

int main()
{
    int temp;
    for (int csize = CACHE_MIN; csize <= CACHE_MAX; csize *= 2)
    {
        for (int stride = 1; stride <= csize / 2; stride *= 2)
        {
            double time = 0.0;
            int steps = 0;
            int tsteps = 0;
            int limit = csize - stride + 1;
            do
            {
                double time0 = get_time();
                for (int i = SAMPLE * stride; i != 0; --i)
                {
                    for (int index = 0; index < limit; index += stride)
                        x[index]++;
                }
                steps++;
                time += get_time() - time0;
            } while (time < 1.0);
            do
            {
                double time0 = get_time();
                for (int i = SAMPLE * stride; i != 0; --i)
                {
                    for (int index = 0; index < limit; index += stride)
                        temp += index;
                }
                tsteps++;
                time -= get_time() - time0;
            } while (tsteps < steps);
        }
    }

    printf("size: %d, stride: %d, time: %d",
           csize * sizeof(int), stride * sizeof(int),
           (int)(time * 1E9 / (steps * SAMPLE * stride * 
            ((limit - 1) / (stride + 1))));
}
```

Figure 4.4: Standard memory hierarchy benchmark
2. For each pair \((csize, stride)\), the benchmark traverses the array \(x\) with the specified stride enough times \((\text{SAMPLE} \times \text{steps})\) to ensure that the total time spent is at least one second.

3. The measurement for the same pair \((csize, stride)\) is repeated using the exact same looping code structure, and the same number of times, but this time accessing a single scalar variable \((\text{temp})\) instead of elements of \(x\).

The benchmark has problems at both the algorithmic and implementation level, as summarized below.

1. Algorithmic Level

   (a) The benchmark considers all levels of the memory hierarchy simultaneously, so each timing result is possibly influenced by several parameters from different cache levels. Therefore, the interpretation of the timing results is complex.

   (b) The benchmark does not interpret the timing results to produce actual memory hierarchy parameters itself, but rather produces a set of measurements that need to be interpreted manually.

   (c) The benchmark uses only array sizes restricted to powers of 2, which prevents it from measuring cache capacities that are not a power of 2. However, an increasing number of caches on modern architectures, such as the Level 3 cache on the Itanium 2, have capacities that are not a power of 2.

2. Implementation Level
(a) The source code uses a very complex looping structure, which is the source of substantial loop overhead. An attempt is made to account for that overhead by measuring and subtracting the time of execution of a cloned version the same looping structure which does not perform any memory accesses. Unfortunately, there is no control over the back-end compiler, so different code may be produced for the two replicas, thus yielding inaccurate results.

(b) All memory accesses are independent, which allows an aggressively optimizing compiler to schedule them in a way so that some overlap. This is mainly a problem for measuring access latency, but also cause inaccuracies in measuring other parameters.

(c) Both memory read and write are performed on the current array element, which introduces interference with write buffers and further prohibit the measurement of these operations in isolation.

(d) The addressing mode used to access array elements involves base address and offset and on many RISC architectures this operation requires an extra address computation instruction before performing the actual memory access instruction.

(e) The source code does not use the values of accessed array elements and more importantly the value of the temp variable for anything meaningful, so a smart optimizing compiler can prune portions of the code during dead code elimination.

(f) There is a constant stride between accesses of array elements and some modern architectures provide hardware that is able to prefetch constant
stride accesses to memory into the higher levels of the memory hierarchy.

(g) It is implicitly assumed and very important for the benchmark that the array \( x \) is stored in a contiguous chunk of memory. In reality, it is only

guaranteed to be contiguous in virtual memory and can be fragmented

in physical memory. In many cases lower cache levels are physically

addressed, which invalidates this important assumption.

The existing systems we examined all use this micro-benchmark in one form or another, although some of them attempt to address some of these problems in various ways. Our approach is very different at the algorithmic level, and it

eliminates the implementation problems discussed above.

4.4.2 Compactness of Sequences

Our micro-benchmarks measure the associativity \( (A) \), block size \( (B) \), capacity \( (C) \), and hit latency \( (l) \) of caches. The first three parameters are sometimes referred to as the \( \langle A, B, C \rangle \) of caches.

\[
\begin{align*}
\text{tag} & \quad \text{index} & \quad \text{offset} \\
\hline
\text{t=20} & \quad \text{i=7} & \quad \text{b=5} \\
\end{align*}
\]

Figure 4.5: Memory address decomposition on Pentium III

Figure 4.5 shows the typical structure of a memory address. We use the Intel Pentium III architecture in the following explanations. On these machines, the L1 data cache is organized as \( \langle A, B, C \rangle = \langle 4, 32, 16KB \rangle \). Therefore the cache contains

\[
C \div B = 16384 \div 32 = 512 \text{ individual blocks, divided into } 512 \div A = 512 \div 4 = 128
\]

sets of 4 blocks each. The highest \( t = 20 \) bits constitute the block \( \text{tag} \), \( i = 7 \) bits
are needed to index one of the 128 sets, and \( b = 5 \) bits are needed to store the offset of a particular byte within the 32-byte block.

**Definition 1.** For a cache with associativity \( A \) and capacity \( C \), we define the stride \( T \) of that cache as \( T \equiv \frac{C}{A} \).

Note that \( T = 2^{i+b} \), and thus \( C = A \times 2^{i+b} \). Lemma 1 gives another characterization of \( T \).

**Lemma 1.** Consider a cache with stride \( T \), and addresses \( m_0 \) and \( m \) aligned to a cache block boundary. The address \( m \) maps to the same cache set as \( m_0 \) iff \( m = m_0 + k \times T \) for some integer \( k \).

**Proof.** Follows directly from the definition.

Unlike cache stride, associativity and capacity do not have to be a power of 2. For example, some versions of the Intel Itanium 2 have a 24-way set associative L3 cache with a capacity of 6MB.

If \( W \) is a set of addresses, we define \( \text{project}_i (W) \) to be the subset of \( W \) containing only the addresses that map to cache set \( i \), and \( \text{indices} (W) \) to be the set of cache indices of the elements of \( W \).

**Definition 2.** For a set of addresses \( W \), and an index \( i \),

\[
\text{project}_i (W) \equiv \{ m \in W : \text{index} (m) = i \}
\]

**Definition 3.** For a set of addresses \( W \),

\[
\text{indices} (W) \equiv \{ i : \text{project}_i (W) \neq \emptyset \}
\]

We assume that set-associative caches implement the least-recently-used (LRU) replacement policy. This assumption is reasonable because most modern processors
implement variants of this policy. Moreover, our experimental results show that our micro-benchmarks can be accurate even when the policy is not LRU.

**Sequences**

Some of our micro-benchmarks access sequences of \( N \) addresses, where successive addresses are separated by a stride \( S = 2^\sigma \) as shown in Figure 4.6(a). Such sequences are completely characterized by their starting address \( m_0 \), stride \( S \) and number of elements \( N \). Therefore we use the notation \( \langle m_0, S, N \rangle \) to represent them. To measure parameters of multilevel memory hierarchies our micro-benchmarks use sequences of sequences, as shown in Figure 4.6(b). To represent them we use the notation \( W = \langle \langle m_0, s, n \rangle, S, N \rangle \).

![Diagram](image)

(a) \( \langle m_0, S, N \rangle \)

(b) \( \langle \langle m_0, s, n \rangle, S, N \rangle \)

Figure 4.6: Sequences of sequences

**Definition 4.**

\( \langle m_0, S, N \rangle \equiv [m_0, m_0 + S, \ldots, m_0 + (N - 1) S] \)
(b) \( \langle \langle m_0, s, n \rangle, S, N \rangle \equiv \bigcup_{i \in [0,N-1]} \langle m_0 + i \times S, s, n \rangle \)

In Definition 4(b), we call each subsequence \( \langle m_0 + i \times S, s, n \rangle \) of \( \langle \langle m_0, s, n \rangle, S, N \rangle \) an *inner* subsequence.

Notice that the sequence of addresses in Figure 4.6(b) can also be expressed as \( \langle \langle m_0, S, N \rangle, s, n \rangle \). This property is expressed in Lemma 2.

**Lemma 2.** \( \langle \langle m_0, s, n \rangle, S, N \rangle \equiv \langle \langle m_0, S, N \rangle, s, n \rangle \)

**Compactness**

We determine cache parameters by measuring the average time per memory access when accessing the elements of certain sets of memory addresses.

When all addresses of an address sequence \( W \) can coexist together in a cache we say that \( W \) is *compact* with respect to that cache and the average access time is the cache hit latency \( l_{hit} \). When the sequence is not compact and we repeatedly access its elements the cache will suffer some misses. If every single access is a cache miss, we say that \( W \) is *non-compact* and the average access time is the cache miss latency \( l_{miss} \), which is typically much greater than \( l_{hit} \). Finally, when some accesses are cache hits and some are cache misses, the average access time is between \( l_{hit} \) and \( l_{miss} \) and we say that \( W \) is *semi-compact*. Definition 5 presents this concepts formally.

**Definition 5.** For a cache with associativity \( A \),

\[
compact(W) \equiv \forall i \in \text{indices}(W) : |\text{project}_i(W)| \leq A
\]

\[
non-compact(W) \equiv \forall i \in \text{indices}(W) : |\text{project}_i(W)| > A
\]

\[
semi-compact(W) \equiv \neg compact(W) \land \neg non-compact(W)
\]
The definition says that, for any cache index from the set of indices for \( W \), a compact sequence will have at most \( A \) elements with this index, while a non-compact sequence will have at least \( A + 1 \) elements with this index. A sequence is semi-compact if there is an index with at most \( A \) elements, as well as an index with at least \( A + 1 \) elements.

**Lemma 3.** Compact sequences have the following properties.

(a) For a cache with capacity \( C \) and block size \( B \), and an address \( m_0 \), aligned on a cache block boundary, the half-open interval \([m_0, m_0 + C)\) is compact.

(b) A subset of a compact sequence is compact.

(c) If \( \text{indices}(W_1) \cap \text{indices}(W_2) = \emptyset \), and \( W_1 \) and \( W_2 \) are compact, then \( W_1 \cup W_2 \) is compact.

(d) If \( \text{indices}(W_1) \cap \text{indices}(W_2) = \emptyset \), and \( W_1 \cup W_2 \) is non-compact, then \( W_1 \) and \( W_2 \) are non-compact.

(e) If \( W_1 \) and \( W_2 \) are non-compact then \( W_1 \cup W_2 \) is non-compact.

*Proof.* (a) The interval \([m_0, m_0 + C)\) is equivalent to the sequence \( W = \langle m_0, 1, C \rangle = \langle \langle m_0, 1, B \rangle, B, \frac{C}{B} \rangle \). Because \( m_0 \) is aligned on \( B \), the cache lines used by \( W \) are the same as the cache lines used by \( \hat{W} = \langle m_0, B, \frac{C}{B} \rangle \), in which only one address is mapped to a single cache line. Furthermore \( \hat{W} \) can be expressed as \( \langle \langle m_0, B, \frac{T}{B} \rangle, T, \frac{C}{T} \rangle \). From Lemma 1, all inner subsequences \( \hat{w}_i = \langle m_0 + i \times T, B, \frac{T}{B} \rangle \) map exactly one element to each cache set. Therefore \( \hat{W} \) maps exactly \( A = \frac{C}{T} \) elements to each cache set, and by Definition 5 it is compact. Because \( W \) uses the exact same cache lines, it is also compact.

Results (b)-(e) follow directly from Definition 5. \( \square \)
4.4.3 L1 Data Cache

To measure the parameters of the L1 data cache our micro-benchmarks measure
the average time per element to access the elements of certain compact and non-
compact fixed stride sequences.

Figure 4.7: Example of (semi-/non-)compact

Figure 4.7 gives some intuition about the compactness properties of a sequence
$W = \langle m_0, S, N \rangle$ where $S \leq T$. When $N \times S \leq C$ the sequence is compact as it
maps at most $A$ addresses to each cache set. When $N \times S \geq C + T$ the sequence
is non compact, as it maps at least $A + 1$ addresses to each cache set. When
$C < N \times S < C + T$, the sequence maps $A$ addresses to some of the cache sets and
$A + 1$ address to the rest of the cache sets. For $S \geq T$ there are no semi-compact
sequences, and for $S < T$, $W$ is semi-compact for $\frac{T}{S} - 1$ different values of $N$. For
example, for $S = \frac{T}{2}$ there is only one $N = \frac{C+S}{S}$ for which the $W$ is semi-compact.

Theorem 1 describes the necessary and sufficient conditions for compactness
and non-compactness of a sequence of this type for a given cache. Informally, this
Theorem says that as the stride $S$ gets bigger, the maximum length of a compact sequence with that stride decreases until it bottoms out at $A$, while the minimum length of a non-compact sequence with that stride decreases until it bottoms out at $A + 1$.

**Theorem 1.** Consider a cache with parameters $\langle A, B, C \rangle$ and a sequence $W = \langle m_0, S, N \rangle$.

(a) compact $(W) \Leftrightarrow N \leq N_c = A \left\lceil \frac{T}{S} \right\rceil$

(b) non-compact $(W) \Leftrightarrow N \geq N_{nc} = (A + 1) \left\lceil \frac{T}{S} \right\rceil$

**Proof.** There are two cases to consider.

- **$S \geq T$.** In this case $N_c = A$ and $N_{nc} = A + 1$.

  Since both $S$ and $T$ are powers of 2, $S$ must be an integer multiple of $T$. From Lemma 1 it follows that all $N$ addresses in the sequence map to the same cache set.

  Therefore the sequence is compact iff for $N \leq A = N_c$ and non-compact iff $N \geq A + 1 = N_{nc}$.

- **$S < T$.** Since $S$ and $T$ are both powers of 2, $\frac{T}{S}$ is an integer, and $N_c = A \times \frac{T}{S}$ and $N_{nc} = (A + 1) \times \frac{T}{S}$.

  Let $N = p \times \frac{T}{S} + r$, where $0 \leq r < p$. We can divide $W$ into $p + 1$ parts, in which the first $p$ have $\frac{T}{S}$ elements, and the last one has $r$ elements. Furthermore, we represent $W$ as the union of two sequences of sequences: one with $p + 1$ subsequences of length $r$ and one with $p$ subsequences of length $\frac{T}{S} - r$. This
Figure 4.8: Decomposition of $W = \langle m_0, S, N \rangle$

is presented pictorially in Figure 4.8.

$$W = \langle \langle m_0, S, r \rangle, T, p + 1 \rangle$$

$$\cup \langle \langle m_0 + r \times S, S, \frac{T}{S} - r \rangle, T, p \rangle$$

$$= \langle \langle m_0, T, p + 1 \rangle, S, r \rangle$$

$$\cup \langle \langle m_0 + r \times S, T, p \rangle, S, \frac{T}{S} - r \rangle$$

From Lemma 1, the elements of each inner subsequence map to the same

cache set, whereas elements from different subsequences map to different

cache sets. Therefore $r$ cache sets will have $p + 1$ different addresses mapped
to each of them and $\frac{T}{S} - r$ cache sets will have $p$ addresses mapped to each
of them.

- $N < N_c$. In this case $p < A$, i.e. $p + 1 \leq A$. Therefore $\frac{T}{S}$ cache sets
  have at most $A$ different addresses mapped to each of them, so $W$ is compact.

- $N = N_c$. In this case $p = A$ and $r = 0$. Therefore $\frac{T}{S}$ cache sets have
  exactly $A$ different addresses mapped to each of them, so $W$ is compact.

- $N_c < N < N_{nc}$. In this case $p = A$ and $0 < r < \frac{T}{S}$. Therefore $r$
cache sets have exactly $A + 1$ different addresses mapped to each of
them and $\frac{T}{S} - r$ different cache sets have exactly $A$ different addresses mapped to each of them, so $W$ is neither compact nor non-compact (it is semi-compact).

- $N \geq N_{nc}$. In this case $p \geq A + 1$. Therefore $\frac{T}{S}$ cache sets have at least $A + 1$ different addresses mapped to each of them, so $W$ is non-compact.

The required result follows directly from this.

\[\square\]

**Algorithms for Measuring Parameters**

In this section we use the function `is_compact(W)` to determine empirically if $W$ is compact. Our implementation of this function repeatedly accesses each address in $W$, computes the average time per access $l$, and declares the sequence to be compact if $l$ is close to the hit latency of the cache $l_{hit}$, which is measured as described below.

Although this procedure seems simple in principle, the timing measurements require some special care to avoid the problems discussed in Section 4.4.1 and we discuss how we address these in Section 4.4.3.

**Cache Latency**

We determine $l_{hit}$ by measuring the average time per access of the sequence $(m_0, 1, 1)$, which is compact since it contains a single element.

**Capacity and Associativity**

Theorem 1 suggests a method for determining the capacity $C$ and the associativity $A$ of the cache. First, we find $A$ by determining the asymptotic limit of the length of a compact sequence as the stride is increased. The smallest value of the
Pseudo-code for measuring $C$ and $A$ of the L1 data cache is shown in Figure 4.9. The algorithm can be described as follows. Start with the sequence $\langle m_0, S, N \rangle = \langle m_0, 1, 1 \rangle$, which is compact, and keep doubling $N$ until the sequence is not compact. Let $N_{old}$ be the first $N$ for which this happens. Now start doubling the stride $S$, and for each $S$ compute the smallest $N$, for which $\langle m_0, S, N \rangle$ is not compact. This value of $N$ can be found by using binary search in the interval $[1, N_{old}]$. If $N \neq N_{old}$, Let $N_{old} = N$ and recompute $N$ for the next $S$. Repeat this step until $N = N_{old}$. At this point, declare $A = N - 1$ and the $C = \frac{S}{2} \times A$.

The largest stride $S$ used in this algorithm is $2T$. We will exploit this fact when
we consider multi-level cache hierarchies.

Note that the number of addresses accessed by the algorithm in this micro-benchmark is on the order of the associativity of the cache, which is superior to previous approaches because non-compactness produces a very pronounced performance drop, which is much easier to detect automatically.

Block Size

For given cache parameters $C$, $A$ and $T$, $\langle m_0, T, 2A \rangle$ is non-compact since all $2A$ addresses map to the same cache set. This sequence can also be expressed as $\langle\langle m_0, T, A \rangle, C, 2 \rangle$. If we offset the second half of the sequence by a constant $\delta$, as shown in Figure 4.10, we get the sequence $\langle\langle m_0, T, A \rangle, C + \delta, 2 \rangle$.

![Figure 4.10: Modified address sequence for measuring $B$](image)

The addresses in each of the inner subsequences $\langle m_0, T, A \rangle$ and $\langle m_0 + C + \delta, T, A \rangle$ map to a single cache set. When $0 \leq \delta < B$ this cache set is the same for both subsequences. When $\delta \geq B$ they map to two different cache sets. Therefore the smallest value of $\delta$ for which the full sequence $\langle\langle m_0, T, A \rangle, C + \delta, 2 \rangle$ is compact is $\delta = B$. Figure 4.11 shows pseudo-code for the algorithm.

Implementation of is_compact

The algorithms in Section 4.4.3 call the function is_compact($W$) to determine whether sequence $W$ is compact. We now describe how this function is imple-
\( \delta \leftarrow 1 \)

```c
while (!\text{is} \text{compact}(\langle \langle m_0, T, A \rangle, C + \delta, 2 \rangle))
    \delta \leftarrow 2 \times \delta;
return \delta;
```

Figure 4.11: Algorithm for measuring \( B \)

The array of elements is declared of type pointer (\texttt{void *}) instead of integer (\texttt{int}) as in the Saavedra benchmark. The array is initialized in such a way that each element contains the address of the element that should be accessed immediately after it. A local variable \( p \) is initialized with the address of the element that should be accessed first. This initialization is performed off-line, before the actual timing.

A simplified version of the timing routine is presented in Figure 4.12. The variable \( R \) is chosen so that the loop executes for at least a predetermined amount of time \( t \). Larger values of \( R \) are likely to produce more accurate timing results at the expense of additional running time. In our implementation, we use \( t = 1 \) second.

In addition, in the actual implementation, the \texttt{while} loop is unrolled several times to avoid loop overhead.

It is easy to see that the only operation performed in the loop body is \( p \leftarrow *(\texttt{void **})p \), which reads the memory address stored at address \( p \) and updates \( p \) with it, effectively following the pointer chain preprogrammed inside the array.

The following points address the implementation problems of the Saavedra benchmark discussed in Section 4.4.1.

(a) The code in Figure 4.12 uses the simplest possible looping structure, and
start\textbf{Time} \leftarrow \textbf{get\_time}();

while (--R)

\hspace{1em} p \leftarrow \ast(\textbf{void **})p;

time\textbf{Per\_Access} \leftarrow (\textbf{get\_time}() - \textbf{start\_Time}) \div R;

\text{printf("", p);

Figure 4.12: Improved timing of memory accesses

loop overhead can be reduced as much as needed by sufficient unrolling. In our implementation we unroll 256 times.

(b) Each of the memory accesses depends on the previous one to produce the actual address to access, so aggressive compilers cannot take advantage of instruction-level parallelism and overlap them.

(c) Each memory access constitutes precisely one memory read instruction, so the actual timing corresponds exactly to the average latency per access.

(d) All modern architectures today support an indirect addressing mode, so each operation should be translated to a single machine instruction (e.g., \texttt{lea eax, [eax]} on x86 ISA).

(e) The final value of the variable \( p \) is used by the \texttt{printf} statement, so the compiler is not able to optimize the memory accesses away by dead code elimination.

