CO-ADAPTING SCIENTIFIC APPLICATIONS AND
ARCHITECTURES TOWARD ENERGY-EFFICIENT HIGH
PERFORMANCE COMPUTING

A Dissertation in
Computer Science and Engineering

by

Konrad Malkowski

© 2008 Konrad Malkowski

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

December 2008
The dissertation of Konrad Malkowski was reviewed and approved* by the following:

Padma Raghavan  
Professor of Computer Science and Engineering  
Dissertation Adviser  
Chair of Committee

Mary Jane Irwin  
Professor of Computer Science and Engineering

Mahmut Taylan Kandemir  
Associate Professor of Computer Science and Engineering  
Director of Graduate Affairs

Jia Li  
Associate Professor of Statistics

Michael P. Perrone  
Manager, Cell Solutions Department, at IBM TJ Watson Research Division  
Special Member

*Signatures are on file in the Graduate School.
Abstract

Many fields in science and engineering rely on computational models for design and discovery. These models are typically derived from either experimental data or from a mathematical description of the physical phenomena. In most cases, the underlying computer representation is sparse, involving matrices with few non-zero entries per row, graphs with just a few edges, or meshes with few neighboring elements per node. Such models are computationally solved at a scale corresponding to the desired level of detail and require large multi-processor systems to assure reasonable execution times. Owing to micro-architectural trends, the power and cooling costs in such multi-processor systems are high, and the efficiency of sparse applications is relatively low compared to theoretical peak performance. Therefore, in this thesis, we consider the challenges of enabling energy-efficient sparse scientific computing.

Our goal is to improve the energy efficiency of sparse scientific applications while maintaining or improving performance. Therefore, we investigate opportunities for energy and performance improvements in order of decreasing scale of size. First, we consider opportunities for improving energy efficiency at the whole multi-processor system scale. We then investigate single node optimizations, starting with single-core and moving towards multi-core power and performance optimizations.

At the multi-processor system scale, we start by considering opportunities for improving workload distribution in parallel applications where computational dependencies and parallelism across the computer nodes are described by a tree. We introduce a new multi-pass workload-to-processor distribution technique and a workload distribution quality measure which we call
critical overload. Our mapping scheme decreases the worst-case critical overload for some cases from approximately 60% to 27%. On average, our technique halves the critical overload for our test suite from nearly 50% for the original mapping to 25%. However, the workload imbalance is not completely eliminated. We therefore consider opportunities for converting the remaining workload imbalances into energy savings by using dynamic voltage and frequency scaling (DVFS) and “just-in-time” computation principles. We introduce three schemes that offer up to a 21% reduction in energy consumption on average across our test suite without degrading application performance.

At the scale of a single node of a multi-processor system, we consider power and performance optimizations for single-core and multi-core architectures. We begin by investigating interactions between hardware features and sparse applications properties on single-core processors. We show that energy savings as high as 64% are possible (without degradations in application performance) when the right combination of hardware, code optimizations, and data structures are used. Next, we propose new hardware features aimed at optimizing performance and energy consumption of sparse scientific applications on single-core processors. We introduce a load miss predictor and a phase-aware adaptive hardware selection control. Combined with DVFS, our load miss predictor reduces system power by 7.3% and energy by 17.3% while maintaining an 8.7% improvement in execution time. Our phase-aware, adaptive hardware selection controller reduces the power consumption of the test applications by as much as 39% and energy consumption by as much as 37% without negatively impacting the performance of applications.

In the multi-core, single node design space, we consider the problem of energy-efficient data lookup in shared non-uniform cache access architecture (NUCA) L2 caches. We propose
a novel tree-based cache architecture as an alternative to static placement NUCA (S-NUCA), and dynamic placement, broadcast-based NUCA (D-NUCA) architectures. In the best instance, in the single-program scenario, our tree-based cache architecture, T-NUCA, outperforms the D-NUCA by 27%, at a 35% overhead in number of transistors used, while reducing the network link use by 48%, energy consumption by 93%, and EDP by 95%. Relative to S-NUCA, in the best instance, our T-NUCA reduces the execution time by 38%, increases the link use by 161% and energy consumption by 32% due to directory overheads, and reduces the EDP by 19%.

We conclude this thesis with a discussion of our results, associated open problems and future research directions.
Table of Contents

List of Tables ............................................................. xi

List of Figures ............................................................ xii

Acknowledgments ......................................................... xviii

Chapter 1. Introduction .................................................. 1

Chapter 2. Computer Architecture Trends ............................ 6
  2.1 Energy Impacts on Peta-Scale Computing ....................... 6
  2.2 Energy Impacts on Processor Design ............................. 9
  2.3 Transistor Scaling - Performance and Power Impacts .......... 11
  2.4 Wire Delay, Switching Speed and Interconnect delays .......... 14
  2.5 Chapter Summary .................................................. 15

Chapter 3. Scientific Workloads and Benchmarks .................. 17
  3.1 Compute Kernels .................................................. 17
    3.1.1 Sparse Matrix-Vector Multiplication ...................... 18
    3.1.2 Equake - Application Specific Sparse Matrix-Vector Multiplication 20
    3.1.3 Conjugate Gradient ......................................... 21
    3.1.4 Preconditioned Conjugate Gradient Solver ................ 22
    3.1.5 Multi-grid ................................................ 25
    3.1.6 Sparse Cholesky Matrix Factorization ..................... 27
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.2</td>
<td>Empirical Results</td>
<td>61</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Performance and Energy Metrics: Profiles and Summary</td>
<td>63</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Relative Incremental Improvements (RII): Measuring Incremental Impact per Feature Addition</td>
<td>66</td>
</tr>
<tr>
<td>6.2.3</td>
<td>Results for Equake-A and Equake-AT</td>
<td>69</td>
</tr>
<tr>
<td>6.3</td>
<td>Chapter Summary</td>
<td>71</td>
</tr>
<tr>
<td>7</td>
<td>Memory Subsystem Impacts on the Energy Efficiency of Conjugate Gradient Algorithm</td>
<td>73</td>
</tr>
<tr>
<td>7.1</td>
<td>Modeling Performance and Power Characteristics</td>
<td>75</td>
</tr>
<tr>
<td>7.1.1</td>
<td>Base Architecture</td>
<td>76</td>
</tr>
<tr>
<td>7.1.2</td>
<td>Memory Subsystem Optimizations</td>
<td>78</td>
</tr>
<tr>
<td>7.1.3</td>
<td>Emulating Low-Power Modes of Caches and Processors</td>
<td>79</td>
</tr>
<tr>
<td>7.2</td>
<td>Empirical Results</td>
<td>80</td>
</tr>
<tr>
<td>7.2.1</td>
<td>Power and Performance Tradeoffs Using DVFS and Low Power Modes of Caches</td>
<td>81</td>
</tr>
<tr>
<td>7.2.2</td>
<td>Impact of L2-Cache and Memory Prefetchers</td>
<td>83</td>
</tr>
<tr>
<td>7.2.3</td>
<td>Impact of Sparse Matrix Properties</td>
<td>85</td>
</tr>
<tr>
<td>7.2.4</td>
<td>Improving Performance and Power</td>
<td>89</td>
</tr>
<tr>
<td>7.3</td>
<td>Chapter Summary</td>
<td>91</td>
</tr>
<tr>
<td>8</td>
<td>Hiding Cache Miss Penalties with the Load Miss Predictor</td>
<td>92</td>
</tr>
<tr>
<td>8.1</td>
<td>Sparse Scientific Computing Applications</td>
<td>93</td>
</tr>
<tr>
<td>8.2</td>
<td>Designing a LMP</td>
<td>93</td>
</tr>
</tbody>
</table>
List of Tables

2.1 Distance travelled in one cycle at speed of light. ................................. 15

3.1 Sparse matrices used in evaluating effectiveness of workload balancing schemes, and opportunities for converting workload imbalance into energy savings. .... 30

3.2 Sparse matrices used in evaluating architecture energy and performance trade-offs for sparse applications. ................................................................. 31

4.1 Description of test matrices, relative critical loads (RCL), and numeric factor-ization times using the original proportional scheme and the multi-pass scheme. 42

5.1 Voltage, frequency and power levels used to explain the behavior of tree DVFS scaling algorithms. ................................................................. 51

5.2 Voltage, frequency, and power values used in our experiments evaluating tree DVFS scaling algorithms when 5 processor voltage levels are available. .... 56

8.1 Wattch configuration for base system representative of a modern 4-way issue out-of-order processor. ................................................................. 98

10.1 L2 cache configuration parameters from CACTI6.0. ......................... 147

10.2 Summary of cache configuration parameters for the L1 and L2 caches for the S-NUCA, D-NUCA and T-NUCA configurations. .................. 147
List of Figures

2.1 Performance of the fastest machines on Top500 list with corresponding power consumption. .......................................................... 8

3.1 The sparse matrix storage formats in an unoptimized version of SMV (SMV-U) and the optimized version with 2x1 blocking (SMV-O). ............. 19

3.2 Mapping of sparse matrix, and source and destination vector elements to main memory for Equake-A and Equake-AT .............................................. 21

3.3 The Conjugate Gradient (CG) scheme. ............................................. 24

3.4 The Preconditioned Conjugate Gradient (PCG) scheme. ...................... 24

3.5 A generic V-cycle Multi-grid algorithm ............................................. 26

3.6 The structure of sparse matrix $A$ during three steps of sparse matrix factorization. 29

4.1 Critical overload for the original proportional heuristic, and the Robin Hood and multi-pass schemes for augustus7 on 8 – 64 processors. .............. 44

4.2 Quality of assignments for worst load instances and the best observed improvement instances. ................................................................. 46

4.3 Cumulative critical overload over all matrices for each processor, from assignments using the original proportional scheme and from our multi-pass scheme. 46

4.4 Cumulative critical overload for 16 – 64 processors for each matrix, from assignments using the original proportional scheme and from our multi-pass scheme. 48
5.1 Supernodal tree from DSCPACK extended with performance and energy consumption information. 51

5.2 Algorithm I - DVFS tree scaling example. 52

5.3 Two examples where the results of Algorithm I can be improved further. 54

5.4 Algorithm VS2 - DVFS tree scaling example. 54

5.5 Algorithm VS3 - DVFS tree scaling example. 55

5.6 Percentage energy savings resulting from applying: Algorithm I, VS2 and VS3. 56

6.1 Execution time in seconds for SMV-U and SMV-O when features are added in the order 'B+W+MO+MP+LP'. 65

6.2 Energy consumption (in Joules) for SMV-U and SMV-O when features are added in the order 'B+W+MO+MP+LP'. 65

6.3 Relative improvements (RI) in execution time and energy for SMV-U and SMV-O. 66

6.4 Relative incremental improvements (RII) in execution time and energy for SMV-U and SMV-O. 68

6.5 Execution time and energy for equake on the base configuration 'B' and with all optimizations 'all'. Equake-A indicates equake with its original SMV and Equake-AT is equake with a slightly tuned SMV. 70

6.6 Relative incremental improvements (RII) in execution time for Equake-A with original SMV, and Equake-AT with a slightly tuned SMV. 71

7.1 Conceptual view of the reference processor architecture for analyzing impacts of memory subsystem and software optimizations. 76

7.2 A memory prefetcher implementation. 79
7.3 Execution time and average system power for CG-U and CG-O for bcsstk31 with RCM. ................................................................. 82

7.4 Relative Time, Power and Energy of unoptimized CG (CG-U) and optimized CG (CG-O) algorithms solving the bcsstk31 matrix. .................................................. 84

7.5 Relative execution time and relative power for CG-O on bcsstk31 with RCM, CG-O bcsstk31 with a random ordering, CG-U on bcsstk31 with a random ordering and the NAS benchmark CG (CG NAS) on a matrix with dimensions similar to bcsstk31. .......................................................... 86