(f) For a correct implementation of \texttt{is\_compact}(W), it is important that we repeatedly access all elements of the sequence, but the actual order in which we access them is irrelevant. To prevent hardware constant stride prefetchers,
like those on the IBM Power architecture, from interfering with our timings, we initialize the array elements by chaining the pointers so that we visit the elements in a pseudo-random order.

Suppose the address sequence is $m_0, m_1, \ldots, m_{n-1}$. One way to reorder this sequence is to choose a number $p$, such that $p$ and $n$ are mutually prime. Then, after element $m_i$, visit element $m_{(i+p) \mod n}$ instead of element $a_{(i+1) \mod n}$. As $p$ and $n$ are mutually prime, the recurrence $i \leftarrow (i + p) \mod n$ is guaranteed to generate all the integers between 0 and $n - 1$ before repeating itself.

(g) All modern processors have virtually indexed L1 data caches and therefore physical continuity is not an issue. Lower levels of the memory hierarchy are usually physically indexed, so physical continuity is important for lower levels of the memory hierarchy, as we discuss in Section 4.4.4.

### 4.4.4 Lower Levels of the Memory Hierarchy

We denote the cache at level $i$ as $C_i$, its $\langle A, B, C \rangle$ parameters as $\langle A_i, B_i, C_i \rangle$, its stride as $T_i$ and its hit latency as $l_i$. We extend the notation from the previous section, so that $\text{compact}_i (W)$ denotes that $\text{compact} (W)$ holds with respect to $C_i$. We extend $\text{non-compact}$ and $\text{semi-compact}$ in the same way.

Measuring parameters of lower levels of the memory hierarchy is considerably more difficult than measuring the parameters of the L1 data cache. One reason why the algorithms described in Section 4.4.3 cannot be used directly is that $C_i$ is accessed only if $C_{i-1}$ suffers a miss. Therefore compactness with respect to $C_i$ of a sequence of addresses can be accurately determined empirically only if this...
sequence is non-compact with respect to \( C_1, C_2, \ldots, C_{i-1} \).

Our solution to this problem is to transform any sequence \( W \) into a new sequence \( W^* \), with the following properties.

1. \( \text{compact}_i (W^*) \Leftrightarrow \text{compact}_i (W) \)

2. \( \text{non-compact}_j (W^*) \), for all \( j \in [1, i - 1] \)

Intuitively, \( W \) is of the form presented in Figure 4.6(a). We want to transform it to \( W^* \), which is a sequence of sequences of the form presented in Figure 4.6(b), so that the extra memory accesses exhaust the associativity at cache levels above \( C_i \). Such a transformation may be necessary because on some architectures, lower level caches are less associative than higher level caches. For example some versions of the IBM Power 3 have 8MB, 8-way set associative \( C_2 \) and 64KB, 128-way set associative \( C_1 \). Therefore the final iteration of the algorithm in Figure 4.9 should be examining the sequence \( W = \langle m_0, 2\text{MB}, 9 \rangle \) and declaring it non-compact. Without transforming \( W \) this will not happen, because although the sequence is non-compact with respect to \( C_2 \), it is compact with respect to \( C_1 \). As we discuss later, the corresponding \( W^* \) we use for such \( W \) is \( W^* = \langle \langle m_0, 512, 15 \rangle, 2\text{MB}, 9 \rangle \), which is non-compact with respect to \( C_1 \). Another way to view this sequence is \( W^* = \langle \langle m_0, 2\text{MB}, 9 \rangle, 512, 15 \rangle \), i.e., 15 copies of the original sequence \( W \) shifted by a factor of 512. Each of these copies behaves identically to the original \( W \) with respect to \( C_2 \), but together they force non-compactness with respect to \( C_1 \).

To generalize Theorem 1 to sequences of sequences we first prove Lemma 4.

**Lemma 4.** Consider a cache with parameters \( \langle A, B, C \rangle \) and stride \( T \). If \( W_1 = \langle m_0, S, N \rangle \) and \( W_2 = \langle m_0 + \delta, S, N \rangle \), where \( m_0 \) and \( m_0 + \delta \) is aligned on a cache
block boundary, and \(0 < \delta < \min(S, T)\), then indices\((W_1)\) and indices\((W_2)\) are disjoint.

**Proof.** We will consider two cases.

First, let \(S \geq T\). From Lemma 1 all elements of \(W_1\) map to the same cache set \(i_1\); similarly all elements of \(W_2\) map to the same cache set \(i_2\). Because \(\delta < T\), \(m_0\) and \(m_0 + \delta\) map to different cache sets, so \(i_1 \neq i_2\).

Second, let \(S < T\). Let \(N = p \times \frac{T}{S} + r\), where \(0 \leq r < \frac{T}{S}\). Then:

\[
W_1 = \langle m_0, S, N \rangle \subset \langle m_0, S, (p + 1) \times \frac{T}{S} \rangle = \hat{W}_1 \\
W_2 = \langle m_0 + \delta, S, N \rangle \subset \langle m_0 + \delta, S, (p + 1) \times \frac{T}{S} \rangle = \hat{W}_2
\]

Now we split \(W_1\) and \(W_2\), which both have \((p + 1) \times \frac{T}{S}\) elements, into two sequences of \(p + 1\) subsequences with \(\frac{T}{S}\) elements each.

\[
\hat{W}_1 = \langle \hat{w}_1 = \langle m_0, S, \frac{T}{S} \rangle, T, p + 1 \rangle \\
\hat{W}_2 = \langle \hat{w}_2 = \langle m_0 + \delta, S, \frac{T}{S} \rangle, T, p + 1 \rangle
\]

From Lemma 1, indices\((\hat{W}_1)\) = indices\((\hat{w}_1)\) and indices\((\hat{W}_2)\) = indices\((\hat{w}_2)\).

The addresses of the last elements of \(\hat{w}_1\) and \(\hat{w}_2\) are \(m_0 + T - S\) and \(m_0 + \delta + T - S\) respectively. Therefore, all addresses in \(\hat{w}_1\) and \(\hat{w}_2\) are contained in the half-open interval \([m_0, m_0 + T]\). Any two addresses \(m_1 \in \hat{w}_1\) and \(m_2 \in \hat{w}_2\) are aligned on a cache block boundary and therefore from Lemma 1 they map to different cache sets. Therefore indices\((\hat{w}_1)\) and indices\((\hat{w}_2)\) are disjoint, so indices\((\hat{W}_1)\) and indices\((\hat{W}_2)\) are disjoint, which implies that indices\((W_1)\) and indices\((W_2)\) are disjoint.

**Theorem 2.** Consider a cache with parameters \(\langle A, B, C \rangle\) and stride \(T\), and a sequence of sequences \(W^* = \langle \langle m_0, s, n \rangle, S, N \rangle\), where \((n - 1) \times s < \min(T, S)\) and \(B \leq s\).
(a) \textit{compact} (W^*) \Leftrightarrow N \leq N_c = \left\lceil \frac{T}{S} \right\rceil

(b) \textit{non-compact} (W^*) \Leftrightarrow N \geq N_{nc} = (A + 1) \left\lceil \frac{T}{S} \right\rceil

\textbf{Proof.} From Lemma 2 and Definition 4,

\[ W^* = \langle (m_0, S, N), s, n \rangle = \bigcup_{i \in [0, n-1]} \langle m_0 + i \times s, S, N \rangle. \]

From Theorem 1 each of the sequences \( w_i = \langle m_0 + i \times s, S, N \rangle \) for \( i \in [0, n-1] \) is compact for \( N \leq N_c \), non-compact for \( N \geq N_{nc} \), and semi-compact otherwise.

From Lemma 4, \textbf{indices} \( (w_i) \) are pairwise disjoint sets for all \( i \in [0, n-1] \). The required result follows from Lemma 3.

Note that Theorem 1 is a special case of Theorem 2 for \( n = 1 \). In this case the constraint \( (n - 1) \times s < T \) is trivially true and the sequence \( \langle m_0, s, n \rangle \) has a single element \( (m_0) \).

\textbf{Two Cache Levels}

Consider two cache levels, \( C_1 = \langle A_1, B_1, C_1 \rangle \) and \( C_2 = \langle A_2, B_2, C_2 \rangle \).

To apply the algorithms in Section 4.4.3 to measure parameters for \( C_2 \), we replace each sequence \( W \) in those algorithms with a sequence of sequences \( W^* \), such that \textbf{compact}_2 (W^*) \Leftrightarrow \textbf{compact}_2 (W) \) and \textbf{non-compact}_1 (W^*).

Ideally we would have a general construction that could construct such a \( W^* \) from any \( W = \langle m_0, S, N \rangle \). Since we do not have such a general construction, we will present an approach, which works for the particular sequences used by the algorithms in Section 4.4.3. In particular we will restrict ourselves to sequences for which \( S \leq 2T \) (because \( 2T \) is the largest stride used by these algorithms). Furthermore, it is invariably the case that \( C_2 \geq 2C_1 \), so if \( (N - 1) \times S \leq 2C_1 \) the sequence
$W$ can be assumed compact without performing an empirical measurement. Therefore we can restrict ourselves to sequences for which $(N - 1) \times S > 2C_1$.

With these restrictions, we choose

$$W^* = \langle m_0, S, N \rangle = \langle \langle m_0, s, n \rangle, S, N \rangle,$$

where:

$$s = T_1$$
$$n = \left\lceil \frac{A_1 + 1}{N} \right\rceil.$$

**Lemma 5.** If $S \leq 2T_2$ and $(N - 1) \times S > 2C_1$ then

(a) $\text{compact}_2(W^*) \Leftrightarrow \text{compact}_2(W)$ and

(b) $\text{non-compact}_1(W^*)$.

**Proof.**

(a) First we show that Inequality (4.1) holds.

$$\left(\left\lceil \frac{A_1 + 1}{N} \right\rceil - 1\right) \times T_1 < \frac{S}{2} \tag{4.1}$$

The opposite is impossible, because then:

$$\frac{S}{2} \leq \left(\left\lceil \frac{A_1 + 1}{N} \right\rceil - 1\right) \times T_1 \leq \left(\frac{A_1 + 1}{N} - 1\right) \times T_1 \leq \frac{A_1}{N} \times T_1 < \frac{S}{2C_1} \times A_1 \times T_1 = \frac{S}{2} \Rightarrow \frac{S}{2} < \frac{S}{2}.$$

From (4.1) and $S \leq 2T_2$ we conclude that $\left(\left\lceil \frac{A_1 + 1}{N} \right\rceil - 1\right) \times T_1 < \min(S, T_2)$. Therefore we can apply Theorem 2 to $W^*$, and so $\text{compact}_2(W^*) \Leftrightarrow N \leq N_c$, where $N_c = A_2 \times \left\lceil \frac{T_2}{S} \right\rceil$. On the other hand, from Theorem 1, $\text{compact}_2(W) \Leftrightarrow N \leq N_c$. Therefore $\text{compact}_2(W^*) \Leftrightarrow \text{compact}_2(W)$. 
(b) From \((N - 1) \times S > 2C_1\) we obtain:

\[
N > \frac{2C_1 + S}{S} \geq \frac{A_1 \times T_1 + T_1}{S} = (A_1 + 1) \times \frac{T_1}{S} \\
\geq (A_1 + 1) \times \left\lceil \frac{T_1}{S} \right\rceil \Rightarrow N > (A_1 + 1) \times \left\lceil \frac{T_1}{S} \right\rceil
\]

From Theorem 2, it follows that non-compact_k (\(W^*\))

\[\square\]

Multiple Cache Levels

To generalize the approach from Section 4.4.4 to multiple cache levels \(C_1, C_2, \ldots, C_k\), we replace \(W\) with \(W^* = (\langle m_0, s, n \rangle, S, N)\), where

\[
s = \min_{i<k} T_i \\
n = \max_{i<k} \left\lceil \frac{A_i + 1}{N} \right\rceil \times \frac{T_i}{s}
\]

Lemma 6. If \(S \leq 2T_k\) and \((N - 1) \times S > 2C_i\) for all \(i \in [1, k - 1]\) then

(a) compact_k (\(W^*\)) ⇔ compact_k (\(W\)) and

(b) non-compact_i (\(W^*\)) for all \(i \in [1, k - 1]\).

Proof.

(a) By analogy with Inequality (4.1), Inequality (4.2) holds.

\[
\max_{i \in [1, k-1]} \left( \left\lceil \frac{A_i + 1}{N} \right\rceil \times \frac{T_i}{s} - 1 \right) \times T_i < \frac{S}{2}
\]

Therefore:

\[
(n - 1) \times s = \left( \max_{i < k} \left\lceil \frac{A_i + 1}{N} \right\rceil \times \frac{T_i}{s} - 1 \right) \times s \\
= \max_{i < k} \left\lceil \frac{A_i + 1}{N} \right\rceil \times T_i - s \\
< \frac{S}{2} - s \\
< \min (S, T_k)
\]
From Theorem 2, applied to $W^*$, it follows that if $N_c = A_2 \times \left\lceil \frac{T_k}{S} \right\rceil$, then $\text{compact}_k(W^*) \iff N \leq N_c$. From Theorem 1, $\text{compact}_k(W) \iff N \leq N_c$ for the same $N_c$. Therefore $\text{compact}_k(W^*) \iff \text{compact}_k(W)$.

(b) $(N - 1) \times S > 2C_i$ for all $i \in [1, k - 1]$ and by analogy with the proof of Theorem 2(b), $\text{non-compact}_i(W^*)$ holds for all $i \in [1, k - 1]$.

\[ \square \]

**Algorithms for Measuring Parameters**

We use the function $\text{is}\_\text{compact}_i(W)$ to determine empirically if $\text{compact}_i(W)$ holds. Our implementation of this function repeatedly accesses each address in $W$, computes the average time per access $l$, and declares the sequence to be compact if $l$ is close to $l_i$ (the hit latency of $C_i$).

Given the transformation from $W$ to $W^*$ as discussed in Section 4.4.4, we can use the algorithms in Section 4.4.3 to measure latency, capacity and associativity at any cache level.

**Implementation of is\_compact**

There is one important complication when measuring parameters of lower cache levels. On modern platforms $C_1$ is typically virtually indexed, but lower levels are always physically indexed. This is a problem because continuity in virtual memory is not a sufficient condition for continuity in physical memory, and thus a fixed stride sequence of addresses in the virtual address space may not map to a fixed stride sequence in physical address space.

To measure parameters of lower cache levels it is therefore necessary to allocate physically contiguous memory. There are two ways to acquire such memory in a
modern operating system: (i) request physically contiguous pages from the kernel, or (ii) request virtual memory backed by a super-page.

The first approach is generally possible only in kernel mode, and there are strict limits on the amount of allocatable memory. It is mainly used for direct memory access (DMA) devices. Another, somewhat smaller problem is that such memory regions typically consist of many pages and TLB misses might introduce inter-level interference noise in our cache measurements.

The second approach is more promising, but currently there is no portable way to request super-pages from all operating systems. To address this problem, in our implementation we provide OS-specific memory allocation and deallocation routines, which are then used by the cache micro-benchmarks to allocate memory supported by super-pages. We have implemented this approach for Linux, and we will implement it for other operating systems in the near future.

There has been some work on transparently supporting variable size pages in the OS [47]. When such support becomes generally available, our OS-specific solution will not be required.

4.4.5 Measuring TLB Parameters

The general structure of a virtual memory address is shown in Figure 4.13 (the field widths are Intel Pentium III specific). The low-order bits contain the page offset, while the hi-order bits are used for indexing page tables during the translation to a physical address. Because the translation from virtual to physical address is too expensive to perform on every memory access, a TLB is used to cache and reuse the results.

A TLB has a certain number of entries $E$ each of which can cache the address
translation for a single virtual memory page of size $P$. Even though the TLB does not store the actual data but only its physical address and a few flags, it uses the upper portion of the virtual address in a way a normal cache does (for encoding index and tag), and so we can consider it a normal cache $C_{TLB} = \langle A, B, C \rangle = \langle A_{TLB}, P, E \times P \rangle$. Ideally we would like to use our cache parameter measurement algorithms discussed in Section 4.4.3, but some complications arise as outlined below.

1. **Variable page size**: measuring parameters for caches with variable block size is not possible with our current algorithms. On current operating systems, the default is to use only a single page size, and therefore there is no immediate danger of measurement failure. Furthermore, [47] suggests that when transparent support for multiple page sizes becomes available, TLB misses will be automatically minimized and will have negligible impact on performance. At that point measuring the TLB parameters would not be necessary.

2. **Replacement policy**: typically a TLB has a high associativity and LRU is impractical to implement because of speed issues. In practice processors use much simpler replacement policies like round-robin or random. Some even perform a software interrupt on a TLB miss and leave to the operating system to do the replacement. Surprisingly these inconsistencies do not prevent us from producing accurate measurement results.
3. **Ensuring TLB access:** As in the case of lower cache levels, we need to make sure that the TLB is accessed when memory references are issued by the processor. In modern platforms this is ensured by the fact that L1 data caches are usually physically tagged, but even more importantly by the fact that TLB caches memory protection information which is needed to complete the particular memory operation.

4. **Physical Continuity:** As with lower cache levels, we need physically contiguous memory to perform TLB measurements. Unfortunately, using super-pages is not an alternative for obvious reasons, and so a kernel module is required.

For a sequence \( W = (m_0, S, N) \), let \( N = p \times \left\lceil \frac{T_1}{S} \right\rceil + r \), where \( 0 \leq r < \left\lceil \frac{T_1}{S} \right\rceil \).

To measure TLB parameters using the algorithms described in Section 4.4.3, we transform \( W \) into (\( T_1 \) and \( B_1 \) are the stride and the block size of \( C_1 \) respectively):

\[
W^* = \langle \left\langle m_0, S, \left\lceil \frac{T_1}{S} \right\rceil \right\rangle, T_1 + B_1, p \rangle \cup \langle m_0 + (p - 1) \times (T_1 + B_1), S, r \rangle
\]

We assume that the \( C_1 \) has at least twice as many blocks as there are entries in the TLB, i.e., \( \frac{C_1}{B_1} \geq 2 \frac{\text{true}}{B_{\text{TLB}}} \), which is true for all modern platforms today. Under this assumption, it is easy to see that \( \text{compact}_1(W^*) \).

Because we do not have a portable solution to (4) above, our experience with measuring TLB parameters is limited. None of the other tools produced any correct results on any of the tested platforms. Therefore, we describe our limited experimental results in this section.

Using the algorithms in Section 4.4.3 with the modified sequences \( W^* \), we were able to accurately measure the TLB parameters of a Pentium III as 64 page entries,
4-way set associative, and page size of 4KB. We also measured the TLB parameters of a Pentium 4 as 65 page entries, fully-associative, with a page size of 4KB. On the Pentium 4 our measurement is close to the correct one (measured associativity 65 vs. actual associativity of 64).\footnote{This problem may be similar to the one we discuss about the L1 data cache of Power 3 in Section A.2.1}

### 4.5 Future Work

We are actively designing and developing new micro-benchmarks and we are currently working on:

- implementing OS support for Solaris, AIX, etc.,
- improving quality of TLB measurements,
- measuring instruction cache parameters,
- cache bandwidth, parallelism, write mode, and sharedness (unified or dedicated).

Chapter 5

Global Search in ATLAS

ATLAS performs a global search to determine optimal values for the optimization parameters listed in Table 3.1. In principle, the search space is unbounded because most of the parameters, such as $N_B$, are integers. Therefore, it is necessary to bound the search space using parameters of the machine hardware; for example, $M_U$ and $N_U$, the dimensions of the register tile, must be less than the number of registers.

5.1 Estimating hardware parameters

The machine parameters measured by ATLAS are the following.

- $C_1$: the size of the L1 data cache.
- $N_R$: the number of floating-point registers.
- FMA: the availability of a fused multiply-add instruction.
- $L_s$: software latency

Although $L_s$ is not a hardware parameter per se, it is directly related to the latency of floating point multiplication, as explained in Section 3.4. ATLAS measures this optimization parameter directly using a micro-benchmark.

The micro-benchmarks used to measure machine parameters are independent of matrix multiplication. For example, the micro-benchmark for estimating $C_1$ is similar to the one discussed in Hennessy and Patterson [33].
Two other machine parameters are critical for performance: (i) the L1 instruction cache size, and (ii) the number of outstanding loads that the hardware supports. ATLAS does not determine these parameters explicitly using micro-benchmarks; instead, they are considered implicitly during the optimization of matrix multiplication code. For example, the size of the L1 instruction cache limits the $K_U$ parameter in Figure 3.2. Rather than estimate the size of the instruction cache directly by running a micro-benchmark and using that to determine the amount of unrolling, ATLAS generates a suite of mini-MMM kernels with different $K_U$ values and selects the kernel that achieves the best performance.

5.2 Search strategy

To find optimal values for the optimization parameters in Table 3.1, ATLAS uses orthogonal line search, which finds an approximation to the optimal value of a function $y = f(x_1, x_2, \ldots, x_n)$, an $n$-dimensional optimization problem, by solving a sequence of $n$ 1-dimensional optimization problems corresponding to each of the $n$ parameters. When optimizing the value of parameter $x_i$, it uses reference values for parameters $(x_{i+1}, x_{i+2}, \ldots, x_n)$ that have not yet been optimized. Orthogonal line search is heuristic because it does not necessarily find the optimal value even for a convex function, but with luck, it may come close.

To specify an orthogonal line search, it is necessary to specify (i) the order in which the parameters are optimized, (ii) the set of possible values considered during the optimization of each parameter, and (iii) the reference value used for parameter $k$ during the optimization of parameters 1, 2, \ldots, $k - 1$.