7.6 Opportunities for power and performance optimizations for CG algorithm. .... 89

8.1 The Load Miss Predictor design. ................................................................. 94

8.2 Impact of LMP bypass cache replacement policy on SPEC2000 performance with “Ideal” LMP. ................................................................. 101

8.3 Impact of “ideal”, “bimodal” and “2 level 8 bit (L2H8)” predictors on execution time of SPEC 2000 benchmark suite at 2000MHz. .................................................. 102

8.4 Impact of the Regular L1 and Bypass L1 data cache sizes on execution of spec2000 benchmarks. ................................................................. 103

8.5 Impact of Regular L1 and Bypass L1 cache size on performance and power of sparse benchmarks. ................................................................. 105

8.6 Impact of our final LMP configuration on execution time, average power and energy consumption of SPEC 2000 benchmark suite at 2000MHz. ................. 107

8.7 Impact of LMP on execution time, power, energy and on average load latency for SMV-O, SMV-U, MG, and CG. ................................................................. 108
8.8 Impact of LMP and DVFS on execution time, power and energy of SMV-U, SMV-O, CG and MG sparse benchmarks. ........................................... 109

8.9 Relative average time, power and energy, using DVFS with and without LMP for 1300MHz to 2000MHz. ........................................... 110

9.1 Impact of PCG phases on hardware metrics. ........................................... 116

9.2 Finite state machine (fsm) implemented by our adaptive hardware. .......... 119

9.3 High level view of the proposed hardware implementation of the switching heuristic. ........................................... 121

9.4 Impacts of adaptive hardware and automatic phase detection on hardware metrics. 124

9.5 Response of adaptive hardware to changes in application phases. .......... 127

9.6 Our adaptive hardware and switching heuristic have no significant impact on performance of test sparse applications. ........................................... 129

9.7 Comparison of our adaptive hardware technique and a performance optimized processor at 1000MHz with prefetchers turned on. ........................................... 130

10.1 Conceptual design of a CMP with D-NUCA. ........................................... 139

10.2 **Left:** Conceptual 2D T-NUCA cache design showing the additional tree directory network laid out in the chip. **Right:** Conceptual 3D T-NUCA design showing how directories are laid out and connected to each other creating a tree. 140

10.3 Single-program scenario - Average hop count and number of links used per access for S-NUCA (**Left**), D-NUCA (**middle**), and T-NUCA (**right**). .......... 150
10.4 Single-program scenario - Time (Left), energy (middle) and EDP (right) results for ART, EQUAKE, LU and CHOLESKY benchmarks on S-NUCA, D-NUCA, and T-NUCA. ................................. 151

10.5 T-NUCA - Single-program environment - Left: Distribution of accesses satisfied at each level of the tree directory for the whole CMP. Right: Distribution of accesses satisfied at each level of the tree directory for the busiest core. ................................. 153

10.6 T-NUCA-YX - Single-program environment - Time, energy and EDP results for ART, EQUAKE, LU and CHOLESKY benchmarks run on T-NUCA-YX, relative to S-NUCA (left) and relative to T-NUCA (right). ................................................. 154

10.7 T-NUCA-YX - Single-program environment - Left: Distribution of accesses satisfied at each level of the tree directory for the whole CMP. Right: Distribution of accesses satisfied at each level of the tree directory for the busiest core. ................................. 155

10.8 Multi-program scenario. Left: Average number of links utilized per access, and the average distance in hops of where data is found on a L2 read. Middle: Distribution of accesses satisfied at each level of the tree directory for the whole CMP and the busiest core. Right: Execution time, energy and EDP results with S-NUCA, D-NUCA and T-NUCA. ................................. 157

10.9 Multi-program scenario. Per component energy contribution for S-NUCA, D-NUCA, and T-NUCA. ................................. 158

10.10 Multi-program scenario, thread-to-core mapping impacts. Left: Execution time, Middle: Energy consumption, and Right: EDP relative to S-NUCA with row-wise mapping. ................................. 159
10.11 Multi-program scenario, thread-to-core mapping impacts. Distributions of directory hits among T-NUCA tree levels for various thread-to-core mappings. 160

A.1 Idealized gate design. 176
A.2 A more realistic model of gate design. 177
A.3 Comparison between power and energy consumed by computer system. 178
A.4 Comparison between time, energy and energy delay product of a computer system. 179
Acknowledgments

I dedicate this dissertation to my family. I am thankful to my parents for supporting me through college in the United Stated of America, and thus giving me the opportunity to make my dreams come true. I am grateful for their encouragement and instilling into me the importance of working hard, being a good person, always striving to improve myself, and learning for myself. I am eternally grateful to my wife Shannon for her support and love during my work on this dissertation, and for sharing a cheerful smile and a spunky attitude with me every day.

I am most grateful and indebted to my thesis advisor, Professor Padma Raghavan, for the large doses of patience, guidance, and strong encouragement she has shown me during my time here at Penn State. I want to thank my committee members, Professor Mary Jane Irwin, Professor Mahmut Kandemir, Professor Jia Li, and Doctor Michael Perrone for their insightful commentary and guidance on my work.

Finally, special thanks to our cat Spooky who was always eager to keep me company and nap with me at odd hours of the day and night.
Chapter 1

Introduction

Many fields in science and engineering, such as astrophysics [61], climate modeling [122], fluid flow [39], aerodynamics [74], materials processing [134], and epidemiology [82], rely on computational models for design and discovery. These models are typically derived from large data sets or are generated from formal mathematical descriptions. The underlying computer representations are usually sparse [81], involving large (often unstructured) matrices with relatively few nonzero elements, sparse graphs, or adaptively refined meshes that discretize two- or three-dimensional domains to represent regions of interest with a greater density of points. These models are typically solved at a level of detail guaranteeing appropriate solution accuracy and generally require multi-processor systems for reasonable execution times. The models usually have high computational and power requirements and low computational efficiency due to their sparsity.

Until recently, the scale of the models was limited by the capability of the multi-processor systems used to solve them. Presently, the scale of computational models in large computer centers is limited by multi-processor energy consumption and is due to architectural trends, problems with heat dissipation, and limits of the air cooling technology [120, 121, 130]. Owing to Moore’s Law [150] and advances in manufacturing technology and micro-architecture design, multi-processor system performance has continued to grow with each new processor
generation at the cost of increased energy consumption and higher heat density. Now, with machines consisting of thousands of processors and power consumption in the Megawatt (MW) range (e.g., BlueGene/L, 130 thousand processors, 1.5 Megawatts [179, 184]; Earth Simulator, 5120 processors, 7 Megawatts [178]), many computing sites are faced with expensive infrastructure upgrades to accommodate the expected power capacity growth [92, 167]. These challenges and costs have resulted in significant research effort towards energy-efficient scientific computing [30, 64, 88, 91, 110, 116, 140].

In this thesis, we address the challenge of enabling energy-efficient sparse scientific computing by characterizing the interactions between applications and architectural attributes. Our goal is to improve energy efficiency while maintaining or improving application performance. We approach the problem in order of decreasing scale of size. First, we consider optimizations at the entire multi-processor system scale where we expect to achieve greatest performance and energy improvements. We then move towards single node power and performance optimizations for both single-core and multi-core architectures.

In Chapters 2 and 3, we discuss the background material. Chapter 2 explains the trends in computer architecture that have caused energy efficiency to become a concern in large-scale computing. Chapter 3 introduces the sparse compute kernels representative of real-world sparse scientific applications, presents the sparse matrices used as inputs for these kernels, and introduces standard benchmark suites.

In Chapter 4, we investigate the problem of balanced computational workload distribution in large multi-processor systems. Balanced workload distribution and data partitioning has a direct relationship to the maximum achievable speedup on a multi-processor computer. In addition, achieving the maximum speedup results in a reduction of the overall energy costs
associated with the computation. For example, for a parallel application running on the Earth Simulator, which consumes 7 MW of power on average [178], we can expect a 20% reduction in energy consumption when a better workload distribution reduces execution time by 20%. In Chapter 4, we focus on scientific applications with computation dependencies represented by a tree. We propose a new multi-pass workload distribution scheme and a critical overload metric for quantifying the quality of workload distribution. In our test suite, our scheme decreases the worst case critical overload from approximately 60% to 27% and halves the critical overload from nearly 50% for the original mapping to 25% on average.

In Chapter 5, we look for opportunities to reduce the energy consumption in large multi-processor computer systems. We start by considering the problem of workload distribution in tree-based computations. We observe that for certain types of problems, it is impossible to generate a fully-balanced workload distribution. For such problems, certain processors complete their local computations ahead of others and sit idle, waiting for the processors with more work to finish, before moving onto the next computation stage. Therefore, we investigate opportunities for converting such workload imbalances into energy savings by using information stored in the computational dependency tree with dynamic voltage and frequency scaling (DVFS). We introduce three energy reduction schemes that, in our test suite, offer average energy savings of up to 21% without reducing the application performance.

In Chapters 6, 7, 8 and 9, we focus on single node optimizations for power-aware high performance sparse scientific computing. While sparse codes enable the computational costs of modeling applications to scale linearly as the problem size increases [84, 112], their performance depends largely on the memory subsystem design of the computer. Unlike dense codes [124],
which inherently have a large number of floating point operations per data access, the performance of sparse codes is typically limited by data access operations [95]. Consequently, sparse codes use only a small fraction of the computing power of modern microprocessors despite sophisticated attempts at performance tuning [187, 191]. This presents a unique opportunity for architectural optimizations as power-aware microprocessor, memory, and network designs become essential for scaling to future systems [65].

In Chapters 6 and 7, we investigate interactions between hardware features and sparse software properties. We show that with proper support (i.e., data structures, matrix properties, software tuning, and hardware features present in the system), energy consumption can be reduced by up to 64%. At the same time, the application performance relative to the unoptimized system can be maintained or exceeded.

In Chapter 8, we introduce the load miss predictor, a cache subsystem enhancement aimed at reducing the memory access latency for sparse codes. Our results show that the load miss predictor improves the performance of sparse scientific applications by up to 16.7% without negatively impacting the performance of other types of applications. Combined with DVFS, our load miss predictor on average reduces system power by 7.3% and energy by 17.3% while maintaining an 8.7% improvement in execution time.

In Chapter 9, we introduce a phase-aware adaptive hardware selection scheme where the prefetcher activity and processor performance are automatically adapted to application needs. Our scheme effectively adapts to application requirements. For codes with significantly long memory bound phases, energy consumption is reduced by up to 37% without degrading application performance.
In Chapter 10, we consider the problem of energy-efficient data lookup in a shared, non-uniform cache access architecture (NUCA) L2 cache in multi-core systems. We propose a novel, tree-based directory NUCA architecture as an alternative design to the S-NUCA and D-NUCA architectures. We evaluate the performance and energy efficiency of T-NUCA against S-NUCA and D-NUCA in two execution scenarios: (i) a single-program scenario, where one application is assigned all cores on the chip, and (ii) a multi-program scenario, where eight applications are assigned to the chip at the same time. In the single-program scenario, our proposed T-NUCA architecture outperforms the D-NUCA by 27%, at a 35% overhead in number of transistors used, while reducing the network link use by 48%, energy consumption by 93%, and EDP by 95%. Relative to S-NUCA, our T-NUCA reduces the execution time by 38%, increases the link use by 161% and energy consumption by 32% due to directory overheads, and reduces EDP by 19%.

In the multi-program scenario, our T-NUCA design reduces the execution time by 17%, energy by 93% and EDP by 94% over D-NUCA. Relative to S-NUCA, our T-NUCA design reduces the execution time by 71%, energy by 58% and EDP by 88%.