The optimization sequence used in ATLAS is the following.

1. Finding $N_B$. 

86
2. Finding $M_U$ and $N_U$.

3. Finding $K_U$.

4. Finding $L_s$.


6. Finding $NCN_B$: a non-copy version of $N_B$.

7. Finding parameters for clean-up codes.

We now discuss each of these steps in greater detail.

### 5.3 Finding $N_B$

In this step, ATLAS generates a number of mini-MMMs for matrix sizes $N_B \times N_B$ where $N_B$ is a multiple of four that satisfies the following inequality:

$$16 \leq N_B \leq \min \left( 80, \sqrt{C_1} \right)$$  \hspace{1cm} (5.1)

The reference values of $M_U$ and $N_U$ are set to the values closest to each other that satisfy (3.1). For each matrix size, ATLAS tries two extreme cases for $K_U$ – no unrolling ($K_U = 1$) and full unrolling ($K_U = N_B$).

The $N_B$ value that produces the highest MFLOPS is chosen as “best $N_B$” value, and it is used from this point on in all experiments as well as in the final version of the optimized mini-MMM code.
5.4 Finding $M_U$ and $N_U$

This step is a straightforward search that refines the reference values of $M_U$ and $N_U$ that were used to find the best $N_B$. ATLAS tries all possible combinations of $M_U$ and $N_U$ that satisfy inequality (3.1). Cases when $M_U$ or $N_U$ is 1 are treated specially. A test is performed to see if $1 \times 9$ unrolling or $9 \times 1$ unrolling is better than $3 \times 3$ unrolling. If not, unrolling factors of the form $1 \times U$ and $U \times 1$ for values of $U$ greater than 3 are not checked.

5.5 Finding $K_U$

This step is another simple search. Unlike $M_U$ and $N_U$, $K_U$ does not depend on the number of available registers, so it can be made as large as desired without causing register spills. The main constraint is the instruction cache size. ATLAS tries values for $K_U$ between 4 and $\frac{N_B}{2}$ as well as the special values 1 and $N_B$. The value that gives best performance (based on $N_B$, $M_U$ and $N_U$ as determined from the previous steps) is declared the optimal value for $K_U$.

5.6 Finding $L_s$

In this step, ATLAS uses $L_s$ values in the interval $[1, 6]$ to schedule the computations in the micro-MMM of Figure 3.2 to determine the best choice for $L_s$. It also ensures that the chosen value divides $M_U \times N_U \times K_U$ to facilitate instruction scheduling.
5.7 Finding $F_F$, $I_F$, and $N_F$

In this step, ATLAS searches for the values of $F_F$, $I_F$ and $N_F$. First, ATLAS determines the value of $F_F$ (0 or 1). Then, it searches for the best value of the pair $(I_F, N_F)$ where $I_F$ is in the interval $[2, M_U+N_U]$ and $N_F$ is in the interval $[1, M_U+N_U-I_F]$.

5.8 Finding $N CN_B$

For the non-copying version of mini-MMM, ATLAS uses the same values of $M_U$, $N_U$, $F_F$, $I_F$, and $N_F$ that it uses for the copying version. Without copying, the likelihood of conflict misses is higher, so it makes sense to use a smaller L1 cache tile size than in the version of mini-MMM that performs copying. ATLAS searches for an optimal value of $N CN_B$ in the range $[N_B : -4 : 4]$. We would expect performance to increase initially as the tile size is decreased, but decrease when the tile size becomes too small. ATLAS terminates the search when the performance falls by 20% or more from the best performance it finds during this search. Finally, some restricted searches for better values of $K_U$ and $L_s$ are done.

5.9 Finding parameters for clean-up codes

If the tile size is not a multiple of the original matrix size, there may be left-over rows and columns, at the boundaries of the matrices, forming fractional tiles. To handle these fractional tiles, ATLAS generates clean-up code – a special mini-MMM in which one or more of the dimensions of the three tiles is smaller than $N_B$. For $M$ and $N$ clean-up only the corresponding dimension is smaller than $N_B$, while for $K$ cleanup, any of the three dimensions can be smaller than $N_B$. 
For example, ATLAS generates $K$ clean-up codes as follows. For each value of $L$, representing the size of the $K$ dimension, starting with $L = N_B - 1$ and going down, it generates a specialized version of the mini-MMM code in which some of the loops are fully unrolled. Full unrolling is possible because the shapes of the operands are completely known. When the performance of the general version falls within 1% of the performance of the current specialized version, the generation process is terminated. The current $L$ is declared to be the *Crossover Point*. At runtime, the specialized versions are invoked when the dimension of the left-over tile is greater than $L$ and the general version is invoked otherwise.

For $M$ and $N$ clean-up ATLAS produces only a general version, as these are outer loops in the outermost loop nest in Figure 3.2 and they are not as crucial to performance as $K$ clean-up is. The use of clean-up code in ATLAS is discussed in more detail in [58].

### 5.10 Discussion

In optimization problems, there is usually a trade-off between search time and the quality of the solution. For example, we can refine the parameters found by ATLAS by repeating the orthogonal line search some number of times, using the values determined by one search as the reference values for the next search. It is also possible to use more powerful global search algorithms like simulated annealing. However, the potential for obtaining better solutions must be weighed carefully against the increase in installation time. We will address this point in the conclusions.
Chapter 6

Model-Driven Optimization

In this section, we present analytical models for estimating optimal values for the parameters in Table 3.1. To avoid overwhelming the reader, we first present models that ignore interactions between different levels of the memory hierarchy (in this case, L1 data cache and registers). Then, we refine the models to correct for such interactions.

6.1 Estimating hardware parameters

Model-based optimization requires more machine parameters than the ATLAS approach because there is no search. The hardware parameters required by our model are as follows.

- $C_1, B_1$: the capacity and the line size of the L1 data cache.
- $C_I$: The capacity of the L1 instruction cache.
- $L_X$: hardware latency of the floating-point multiply instruction
- $|ALU_{FP}|$: number of floating-point functional units
- $N_R$: the number of floating-point registers.
- $FMA$: the availability of a fused multiply-add instruction.

Empirical optimizers use the values of machine parameters only to bound the search space, so approximate values for these parameters are adequate. In contrast, analytical models require accurate values for these parameters. Therefore, we have developed a tool called X-Ray [66], which accurately measures these values.
6.2 Modeling strategy

We build analytical models which compute optimization parameters directly from the corresponding hardware parameters. We explain our models for all optimization parameters in the following sections.

6.3 Estimating $N_B$

We present our model for estimating $N_B$ using a sequence of refinements for increasingly complex cache organizations. We start with the mini-MMM code in Figure 6.1, and then adjust the model to take register tiling into account.

\[
\begin{align*}
\text{for } j' & \in [0 : 1 : N_B - 1] \\
\text{for } i' & \in [0 : 1 : N_B - 1] \\
\text{for } k' & \in [0 : 1 : N_B - 1] \\
C_{i'j'} & = C_{i'j'} + A_{i'k'} \times B_{k'j'}
\end{align*}
\]

Figure 6.1: Schematic Pseudo-Code for mini-MMM

6.3.1 Fully associative, optimal replacement, unit line size

The goal is to find the value of $N_B$ that optimizes the use of the L1 data cache. First, we consider a simple cache of capacity $C_1$, which is fully-associative with an optimal replacement policy and a unit line size. There are no conflict misses, and spatial locality is not important.

The working set in memory of the mini-MMM loop nest in Figure 6.1 consists of three $N_B \times N_B$ tiles, one from each of the matrices $A$, $B$, and $C$. For the rest of this section, we will refer to these tiles just as $A$, $B$, and $C$. This working set fits
entirely in the cache if Inequality (6.1) holds.

\[3N_B^2 \leq C_1\]  

(6.1)

A more careful analysis shows that it is not actually necessary for all three \(N_B \times N_B\) blocks to reside in the cache for the entire duration of the mini-MMM computation. Consider the mini-MMM code shown in Figure 6.1. Because \(k'\) is the innermost loop, elements of \(C\) are computed in succession; once a given element of \(C\) has been computed, subsequent iterations of the loop nest do not touch that location again. Therefore, with this loop order, it is sufficient to hold a single element of \(C\) in the cache, rather than the entire array. The same reasoning shows that it is sufficient to hold a single column of \(B\) in the cache. Putting these facts together, we see that with this loop order, there will be no capacity misses if the cache can hold all of \(A\), a single column of \(B\), and a single element of \(C\). This leads to Inequality (6.2).

\[N_B^2 + N_B + 1 \leq C_1\]  

(6.2)

6.3.2 Correcting for non-unit line size

In reality, caches have non-unit line size. Assume that the line size is \(B_1\). If the three tiles are stored in column major order, both \(B\) and \(C\) are walked by columns and \(A\) is in the cache for the entire duration of the mini-MMM. This leads to the refined constraint shown in Inequality (6.3).

\[\left\lceil \frac{N_B^2}{B_1} \right\rceil + \left\lceil \frac{N_B}{B_1} \right\rceil + 1 \leq \frac{C_1}{B_1}\]  

(6.3)
6.3.3 Correcting for LRU replacement policy

We can further relax the restrictions of our cache organization to allow for Least Recently Used (LRU) replacement instead of optimal replacement. To determine the effects of LRU replacement on the optimal tile size $N_B$, we must examine the history of memory accesses performed by the loop nest. This analysis is in the spirit of Mattson et al. [42], who introduced the notions of stack replacement and stack distance.

We start with the innermost loop of the mini-MMM loop nest. A single iteration $\langle j, i, k \rangle$ of this loop touches elements

$$A_{ik}; B_{kj}; C_{ij};$$

where the most recently accessed element is written rightmost in this sequence.

Extending this analysis to the middle loop, we see that the sequence of memory access for a given value of the outer loop indices $\langle j, i \rangle$ is the following (as before, the most recently accessed element is rightmost):

$$A_{i0}; B_{0j}; C_{ij}; A_{i1}; B_{1j}; C_{ij}; \ldots; A_{i,N_B-1}; B_{N_B-1,j}; C_{ij};$$

Note that the location $C_{ij}$ is touched repeatedly, so the corresponding history of memory accesses from least recently accessed to most recently accessed is the following:

$$A_{i0}; B_{0j}; A_{i1}; B_{1j}; \ldots; A_{i,N_B-1}; B_{N_B-1,j}; C_{ij};$$

Extending this to a single iteration $j$ of the outermost loop, we see that the
sequence of memory accesses is the following (in left-to-right, top-to-bottom order):

\[
\begin{align*}
    A_{00} &; B_{0j}; & \cdots & A_{0,N_B-1}; B_{N_B-1,j}; & C_{0j}; \\
    A_{10} &; B_{0j}; & \cdots & A_{1,N_B-1}; B_{N_B-1,j}; & C_{1j}; \\
    & \vdots & & & \\
    A_{N_B-1,0} &; B_{0j}; & \cdots & A_{N_B-1,N_B-1}; B_{N_B-1,j}; & C_{N_B-1,j};
\end{align*}
\]

Note that the column of \( \mathbf{B} \) is reused \( N_B \) times, and thus the corresponding history of memory accesses from least recently accessed to most recently accessed is

\[
\begin{align*}
    A_{00} &; \cdots A_{0,N_B-1}; C_{0j}; \\
    A_{10} &; \cdots A_{1,N_B-1}; C_{1j}; \\
    & \vdots & & & \\
    A_{N_B-1,0} &; \cdots A_{N_B-1,N_B-1}; C_{N_B-1,j};
\end{align*}
\]

We do not want to evict the oldest element of this history (\( A_{00} \)) because, as we discussed before, \( \mathbf{A} \) is completely reused in all iterations of the outermost loop. Therefore we need to choose \( N_B \) is such a way that this whole history fits in the cache.

Furthermore, after the \( j \)th iteration of the outermost loop is complete, the \( j + 1 \)st iteration will bring in the \( j + 1 \)st column of \( \mathbf{B} \), which participates in an inner product with all the rows of \( \mathbf{A} \). Because of LRU, this new column will not be able to “optimally” replace the old \( j \)th column of \( \mathbf{B} \), since the old column of \( \mathbf{B} \) has been used quite recently. For the same reason the new element of \( \mathbf{C} \), namely \( C_{0,j+1} \), will not be able to optimally replace the old \( C_{0j} \). To account for this, we need extra storage for an extra column of \( \mathbf{B} \) and an extra element of \( \mathbf{C} \).

Putting all this together, we see that if the cache is fully-associative with capacity \( C_1 \), line size \( B_1 \) and has an LRU replacement policy, we need to cache all of
A, two columns of B and a column plus an element of C. This result is expressed formally in Inequality (6.4).

\[
\left\lceil \frac{N_B^2}{B_1} \right\rceil + 3 \left\lceil \frac{N_B}{B_1} \right\rceil + 1 \leq \frac{C_1}{B_1}
\]  

(6.4)

Finally, to model the mini-MMM code of Figure 3.2, which includes register tiling, we need to take into account interactions between the register file and the L1 cache. Thus far, we implicitly assumed that the computation works directly on the scalar elements of the tiles. As Figure 3.2 shows, the mini-MMM loop nest actually works on register tiles. We refine Inequality (6.4) by recognizing that considerations of rows, columns, and elements of A, B, and C respectively must be replaced by considerations of horizontal panels, vertical panels, and register tiles instead. Taking this into account, we get Inequality (6.5).

\[
\left\lceil \frac{N_B^2}{B_1} \right\rceil + 3 \left\lceil \frac{N_B \times N_U}{B_1} \right\rceil + \left\lceil \frac{M_U}{B_1} \right\rceil \times N_U \leq \frac{C_1}{B_1}
\]  

(6.5)

### 6.3.4 Correcting to avoid micro-MMM clean-up code

Note that estimating \(N_B\) using Inequality (6.4), it is possible to get a value for \(N_B\) that is not an exact multiple of \(M_U\) and \(N_U\). This requires the generation of clean-up code for fractional register tiles at the boundaries of mini-MMM tiles. This complicates code generation, and generally lowers performance. We avoid these complications by trimming the value of \(N_B\) determined from Inequality (6.4) so that it becomes a multiple of \(M_U\) and \(N_U\). The ATLAS Code Generator requires \(N_B\) to be an even integer, so we enforce this constraint as well.

If \(N_B'\) is the tile size obtained by using Inequality (6.4), we set \(N_B\) to the value \(\left\lfloor \frac{N_B'}{lcm(M_U, N_U, 2)} \right\rfloor \times lcm(M_U, N_U, 2)\).
Note this requires that the value of $N_B$ be determined after the values of $M_U$ and $N_U$ have been determined as described below.

### 6.3.5 Other cache organizations

If the cache organization is not fully-associative, conflict misses must be taken into account. Although there is some work in the literature on modeling conflict misses [15, 18], these models are computationally intractable. Therefore, we do not model conflict misses, although there are some general remarks we can make.

If $A$, $B$, and $C$ are copied to $3N_B^2$ contiguous storage locations, Inequality (6.1) can also be viewed as determining the largest value of $N_B$ for which there are no capacity or conflict misses during the execution of the mini-MMM in any cache organization. Although ATLAS usually copies tiles, the code in Figure 3.2 shows that the three copied tiles are not necessarily adjacent in memory. However, if the set-associativity of the L1 data cache is at least 3, there will be no conflict misses.

Inequality (6.2) determines the largest $N_B$ for which there are no capacity misses during the execution of the mini-MMM, although there may be conflict misses if the cache is direct-mapped or set-associative. Notice that these conflict misses arise even if data from all three matrix tiles is copied into contiguous memory, because the amount of data touched by the program is more than the capacity of the cache, and some elements will map to the same cache set.

### 6.4 Estimating $M_U$ and $N_U$

One can look at the register file as a software-controlled, fully-associative cache with unit line size and a capacity equal to the number of available registers $N_R$. Therefore we can use a variant of Inequality (6.2) to estimate the optimal register
The ATLAS Code Generator uses the KIJ loop order to tile for the register file, and thus we need to cache the complete $M_U \times N_U$ tile of $C$, an $1 \times N_U$ row of $B$ and a single element of $A$. Therefore the analog of Inequality (6.2) for registers is Inequality (6.6), shown below.

$$M_U \times N_U + N_U + 1 \leq N_R$$ \hspace{1cm} (6.6)

Because the register file is software controlled, the ATLAS Code Generator is free to allocate registers differently than Inequality (6.6) prescribes. In fact, as discussed in Chapter 3, it allocates to registers a $M_U \times 1$ column of $A$, rather than a single element of $A$. Furthermore, it needs $L_s$ registers to store temporary values of multiplication operations to schedule for optimal use of the floating point pipelines. Taking into account these details, we refine Inequality (6.6) to obtain Inequality (6.7).

$$M_U \times N_U + N_U + M_U + L_s \leq N_R$$ \hspace{1cm} (6.7)

$N_R$ is a hardware parameter, whose effective value is measured by the micro-benchmarks. The value of the optimization parameter $L_s$ is estimated as discussed in Section 6.6. Therefore the only unknowns in Inequality (6.7) are $M_U$ and $N_U$. We estimate their values using the following procedure.

- Let $M_U = N_U = u$. Solve Inequality (6.7) for $u$.
- Let $M_U = \max (u, 1)$. Solve Inequality (6.7) for $N_U$.
- Let $N_U = \max (N_U, 1)$
- Let $\langle M_U, N_U \rangle = \langle \max (M_U, N_U), \min (M_U, N_U) \rangle$. 


6.5 Estimating $K_U$

Although $K_U$ is structurally similar to $M_U$ and $N_U$, it is obviously not limited by the size of the register file. Therefore the only practical limit for $K_U$ is imposed by the size of the instruction cache. To avoid micro-MMM clean-up code, we trim $K_U$ so that $N_B$ is a multiple of $K_U$. Note that in case of complete unrolling ($K_U = N_B$), $K_U$ is not changed by this update.

Therefore our model for estimating $K_U$ is to unroll the loop as much as possible within the size constraints of the L1 instruction cache, while ensuring that $K_U$ divides $N_B$. On most platforms, we found that the loop can be unrolled completely ($K_U = N_B$).

6.6 Estimating $L_s$

$L_s$ is the optimization parameter that represents the skew factor the ATLAS Code Generator uses when scheduling dependent multiplication and addition operations for the CPU pipeline.

Studying the description of the scheduling in Chapter 3, we see that the schedule effectively executes $L_s$ independent multiplications and $L_s - 1$ independent additions between a multiplication $mul_i$ and the corresponding addition $add_i$. The hope is that these $2 \times L_s - 1$ independent instructions will hide the latency of the multiplication. If the floating-point units are fully pipelined and the latency of multiplication is $L_x$, we get the following inequality, which can be solved to obtain a value for $L_s$.

$$2 \times L_s - 1 \geq L_x \quad (6.8)$$
On some machines, there are multiple floating-point units. If $|ALU_{FP}|$ is the number of floating-point ALUs, Inequality (6.8) gets refined as follows.

$$\frac{2 \times L_s - 1}{|ALU_{FP}|} \geq L_x$$

(6.9)

Solving Inequality (6.9) for $L_s$, we obtain Inequality (6.10).

$$L_s = \left\lceil \frac{L_x \times |ALU_{FP}| + 1}{2} \right\rceil$$

(6.10)

Of the machines in our study, only the Intel Pentium machines have floating-point units that are not fully pipelined; in particular, multiplications can be issued only once every 2 cycles. Because Pentium machines have only one floating-point functional unit and ATLAS does not schedule back-to-back multiply instructions, this does not introduce any error in our model. Therefore, Inequality (6.8) holds.

### 6.7 Estimating other parameters

Our experience shows that performance is insensitive to the values of the scheduling optimization parameters $F_F$, $I_F$, and $N_F$. Therefore we set $F_F = 1 (true)$, $I_F = 2$, and $N_F = 2$.

FMA is a hardware parameter, independent of the specific application. If our micro-benchmarks determine that the architecture supports a fused multiply-add instruction, we set this parameter appropriately.

Finally, we set $NCN_B = N_B$. That is, we use the same tile size for the non-copying version of mini-MMM as we do for the copying version. In our experiments, ATLAS always decided to use the copying version of mini-MMM\footnote{Using the non-copy version is mainly beneficial when the matrices involved in the computation are either very small or are long and skinny [56].}, so the value of
this parameter was moot. A careful model for $NCN_B$ is difficult because it is hard to model conflict misses analytically. There is some work on this in the compiler literature but most of the models are based on counting integer points within certain parameterized polyhedra and appear to be intractable [15, 18]. Fraguela et al. have proposed another approach to modeling conflict misses when the sizes of matrices are known [26]. In some compilers, this problem is dealt with heuristically by using the effective cache capacity, defined to be a fraction (such as $\frac{1}{3}$) of the actual cache capacity, when computing the optimal tile size. In our context, we could set $NCN_B$ to the value determined from Inequality (6.4) with $C_1$ replaced with $\frac{C_1}{3}$. We recommend this approach should it become necessary to use a smaller tile size on some architectures.

### 6.8 Discussion

We have described a fairly elaborate sequence of models for estimating the optimal value of $N_B$. In practice, the value found by using Inequality (6.3), a relatively simple model, is close to the value found by using more elaborate models such as Inequalities (6.4) and (6.5).
Chapter 7

Model Refinement and Local Search

Here we address two problems in using model-driven optimization in the context of general-purpose compilers.

The first problem is that there can be a potential performance gap between code produced by library generators and code produced by using model-driven optimization in case the model is not an accurate abstraction of the architecture. Such performance penalty is unacceptable for critical applications that will be run many times. On the other hand, global search as performed by ATLAS does not scale well to large programs or to complex architectures, so it cannot be used in general-purpose compilers. Dongarra and co-workers are exploring faster search algorithms like the simplex method [22], but it is not clear that these algorithms alone are adequate.