We conclude this thesis with Chapter 11 which contains a discussion of the results, open problems, and possible extensions of our research. The last chapter is followed by an Appendix which contains the definitions of the power and performance metrics used in this dissertation and a discussion of Moore’s Law. We encourage the reader to refer to the Appendix whenever they are unsure of the meaning of a particular term. This thesis is based on earlier publications which are referenced and listed in bibliography located at the end of this document.
Chapter 2

Computer Architecture Trends

Due to computer architecture trends, the scaling and performance of computational models is limited by large-scale computer system energy requirements. Therefore, in this chapter, we discuss these architectural power and performance trends. In Section 2.1, we examine the impacts of high energy consumption on peta-scale computing. In Section 2.2, we discuss impacts of high energy consumption on processor design. In Section 2.3, we discuss how transistor scaling affects energy consumption and performance of processors. In Section 2.4, we discuss wire delay and transistor switching speed limits to the performance of single core processors. We conclude this chapter in Section 2.5.

2.1 Energy Impacts on Peta-Scale Computing

In Chapter 1, we identified heat dissipation, cooling costs and power consumption as limiters of performance in large computer installations. It is estimated that for each Watt (W) of electricity required to power the computers, an additional Watt is required for the cooling system [90, 167]. In many instances these costs can exceed the costs of initial system procurement [87, 90, 177]. Furthermore, the large computer centers face the prospect of expensive infrastructure upgrades in order to meet the power requirements of new machines [71, 87, 90, 92, 167, 176, 193].
To project energy requirements of a peta-scale class computer, consider building such a computer out of current (year 2007) top of the line, off the shelf components. Building this system using a high performance, high power processor such as dual core AMD Windsor AMD 64-X2 5200 [5], running at 2.6GHz, will require a minimum of 48077 processors to attain peak theoretical performance of 1 Peta-Flop ($10^{12}$ floating point operations per second). With each processor rated at 89 Watts, the system power requirements for the processors alone are around 4.3 Megawatts (MW). Inclusion of memory (3.4W per 2GB of memory per node [99]), hard drive (5-16W per node [174, 175]), and network interconnect (5W per node [147]) power contribution into our projections increases the overall system power consumption to 4.9–5.5MW (observe that the processor power consumption is a significant fraction of the total power). Cooling requirements double the power consumption to approximately 9.8MW.

In addition to high energy costs associated with running such a system, there are additional costs incurred due to high power density. Higher temperatures lead to higher hardware failure rates [55, 89] and increase the costs of maintenance and overheads related to ensuring computation and system reliability such as checkpointing. Thus, in order to reduce these costs, maintaining energy efficiency is now as important to developers of scientific applications as maintaining iso-efficiency.
Fig. 2.1. Performance of the fastest machines on Top500 list with corresponding power consumption.

This shift of focus, to a more power aware scientific computing, is demonstrated by the increase of research in this field [30, 64, 88, 91, 110, 116, 140]. Additionally, a series of power-aware clusters, such as GreenDestiny [192] and BlueGene/L [184], were introduced over past few years. These systems step away from the traditional design paradigm of using a few fast, power hungry processors and instead use many low power processors and components to balance the system. For example, Earth Simulator, an older system developed by NEC in Japan, is rated to consume around 7 MW of electricity at the theoretical peak performance of 40.96 Terra-FLOPS [178], while the IBM BlueGene/L system requires “only” 1.2-1.5 MW at
the theoretical peak performance of 367 Terra-FLOPS [179]. Figure 2.1 demonstrates how the power requirements of the fastest computer of the Top500 list [51] evolved from year 1996 to 2007. The IBM BlueGene/L marks a significant drop in the overall power consumption due to the change in the design philosophy. However, a projection into the peta-scale and beyond regime demonstrates that power requirements will continue to grow. This projection suggests that power-consumption is a function of the computational requirements and that as the number of processors required to obtain a solution at higher resolutions increases, so will the overall power requirements of the systems leading to even higher costs.

2.2 Energy Impacts on Processor Design

In Section 2.1, we discussed the energy trends in large-scale computing, and their impacts on computer centers and peta-scale system design. We showed that power consumption of the peta-scale clusters will be largely dominated by the average power consumption of the processors used. In this section, we show how the energy consumption and heat dissipation limit the performance of processors and influence architecture design changes. We also discuss how processor manufacturers are working towards limiting the processor power consumption by moving to the multi-core design. We begin with the analysis of single-core processor limits, then we discuss multi-core processor design.

Consider the single-core processor design. Until recently, performance gains were achieved by increasing the pipeline lengths, adding more speculation to the execution pipeline and increasing processor frequency. The focus was on utilizing instruction level parallelism, leading to a situation where a significant portion of the processor area is dedicated to the support of instruction level parallelism and speculation, with only a small portion doing performing actual
computations [182, 183]. All of these trends led to high processor power consumption and heat dissipation.

We observe, that with air cooled heat dissipation technology it is no longer economical to pursue performance by adding hardware features that result in higher heat density, such as increased pipeline depths or higher frequencies. The maximum efficiency of air cooled heat sinks is $75 \frac{W}{cm^2}$ [120,121,130]. At the same time many modern processors can easily produce in excess of 125W of power (which is the limit of air cooled technology). Additionally, there are certain classes of codes which do not benefit significantly from the deep pipelines, for example, codes with large number of branches and operating system codes [149]. In general, there exists a cross-over point beyond which further increases in single-core processor operational frequency cannot be justified by the amount of electrical energy required to power the processor [165].

To further improve processor performance and avoid complex and expensive liquid cooling systems, manufacturers have at first embraced the symmetric multi-threading (SMT) [126] and then the multi-core design paradigm [21,117,149]. In the SMT processor design multiple hardware threads are sharing the processor resources in order to use the processor more efficiently. This design relies on the fact that, in deeply pipelined architectures branches, cache misses and data dependencies introduce pipeline bubbles. In a single threaded architecture these pipeline bubbles are wasted and decrease performance. Additionally, due to data dependencies not all functional units are always fully utilized in a single threaded machine. SMT design allows for efficient use of these underutilized resources.

In the multi-core processor design, multiple cores are located on a processor die instead of one large and power-hungry superscalar processor. This shift in processor design enables peak processor performance to continue following the Moore’s Law [150] curve, as the advances in
miniaturization allow chip manufacturers to place more cores on a chip within a constant power
envelope [165]. Furthermore, the focus is now on thread-level parallelism instead of instruction-
level parallelism. This presents a new set of challenges for both programmers and architecture
designers as we move towards systems with 80 and more cores on chip [1]. Some of the problems
are efficient data sharing, efficient data lookup among cores, efficient programming techniques,
OS scheduling and others.

It is presently an open question what type of multi-core design will succeed as there
are varying competing alternatives: homogenous vs. heterogeneous systems, simple cores vs.
complex cores [13,55,92,171]. Each design choice offers unique advantages as well as disadvan-
tages. Homogenous cores are arguably easier to program and can be used efficiently to increase
performance of applications with significant data and task parallelism. On the other hand, the
heterogeneous cores increase programming complexity at the benefit of providing higher effi-
ciency and performance for certain types of codes. Similarly, more simple cores can be placed
on the single processor die, promising significant performance gains and increased energy effi-
ciency provided that the parallelism they offer can be fully utilized by software. Complex cores,
on the other hand, benefit applications which have very little large granularity parallelism.

2.3 Transistor Scaling - Performance and Power Impacts

In this section, we discuss the trends in transistor count scaling in modern processors,
and their impacts on power and performance. We begin by investigating the forces behind the
transistor count scaling. The doubling of the number of transistors per processor, and corre-
sponding processor performance increases in accordance with Moore’s Law, are in part due to
the ingenuity of engineers and designers, and in part due to technology scaling, resulting in
smaller transistor feature sizes. Borkar et al. [19] stipulates that with each improvement in the lithography process the feature size decreases by a factor of 0.7 in one dimension. This scaling effectively allows designers to place a fixed number of transistors in half the area required by the previous technology, and allows for a corresponding increase in the processor frequency due to shorter distances on the chip. Alternatively, power consumption can be reduced if the frequency is kept constant across process technologies. Optionally, the chip area can remain constant, the number of transistors doubled, and the new transistors can be used to implement additional features at a higher frequency and power consumption.

Until recently, most processor designers chose to use the improvements in the lithography process and feature miniaturization to increase the performance of single-core processors. This was done by increasing the number of transistors inside of the core, adding additional hardware features and increasing processor frequency [171]. The cost was higher heat density and increased system power consumption as the result of higher dynamic power consumption and static leakage power.

- The dynamic power consumption is associated with the number of transistors on the chip that are switching at any given time (represented by $\alpha$). It is proportional to the frequency $f$ of the processor and supply voltage $V_{dd}$. It is approximated by $P_d = \alpha \times V_{dd}^2 \times f$.

- The leakage power is the result of the electrical properties of the gate design [69], and can be estimated by $P_l = \beta \times I_l \times V_{dd}$, where $\beta$ is the number of powered up transistors on chip, $I_l$ is the leakage current, and $V_{dd}$ is the voltage difference between substrate and gate electrode.
• The leakage current consists of sub-threshold leakage (source to drain leakage) and gate leakage (gate to substrate leakage).

  – The gate leakage is the result of reduction in the Complementary Metal Oxide Semiconductor (CMOS) process size. As the transistors become smaller the thickness of silicon dioxide between the gate electrode and the substrate decreases. While ideally it is depleted of charge and serves as an insulator between gate and substrate when the transistor is turned off, presently it is only approximately five atomic layers thick. This small size allows for electric charges to travel between the gate and the substrate [69].

  – The sub-threshold current is related to the source-to-drain leakage and is a function of the channel length. As the feature sizes decrease from one process generation to the next, the channel length shrinks. The shorter gate length results in faster switching gates. However the gate control becomes harder. The voltage on the drain reduces the energy barrier in the channel, resulting in lower threshold voltage, and enables the charges to flow through the transistor even if no actual voltage is applied to the gate electrode [69]. Presently, the sub-threshold current increases fivefold with every process generation.

It is estimated that the power leakage is around 30% of total chip consumption in 65nm technology [107, 172], and that with each successive smaller technology point it will continue to increase [19, 115]. To reduce the amount of leakage and improve the performance of transistors, new materials and manufacturing processes are researched. For example, the use of High-K
Dielectric materials enabled IBM, Intel and AMD [123] to reduce the leakage by 80%, while improving performance and reducing overall power consumption. To further improve electrical properties of transistors, new transistor designs, such as FinFET’s [69], are currently under development.

2.4 Wire Delay, Switching Speed and Interconnect delays

The advances in processor manufacturing technology enable us to place more transistors per fixed area of processor die. However, as the switching elements (gates) and local interconnect wires within the cells of the chip scale down with the advances in fabrication technology, the global wires that need to transmit the data across the chip do not [6, 18, 92]. Thus, as the frequency of the processor increases and the underlying local structures shrink and switch faster, the distance in global wires that can be traveled by electrical signals within once clock cycle decreases. This distance is determined by signal switching speed, capacitive and resistive properties of the wire, width and length of the wire [152], and electro-magenetic permittivity of the material surrounding the wires [23] (Refer to the Appendix for more details on signal propagation speed in wires).

In addition to wire delay, the performance of single processor is limited by the finite amount of main memory that can be used to solve problems, and thermal and power limits which control the processor’s maximum operational frequency. There exists a cross-over point beyond which further increases in single-core processor operational frequency cannot be justified by the amount of electrical energy required to power the processor [165]. To overcome these memory

---

<sup>1</sup>Dielectric materials are insulators, frequently used in capacitors and transistors. $K$ represents the dielectric constant, which is a material property [57, 58].
and performance limitations multiple computer nodes are used to solve large problems [119]. However, when using multiple processors, the speed of light becomes an additional constraint on available performance, as it determines the minimum latency required for signals to propagate over the interconnect. Table 2.1 lists the distance the light can traverse in one processor cycle. From that table we can see that while light travels fast (299,792, 458\frac{m}{s} or approximately 30\frac{cm}{ns}), delays resulting from transmitting data across the network interconnect, or just system bus in a symmetric multi processor, can amount to hundreds or thousands of wasted computational cycles [167]. The resistance, capacitance, and inductance of wires further limit the rate at which the data is transfered over the wires [57, 58].