The second problem is performance portability. It may seem that the use of global search ensures that library generators will work “out of the box” on new architectures, whereas model-driven optimization may fail if the model is a poor abstraction of the new architecture. However, global search is not a panacea. In particular, if the code generator does not exploit aspects of an architecture that are key to performance, the resulting code may be poor regardless of how exhaustive the search for optimal parameter values is. The methodology used in the ATLAS system to adapt to new architectures for which search alone is not sufficient is to include a collection of user-contributed hand-tuned kernels in the distribution; during the search process, the performance of these codes is evaluated, and if one of them performs better than the code generated by the code generator, it is
used to produce the library. This methodology cannot be used for model-driven optimization because it runs counter to the spirit of using models to optimize programs.

The approach that we advocate to address both problems is to use a combination of model refinement and local search. To close the performance gap with code produced by empirical optimization, we advocate using local search in the neighborhood of the parameter values produced by using the model. Of course local search alone may not be adequate if the model is not a good abstraction of the architecture. In that case, we advocate using model refinement in the same spirit as ATLAS incorporates user-contributed code - we study the new architecture and refine the model as needed. Note that like the production of user-contributed code, model refinement must be done manually. Intuitively, in our approach, small performance problems are tackled using local search, while large performance problems are tackled using model refinement.

The cases where we found that it is necessary to refine the models described in Chapter 6 are presented along with our experimental results in Chapter 8, as we believe they are much better justified in this context.

Now we describe some very simple local search techniques, targeted at possible inaccuracies in the models for $N_B$, $M_U$, $N_U$, and $L_s$.

**Local Search for $N_B$**

If $N_{B_M}$ is the value of $N_B$ estimated by the model, we can refine this value by local search in the interval $[N_{B_M} - \text{lcm}(M_U, N_U, 2) : 2 : N_{B_M} + \text{lcm}(M_U, N_U, 2)]$. This ensures that we examine the first values of $N_B$ in the neighborhood of $N_{B_M}$ that are divisible by both $M_U$ and $N_U$. 
Local Search for $M_U$, $N_U$, and $L_s$

Unlike sensitivity graphs for $N_B$, sensitivity graphs for $M_U$ and $N_U$ tend to be convex in the neighborhood of model-predicted values. This is probably because register allocation is under compiler control, and there are no conflict misses. Hence, we use a simple hill-climbing search strategy to improve these parameters.

We start with the model predicted values for $M_U$, $N_U$, and $L_s$ and determine if performance improves by changing each of them by $+1$ and $-1$. We continue following the path of increasing performance until we stop at a local maximum. On platforms on which there is a Fused-Multiply-Add instruction ($FMA = 1$), the optimization parameter $L_s$ has no effect on the generated code and in that case we only consider $M_U$ and $N_U$ for the hill-climbing local search.
Chapter 8

Experimental Results

8.1 Global search vs. model-driven optimization

*Models are to be used, not believed.*

H. Theil ‘Principles of Econometrics’

In this section, we present the results of running ATLAS CGw/s and ATLAS Model on ten common platforms. For all experiments we used the latest stable version of ATLAS, which as of this writing is 3.6.0. Where appropriate, we also present numbers for ATLAS Unleashed and vendor supported, native BLAS.

We performed our experiments on the following platforms.

- RISC, Out-of-order
  - DEC Alpha 21264
  - IBM Power 3
  - IBM Power 4
  - SGI R12K
- RISC In-order
  - Sun UltraSPARC IIIi
- EPIC In-order
  - Intel Itanium 2
- CISC, Out-of-order

105
- AMD Opteron 240
- AMD Athlon MP
- Intel Pentium III
- Intel Pentium 4

For each platform, we present the following results.

- **Times:**
  - *X-Ray:* time taken by X-Ray to determine hardware parameters;
  - *ATLAS Micro-benchmarks:* time taken by the micro-benchmarks in ATLAS to determine hardware parameters;
  - *ATLAS Optimization Parameter Search:* time taken by global search in ATLAS for determining optimization parameter values.

We do not report the actual installation time of any of the versions of ATLAS because most of this time is spent in optimizing other BLAS kernels, generating library code, building object modules, etc.

We do not discuss the timing results in detail as they are not particularly surprising. X-Ray is faster than ATLAS in measuring hardware parameters on nine out of the ten platforms and has comparable timing (10% slower) on one (IBM Power 3). Moreover, while ATLAS CGw/S spends a considerable amount of time, ranging between 8 minutes on the DEC Alpha to more than 8 hours on the Intel Itanium 2, to find the optimal values for the optimization parameters, the model-based approach takes no measurable time.
• **Performance:**

  – *Optimization parameter values:* values determined by ATLAS CGw/S and ATLAS Model. Where appropriate, we also report these values for ATLAS Unleashed.


  – *MMM performance:* for matrices sized $100 \times 100$ to $5000 \times 5000$. We report performance of complete MMM computations using (i) vendor supported, native BLAS, and the code produced by (ii) ATLAS CGw/S, (iii) ATLAS Model, (iv) ATLAS Unleashed, and (v) the native FORTRAN compiler. On each platform, the code produced by ATLAS is compiled with the native C compiler. The input FORTRAN program is the standard triply-nested loop shown in Figure 3.1.

  For vendor supported, native BLAS (labeled “BLAS” on all figures) we used the following libraries and corresponding versions, which were current at the time of our experiments:

  * DEC Alpha: CXML 5.2
  * IBM Power 3/4: ESSL 3.3
  * SGI R12K: SCSL 6.5
  * SUN UltraSPARC IIIi: Sun One Studio 8
  * Intel Itanium 2, Pentium III/4: MKL 6.1
  * AMD Opteron, Athlon: ACML 2.0

• **Sensitivity Analysis:** this describes the relative change of performance as we change one of the optimization parameters, keeping all other parameters
fixed to the values found by ATLAS CGw/S. The sensitivity analysis explains how variations in the values of the optimization parameters influence the performance of the generated mini-MMM kernel.

- $N_B$: change in mini-MMM performance when the value of $N_B$ is changed
- $M_U$, $N_U$: change in mini-MMM performance when values of $M_U$ and $N_U$ are changed. Because optimal values of $M_U$ and $N_U$ depend on the same hardware resource ($N_R$), we vary them together.
- $K_U$: change in mini-MMM performance when value of $K_U$ is changed.
- $L_s$: change in mini-MMM performance when $L_s$ is changed.
- $F_F$, $I_F$ and $N_F$: we do not show sensitivity graphs for these parameters because performance is relatively insensitive to their values.
### 8.1.1 DEC Alpha 21264

**Table 8.1: DEC Alpha 21264: Platform Specification**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>Out-Of-Order, RISC</td>
</tr>
<tr>
<td>CPU Core Frequency</td>
<td>833 MHz</td>
</tr>
<tr>
<td>L1 Data Cache</td>
<td>64 KB, 64 B/line, 2-way</td>
</tr>
<tr>
<td>L1 Instruction Cache</td>
<td>64 KB, 64 B/line, 2-way</td>
</tr>
<tr>
<td>L2 Unified Cache</td>
<td>4 MB, 64 B/line, 1-way</td>
</tr>
<tr>
<td>Floating-Point Registers</td>
<td>32</td>
</tr>
<tr>
<td>Floating-Point Functional Units</td>
<td>2</td>
</tr>
<tr>
<td>Floating-Point Multiply Latency</td>
<td>4</td>
</tr>
<tr>
<td>Has Fused Multiply Add</td>
<td>No</td>
</tr>
<tr>
<td>Operating System</td>
<td>Tru64 v5.1B (rev.2650)</td>
</tr>
<tr>
<td>C Compiler</td>
<td>Compaq C v6.5-003</td>
</tr>
<tr>
<td>Fortran Compiler</td>
<td>GNU Fortran 3.3</td>
</tr>
</tbody>
</table>

**Table 8.2: DEC Alpha 21264: Optimization Parameters**

<table>
<thead>
<tr>
<th>N_B</th>
<th>M_U, N_U, K_U</th>
<th>L_s</th>
<th>FMA</th>
<th>F_F, I_F, N_F</th>
<th>MFLOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>CGw/S</td>
<td>72</td>
<td>4, 4, 72</td>
<td>4</td>
<td>0</td>
<td>1, 7, 1</td>
</tr>
<tr>
<td>Model</td>
<td>84</td>
<td>4, 4, 84</td>
<td>4</td>
<td>0</td>
<td>0, 2, 2</td>
</tr>
<tr>
<td>Unleashed</td>
<td>80</td>
<td>4, 4, 84</td>
<td>4</td>
<td>0</td>
<td>0, 2, 2</td>
</tr>
</tbody>
</table>

**Table 8.3: DEC Alpha 21264: Timings**

<table>
<thead>
<tr>
<th></th>
<th>Search</th>
<th>Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Machine Parameters</td>
<td>148s</td>
<td>101s</td>
</tr>
<tr>
<td>Optimization Parameters</td>
<td>556s</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>704s</td>
<td>101s</td>
</tr>
</tbody>
</table>
Figure 8.1: DEC Alpha 21264: MMM Performance

Figure 8.2: DEC Alpha 21264: Sensitivity of performance to $N_B$
Figure 8.3: DEC Alpha 21264: Sensitivity of performance to $M_U$ and $N_U$

Figure 8.4: DEC Alpha 21264: Sensitivity of performance to $K_U$

Figure 8.5: DEC Alpha 21264: Sensitivity of performance to $L_s$
mini-MMM

On this machine the model-determined optimization parameters provided performance of about 100 MFLOPS (7%) slower than the ones determined by search. The reason of the difference is the suboptimal selection of the $N_B$ parameter (84 for Atlas Model vs. 72 for ATLAS CGw/S), as can be seen in the $N_B$ sensitivity graph of Figure 8.2(b).

MMM Performance

Figure 8.1 shows the MMM performance. ATLAS Unleashed produces the fastest BLAS implementation because it uses highly-optimized, hand-tuned BLAS kernels written by Goto. A newer version of these kernels is described in [35]. The native BLAS library is only marginally slower.

Although the gap in performance of the mini-MMM codes produced by ATLAS CGw/S and ATLAS Model is 100 MFLOPS, the gap in performance of complete MMM computations is only about 50 MFLOPS (4%) for large matrices. Finally, we note that the GNU FORTRAN compiler is unable to deliver acceptable performance. We did not have access to the Compaq FORTRAN compiler, so we did not evaluate it.

Sensitivity Analysis

Figure 8.3 shows the sensitivity of performance to the values of $M_U$ and $N_U$. The optimal value is $(4, 4)$, closely followed by $(3, 6)$, and $(6, 3)$. These match our expectations that optimal unroll factors are as close to square as possible, while dividing the tile size $N_B = 72$ without reminder.

Figure 8.2(a) shows the sensitivity of performance to the value of $N_B$. Fig-
Figure 8.2(b) shows a scaled-up version of this graph in the region of the optimal $N_B$ value. The optimal value for $N_B$ is 88. ATLAS does not find this point because it does not explore tile sizes greater than 80, as explained in Section 5, but it chooses a tile size of 72, which is close to optimal. If we use Inequality (6.5) to determine $N_B$ analytically, we obtain $N_B = 84$. Note that using the simpler model of Inequality (6.3), we obtain $N_B = 90$, which appears to be almost as good as the value determined by the more complex model.

The $N_B$ sensitivity graph of Figure 8.2(b) has a saw-tooth of periodicity 4, with notable peaks occurring with a periodicity of 8. The saw-tooth of periodicity 4 arises from the interaction between cache tiling and register tiling - the register tile is (4, 4), so whenever $N_B$ is divisible by 4, there is no clean-up code for fractional register tiles in the mini-MMM code, and performance is good. We do not yet understand why there are notable peaks in the saw-tooth with a periodicity of 8.

Figure 8.4 shows the sensitivity of performance to the value of $K_U$. On this machine the entire mini-MMM loop body can fit into the L1 instruction cache for values of $K_U$ up to $N_B$. Performance is relatively insensitive to $K_U$ as long as the value of this parameter is sufficiently large ($K_U > 7$).

Figure 8.5 shows the sensitivity of performance to the value of $L_s$. The graph is convex upwards, with a peak at 4. The multiplier on this machine has a latency of 4 cycles, hence the model for $L_s$ in Chapter 6 computes $L_s = 5$, which is close to optimal. The inverted-U shape of this graph follows our expectations. For very small values of $L_s$, dependent multiplications and additions are not well separated and CPU pipeline utilization is low. As $L_s$ grows, the problem gradually disappears, until the performance peak is reached when the full latency of the multiplication is hidden. Increasing $L_s$ further does not improve performance as
there is no more latency to hide. On the contrary, more temporary registers are needed to save multiplication results, which causes more register spills to memory, decreasing performance.
### 8.1.2 IBM Power 3

#### Table 8.4: IBM Power 3: Platform Specification

<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>Out-Of-Order, RISC</td>
</tr>
<tr>
<td>CPU Core Frequency</td>
<td>375 MHz</td>
</tr>
<tr>
<td>L1 Data Cache</td>
<td>64 KB, 128 B/line, 128-way</td>
</tr>
<tr>
<td>L1 Instruction Cache</td>
<td>32 KB, 128 B/line, 128-way</td>
</tr>
<tr>
<td>L2 Unified Cache</td>
<td>4 MB, 128 B/line, direct-mapped</td>
</tr>
<tr>
<td>Floating-Point Registers</td>
<td>32</td>
</tr>
<tr>
<td>Floating-Point Functional Units</td>
<td>2</td>
</tr>
<tr>
<td>Floating-Point Multiply Latency</td>
<td>4</td>
</tr>
<tr>
<td>Has Fused Multiply Add</td>
<td>Yes</td>
</tr>
<tr>
<td>Operating System</td>
<td>AIX</td>
</tr>
<tr>
<td>C Compiler</td>
<td>XL C for AIX v.5</td>
</tr>
<tr>
<td>Fortran Compiler</td>
<td>XL Fortran for AIX</td>
</tr>
</tbody>
</table>

#### Table 8.5: IBM Power 3: Optimization Parameters

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CGw/S</td>
<td>80</td>
<td>4, 5, 80</td>
<td>6</td>
<td>1</td>
<td>0, 8, 1</td>
<td>1264</td>
</tr>
<tr>
<td>Model</td>
<td>84</td>
<td>4, 4, 84</td>
<td>4</td>
<td>1</td>
<td>0, 2, 2</td>
<td>1225</td>
</tr>
<tr>
<td>Unleashed</td>
<td>80</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1257</td>
</tr>
</tbody>
</table>

#### Table 8.6: IBM Power 3: Timings

<table>
<thead>
<tr>
<th></th>
<th>Search</th>
<th>Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Machine Parameters</td>
<td>139s</td>
<td>154s</td>
</tr>
<tr>
<td>Optimization Parameters</td>
<td>1984s</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>2123s</td>
<td>154s</td>
</tr>
</tbody>
</table>
Figure 8.6: IBM Power 3: MMM Performance

Figure 8.7: IBM Power 3: Sensitivity of performance to $N_B$
Figure 8.8: IBM Power 3: Sensitivity of performance to $M_U$ and $N_U$

Figure 8.9: IBM Power 3: Sensitivity of performance to $K_U$

Figure 8.10: IBM Power 3: Sensitivity of performance to $L_s$
mini-MMM

Mini-MMM code produced by ATLAS Model is about 40 MFLOPS (3%) slower than mini-MMM code produced by ATLAS CGw/S. Figure 8.7(b) shows that one reason for this difference is the sub-optimal choice of $N_B$; fixing the values of all parameter other than $N_B$ to the ones chosen by ATLAS CGw/S and using the model-predicted value of 84 for $N_B$ results in mini-MMM code that performs about 100 MFLOPS worse than the mini-MMM code produced by ATLAS CGw/S.

MMM Performance

For multiplying large matrices, the handwritten BLAS as well as the codes produced by ATLAS CGw/S, ATLAS Model, and ATLAS Unleashed perform almost identically.

Sensitivity Analysis

Figure 8.7(a) shows the sensitivity of performance to the value of $N_B$. Figure 8.7(b) shows a scaled-up version of this graph in the region of the optimal $N_B$ value. Figure 8.8 shows the sensitivity of performance to the values of $M_U$ and $N_U$. Figure 8.9 shows the sensitivity of performance to the value of $K_U$. On this machine, the entire mini-MMM loop body can fit into the L1 instruction cache for values of $K_U$ up to $N_B$. Performance is relatively insensitive to $K_U$ as long as the value of this parameter is sufficiently large ($K_U > 5$). We do not understand the sudden drop in performance at $K_U = 3$. Figure 8.10 shows the sensitivity of performance to the value of $L_s$. The Power 3 platform has a fused multiply-add instruction, which the ATLAS micro-benchmarks and X-ray find, and the Code Generator exploits, so performance does not depend on the value of $L_s$. 
### 8.1.3 IBM Power 4

#### Table 8.7: IBM Power 4: Platform Specification

<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>Out-Of-Order, RISC</td>
</tr>
<tr>
<td>CPU Core Frequency</td>
<td>1450 MHz</td>
</tr>
<tr>
<td>L1 Data Cache</td>
<td>32 KB, 128 B/line, 2-way</td>
</tr>
<tr>
<td>L1 Instruction Cache</td>
<td>64 KB, 128 B/line, 1-way</td>
</tr>
<tr>
<td>L2 Unified Cache</td>
<td>1.5 MB, 128 B/line, 8-way</td>
</tr>
<tr>
<td>L3 Cache</td>
<td>32 MB, 512B/line, 8-way</td>
</tr>
<tr>
<td>Floating-Point Registers</td>
<td>32</td>
</tr>
<tr>
<td>Floating-Point Functional Units</td>
<td></td>
</tr>
<tr>
<td>Floating-Point Multiply Latency</td>
<td></td>
</tr>
<tr>
<td>Has Fused Multiply Add</td>
<td>Yes</td>
</tr>
<tr>
<td>Operating System</td>
<td>AIX</td>
</tr>
<tr>
<td>C Compiler</td>
<td>XL C for AIX v.5</td>
</tr>
<tr>
<td>Fortran Compiler</td>
<td>XL Fortran for AIX</td>
</tr>
</tbody>
</table>

#### Table 8.8: IBM Power 4: Optimization Parameters

<table>
<thead>
<tr>
<th>Model</th>
<th>$N_B$</th>
<th>$M_U, N_U, K_U$</th>
<th>$L_A$</th>
<th>FMA</th>
<th>$F_F, I_F, N_F$</th>
<th>MFLOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>CGw/S</td>
<td>64</td>
<td>4, 4, 64</td>
<td>1</td>
<td>1</td>
<td>1, 8, 1</td>
<td>3468</td>
</tr>
<tr>
<td>Model</td>
<td>56</td>
<td>4, 4, 56</td>
<td>6</td>
<td>1</td>
<td>0, 2, 2</td>
<td>3400</td>
</tr>
<tr>
<td>Unleashed</td>
<td>64</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3468</td>
</tr>
</tbody>
</table>

#### Table 8.9: IBM Power 4: Timings

<table>
<thead>
<tr>
<th></th>
<th>Search</th>
<th>Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Machine Parameters</td>
<td>175s</td>
<td>125s</td>
</tr>
<tr>
<td>Optimization Parameters</td>
<td>2390s</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>2665s</td>
<td>125s</td>
</tr>
</tbody>
</table>
Figure 8.11: IBM Power 4: MMM Performance

Figure 8.12: IBM Power 4: Sensitivity of performance to $N_B$
Figure 8.13: IBM Power 4: Sensitivity of performance to $M_U$ and $N_U$

Figure 8.14: IBM Power 4: Sensitivity of performance to $K_U$

Figure 8.15: IBM Power 4: Sensitivity of performance to $L_s$
mini-MMM

On this machine, mini-MMM code produced by ATLAS Model is about 70 MFLOPS (2%) slower than mini-MMM code produced by ATLAS CGw/S. Figure 8.12(b) shows that one reason for this difference is a slightly sub-optimal choice of $N_B$; fixing the values of all parameter other than $N_B$ to the ones chosen by ATLAS CGw/S and using the model-predicted value of 56 for $N_B$ results in mini-MMM code that performs slightly worse than the mini-MMM code produced by ATLAS CGw/S.

MMM Performance

Figure 8.11 shows MMM performance. For large matrices, the hand-tuned BLAS performs the best, although by a small margin. The code produced by ATLAS Model, ATLAS CGw/S and ATLAS Unleashed perform almost identically. On this machine the native IBM XL Fortran compiler produced relatively good results for small matrices.

Sensitivity Analysis

Figure 8.13 shows the sensitivity of performance to changes in the values of $M_U$ and $N_U$. The parameter values (4, 4) perform best, and these are the values used by both ATLAS CGw/S and ATLAS Model.

Figure 8.12(a) shows the sensitivity of performance to the value of $N_B$. Figure 8.12(b) shows a scaled-up version of this graph in the neighborhood of the $N_B$ value determined by ATLAS CGw/S. Figure 8.12(a) shows that on this machine, $N_B$ values between 150 and 350 give the best performance of roughly 3.5 GFLOPS. Using Inequality (6.1) for the L2 cache (capacity of 1.5 MB) gives $N_B = 254$, while
Inequality (6.5) gives $N_B = 436$, showing that on this machine, it is better to tile for the L2 cache rather than the L1 cache.

Figure 8.14 shows the sensitivity of performance to the value of $K_U$. The L1 instruction cache on this machine is large enough that we can set $K_U$ to $N_B$. As on the Power 3, unrolling by 3 gives poor performance for reasons we do not understand.