<table>
<thead>
<tr>
<th>Frequency (GHz)</th>
<th>Cycle Period (s)</th>
<th>Distance Travelled (m)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5E-10</td>
<td>0.149896229</td>
</tr>
<tr>
<td>3</td>
<td>3.33333E-10</td>
<td>0.099930819</td>
</tr>
<tr>
<td>4</td>
<td>2.5E-10</td>
<td>0.074948115</td>
</tr>
</tbody>
</table>

2.5 Chapter Summary

In this chapter, we discussed trends in the computer architecture that have resulted in energy efficiency becoming a significant concern in large scale scientific computing. We began in Section 2.1, with the discussion of energy impacts on peta-scale class computing. Then, in Section 2.2, we discussed heat dissipation and power consumption in light of single-core and
multi-core architectures. In Sections 2.3 and 2.4, we discussed transistor scaling, and the effects it has on performance, power and wire delay. In the next chapter, we introduce compute kernels, standard benchmark suites and data sets used in this thesis.
Chapter 3

Scientific Workloads and Benchmarks

In the previous chapter, we discussed architectural trends that lead to energy being the scaling limiter for computational models. In this chapter, we introduce the software kernels and benchmarks used in evaluating opportunities for improving performance and energy efficiency of sparse scientific applications. In Section 3.1, we introduce the sparse compute kernels representative of sparse linear solvers used in large-scale scientific applications. In Section 3.2, we present sparse matrices used as input sets for these compute kernels. In Section 3.3, we discuss the standard benchmark suites, such as SPEC or NAS, used in this thesis. We conclude this chapter in Section 3.4.

3.1 Compute Kernels

In this section, we introduce the scientific compute kernels representative of many sparse scientific applications, which we use to evaluate power and performance optimizations and trade-offs between computer hardware and applications. In our experiments, we consider various level of optimizations for our kernels. In Section 3.1.1, we discuss sparse matrix-vector multiplication which is at the core of many sparse iterative solvers. In Section 3.1.2, we discuss equake from the SPEC benchmark suite, which contains an application specific sparse matrix-vector multiplication algorithm. In Section 3.1.3, we discuss the Conjugate Gradient solver. In Section 3.1.4, we discuss Preconditioned Conjugate Gradient, and in Section 3.1.5, we discuss the Multi-grid
solver. Finally, in Section 3.1.6, we discuss DSCPACK [169] - a sparse direct solver, based on Cholesky factorization.

### 3.1.1 Sparse Matrix-Vector Multiplication

The sparse matrix-vector (SMV) multiplication kernel is at the heart of many iterative sparse application solvers. The general purpose forms of SMV use standard data structures [173] for storing the sparse matrix $A$, the source vector $x$ and the destination vector $y$. The latter two are stored as simple arrays in contiguous locations in memory. Only the nonzeroes in the matrix and its corresponding indices are explicitly stored using a standard sparse matrix format with a list of subscripts and nonzeroes and a list to index into these two lists for each row.

The standard sparse matrix-vector multiplication routine requires one floating-point multiplication and addition per nonzero element in $A$. Note that in addition to the nonzero element, its indices in the matrix also have to be loaded, thus increasing the number of data accesses per floating point operation. There is potential for re-use with elements of the source vector $x$, but the access pattern on $x$ depends on the sparsity structure of $A$ which can be re-ordered to a ‘band form’ using, for example, a Reverse Cuthill McKee (RCM) scheme [43] to improve locality of access in $x$. Such re-orderings can be used with other techniques like register-blocking and loop-unrolling to further improve the performance [187, 191]; some of these techniques may actually increase floating-point operations while decreasing loads from memory.

SPARSITY [95] is a library consisting of highly optimized matrix-vector multiplication routines. It utilizes register and cache blocking to improve performance, typically after an RCM ordering is applied to the matrix. Figure 3.1 shows how SPARSITY reduces the row and column index overhead while enabling data local accesses on a sample sparse matrix by using a “2x1”
register blocking with loop unrolling. Although there is a slight increase in floating point operations and storage requirements due to the addition of explicit zeroes into the matrix structure, the scheme considerably outperforms the unoptimized kernel [95].

Fig. 3.1. The sparse matrix storage formats in an unoptimized version of SMV (SMV-U) and the optimized version with 2x1 blocking (SMV-O).

In this thesis, we use two sparse matrix-vector multiplication routines from SPARSITY with varying levels of tuning:
• SMV-U, a natural implementation of sparse matrix-vector multiplication or, equivalently, an untuned version of the code in SPARSITY [95].

• SMV-O from SPARSITY [95] with with 2x1 blocking and loop unrolling (SMV-O) which decreases loads at the expense of a slight increase in floating point operations [96]. This level of loop unrolling and register blocking resulted in the best performance on our simulated computer architectures.

3.1.2 Equake - Application Specific Sparse Matrix-Vector Multiplication

In this section, we consider an application-specific sparse matrix-vector multiplication kernel, found in the equake code from SPEC CFP 2000 [41]. This application simulates the propagation of elastic waves in large, highly heterogeneous valleys and more than 90% of its execution time is spent in the custom SMV function. The matrix is stored in an application specific data structure that reflects the relationship of the matrix to the mesh. Furthermore, corresponding portions of the source vector may also not be contiguous in memory. We use Equake-A to denote the original SMV kernel for this application specific format in equake. We make a minor change in this kernel to obtain the tuned version Equake-AT while still using the application specific data structure. The tuning reflects a change in how memory is allocated in the code to ensure a greater degree of contiguous allocations to improve the locality of data accesses. We used Equake-AT in quake and we verified that the simulation results were correct and unchanged after the replacement of the original kernel with its modified version. The original data structure and its mapping to memory in Equake-A and the modified mapping to memory in Equake-AT are shown in Figure 3.2.
Fig. 3.2. Mapping of sparse matrix, and source and destination vector elements to memory in Equake-A and an illustration of tuning to improve contiguous allocation of data elements to increase locality of access in Equake-AT.