Figure 8.15 shows the sensitivity of performance to the value of $L_s$. The Power 4 platform has a fused multiply-add instruction, which the ATLAS micro-benchmarks find and the Code Generator exploits, so performance does not depend on the value of $L_s$. 
### 8.1.4 SGI R12K

#### Table 8.10: SGI R12K: Platform Specification

<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>Out-Of-Order, RISC</td>
</tr>
<tr>
<td>CPU Core Frequency</td>
<td>270 MHz</td>
</tr>
<tr>
<td>L1 Data Cache</td>
<td>32 KB, 32 B/line, 2-way</td>
</tr>
<tr>
<td>L1 Instruction Cache</td>
<td>32 KB, 32 B/line, 2-way</td>
</tr>
<tr>
<td>L2 Unified Cache</td>
<td>4 MB, 32 B/line, 1-way</td>
</tr>
<tr>
<td>Floating-Point Registers</td>
<td>32</td>
</tr>
<tr>
<td>Floating-Point Functional Units</td>
<td>2</td>
</tr>
<tr>
<td>Floating-Point Multiply Latency</td>
<td>2</td>
</tr>
<tr>
<td>Has Fused Multiply Add</td>
<td>Yes</td>
</tr>
<tr>
<td>Operating System</td>
<td>IRIX64</td>
</tr>
<tr>
<td>C Compiler</td>
<td>SGI MIPSPro C 7.3.1.1m</td>
</tr>
<tr>
<td>Fortran Compiler</td>
<td>SGI MIPSPro FORTRAN 7.3.1.1m</td>
</tr>
</tbody>
</table>

#### Table 8.11: SGI R12K: Optimization Parameters

<table>
<thead>
<tr>
<th>$N_B$</th>
<th>$M_U$, $N_U$, $K_U$</th>
<th>$L_s$</th>
<th>$FMA$</th>
<th>$F_F$, $I_F$, $N_F$</th>
<th>MFLOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>CGw/S</td>
<td>64</td>
<td>4, 5, 32</td>
<td>3</td>
<td>0</td>
<td>1, 8, 1</td>
</tr>
<tr>
<td>Model</td>
<td>58</td>
<td>5, 4, 58</td>
<td>1</td>
<td>1</td>
<td>0, 2, 2</td>
</tr>
<tr>
<td>Unleashed</td>
<td>64</td>
<td>4, 5, 32</td>
<td>3</td>
<td>0</td>
<td>1, 8, 1</td>
</tr>
</tbody>
</table>

#### Table 8.12: SGI R12K: Timings

<table>
<thead>
<tr>
<th></th>
<th>Search</th>
<th>Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Machine Parameters</td>
<td>251s</td>
<td>117s</td>
</tr>
<tr>
<td>Optimization Parameters</td>
<td>5015s</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>5266s</td>
<td>117s</td>
</tr>
</tbody>
</table>
Figure 8.16: SGI R12K: MMM Performance

Figure 8.17: SGI R12K: Sensitivity of performance to $N_B$
Figure 8.18: SGI R12K: Sensitivity of performance to $M_U$ and $N_U$

Figure 8.19: SGI R12K: Sensitivity of performance to $K_U$

Figure 8.20: SGI R12K: Sensitivity of performance to $L_s$
mini-MMM

On this machine, mini-MMM code produced by ATLAS Model is about 20 MFLOPS (4%) slower than mini-MMM code produced by ATLAS CGw/S. The performance of both codes is similar to that of mini-MMM code produced by ATLAS Unleashed.

MMM Performance

Figure 8.16 shows MMM performance. The hand-coded BLAS performs best by a small margin. On this machine the native compiler (in this case, the SGI MIPSPro) generates relatively good code that is only 20% lower in performance than the hand-coded BLAS, at least for small matrices.

Sensitivity Analysis

Figure 8.18 shows the sensitivity of performance to the values of $M_U$ and $N_U$. This machine has a relatively large number of registers (32), so there is a fairly broad performance plateau in this graph.

Figure 8.17(a) shows the sensitivity of performance to the value of $N_B$. Figure 8.17(b) shows a scaled-up version of this graph in the region of the optimal $N_B$ value. Figure 8.17(a) shows that $N_B$ values between 300 and 500 give the best performance of roughly 510 MFLOPS. Using Inequality (6.1) for the L2 cache (capacity of 4MB) gives $N_B = 418$, while Inequality (6.5) gives $N_B = 718$, showing that it is better to tile for the L2 cache rather than the L1 cache.

Figure 8.19 shows the sensitivity of performance to the value of the $K_U$. The instruction cache is large enough that full unrolling ($K_U = N_B$) is possible.

Figure 8.20 shows the sensitivity of performance to the value of the $L_s$. The R12K processor has a fused multiply-add instruction, so we would expect the per-
formance of the generated code to be insensitive to the value of $L_s$. While this is borne out by Figure 8.20, notice that Table 8.11 shows that the micro-benchmark used by ATLAS did not discover the fused multiply-add instruction on this machine ($FMA = 0$)! It is worth mentioning that on this platform the FMA instruction, while present in the ISA, is not backed up by a real FMA pipeline in hardware. Instead it allows the two separate functional units (for multiplication and addition respectively) to be used sequentially saving one latency cycle. Therefore, in theory, peak performance is achievable even by using separate multiply and add instructions. Although the ATLAS Code Generator schedules code using $L_s = 3$, the SGI MIPSPro compiler is clever enough to discover the separated multiplies and adds, and fuse them. In fact the compiler is able to do this even when $L_s = 20$, which is impressive.
### 8.1.5 Sun UltraSPARC IIIi

#### Table 8.13: Sun UltraSPARC IIIi: Platform Specification

<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>Out-Of-Order, RISC</td>
</tr>
<tr>
<td>CPU Core Frequency</td>
<td>1060 MHz</td>
</tr>
<tr>
<td>L1 Data Cache</td>
<td>64 KB, 32 B/line, 4-way</td>
</tr>
<tr>
<td>L1 Instruction Cache</td>
<td>32 KB, 32 B/line, 4-way</td>
</tr>
<tr>
<td>L2 Unified Cache</td>
<td>1 MB, 32 B/line, 4-way</td>
</tr>
<tr>
<td>Floating-Point Registers</td>
<td>32</td>
</tr>
<tr>
<td>Floating-Point Functional Units</td>
<td>2</td>
</tr>
<tr>
<td>Floating-Point Multiply Latency</td>
<td>4</td>
</tr>
<tr>
<td>Has Fused Multiply Add</td>
<td>No</td>
</tr>
<tr>
<td>Operating System</td>
<td>SUN Solaris 9</td>
</tr>
<tr>
<td>C Compiler</td>
<td>SUN C 5.5</td>
</tr>
<tr>
<td>Fortran Compiler</td>
<td>SUN FORTRAN 95 7.1</td>
</tr>
</tbody>
</table>

#### Table 8.14: Sun UltraSPARC IIIi: Optimization Parameters

<table>
<thead>
<tr>
<th>Model</th>
<th>$N_B$</th>
<th>$M_U$, $N_U$, $K_U$</th>
<th>$L_s$</th>
<th>$F_M$, $I_F$, $N_F$</th>
<th>MFLOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>CGw/S</td>
<td>44</td>
<td>4, 3, 44</td>
<td>5</td>
<td>0, 0, 3, 2</td>
<td>986</td>
</tr>
<tr>
<td>Model</td>
<td>84</td>
<td>4, 4, 84</td>
<td>4</td>
<td>0, 0, 2, 2</td>
<td>1149</td>
</tr>
<tr>
<td>Unleashed</td>
<td>168</td>
<td></td>
<td></td>
<td></td>
<td>1695</td>
</tr>
</tbody>
</table>

#### Table 8.15: Sun UltraSPARC IIIi: Timings

<table>
<thead>
<tr>
<th></th>
<th>Search</th>
<th>Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Machine Parameters</td>
<td>203s</td>
<td>112s</td>
</tr>
<tr>
<td>Optimization Parameters</td>
<td>1254s</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>1457s</td>
<td>112s</td>
</tr>
</tbody>
</table>
Figure 8.21: Sun UltraSPARC IIIi: MMM Performance

Figure 8.22: Sun UltraSPARC IIIi: Sensitivity of performance to $N_B$
Figure 8.23: Sun UltraSPARC IIIi: Sensitivity of performance to $M_{U}$ and $N_{U}$

Figure 8.24: Sun UltraSPARC IIIi: Sensitivity of performance to $K_{U}$

Figure 8.25: Sun UltraSPARC IIIi: Sensitivity of performance to $L_{S}$
mini-MMM

On this machine, mini-MMM code produced by ATLAS Model is about 160 MFLOPS (17%) faster than mini-MMM code produced by ATLAS CGw/S. The main reason for this is that the micro-benchmarks used by ATLAS incorrectly measured the capacity of the L1 data cache as 16 KB, rather than 64 KB. Therefore ATLAS only searched for $N_B$ values less than 44. Our micro-benchmarks on the other hand correctly measured the capacity of the L1 cache, so the model estimated $N_B = 84$, which gave better performance as can be seen in Figure 8.22(b).

MMM Performance

Figure 8.21 shows the MMM performance. On this machine, the hand-coded BLAS and ATLAS Unleashed perform roughly 50% better than the code produced by ATLAS CGw/S. The reason for this difference is that the mini-MMM code in ATLAS Unleashed (and perhaps the hand-coded BLAS) pre-fetches portions of the A and B matrices required for the next mini-MMM. This may be related to the level-3 pre-fetching idea of Gustavson et al. [4].

Sensitivity Analysis

Figure 8.23 shows the sensitivity of the performance to the values of $M_U$ and $N_U$. Figure 8.22(a) shows the performance sensitivity to the value of $N_B$. Figure 8.22(b) shows a scaled-up version of this graph in the region of the optimal $N_B$ value. On this machine, as on many other machines, it is better to tile for the L2 cache, as can be seen in Figure 8.22(a). Using Inequality (6.1) for the L2 cache (with capacity of 1 MB), we obtain $N_B = 208$, which gives roughly 1380 MFLOPS. Using Inequality (6.5), we obtain $N_B = 356$, which is close to the $N_B$ value in
Figure 8.22(a) where the performance drops rapidly.

Figure 8.24 shows the sensitivity of performance to the value of the $K_U$. On this machine, the instruction cache is large enough that full unrolling ($K_U=N_B$) is possible.

Figure 8.25 shows the sensitivity of performance to the value of the $L_s$. This machine does not have a fused multiply-add instruction, so the value of the $L_s$ parameter affects performance. Both the model and ATLAS CGw/S find good values for this parameter.
### Intel Itanium 2

#### Table 8.16: Intel Itanium 2: Platform Specification

<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>In-Order, EPIC, IA-64</td>
</tr>
<tr>
<td>CPU Core Frequency</td>
<td>1500 MHz</td>
</tr>
<tr>
<td>L1 Data Cache</td>
<td>16 KB, 64 B/line, 4-way</td>
</tr>
<tr>
<td>L1 Instruction Cache</td>
<td>16 KB, 64 B/line, 4-way</td>
</tr>
<tr>
<td>L2 Unified Cache</td>
<td>256 KB, 128 B/line, 8-way</td>
</tr>
<tr>
<td>L3 Cache</td>
<td>3 MB, 128B/line, 12-way</td>
</tr>
<tr>
<td>Floating-Point Registers</td>
<td>128</td>
</tr>
<tr>
<td>Floating-Point Functional Units</td>
<td>2</td>
</tr>
<tr>
<td>Floating-Point Multiply Latency</td>
<td>4</td>
</tr>
<tr>
<td>Has Fused Multiply Add</td>
<td>Yes</td>
</tr>
<tr>
<td>Operating System</td>
<td>Linux 2.4.18-e.31smp</td>
</tr>
<tr>
<td>C Compiler</td>
<td>GNU C/C++ 3.3</td>
</tr>
<tr>
<td>Fortran Compiler</td>
<td>GNU Fortran 3.3</td>
</tr>
</tbody>
</table>

#### Table 8.17: Intel Itanium 2: Optimization Parameters

<table>
<thead>
<tr>
<th>Model</th>
<th>(N_U), (N_U), (K_U), (L_s), FMA, (F_F), (I_F), (N_F)</th>
<th>MFLOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>CGw/S</td>
<td>80, 10, 10, 4, 1, 0, 19, 1</td>
<td>4028</td>
</tr>
<tr>
<td>Model</td>
<td>30, 10, 10, 8, 1, 0, 2, 2</td>
<td>1806</td>
</tr>
<tr>
<td>Unleashed</td>
<td>120</td>
<td>4891</td>
</tr>
</tbody>
</table>

#### Table 8.18: Intel Itanium 2: Timings

<table>
<thead>
<tr>
<th></th>
<th>Search</th>
<th>Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Machine Parameters</td>
<td>1555s</td>
<td>143s</td>
</tr>
<tr>
<td>Optimization Parameters</td>
<td>30710s</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>32265s</td>
<td>143s</td>
</tr>
</tbody>
</table>
Figure 8.26: Intel Itanium 2: MMM Performance

Figure 8.27: Intel Itanium 2: Sensitivity of performance to $N_B$
Figure 8.28: Intel Itanium 2: Sensitivity of performance to $M_U$ and $N_U$

Figure 8.29: Intel Itanium 2: Sensitivity of performance to $K_U$

Figure 8.30: Intel Itanium 2: Sensitivity of performance to $L_s$
mini-MMM

On this machine, the mini-MMM code produced by ATLAS Model is about 2.2 GFLOPS (55%) slower than mini-MMM code produced by ATLAS CGw/S. This is a rather substantial difference in performance, so it is necessary to examine the sensitivity graphs to understand the reasons why ATLAS Model is doing so poorly.

Figure 8.27(b) shows that one reason for this difference is that ATLAS Model used $N_B = 30$, whereas ATLAS CGw/S used $N_B = 80$. ATLAS CGw/S uses $N_B = 80$ because it disregards the L1 data cache size (16KB) and considers directly the L2 cache size (256KB), and therefore the expression $\min(80, \sqrt{C})$ in Inequality (5.1) evaluates to 80, the largest possible value of $N_B$ in the search space used by ATLAS.

While the value $N_B = 30$ used by ATLAS Model is correct with respect to the L1 data cache size, the Intel Itanium 2 does not allow storing floating point numbers in the L1 data cache, and thus L2 has to be considered instead. Once we incorporate in X-Ray the ability to measure this specific hardware feature, the shortcoming of ATLAS Model will be resolved.

MMM Performance

Figure 8.26 shows MMM performance. The hand-written BLAS and ATLAS Unleashed give the best performance. The code produced by ATLAS CGw/S runs about 1.5 GFlops slower than the hand-written BLAS, while the code produced by ATLAS Model runs about 3.5 GFlops slower.

Sensitivity Analysis

Figure 8.28 shows the sensitivity of performance to the values of $M_U$ and $N_U$. The Itanium 2 has 128 general-purpose registers, so the optimal register tiles are
Figure 8.31: Intel Itanium 2: Sensitivity of performance to $K_U$

relatively large. There is a broad plateau of $(M_U, N_U)$ values that give excellent performance.

Figure 8.27(a) shows the sensitivity of performance to the value of $N_B$. Figure 8.27(b) shows a scaled-up version of this graph in the region of the optimal $N_B$ value. Figure 8.27(a) shows that on this machine, the best performance is obtained by tiling for the L3 cache! Indeed, using Inequality (6.1) for the L3 cache (capacity of 3 MB), we obtain $N_B = 360$, which gives roughly 4.6 GFLOPS. Figure 8.27(a) shows that this value is close to optimal. Using Inequality (6.5), we obtain $N_B = 610$, which is close to the $N_B$ value in Figure 8.27(a) where the performance starts to drop.

Figure 8.29 shows the sensitivity of performance to the value of $K_U$. On the Itanium 2, unlike on other machines in our study, performance is highly sensitive to the value of $K_U$. The main reason is the large register tile $(M_U, N_U) = (10, 10)$; after unrolling the micro-MMM loops, we get a very long straight-line code sequence. Furthermore, unrolling of the $k''$ loop creates numerous copies of this code sequence. Unfortunately, the L1 instruction cache on this machine has a capacity of 32 KB, so it can hold only about 9 copies of the micro-MMM code sequence. Therefore, performance drops off dramatically for values of $K_U$ greater than 9.
Since this is the only machine in our study in which the $K_U$ parameter mattered, we decided to investigate the sensitivity graph more carefully. Figure 8.31 shows a magnified version of Figure 8.29 in the interval $K_U \in [0, 15]$. We would expect the $K_U$ sensitivity graph to exhibit the typical inverted-U shape, and it more or less does. However, performance for $K_U = 7$ is significantly worse than the performance for $K_U = 6$, and $K_U = 8$, which appears anomalous.

The anomaly arises from clean-up code that is required when $K_U$ does not divide $N_B$ evenly (see the $k'$ loop in the tiled code in Figure 3.2). If we unroll the $k'$ loop by $K_U$, the number of times the completely unrolled micro-MMM code is replicated inside the mini-MMM is not $K_U$, but $K_U + N_B \% K_U$ ($\%$ is the remainder from integer division). The first term in the sum is the expected number of repetitions inside the unrolled $k'$ loop, while the second part is the clean-up code that takes care of the case when $K_U$ does not divide $N_B$ exactly. This second piece of code is still part of the mini-MMM loop nest, and it has to be stored in the L1 instruction cache during execution to achieve optimal performance.

For $N_B = 80$, performance increases initially as $K_U$ increases because loop overhead is reduced. When $K_U = 6$, there are 8 copies of the unrolled micro-MMM code in the mini-MMM, and this is close to the I-cache limit. When $K_U = 7$, there are $7 + 80 \% 7 = 10$ copies of the micro-MMM code, which exceeds the I-cache limit, and performance drops substantially. However, when $K_U = 8$, there is no clean-up code, and there are only 8 copies of the unrolled micro-MMM code, so performance goes up again. Beyond this point, the code sizes overflows the I-cache and grows larger, and performance degrades gradually. Ultimately, performance is limited by the rate at which L1 I-cache misses can be serviced. For $N_B = 360$, the trends are similar, but the effect of clean-up code is less because the clean-up code performs
a smaller fraction of the computations of the \( k' \) loop (less than 1% compared to about 5% for \( N_B = 80 \)).

Figure 8.30 shows the sensitivity of performance to the value of the \( L_s \). The Itanium 2 has a fused multiply-add instruction, so performance is insensitive to the \( L_s \) parameter.

In summary, the code produced by ATLAS Model on this machine did not perform as well as the code produced by ATLAS CGw/S. However, this is because ATLAS Model tiled for the L1 cache, whereas on this machine, the best performance is obtained by tiling for L3 cache. ATLAS CGw/S gets better performance because the tile size is set to a larger value than the value used by ATLAS Model.
### 8.1.7 AMD Opteron 240

#### Table 8.19: AMD Opteron 240: Platform Specification

<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>Out-Of-Order, CISC, x86-64</td>
</tr>
<tr>
<td>CPU Core Frequency</td>
<td>1400 MHz</td>
</tr>
<tr>
<td>L1 Data Cache</td>
<td>64 KB, 64 B/line, 2-way</td>
</tr>
<tr>
<td>L1 Instruction Cache</td>
<td>64 KB, 64 B/line, 2-way</td>
</tr>
<tr>
<td>L2 Unified Cache</td>
<td>1024 MB, 64 B/line, 16-way</td>
</tr>
<tr>
<td>Floating-Point Registers</td>
<td>8 x87</td>
</tr>
<tr>
<td>Floating-Point Functional Units</td>
<td>ADD + MUL + Memory</td>
</tr>
<tr>
<td>Floating-Point Multiply Latency</td>
<td>4</td>
</tr>
<tr>
<td>Has Fused Multiply Add</td>
<td>No</td>
</tr>
<tr>
<td>Operating System</td>
<td>Linux 2.4.19</td>
</tr>
<tr>
<td>C Compiler</td>
<td>GCC C/C++ 3.3.2</td>
</tr>
<tr>
<td>Fortran Compiler</td>
<td>GNU Fortran 3.3.2</td>
</tr>
</tbody>
</table>

#### Table 8.20: AMD Opteron 240: Optimization Parameters

<table>
<thead>
<tr>
<th>Optimizer</th>
<th>$N_B$</th>
<th>$M_U, N_U, K_U$</th>
<th>$L_{F,U}$</th>
<th>FMA</th>
<th>$F_F, I_F, N_F$</th>
<th>MFLOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>CGw/S</td>
<td>60</td>
<td>6, 1, 60</td>
<td>6</td>
<td>1</td>
<td>0, 6, 1</td>
<td>2072</td>
</tr>
<tr>
<td>Model</td>
<td>88</td>
<td>2, 1, 88</td>
<td>2</td>
<td>0</td>
<td>0, 2, 2</td>
<td>1282</td>
</tr>
<tr>
<td>Unleashed</td>
<td>56</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2608</td>
</tr>
</tbody>
</table>

#### Table 8.21: AMD Opteron 240: Timings

<table>
<thead>
<tr>
<th>Category</th>
<th>Search</th>
<th>Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Machine Parameters</td>
<td>148s</td>
<td>101s</td>
</tr>
<tr>
<td>Optimization Parameters</td>
<td>556s</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>704s</td>
<td>101s</td>
</tr>
</tbody>
</table>
Figure 8.32: AMD Opteron 240: MMM Performance

Figure 8.33: AMD Opteron 240: Sensitivity of performance to $N_B$
Figure 8.34: AMD Opteron 240: Sensitivity of performance to $M_U$ and $N_U$

Figure 8.35: AMD Opteron 240: Sensitivity of performance to $K_U$

Figure 8.36: AMD Opteron 240: Sensitivity of performance to $L_s$
mini-MMM

Table 8.21 shows that on this machine, the mini-MMM code generated by ATLAS Model runs roughly 38% slower than the code generated by ATLAS CGw/S. The values of almost all optimization parameters determined by the two systems are different, so it is not obvious where the problem is. To get some insight, it is necessary to look at the sensitivity graphs.

Figure 8.33(a) shows the performance sensitivity graph for $N_B$. Both 60 and 88 appear to be reasonable values, so the problem with ATLAS Model is not in its choice of $N_B$. Because $K_U$ is bound to the value of $N_B$, the only remaining differences are those between $M_U$, $N_U$, $L_s$, and $FMA$. Table 8.20 shows that ATLAS Model chose $M_U = 2$, $N_U = 1$, $FMA = 0$, while ATLAS CGw/S chose $M_U = 6$, $N_U = 1$, $FMA = 1$. We verified experimentally that if the model had chosen $M_U = 6$ and $FMA = 1$, keeping the rest of the parameters the same, the mini-MMM performance becomes 2050 MFLOPS, closing the performance gap with ATLAS CGw/S.

The parameters values used by ATLAS CGw/S are puzzling for several reasons. First, the Opteron does not have an FMA instruction, so it is not clear why ATLAS CGw/S chose to set $FMA = 1$. Second, choosing 6 and 1 for the values of $M_U$ and $N_U$ violates Inequality (6.7) since the Opteron has only 8 registers.

We address the problem of the register-tile size first. Recall that Inequality (6.7) stems from the fact that ATLAS uses registers to multiply an $M_U \times 1$ vector-tile of matrix $A$ (which we call $\vec{a}$) with a $1 \times N_U$ vector-tile of matrix $B$ (which we call $\vec{b}$), accumulating the result into an $M_U \times N_U$ tile of matrix $C$ (which we call $\vec{c}$). Notice that if $N_U = 1$, then $\vec{b}$ is a single scalar that is multiplied by each element of $\vec{a}$. Therefore no reuse exists for elements of $\vec{a}$. This observation lets us generate the
code in Figure 8.37, which uses 1 register for $\bar{b} (rb)$, 6 registers for $\bar{c} (rc_1 \ldots rc_6)$ and 1 temporary register ($rt$) to hold elements of $\bar{a}$.

Even if there are enough logical registers, this kind of scheduling may be beneficial if the ISA is 2-address rather than 3-address, because one of the operands is overwritten. This is true on the Opteron when the 16 SSE vector registers are used to hold scalar values, which is GCC’s default behavior. Even though Inequality 3.1 prescribes $3 \times 3$ register tiles, the refined model prescribes $14 \times 1$ tiles. Experiments show that this performs better [57].

One might expect that this code will not perform well because there are dependences between most of the instructions that arise from the use of temporary register $rt$. In fact, experiments show that the code in Figure 8.37 performs well because of two architectural features of the Opteron.

1. **Out-of-order execution**: it is possible to schedule several multiplications in successive cycles without waiting for the first one to complete.

2. **Register renaming**: the single temporary register $rt$ is renamed to a different physical register for each pair of multiply-add instructions.

Performing instruction scheduling as described in Chapter 3 requires additional logical registers for temporaries, which in turn limits the sizes of the register tiles. When an architecture provides out-of-order execution and a small number of logical registers, it is better to use the logical registers for allocating larger register tiles and leave instruction scheduling to the out-of-order hardware core, which can use the extra physical registers to hold the temporaries.

These insights permit us to refine the model described in Chapter 6 as follows: for processors with out-of-order execution and a small number of logical registers,
\[ rc_1 \leftarrow \bar{c}_1 \ldots rc_6 \leftarrow \bar{c}_6 \]

\[
\ldots
\
\text{loop } k
\
\{
\quad rb \leftarrow \bar{b}_1
\]

\[
rt \leftarrow \bar{a}_1
\]
\[
rt \leftarrow rt \times rb
\]
\[
rc_1 \leftarrow rc_1 + rt
\]

\[
rt \leftarrow \bar{a}_2
\]
\[
rt \leftarrow rt \times rb
\]
\[
rc_2 \leftarrow rc_2 + rt
\]

\[
\vdots
\]

\[
rt \leftarrow \bar{a}_6
\]
\[
rt \leftarrow rt \times rb
\]
\[
rc_6 \leftarrow rc_6 + rt
\]
\[
}\}
\]

\[
\ldots
\]
\[
\bar{c}_1 \leftarrow rc_1 \ldots \bar{c}_6 \leftarrow rc_6
\]

Figure 8.37: \((M_U, N_U) = (6, 1)\) code for x86 CISC
It is interesting to analyze how the ATLAS search engine settled on these parameter values. Note that on a processor that does not have a fused multiply-add instruction, $FMA = 1$ is equivalent to $FMA = 0$ and $L_s = 1$. The code produced by the ATLAS Code Generator is shown schematically in Figure 8.38. Note that this code uses 6 registers for $\bar{a}$ ($ra_1 \ldots ra_6$), 1 register for $\bar{b}$ ($rb$), 6 registers for $\bar{c}$ ($rc_1 \ldots rc_6$) and 1 temporary register (implicitly by the multiply-add statement). However, the back-end compiler (GCC) reorganizes this code into the code pattern shown in Figure 8.37.

Notice that the ATLAS Code Generator itself is not aware that the code of Figure 8.37 is optimal. However, setting $FMA = 1$ (even though there is no fused-multiply instruction) produces code that triggers the right instruction reorganization heuristics inside GCC, and performs well on the Opteron. This illustrates the well-known point that search does not need to be intelligent to do the right thing! Nevertheless, our refined model explains the observed performance data, makes intuitive sense, and can be easily incorporated into a compiler.

**MMM Performance**

Figure 8.32 shows the MMM performance. ATLAS Unleashed is once again the fastest implementation here, as it uses the highly-optimized, hand-tuned BLAS kernels, using the SSE2 SIMD instructions, for which the ATLAS Code Generator does not generate code. The native BLAS library is about 200 MFLOPS slower on average. ATLAS CGw/S and ATLAS Model perform at the same level as their corresponding mini-MMM kernels.

Refining the model as explained above brings ATLAS Model on par with
\[ rc_1 \leftarrow \bar{c}_1 \ldots rc_6 \leftarrow \bar{c}_6 \]

\[
\ldots
\]

\text{loop } k

\{
\begin{align*}
ra_1 & \leftarrow \bar{a}_1 \\
rb & \leftarrow \bar{b}_1 \\
rc_1 & \leftarrow rc_1 + ra_1 \times rb \\
ra_2 & \leftarrow \bar{a}_2 \\
ra_3 & \leftarrow \bar{a}_3 \\
rc_2 & \leftarrow rc_2 + ra_2 \times rb \\
rc_3 & \leftarrow rc_3 + ra_3 \times rb \\
ra_4 & \leftarrow \bar{a}_4 \\
ra_5 & \leftarrow \bar{a}_5 \\
rc_4 & \leftarrow rc_4 + ra_4 \times rb \\
rc_5 & \leftarrow rc_5 + ra_5 \times rb \\
ra_6 & \leftarrow \bar{a}_6 \\
rc_6 & \leftarrow rc_6 + ra_6 \times rb
\end{align*}
\}

\[
\ldots
\]

\[ \bar{c}_1 \leftarrow rc_1 \ldots \bar{c}_6 \leftarrow rc_6 \]

Figure 8.38: ATLAS unroll \((M_U, N_U) = (6, 1)\) code for x86 CISC

ATLAS CGw/s. To bridge the gap between ATLAS CGw/S and user contributed code, we would need a different code generator – one that understands SIMD and prefetch instructions. GCC exposes these as intrinsic functions and we plan to
explore this in future work.

**Performance Sensitivity Analysis**

Figure 8.34 shows the performance sensitivity to the values of the $M_U$ and $N_U$ optimization parameters. As discussed in Section 8.1.7, the optimal value is $(6, 1)$. From the graph we can see that the only plausible values are those with $N_U = 1$. Furthermore, performance increases while we grow $M_U$ from 1 to 6, while it suddenly drops for $M_U = 7$. This is easily explained by our refined model, as $M_U + 2 \leq N_R$ would require 9 registers, while only 8 are available.

Figure 8.33(a) shows the sensitivity of performance to the value of the $N_B$ optimization parameter. The first drop in performance is the result of L1 data cache misses starting to occur. This fact is accurately captured by our model for $N_B$ in Inequality (6.5). Solving the inequality for $C = 8192$ (the L1 data cache capacity in double-sized floating-point values), we obtain $N_B = 89$. Similarly the second drop in performance in Figure 8.33(a) can be explained by applying the same model to the 1MB L2 cache.

Figure 8.35 shows the performance sensitivity to the value of the $K_U$ optimization parameter. On this machine the entire mini-MMM loop body can fit into the L1 instruction cache for arbitrary $K_U$ values (up to $K_U = N_B$). Performance is relatively insensitive to $K_U$ as long as this unroll factor is sufficiently large ($K_U > 10$).

Figure 8.36 shows the performance sensitivity to the value of the $L_s$ optimization parameter. As we mentioned before, when $FMA = 1$, the $L_s$ optimization parameter does not influence the generated code. Therefore, performance is constant with respect to $L_s$. 

### 8.1.8 AMD Athlon MP

#### Table 8.22: AMD Athlon MP: Platform Specification

<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>Out-Of-Order, CISC, x86</td>
</tr>
<tr>
<td>CPU Core Frequency</td>
<td>1733 MHz</td>
</tr>
<tr>
<td>L1 Data Cache</td>
<td>64 KB, 64 B/line, 2-way</td>
</tr>
<tr>
<td>L1 Instruction Cache</td>
<td>64 KB, 64 B/line, 2-way</td>
</tr>
<tr>
<td>L2 Unified Cache</td>
<td>256 KB, 64 B/line, 16-way</td>
</tr>
<tr>
<td>Floating-Point Registers</td>
<td>8</td>
</tr>
<tr>
<td>Floating-Point Functional Units</td>
<td>ADD + MUL + Memory</td>
</tr>
<tr>
<td>Floating-Point Multiply Latency</td>
<td>4</td>
</tr>
<tr>
<td>Has Fused Multiply Add</td>
<td>No</td>
</tr>
<tr>
<td>Operating System</td>
<td>Linux 2.4.20</td>
</tr>
<tr>
<td>C Compiler</td>
<td>GNU C/C++ 3.2.2</td>
</tr>
<tr>
<td>Fortran Compiler</td>
<td>GNU Fortran 3.2.2</td>
</tr>
</tbody>
</table>

#### Table 8.23: AMD Athlon MP: Optimization Parameters

<table>
<thead>
<tr>
<th>Model, Unleashed</th>
<th>NB, MU, NU, KU, LU, FMA, IP, IP, NF</th>
<th>MFLOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>CGw/S</td>
<td>76, 4, 1, 76, 1, 0, 0, 3, 2</td>
<td>1531</td>
</tr>
<tr>
<td>Model</td>
<td>88, 2, 1, 88, 2, 0, 0, 2, 2</td>
<td>1239</td>
</tr>
<tr>
<td>Unleashed</td>
<td>30, 2, 1, 88, 2, 0, 2, 2</td>
<td>2512</td>
</tr>
</tbody>
</table>

#### Table 8.24: AMD Athlon MP: Timings

<table>
<thead>
<tr>
<th></th>
<th>Search</th>
<th>Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Machine Parameters</td>
<td>220s</td>
<td>121s</td>
</tr>
<tr>
<td>Optimization Parameters</td>
<td>3195s</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>3415s</td>
<td>121s</td>
</tr>
</tbody>
</table>
Figure 8.39: AMD Athlon MP: MMM Performance

Figure 8.40: AMD Athlon MP: Sensitivity of performance to $N_B$
Figure 8.41: AMD Athlon MP: Sensitivity of performance to $M_U$ and $N_U$.

Figure 8.42: AMD Athlon MP: Sensitivity of performance to $K_U$.

Figure 8.43: AMD Athlon MP: Sensitivity of performance to $L_s$. 
The AMD Athlon implements the x86 instruction set, so we would expect the experimental results to be similar to those on the Opteron.

**mini-MMM**

Table 8.24 shows that on this machine, the mini-MMM code generated by ATLAS Model runs roughly 20% slower than the code generated by ATLAS CGw/S. Figure 8.40(a) shows that the choice of $N_B$ made by the model is reasonable, while Figure 8.41 shows that the register-tile values were not chosen optimally by the model, as on the Opteron. The problem and its solution are similar to those on the Opteron.

**MMM Performance**

Figure 8.39 shows MMM performance. ATLAS Unleashed out-performs the other approaches by a significant margin. The hand-coded BLAS do almost as well, followed by ATLAS CGw/S.

**Sensitivity Analysis**

Figure 8.41 shows the sensitivity of performance to the values of $M_U$ and $N_U$.

Figure 8.40(a) shows the sensitivity of performance to the value of $N_B$. Figure 8.40(b) shows a scaled-up version of this graph in the region of the optimal $N_B$ value. Both ATLAS Model and ATLAS CGw/S choose good values of $N_B$. In Figure 8.40(b), the saw-tooth with period 2 arises from the overhead of executing clean-up code when the value of $N_B$ is odd, and therefore not divisible by the value of $M_U (= 2)$. As on other machines, we do not understand the saw-tooth with period 4 that has larger spikes in performance.
Figure 8.44: AMD Athlon MP: Sensitivity of performance to $K_U$

Figure 8.42 shows the sensitivity of performance to the value of $K_U$. The L1 I-cache is large enough to permit full unrolling ($K_U = N_B$). However, the sensitivity graph of $K_U$ is anomalous; performance is relatively low for all values of $K_U$ other than $K_U = N_B$. By examining the code produced by the native compiler (GCC), we found that this anomaly arose from interference between instruction scheduling in ATLAS and instruction scheduling in GCC. Notice that ATLAS CGw/S uses $FMA = 0$, so it attempts to schedule instructions and perform software pipelining in the mini-MMM code. Fully unrolling the $k'$ loop ($K_U = N_B$) produces straight-line code that is easier for GCC to schedule.

To verify this conjecture, we redid the $K_U$ sensitivity study with $FMA$ set to 1. Figure 8.44 shows the results. Setting $FMA = 1$ dissuades the ATLAS Code Generator from attempting to schedule code, so GCC has an easier job, producing a $K_U$ sensitivity graph that is in line with what we would expect.

Notice that our refined model, described in the context of the Opteron, performs extremely well – 1544 MFLOPS for mini-MMM, which is faster than the performance of the mini-MMM produced by ATLAS CGw/S.

Figure 8.43 shows the sensitivity of performance to the value of the $L_s$. 

![Figure 8.44: AMD Athlon MP: Sensitivity of performance to $K_U$](image-url)
### 8.1.9 Intel Pentium III

#### Table 8.25: Intel Pentium III: Platform Specification

<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>Out-Of-Order, CISC, x86</td>
</tr>
<tr>
<td>CPU Core Frequency</td>
<td>1266 MHz</td>
</tr>
<tr>
<td>L1 Data Cache</td>
<td>16 KB, 32 B/line, 4-way</td>
</tr>
<tr>
<td>L1 Instruction Cache</td>
<td>16 KB, 32 B/line, 4-way</td>
</tr>
<tr>
<td>L2 Unified Cache</td>
<td>512 MB, 32 B/line, 8-way</td>
</tr>
<tr>
<td>Floating-Point Registers</td>
<td>8</td>
</tr>
<tr>
<td>Floating-Point Functional Units</td>
<td>1</td>
</tr>
<tr>
<td>Floating-Point Multiply Latency</td>
<td>5</td>
</tr>
<tr>
<td>Has Fused Multiply Add</td>
<td>No</td>
</tr>
<tr>
<td>Operating System</td>
<td>Linux 2.4.20-28.8smp</td>
</tr>
<tr>
<td>C Compiler</td>
<td>GNU C/C++ 3.2</td>
</tr>
<tr>
<td>Fortran Compiler</td>
<td>GNU Fortran 3.2</td>
</tr>
</tbody>
</table>

#### Table 8.26: Intel Pentium III: Optimization Parameters

<table>
<thead>
<tr>
<th></th>
<th>$N_B$</th>
<th>$M_U$, $N_U$, $K_U$</th>
<th>$L_s$</th>
<th>$F_{\text{FMA}}$, $I_{\text{F}}, N_{\text{F}}$</th>
<th>MFLOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>CGw/S</td>
<td>44</td>
<td>4, 1, 44</td>
<td>3</td>
<td>0, 0, 3, 2</td>
<td>894</td>
</tr>
<tr>
<td>Model</td>
<td>42</td>
<td>2, 1, 42</td>
<td>2</td>
<td>0, 0, 2, 2</td>
<td>841</td>
</tr>
<tr>
<td>Unleashed</td>
<td>40</td>
<td></td>
<td></td>
<td></td>
<td>951</td>
</tr>
</tbody>
</table>

#### Table 8.27: Intel Pentium III: Timings

<table>
<thead>
<tr>
<th></th>
<th>Search</th>
<th>Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Machine Parameters</td>
<td>133s</td>
<td>100s</td>
</tr>
<tr>
<td>Optimization Parameters</td>
<td>630s</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>763s</td>
<td>100s</td>
</tr>
</tbody>
</table>
Figure 8.45: Intel Pentium III: MMM Performance

Figure 8.46: Intel Pentium III: Sensitivity of performance to $N_B$
Figure 8.47: Intel Pentium III: Sensitivity of performance to $M_U$ and $N_U$

Figure 8.48: Intel Pentium III: Sensitivity of performance to $K_U$

Figure 8.49: Intel Pentium III: Sensitivity of performance to $L_s$
mini-MMM

On this machine, mini-MMM code produced by ATLAS Model is about 50 MFLOPS (6%) slower than mini-MMM code produced by ATLAS CGw/S. The code produced by ATLAS Unleashed performs roughly 50 MFLOPS better than the code produced by ATLAS CGw/S.

The difference in performance between the codes produced by ATLAS CGw/S and ATLAS Model arises mostly from the sub-optimal register tile chosen by the model, as explained in the context of the Opteron in Section 8.1.7. Using (6, 1) as the register tile raises mini-MMM performance to 916 MFLOPS.

MMM Performance

Figure 8.45 shows MMM performance. The hand-coded BLAS performs at roughly 1100 MFLOPS, whereas the codes produced by ATLAS CGw/S and ATLAS Unleashed perform roughly at 900 MFLOPS. The code produced by ATLAS Model runs roughly at 850 MFLOPS; using the refined model improves performance to a point that is slightly above the performance of code produced by ATLAS CGw/S.

Sensitivity Analysis

Figure 8.47 shows the sensitivity of performance to the values of $M_U$ and $N_U$. Like all x86 machines, the Pentium III has a limited number of logical registers. Our base-line model picked (2, 1) for the register tile, whereas ATLAS CGw/S chose (4, 1). If we use the refined model described in Section 8.1.7, the size of the register tile becomes (6, 1), and mini-MMM performance rises to 916 MFLOPS.

Figure 8.46(a) shows the sensitivity of performance to the value of $N_B$. Figure 8.46(b) shows a scaled-up version of this graph in the region of the optimal
$N_B$ value. The broad peak in Figure 8.46(a) arises from the influence of the L2 cache (capacity of 512 KB). Using Inequality (6.1) for the L2 cache, we obtain $N_B = 104$, which is the $N_B$ values where the peak starts, while Inequality (6.5) gives $N_B = 164$, which corresponds to the $N_B$ value where the peak ends. The L2 cache on the Pentium III is 8-way set-associative, so the drop in performance between $N_B = 104$ and $N_B = 164$ is small.

Figure 8.48 shows the sensitivity of performance to the value of the $K_U$. On this machine, the L1 instruction cache is large enough to permit full unrolling ($K_U = N_B$).

Figure 8.49 shows the sensitivity of performance to the value of the $L_s$. There is no fused multiply-add instruction, so performance is sensitive to the value of $L_s$, but both ATLAS Model and ATLAS CGw/S find reasonable values for this parameter. If we use the refined model described in Section 8.1.7, we set $FMA = 1$, and the value of the $L_s$ parameter becomes irrelevant.
8.1.10 Intel Pentium 4

Table 8.28: Intel Pentium 4: Platform Specification

<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>Out-Of-Order, CISC, x86</td>
</tr>
<tr>
<td>CPU Core Frequency</td>
<td>2000 MHz</td>
</tr>
<tr>
<td>L1 Data Cache</td>
<td>8 KB, 64 B/line, 4-way</td>
</tr>
<tr>
<td>L1 Instruction Cache</td>
<td>12 K uOPs, 6 uOPs/line, 8-way</td>
</tr>
<tr>
<td>L2 Unified Cache</td>
<td>512 KB, 128 B/line, 8-way</td>
</tr>
<tr>
<td>Floating-Point Registers</td>
<td>8</td>
</tr>
<tr>
<td>Floating-Point Functional Units</td>
<td>1</td>
</tr>
<tr>
<td>Floating-Point Multiply Latency</td>
<td>7</td>
</tr>
<tr>
<td>Has Fused Multiply Add</td>
<td>No</td>
</tr>
<tr>
<td>Operating System</td>
<td>Linux 2.4.20-30.9smp</td>
</tr>
<tr>
<td>C Compiler</td>
<td>GNU C v3.2.2</td>
</tr>
<tr>
<td>Fortran Compiler</td>
<td>GNU Fortran 3.2.2</td>
</tr>
</tbody>
</table>

Table 8.29: Intel Pentium 4: Optimization Parameters

<table>
<thead>
<tr>
<th>Model</th>
<th>NB</th>
<th>MU, NU, KU</th>
<th>Ls</th>
<th>FMA</th>
<th>FP, IP, NP</th>
<th>MFLOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>CGw/S</td>
<td>28</td>
<td>3, 1, 28</td>
<td>1</td>
<td>0</td>
<td>0, 2, 1</td>
<td>1504</td>
</tr>
<tr>
<td>Model</td>
<td>30</td>
<td>1, 1, 30</td>
<td>4</td>
<td>0</td>
<td>0, 2, 2</td>
<td>913</td>
</tr>
<tr>
<td>Unleashed</td>
<td>72</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3317</td>
</tr>
</tbody>
</table>

Table 8.30: Intel Pentium 4: Timings

<table>
<thead>
<tr>
<th></th>
<th>Search</th>
<th>Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Machine Parameters</td>
<td>136s</td>
<td>98s</td>
</tr>
<tr>
<td>Optimization Parameters</td>
<td>643s</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>779s</td>
<td>98s</td>
</tr>
</tbody>
</table>
Figure 8.50: Intel Pentium 4: MMM Performance

Figure 8.51: Intel Pentium 4: Sensitivity of performance to $N_B$
Figure 8.52: Intel Pentium 4: Sensitivity of performance to $M_U$ and $N_U$

Figure 8.53: Intel Pentium 4: Sensitivity of performance to $K_U$

Figure 8.54: Intel Pentium 4: Sensitivity of performance to $L_s$
**mini-MMM**

On this machine, the mini-MMM code produced by ATLAS Model is about 600 MFLOPS (40%) slower than the mini-MMM code produced by ATLAS CGw/S. This is mostly because of the sub-optimal register tile used by ATLAS Model; changing it to (6, 1) improves the performance of the mini-MMM code produced by ATLAS Model to 1445 MFLOPS, which is only 50 MFLOPS (3%) less than the performance of the mini-MMM code produced by ATLAS CGw/S.

The mini-MMM produced by ATLAS Unleashed is roughly twice as fast as the mini-MMM produced by ATLAS Model because this code uses the SSE2 vector extensions to the x86 instruction set.

**MMM Performance**

Figure 8.50 shows the MMM performance. The hand-coded BLAS routines for this machine perform best, followed by the code produced by ATLAS Unleashed. Both the hand-coded BLAS and the code produced by ATLAS Unleashed use the SSE2 vector extensions, and this accounts for most of the gap between these codes and the codes produced by ATLAS Model and ATLAS CGw/S. We do not know why the hand-coded BLAS performs substantially better than the code produced by ATLAS Unleashed.

The gap in performance between the codes produced by ATLAS CGw/S and ATLAS Model disappears if the refined model for register tiles is used.

**Sensitivity Analysis**

Figure 8.52 shows the sensitivity of performance to the values of $M_U$ and $N_U$. This figure shows that the best register tile is $(5, 1)$, which produces mini-MMM code
that runs at 1605 MFLOPS. Using (6, 1) as the register tile is not as good because it reduces performance to 1521 MFLOPS.

Figure 8.51(a) shows the sensitivity of performance to the value of \( N_B \). Figure 8.51(b) shows a scaled-up version of this graph in the region of the optimal \( N_B \) value. Both ATLAS Model and ATLAS CGw/S choose good tile sizes for the L1 cache. Tiling for the L2 cache gives slightly better performance. The L2 cache on this machine has a capacity of 256 KB; using Inequalities (6.1) and (6.5), we get \( N_B = 105 \) and \( N_B = 180 \), which agree well with the data.

Figure 8.53 shows the sensitivity of performance to the value of \( K_U \). On this machine, the L1 instruction cache is large enough to permit full unrolling (\( K_U = N_B \)). Figure 8.54 shows the sensitivity of performance to the value of \( L_s \).

8.1.11 Discussion

The experimental results in this section can be summarized as follows. Figure 8.55 describes the analytical models used to compute the values for the optimization parameters. This figure also shows the refined model used to compute the register tile values for the x86 architectures.

Figure 8.56 shows the relative performance of the mini-MMM codes produced by ATLAS Model and by ATLAS Unleashed, using the performance of the codes produced by ATLAS CGw/S as the base line (the 100% line in this figure represents the performance of ATLAS CGw/S on all machines). All the performance numbers for ATLAS Model in this graph are obtained by tiling for the L1 cache.

We see that on all machines other than the Itanium, the codes produced by using the analytical models perform almost as well or slightly better than the codes produced using global search. On the Itanium, we saw that it is best to
• **Estimating** $FMA$: Use the machine parameter $FMA$

• **Estimating** $L_s$: $L_s = \left\lceil \frac{L_s \times |ALU_{FP}| + 1}{2} \right\rceil$

• **Estimating** $M_U$ and $N_U$: $M_U \times N_U + N_U + M_U + L_s \leq N_R$
  
  1. $M_U, N_U \leftarrow u$.
  2. Solve constraint for $u$.
  3. $M_U \leftarrow \max(u, 1)$.
  4. Solve constraint for $N_U$.
  5. $N_U \leftarrow \max(N_U, 1)$.
  6. If $M_U < N_U$ then swap $M_U$ and $N_U$.
  7. **Refined Model**: If $N_U = 1$ then
     - $M_U \leftarrow N_R - 2$
     - $N_U \leftarrow 1$
     - $FMA \leftarrow 1$

• **Estimating** $N_B$: $\left\lceil \frac{N_B^2}{B_1} \right\rceil + 3 \left\lceil \frac{N_B \times N_U}{B_1} \right\rceil + \left\lceil \frac{M_U}{B_1} \right\rceil \times N_U \leq C_B$.
  
  Trim $N_B$, to make it a multiple of $M_U$, $N_U$, and 2.

• **Estimating** $K_U$: Choose $K_U$ as the maximum value for which mini-MMM fits in the L1 instruction cache. Trim $K_U$ to make it divide $N_B$ evenly.

• **Estimating** $F_F$, $I_F$, and $N_F$: $F_F = 0$, $I_F = 2$, $N_F = 2$

Figure 8.55: Summary of Model

tile for the L3 cache, rather than the L1 cache. By using the L2 cache instead, ATLAS CGw/S was able to obtain some of the benefits of tiling for the L3 cache.
Figure 8.56: Summary of mini-MMM Performance. Performance numbers are normalized to that of ATLAS CGw/S, which is presented as 100%.

If we use this value in the model of Figure 8.55, we produce mini-MMM code of comparable performance. Using the actual capacity of the L3 cache gives even
better performance.

In our experiments we noticed that on several platforms, we get better MMM performance by tiling for a lower cache level, such as L2 or L3, rather than L1. This may result in a large value for $N_B$, which may hurt overall performance if the resulting MMM library routine is invoked from other routines such as LU and Cholesky factorizations [32]. It is unclear to us that this is an issue in the context of compilers, where codes like LU and Cholesky would be optimized directly, rather than built upon MMM.

8.2 Model refinement and local search

8.2.1 x86-based architectures

In Section 8.1.7 we noticed that the original model (denoted simply as “Model” in this section) performed almost 40% worse than ATLAS CGw/S. We analyzed the sensitivity of optimization parameters and designed a Refined Model for estimating register tile size on x86-based architectures. As we discuss in Section 8.1.11 this refinement alone is enough to close the performance gap between Model and ATLAS CGw/S on all x86-based architectures. Results are presented in Figure 8.56.

8.2.2 Multilevel memory hierarchies

As discussed in Section 8.1, there are some machines for which tiling for the L2 or L3 cache will give better performance than tiling for the L1 cache. The model presented in Chapter 6 does not account for cache miss penalties at different cache levels, so although we estimate tile sizes for different cache levels, we cannot determine which level to tile for.
One approach to addressing this problem in the context of model-driven optimization is to refine the model to include miss penalties. Our experience however is that it is difficult to use micro-benchmarks to measure miss penalties accurately for lower levels of the memory hierarchy on modern machines, because the actual penalties are variable and depend on the current utilization of the bus. Therefore, we decided to estimate tile sizes for all the cache levels according to Inequalities (6.1) and (6.5), and then empirically determine which one gives the best performance. When combined with the local search techniques presented in Chapter 7, we call this approach *Multilevel (ML) Local Search*.

Notice that in the context of global search, the problem can be addressed by making the search space for $N_B$ large enough. However, this would increase the search time substantially since the size of an L3 cache, which would be used to bound the search space, is typically much larger than the size of an L1 cache. This difficulty highlights the advantage of our approach of using model-driven optimization together with a small amount of search - we can tackle multi-level memory hierarchies without increasing installation time significantly.

ML Local Search resolved the multilevel memory hierarchy problems on all architectures discussed in Section 8.1. Here we present experimental results from two architectures where this ML Local Search was particularly useful – the Intel Itanium 2 and the SGI R12K.

**Intel Itanium 2**

Table 8.31 shows the values of the optimization parameters for Model, Refined Model, Local Search, Multi-Level (ML) Local Search, Global Search, and Unleashed, along with the corresponding performance numbers for mini-MMM.
Table 8.31: Optimization Parameters for Intel Itanium 2

<table>
<thead>
<tr>
<th></th>
<th>$N_B$</th>
<th>$M_U, N_U, K_U$</th>
<th>$L_s$</th>
<th>FMA</th>
<th>$F_P, I_P, N_P$</th>
<th>MFLOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>30</td>
<td>10, 10, 4</td>
<td>1</td>
<td>1</td>
<td>0, 2, 2</td>
<td>3130</td>
</tr>
<tr>
<td>Refined Model</td>
<td>30</td>
<td>10, 10, 4</td>
<td>1</td>
<td>1</td>
<td>0, 2, 2</td>
<td>3130</td>
</tr>
<tr>
<td>Local Search</td>
<td>30</td>
<td>10, 10, 4</td>
<td>1</td>
<td>1</td>
<td>0, 2, 2</td>
<td>3130</td>
</tr>
<tr>
<td>ML Local Search</td>
<td>360</td>
<td>10, 10, 4</td>
<td>1</td>
<td>1</td>
<td>0, 2, 2</td>
<td>4602</td>
</tr>
<tr>
<td>Global Search</td>
<td>80</td>
<td>10, 10, 4</td>
<td>4</td>
<td>1</td>
<td>0, 19, 1</td>
<td>4027</td>
</tr>
<tr>
<td>Unleashed</td>
<td>120</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4890</td>
</tr>
</tbody>
</table>

Figure 8.57: Intel Itanium 2: MMM Performance

Table 8.32 shows the times taken by our micro-benchmarks and by the ATLAS micro-benchmarks for determining machine and optimization parameters.

Table 8.32: Timings for Intel Itanium 2

<table>
<thead>
<tr>
<th></th>
<th>Parameters</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Machine</td>
<td>Optimization</td>
</tr>
<tr>
<td>Model</td>
<td>143</td>
<td>6</td>
</tr>
<tr>
<td>Refined Model</td>
<td>143</td>
<td>6</td>
</tr>
<tr>
<td>Local Search</td>
<td>143</td>
<td>162</td>
</tr>
<tr>
<td>ML Local Search</td>
<td>143</td>
<td>278</td>
</tr>
<tr>
<td>Global Search</td>
<td>1554</td>
<td>29667</td>
</tr>
</tbody>
</table>

Figure 8.57 shows the MMM performance for all these approaches. Refined Model and Local Search are not plotted on this platform, because their performance is virtually equivalent to that of Model. The native BLAS used is MKL 6.1.
Model does not perform well because it tiles for the L1 data cache. For the Itanium, ATLAS used the size of the L2 cache (256KB) to restrict $N_B$, effectively selecting the maximum value in the search range ($N_B = 80$). Nevertheless this tile size is not optimal either. The Multi-level model determined that tiling for the 3 MB L3 cache is optimal, and chooses a value of $N_B = 362$. This is refined to $N_B = 360$ by local search. This improves performance compared to both Model and Global Search.

**SGI R12000**

Table 8.33 shows the values of the optimization parameters for Model, Refined Model, Local Search, Multi-Level (ML) Local Search, Global Search, and Unleashed, along with the corresponding performance numbers for mini-MMM.

<table>
<thead>
<tr>
<th></th>
<th>$N_B$</th>
<th>$M_U, N_U, K_U$</th>
<th>$L_s$</th>
<th>FMA</th>
<th>$F_F, F_F, N_F$</th>
<th>MFLOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>58</td>
<td>5, 4, 58</td>
<td>1</td>
<td>1</td>
<td>0, 2, 2</td>
<td>440</td>
</tr>
<tr>
<td>Refined Model</td>
<td>58</td>
<td>5, 4, 58</td>
<td>1</td>
<td>1</td>
<td>0, 2, 2</td>
<td>440</td>
</tr>
<tr>
<td>Local Search</td>
<td>58</td>
<td>5, 4, 58</td>
<td>1</td>
<td>1</td>
<td>0, 2, 2</td>
<td>440</td>
</tr>
<tr>
<td>ML Local Search</td>
<td>418</td>
<td>5, 4, 16</td>
<td>1</td>
<td>1</td>
<td>0, 2, 2</td>
<td>508</td>
</tr>
<tr>
<td>Global Search</td>
<td>64</td>
<td>4, 5, 64</td>
<td>1</td>
<td>0</td>
<td>1, 8, 1</td>
<td>457</td>
</tr>
<tr>
<td>Unleashed</td>
<td>64</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>463</td>
</tr>
</tbody>
</table>

Table 8.34 shows the times taken by our micro-benchmarks and by the ATLAS micro-benchmarks for determining machine and optimization parameters.

Figure 8.58 shows the MMM performance on the SGI R12K. Refined Model and Local Search are not plotted on this platform, because as Table 8.33 suggests, their performance is virtually equivalent to that of Model. For native BLAS we used SGI SCSL v.1.4.1.3.

The most interesting fact on this platform is that Multi-Level Local Search
Table 8.34: Timings for SGI R12000

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Machine</th>
<th>Optimization</th>
<th>Total (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>118</td>
<td>13</td>
<td>131</td>
</tr>
<tr>
<td>Refined Model</td>
<td>118</td>
<td>13</td>
<td>131</td>
</tr>
<tr>
<td>Local Search</td>
<td>118</td>
<td>457</td>
<td>575</td>
</tr>
<tr>
<td>ML Local Search</td>
<td>118</td>
<td>496</td>
<td>608</td>
</tr>
<tr>
<td>Global Search</td>
<td>251</td>
<td>2131</td>
<td>2382</td>
</tr>
</tbody>
</table>

Figure 8.58: SGI R12000: MMM Performance

successfully finds that it is worth tiling for the L2 cache. By doing this, it achieves better performance than even the native BLAS. Global Search achieves slightly better performance than Model due to the minor differences in several optimization parameters. Unleashed does fine for relatively small matrices but for large ones performs worse than Model and Global Search. Although not entirely visible from the plot, on this platform, the native compiler (SGI MIPSPro) does a relatively good job.

8.2.3 Discussion

The experiments reported in show that the combination of model refinement and local search is effective in closing performance gaps between the model-generated code and the code generated by global search, while keeping code generation time
small. However, it is important to realize that reducing library generation time is not the primary focus of our work; rather, our goal is to find optimization strategies for generating very high-performance code that can be used in general-purpose compilers because they scale to large programs and complex architectures.
Chapter 9

Conclusions and Future Work

\[ \ldots \text{the end of all our exploring} \]
\[ \text{Will be to arrive where we started} \]
\[ \text{And know the place for the first time.} \]

T.S.Eliot, Four Quartets

The experimental results demonstrate that it is possible to use analytical models to determine near-optimal values for the optimization parameters needed in the ATLAS system to produce high-quality BLAS. Our models were designed to be compatible with the ATLAS Code Generator; for example, since ATLAS uses square cache tiles, we had only one parameter $N_B$, whereas a different Code Generator that uses general rectangular tiles may require three cache tile parameters. Van de Geijn and co-workers have considered such models in their work on optimizing matrix multiplication code for multi-level memory hierarchies [30, 31, 34].

Our results show that using models to determine values for the optimization parameters is much faster than using empirical search. However, this does not imply that search has no role to play in the generation of high-performance code. Systems like FFTW and SPIRAL use search not to choose optimal values for transformation parameters, but to choose an optimal algorithm from a whole suite of algorithms. We do not know if model-driven optimization is effective in this context. Even in the relatively simple context of the BLAS, there are aspects of program behavior that may not be worth modeling in practice even if they can be modeled in principle. For example, the analytical models for $N_B$ described in Chapter 6 ignore conflict misses. Although there is some work in the compiler.
literature on modeling conflict misses [15, 18], these models appear to be computationally intractable. Fortunately, the effect of conflict misses on performance can be reduced by appropriate copying. If necessary, the value of $N_B$ found by the model can be refined by local search in the neighborhood of the $N_B$ value predicted by the model. This combination of modeling and local search may be the most tractable approach for optimizing large programs for complex high-performance architectures.

At the end of this dissertation, we are left with the same question that we asked at its beginning: how do we improve the state of the art of compilers? Conventional wisdom holds that current compilers are unable to produce high-quality code because the analytical models they use to estimate optimization parameter values are overly simplistic compared to the complexity of modern high-performance architectures. The results in this dissertation contradict this conventional wisdom, and suggest that there is no intrinsic reason why compilers cannot use analytical models to generate excellent code, at least for the BLAS.

However, it is important not to underestimate the challenge in improving general-purpose compilers to bridge the current performance gap with library generators. Although the techniques used by ATLAS, such as loop tiling, unrolling, and instruction scheduling, have been in the compiler literature for many years, it is not easy to incorporate them into general-purpose compilers. For example, transformations such as tiling are not always legal, so a general-purpose compiler must perform dependence analysis before transforming a program. In contrast, the implementor of a library generator focuses on one application and knows the precise structure of the code to be generated for that application, so she is not encumbered by the baggage required to support the restructuring of general codes.
At the very least, improving the state of the art of compilation technology will require an open compiler infrastructure that permits researchers to experiment easily with different transformations and to vary the parameters of those transformations. This has been a long-standing problem, and no adequate infrastructure exists in spite of many attempts.

An equally important conclusion of this study is that there is still a significant gap in performance between the code generated by ATLAS CGw/S and the vendor BLAS routines. Although we understand some of the reasons for this gap, the problem of automating library generation remains open. The high cost of library and application tuning makes this one of the most important questions we face today.
Appendix A

X-Ray: Experimental Results

A.1 CPU Features

In this section, we will present and discuss the results from running X-Ray on a number of different hardware architectures. In particular, we will show results for the following,

- Intel Itanium 2, 2-way SMP, 1.5GHz
- AMD Opteron 240, 2-way SMP, 1.4GHz
- Sun UltraSPARC IIIi, 2-way SMP, 1GHz
- Intel Pentium 4 Xeon, 2-way SMP, 2.2GHz
- AMD Athlon MP 2800+, 2-way SMP, 2.25GHz
- SGI R12000, 300MHz
- IBM Power 3, 8-way SMP, 375MHz

We also compare our results with those from three other similar platform-independent tools:

- **Calibrator v0.9e** [41] is a memory system benchmark aimed at measuring capacity, block size, and latency at each level of the memory hierarchy and the corresponding parameters for TLBs, such as the number of entries, the page size, and the latency.
• **Imbench v3.0a3** [45, 55, 44] is a suite of benchmarks for measuring operating systems parameters such as thread-creation time and context-switch time. In version 3, the authors have included several hardware benchmarks for measuring CPU frequency, latency and parallelism of different operations, capacity, block size, and latency of each level of the memory hierarchy, and the number of TLB entries.

• **MOB v0.1.1** [9] is an ambitious project to create a benchmark suite capable of measuring a large number of properties of the memory hierarchy, including capacity, block size, associativity, sharedness, replacement policy, write mode, and latency of each level, as well as the corresponding parameters for TLBs.

We also looked at ATLAS v3.6.0 [56], which has a set of micro-benchmarks to aid its empirical search engine. These micro-benchmarks measure the latency of floating-point multiplication, the number of floating-point registers, the existence of a fused multiply-add instruction and the capacity of the L1 data cache. ATLAS needs only approximate values for these hardware parameters because it uses these values to bound the search space for optimization parameters, and not to estimate optimal values for these parameters. The micro-benchmarks in ATLAS are of limited precision, although they are perfectly adequate for what is needed of them in the context of the ATLAS system. Therefore, we decided not compare X-Ray directly to it.

Because all the tools, including X-Ray, measure hardware parameters empirically, the results sometimes vary from one execution to the next. These variations are negligible with X-Ray, but sometimes quite noticeable with the other tools. The results we present are the best ones we obtained in several trial runs.
Table A.1: Summary of Experimental Results: CPU Features

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Tool</th>
<th>Itanium</th>
<th>Opteron</th>
<th>SPARC</th>
<th>Pentium4</th>
<th>Athlon</th>
<th>R12000</th>
<th>Power 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency</td>
<td>Actual</td>
<td>1500</td>
<td>1400</td>
<td>1000</td>
<td>2200</td>
<td>2250</td>
<td>300</td>
<td>375</td>
</tr>
<tr>
<td></td>
<td>X-Ray</td>
<td>1488.16</td>
<td>1379.31</td>
<td>994.21</td>
<td>4324.09</td>
<td>2129.19</td>
<td>298.26</td>
<td>375.43</td>
</tr>
<tr>
<td></td>
<td>lmbench</td>
<td>1497</td>
<td>1392</td>
<td>1001</td>
<td>2164</td>
<td>2117</td>
<td>292</td>
<td>375</td>
</tr>
<tr>
<td>(MHz)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP Add</td>
<td>Actual</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Latency</td>
<td>X-Ray</td>
<td>3.98</td>
<td>3.96</td>
<td>4.04</td>
<td>10</td>
<td>4</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>(cycles)</td>
<td>4.01</td>
<td>4.04</td>
<td>3.88</td>
<td>5</td>
<td>4</td>
<td>2.01</td>
<td>4.01</td>
</tr>
<tr>
<td>FP Multiply</td>
<td>Actual</td>
<td>0.5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.5</td>
</tr>
<tr>
<td>Latency</td>
<td>X-Ray</td>
<td>0.51</td>
<td>0.99</td>
<td>4.11</td>
<td>14.09</td>
<td>4</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>(cycles)</td>
<td>0.34</td>
<td>1.48</td>
<td>2.73</td>
<td>1.02</td>
<td>0.73</td>
<td>3.44</td>
<td>1.34</td>
</tr>
<tr>
<td>FP Add</td>
<td>Actual</td>
<td>0.5</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0.5</td>
</tr>
<tr>
<td>Initiation Interval</td>
<td>X-Ray</td>
<td>0.51</td>
<td>0.99</td>
<td>4</td>
<td>1.01</td>
<td>1</td>
<td>1</td>
<td>0.5</td>
</tr>
<tr>
<td></td>
<td>(cycles)</td>
<td>0.34</td>
<td>1.48</td>
<td>2.85</td>
<td>1.05</td>
<td>0.75</td>
<td>3.44</td>
<td>1.34</td>
</tr>
<tr>
<td>FMA existence</td>
<td>Actual</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>X-Ray</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>lmbench</td>
<td>supported</td>
<td>supported</td>
<td>supported</td>
<td>supported</td>
<td>supported</td>
<td>supported</td>
<td>supported</td>
</tr>
<tr>
<td>(yes/no)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SMP</td>
<td>Actual</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X-Ray</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(count)</td>
<td>supported</td>
<td>supported</td>
<td>supported</td>
<td>supported</td>
<td>supported</td>
<td>supported</td>
<td>supported</td>
</tr>
<tr>
<td>SMT per CPU</td>
<td>Actual</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>X-Ray</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>(count)</td>
<td>supported</td>
<td>supported</td>
<td>supported</td>
<td>supported</td>
<td>supported</td>
<td>supported</td>
<td>supported</td>
</tr>
</tbody>
</table>

Table A.1 shows the CPU parameters measured by X-Ray and lmbench. Calibrator and MOB address only the memory hierarchy, so we do not discuss them here. The parameters that lmbench does not measure are marked “!supported”. Now we discuss the individual parameters in greater detail.
A.1.1 CPU Frequency

Processor frequency is not a valuable parameter by itself; its only purpose is to express the latencies of other hardware parameters in clock cycles. Therefore it is only important that it be a consistent base for relative comparisons. In addition, the actual CPU frequency may differ from the advertised frequency by 5%.

On most of the architectures, the results of both X-Ray and Imbench (in Table A.1) were close to the advertised actual frequencies. The only exception is the Pentium 4, where X-Ray measured double the actual value. The reason for this inconsistency is that the X-Ray methodology for computing frequency assumes that integer adds have an initiation interval of 1, but the Pentium 4 has a double-pumped integer ALU unit. In effect, the integer add instruction has an initiation interval of 0.500 on the P4. For relative comparisons of instruction latencies, this anomaly is not important.

A.1.2 Instruction Latency and Throughput

X-Ray can measure the latency and initiation interval of any instruction; for lack of space, we show only the results for double-precision floating-point addition and multiplication.

One detail is that Imbench measures these values in different units. It measures instruction latency \( l \) in nanoseconds and instruction parallelism \( p \) as a factor of improvement over \( l \). To produce directly comparable results, we converted the latency to cycles, using the Imbench’s measured processor frequency \( F_{CPU} \): 
\[
\text{cycles} = l \times F_{CPU}.
\]
Then we converted the parallelism to our notion of initiation interval using
\[
II = \frac{\text{cycles}}{p}.
\]

Similarly, Calibrator does not implement CPU frequency measurement, but
rather requires the user to input it as an external parameter. We used the advertised frequency of the processors to provide a value for this external parameter.

Both X-Ray and lmbench measure latencies very accurately, but lmbench does not measure some initiation intervals correctly. Two points deserve mention:

- The X-Ray numbers on the Pentium 4 are twice the actual values because they are relative to the integer ALU frequency, which, as mentioned in Section A.1.1, is twice higher than the rest of the core.

- On the Itanium 2 and the Power 3, the initiation interval of both instructions is 0.5, which means that the CPU is dispatching 2 instruction per cycle. This is because these architectures have two FP ALUs.

Paradoxically, X-Ray found that not every CPU with two fully pipelined ALUs has an initiation interval of 0.5. For example the Pentium III and Pentium 4 both have two integer ALUs, but their initiation intervals are 0.67. This anomaly arises from the fact that there are other bottlenecks in the CPU that prevent it from issuing the two integer instructions at a sustained rate. On the Pentium III the bottleneck is that there are only two register read ports, while on the Pentium 4 the bottleneck is the bandwidth between the instruction cache and the CPU core.

This paradox highlights a recurring theme in our investigation. When there is a disagreement between X-Ray and the vendor’s manuals, the values measured by X-Ray are more likely to be relevant to program optimization.
A.1.3 Number of Registers

Registers are often considered a level-0 cache $C_0$, as they are at the top of the memory hierarchy. If a machine has $N$ registers of type $T$, we can characterize $C_0 = \langle A, B, C \rangle = \langle N, \text{sizeof} (T), N \times \text{sizeof} (T) \rangle$. $C_0$ can exhibit spatial locality only in the case of vector registers (MMX, SSE, etc.). Furthermore, it is fully associative and the replacement policy is software controlled.

The only way to directly exercise this control is to program in assembly language. Portable software, on the other hand, is usually written in a high-level language like C and the native compiler is responsible for register allocation, register spills and fills. Nevertheless, when the ultimate goal is high-performance, programmers need to make assumptions about the number of registers available for register allocation and apply optimizing transformations like array scalarization and loop unrolling appropriately (e.g., ATLAS [61]).

Note that we measure the effective number of available registers, which is the value that is relevant for program optimization. This value is often be smaller than the number of actual registers in the given architecture for the following reasons.

- Some registers may be reserved for the Stack Pointer, Frame Pointer, Return Address, etc.
- Some registers may be hardwired with specific values, most often the floating point values 0.0 and 1.0.
- Compilers may use some registers in a special way, and they might not be available to the general register allocator, e.g., accumulators, register windows, etc.
• Compilers might not use all available registers for different reasons, e.g., targeting an older version of the ISA.

By appropriately defining the operation `add`, this method is able to measure all types of registers, including integer, floating point, and vector registers (e.g., MMX, SSE, 3DNow!, Altivec) through compiler intrinsics.

None of lmbench, Calibrator, and MOB try to measure the number of available registers. The ATLAS framework attempts to provide a rough estimate for the number of floating point registers, but they can afford to be conservative, as opposed to precise, because they only use the estimate to bound their search space. Table A.2 summarizes our measurement results.

<table>
<thead>
<tr>
<th>Architecture</th>
<th>available / actual</th>
<th>int</th>
<th>double</th>
<th>MMX</th>
<th>SSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pentium 4</td>
<td>5 / 8</td>
<td>8 / 8</td>
<td>8 / 8</td>
<td>8 / 8</td>
<td>8 / 8</td>
</tr>
<tr>
<td>Itanium 2</td>
<td>123 / 128</td>
<td>128 / 128</td>
<td>n/a</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td>Athlon MP</td>
<td>5 / 8</td>
<td>8 / 8</td>
<td>8 / 8</td>
<td>8 / 8</td>
<td>8 / 8</td>
</tr>
<tr>
<td>Opteron 240</td>
<td>14 / 16</td>
<td>16 / 16</td>
<td>8 / 8</td>
<td>16 / 16</td>
<td></td>
</tr>
<tr>
<td>UltraSPARC IIIi</td>
<td>24 / 32</td>
<td>31 / 32</td>
<td>n/a</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td>R12000</td>
<td>22 / 32</td>
<td>32 / 32</td>
<td>n/a</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td>Power 3</td>
<td>28 / 32</td>
<td>32 / 32</td>
<td>n/a</td>
<td>n/a</td>
<td></td>
</tr>
</tbody>
</table>

As expected the number of available integer registers is always less than the actual number of registers because some registers are reserved for use either by the hardware or by the compiler. The measured number of floating point registers is equal to the actual number in all cases except on the UltraSPARC IIIi machine, where one of the registers is hardwired to 0.0. The measured number of vector registers is always equal to the actual number. We do not provide results for 3DNow! and SSE2 registers, because they are equivalent to MMX and SSE register
respectively.

A.2 Memory Hierarchy

To report cache latency in CPU cycles we use a micro-benchmark for measuring CPU frequency, which is part of X-Ray. In this section we compare the results of running the memory-hierarchy portion of X-Ray on 7 platforms with the results of running the following three tools.

- **Calibrator v0.9e** [41] is a memory system benchmark aimed at measuring capacity, block size, and latency at each level of the memory hierarchy and TLB parameters, such as number of entries, page size, and latency.

- **lmbench v3.0a3** [44, 45, 55] is a suite of benchmarks for measuring operating systems parameters such as thread-creation time and context-switch time. Version 3 contains micro-benchmarks for measuring latency and parallelism of different operations, capacity, block size, and latency of each level of the memory hierarchy, and the number of TLB entries.

- **MOB v0.1.1** [9] is an ambitious project to create a benchmark suite capable of measuring a large number of properties of the memory hierarchy, including capacity, block size, associativity, sharedness, replacement policy, write mode, and latency of each level, as well as the corresponding TLB parameters.

Table A.3 shows the memory hierarchy parameters, along with the results from measuring them with the different tools. Whenever a parameter was not successfully computed, we use the following special entries to specify the reason:
• **n/a** – the tool does not claim to be able to measure this hardware parameter;

• **empty** – the benchmark completed but did not produce a value for this parameter;

• **abort** – an abnormal termination of some kind occurred prevented the benchmark from completion;

• **build** – the benchmark did not build successfully;

• **os** – OS-specific support is required for X-Ray to complete this measurement and we have not implemented such support yet.

### A.2.1 L1 Data Cache

As Table A.3 shows, X-Ray successfully found the correct values for all L1 cache parameters on all the platforms other than the Power 3, where it decided that the cache was 129-way set associative although it is actually 128-way set-associative. For reasons we do not understand, there was no performance loss in the micro-benchmark when moving from 128 to 129 steps, but there was a performance loss in moving from 129 to 130. This anomaly also affected the determination of the cache capacity slightly. The performance of the other tools varies, and the details are presented in Table A.3.

### A.2.2 Lower Level Caches

Lower level caches are physically addressed on all modern machines so we found it necessary to use super-pages to obtain consistent measurements of lower level cache parameters, as discussed in Section 4.4.4. Support for super-pages is very...
OS-specific, so we targeted the Linux system as a proof of concept. Table A.3 shows that X-Ray was able to measure lower level cache parameters correctly on all the Linux machines in our study (Pentium 4, Itanium 2, Athlon MP, and Opteron 240). We are currently working on the implementation for Solaris, IRIX and AIX, which will allow us to test X-Ray on the rest of the machines as well.

The numbers for the AMD machines (Athlon and Opteron) are interesting because they expose the fact that the L1 and L2 caches on these machines implement cache exclusion. Most platforms support cache inclusion, which means that information cached at a particular level of the memory hierarchy should also be cached in all lower levels. This is necessary to support cache-coherence protocols in SMP systems. AMD machines on the other hand use exclusion, so data never resides in both the L1 and L2 caches simultaneously. While this requires the L1 cache to snoop on the bus to resolve coherency issues, it effectively increases the useful capacity of L2 by the capacity of the L1.

X-Ray classified the 512KB, 16-way associative L2 cache of the AthlonMP as an 18-way set-associative cache with a capacity of 576KB (exactly $C_1 + C_2$). Similarly on the Opteron 240, the 1MB L2 was classified as a 17-way set associative cache with an effective capacity 1088KB (exactly $C_1 + C_2$). If the actual capacity of the $L_2$ cache is needed, it can be obtained by subtracting the capacity of the $L_1$ cache, although the combined capacity is what is actually relevant for autonomic code that wants to perform an optimization like cache tiling.

The performance of the other tools varied. Calibrator produced somewhat pessimistic results for cache capacity on some of the Linux machines; we believe this effect, too, arises from non-contiguous physical memory since this reduces the effective cache capacity. lmbench terminates abnormally on some platforms, but
produces accurate results when it terminates cleanly. MOB produced accurate results only for the capacity of the L2 cache of Itanium 2. In all other cases, it either aborted, produced a wrong result or did not produce a result at all.

The cache access latency figures produced by all the tools for lower level caches should be taken with a grain of salt since the actual access time can fluctuate substantially depending on what other memory bus transactions are occurring at the same time.
Table A.3: Summary of experimental results

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Actual</th>
<th>X-Ray</th>
<th>Calibrator</th>
<th>lmbench</th>
<th>MOB</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>L1 C (KB)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pentium 4</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Itanium 2</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>abort</td>
<td>4</td>
</tr>
<tr>
<td>Athlon MP</td>
<td>64</td>
<td>64</td>
<td>64</td>
<td>empty</td>
<td>abort</td>
</tr>
<tr>
<td>Opteron 240</td>
<td>64</td>
<td>64</td>
<td>64</td>
<td>abort</td>
<td>empty</td>
</tr>
<tr>
<td>UltraSPARC IIIi</td>
<td>64</td>
<td>64</td>
<td>64</td>
<td>64</td>
<td>abort</td>
</tr>
<tr>
<td>R12000</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>build</td>
</tr>
<tr>
<td>Power 3</td>
<td>64</td>
<td>64.5</td>
<td>64</td>
<td>64</td>
<td>empty</td>
</tr>
<tr>
<td><strong>L1 B (bytes)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pentium 4</td>
<td>64</td>
<td>64</td>
<td>32</td>
<td>64</td>
<td>abort</td>
</tr>
<tr>
<td>Itanium 2</td>
<td>64</td>
<td>64</td>
<td>64</td>
<td>abort</td>
<td>104</td>
</tr>
<tr>
<td>Athlon MP</td>
<td>64</td>
<td>64</td>
<td>64</td>
<td>empty</td>
<td>abort</td>
</tr>
<tr>
<td>Opteron 240</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>abort</td>
</tr>
<tr>
<td>UltraSPARC IIIi</td>
<td>1024</td>
<td>1088</td>
<td>768</td>
<td>abort</td>
<td>empty</td>
</tr>
<tr>
<td>R12000</td>
<td>1024</td>
<td>1024</td>
<td>8192</td>
<td>8192</td>
<td>build</td>
</tr>
<tr>
<td>Power 3</td>
<td>128</td>
<td>128</td>
<td>128</td>
<td>128</td>
<td>empty</td>
</tr>
<tr>
<td><strong>L1 A (count)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pentium 4</td>
<td>128</td>
<td>128</td>
<td>128</td>
<td>128</td>
<td>abort</td>
</tr>
<tr>
<td>Itanium 2</td>
<td>128</td>
<td>128</td>
<td>128</td>
<td>abort</td>
<td>128</td>
</tr>
<tr>
<td>Athlon MP</td>
<td>128</td>
<td>128</td>
<td>128</td>
<td>empty</td>
<td>abort</td>
</tr>
<tr>
<td>Opteron 240</td>
<td>128</td>
<td>128</td>
<td>128</td>
<td>empty</td>
<td>abort</td>
</tr>
<tr>
<td>UltraSPARC IIIi</td>
<td>128</td>
<td>128</td>
<td>128</td>
<td>128</td>
<td>build</td>
</tr>
<tr>
<td>R12000</td>
<td>128</td>
<td>128</td>
<td>128</td>
<td>128</td>
<td>empty</td>
</tr>
<tr>
<td>Power 3</td>
<td>128</td>
<td>128</td>
<td>128</td>
<td>128</td>
<td>empty</td>
</tr>
<tr>
<td><strong>L2 C (KB)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pentium 4</td>
<td>512</td>
<td>512</td>
<td>384</td>
<td>512</td>
<td>abort</td>
</tr>
<tr>
<td>Itanium 2</td>
<td>256</td>
<td>256</td>
<td>256</td>
<td>abort</td>
<td>256</td>
</tr>
<tr>
<td>Athlon MP</td>
<td>512</td>
<td>512</td>
<td>384</td>
<td>abort</td>
<td>512</td>
</tr>
<tr>
<td>Opteron 240</td>
<td>1024</td>
<td>1024</td>
<td>768</td>
<td>abort</td>
<td>1024</td>
</tr>
<tr>
<td>UltraSPARC IIIi</td>
<td>512</td>
<td>os</td>
<td>1024</td>
<td>abort</td>
<td>512</td>
</tr>
<tr>
<td>R12000</td>
<td>512</td>
<td>os</td>
<td>2048</td>
<td>2048</td>
<td>build</td>
</tr>
<tr>
<td>Power 3</td>
<td>512</td>
<td>os</td>
<td>6144</td>
<td>6144</td>
<td>6144</td>
</tr>
<tr>
<td><strong>L2 B (bytes)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pentium 4</td>
<td>128</td>
<td>128</td>
<td>128</td>
<td>128</td>
<td>abort</td>
</tr>
<tr>
<td>Itanium 2</td>
<td>128</td>
<td>128</td>
<td>128</td>
<td>abort</td>
<td>128</td>
</tr>
<tr>
<td>Athlon MP</td>
<td>128</td>
<td>128</td>
<td>128</td>
<td>empty</td>
<td>abort</td>
</tr>
<tr>
<td>Opteron 240</td>
<td>128</td>
<td>128</td>
<td>128</td>
<td>empty</td>
<td>abort</td>
</tr>
<tr>
<td>UltraSPARC IIIi</td>
<td>128</td>
<td>os</td>
<td>128</td>
<td>128</td>
<td>build</td>
</tr>
<tr>
<td>R12000</td>
<td>128</td>
<td>os</td>
<td>128</td>
<td>128</td>
<td>empty</td>
</tr>
<tr>
<td>Power 3</td>
<td>128</td>
<td>os</td>
<td>128</td>
<td>128</td>
<td>empty</td>
</tr>
<tr>
<td><strong>L2 A (count)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pentium 4</td>
<td>7</td>
<td>21</td>
<td>18</td>
<td>20</td>
<td>abort</td>
</tr>
<tr>
<td>Itanium 2</td>
<td>7</td>
<td>6</td>
<td>4</td>
<td>abort</td>
<td>6</td>
</tr>
<tr>
<td>Athlon MP</td>
<td>7</td>
<td>36</td>
<td>18</td>
<td>abort</td>
<td>3</td>
</tr>
<tr>
<td>Opteron 240</td>
<td>7</td>
<td>23</td>
<td>13</td>
<td>abort</td>
<td>25</td>
</tr>
<tr>
<td>UltraSPARC IIIi</td>
<td>7</td>
<td>os</td>
<td>n/a</td>
<td>n/a</td>
<td>abort</td>
</tr>
<tr>
<td>R12000</td>
<td>7</td>
<td>os</td>
<td>n/a</td>
<td>n/a</td>
<td>abort</td>
</tr>
<tr>
<td>Power 3</td>
<td>7</td>
<td>os</td>
<td>n/a</td>
<td>n/a</td>
<td>abort</td>
</tr>
<tr>
<td><strong>L2 l (cycles)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pentium 4</td>
<td>7</td>
<td>381</td>
<td>372</td>
<td>368</td>
<td>abort</td>
</tr>
<tr>
<td>Itanium 2</td>
<td>7</td>
<td>298</td>
<td>281</td>
<td>abort</td>
<td>empty</td>
</tr>
<tr>
<td>Athlon MP</td>
<td>7</td>
<td>401</td>
<td>401</td>
<td>abort</td>
<td>empty</td>
</tr>
<tr>
<td>Opteron 240</td>
<td>7</td>
<td>158</td>
<td>129</td>
<td>abort</td>
<td>empty</td>
</tr>
<tr>
<td>UltraSPARC IIIi</td>
<td>7</td>
<td>os</td>
<td>164</td>
<td>175</td>
<td>abort</td>
</tr>
<tr>
<td>R12000</td>
<td>7</td>
<td>os</td>
<td>111</td>
<td>122</td>
<td>build</td>
</tr>
<tr>
<td>Power 3</td>
<td>7</td>
<td>os</td>
<td>130</td>
<td>161</td>
<td>empty</td>
</tr>
<tr>
<td><strong>Memory l (cycles)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Notes:
- X-Ray: X-Ray tool results.
- Calibrator: Calibrator tool results.
- lmbench: lmbench tool results.
- MOB: Measurement of Bandwidth.
Table A.4: Summary of Itanium 2 $C_3$ parameters

<table>
<thead>
<tr>
<th></th>
<th>Actual</th>
<th>X-Ray</th>
<th>Calibrator</th>
<th>Imbench</th>
<th>MOB</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C$ (KB)</td>
<td>6144</td>
<td>6144</td>
<td>6144</td>
<td>abort</td>
<td>4096</td>
</tr>
<tr>
<td>$B$ (bytes)</td>
<td>128</td>
<td>128</td>
<td>128</td>
<td>abort</td>
<td>empty</td>
</tr>
<tr>
<td>$A$ (count)</td>
<td>24</td>
<td>24</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>$l$ (cycles)</td>
<td>?</td>
<td>19</td>
<td>14</td>
<td>abort</td>
<td>6</td>
</tr>
</tbody>
</table>
BIBLIOGRAPHY


[56] R. Clint Whaley. Personal communication.