3.1.3 Conjugate Gradient

In this section, we describe the Conjugate Gradient (CG) algorithm with an emphasis on how its core computations access the memory subsystem [14, 173]. Figure 3.3 contains an outline of a generic CG algorithm used in many applications. This CG scheme uses standard
data structures [14, 173] for storing the sparse matrix $A$, and vectors $p, q, r$. Only the nonzeros of the sparse matrix $A$, and its corresponding indices are explicitly stored using a standard sparse format. The vectors $p, q, r$ are stored as one-dimensional arrays in contiguous locations in memory.

A single iteration of the CG requires one matrix-vector multiplication, two vector inner products, three vector additions and two floating point divisions. Among these operations, the matrix-vector multiplication dominates the computational cost, and accounts for more than 90% of the overall execution time. Due to the sparse nature of the matrix $A$, the number of floating point operations per access to the main memory is relatively low during the matrix-vector multiplication. Additionally, the access pattern of the elements in the vector $p$ depends on the sparsity structure of $A$. Sparse matrices from typical scientific applications can be ordered to a band form where the nonzeros are clustered around the diagonal using a scheme such as RCM [68]. This improves the locality of access in $p$. Such re-orderings can be used with other techniques like register-blocking and loop-unrolling to further improve the performance [95, 187]; some of these techniques may actually increase the number of floating-point operations. We explore impacts of these techniques by considering two versions of CG with different levels of tuning in the SMV routine as described in Section 3.1.1. The CG version based on SMV-U is named CG-U, and the version based on optimized SMV-O is named CG-O.

3.1.4 Preconditioned Conjugate Gradient Solver

In this section, we introduce the Preconditioned Conjugate Gradients (PCG) algorithm. PCG is frequently used to solve sparse linear systems instead of the CG algorithm due to increased convergence rates [14, 173]. The PCG [95, 191] kernel we use is not part of a benchmark
suite. However, it reflects the state-of-the art in a tuned implementation of the main computation in all members of perhaps the largest and most popular class of sparse solvers, namely the Krylov subspace iterative methods [14].

A PCG sparse solver has two main phases:

- preconditioner construction (PC Const),

- main routine (PCG routine)

Our in-house implementation uses the incomplete Cholesky factorization (ICF) algorithm to construct the preconditioner. There are two main variants of the ICF implementation. They differ in the choice of fill in dropping policy: level-of-fills and drop-threshold. Level-of-fills allows fill in based on the nonzero structure of the original matrix $A$, and drop-thresholds allows fill in based on the magnitude of the elements. Consequently, the memory access pattern of level-of-fills method is more predictable than that of drop-threshold. Our implementation uses a left-looking ICF which has better cache utilization compared to that of a right-looking implementation [9, 145].

Figure 3.4 contains an outline of a generic PCG algorithm used to solve sparse linear systems. Our PCG scheme uses row compressed sparse data structures for storing the sparse matrix $A$ and the preconditioner matrix $M$. Only the non-zeroes of the sparse matrices $A$ and $M$, and their corresponding indices are explicitly stored. The vectors $p, q, r, z$ are stored as one-dimensional arrays in contiguous locations in memory.

A single iteration of the PCG requires one sparse linear solution (preconditioner application), one sparse matrix-vector multiplication, two vector inner products, three vector additions and several floating point operations. Among these operations, a sparse linear solution and the
CG(A,b)
% A - input matrix, b - right hand side
x₀ is initial guess;
\[ r = b - Ax₀; \quad \rho = r^Tr; \quad p = r \]
for i=1,2,... do
    \[ q = Ap \]
    \[ \alpha = \rho/(p^Tq) \]
    \[ x_i = x_{i-1} + \alpha p \]
    \[ \tilde{r} = r - \alpha q \]
    if \( ||\tilde{r}||/||b|| \) is small enough then stop
    \[ \tilde{\rho} = r^T\tilde{r} \]
    \[ \beta = \tilde{\rho}/\rho \]
    \[ p = r + \beta p \]
    \[ r = \tilde{r}; \quad \rho = \tilde{\rho} \]
end for

Fig. 3.3. The Conjugate Gradient (CG) scheme.

PCG(A,M,b)
% A - input matrix,
% M - preconditioned matrix,
% b - right hand side
x₀ is initial guess;
\[ r = b - Ax₀; \]
Solve \( Mz = r; \quad \rho = r^Tz; \quad p = z \)
for i=1,2,... do
    \[ q = Ap \]
    \[ \alpha = \rho/(p^Tq) \]
    \[ x_i = x_{i-1} + \alpha p \]
    \[ \tilde{r} = r - \alpha q \]
    if \( ||\tilde{r}||/||b|| \) is small enough then stop
    Solve \( Mz = \tilde{r} \)
    \[ \tilde{\rho} = \tilde{r}^Tz \]
    \[ \beta = \tilde{\rho}/\rho \]
    \[ p = z + \beta p \]
    \[ r = \tilde{r}; \quad \rho = \tilde{\rho} \]
end for

Fig. 3.4. The Preconditioned Conjugate Gradient (PCG) scheme.
sparse matrix-vector multiplication dominate the computational cost of the overall execution time. As a result, the PCG performance depends on the performance of the sparse linear solver and matrix-vector multiplication.

3.1.5 Multi-grid

The Multi-grid method [173] is another approach to solving sparse linear systems of equations. This method relies on working on several meshes of varying detail in order to accelerate the convergence to solution of a problem. In the most general case of the algorithm, at each level $i$, the mesh representing the problem contains half as many points in each direction as the mesh at level $i-1$. For certain problems the Multi-grid method offers significant performance gains over Krylov subspace based methods. However, unlike Krylov subspace methods, which tend to be as general as possible, the Multi-grid methods have to be significantly adapted to the problem they are to solve [173].

One of the simplest Multi-grid implementations is the V-cycle algorithm. The V-cycle algorithm consists of three main stages: restriction, coarse grid solution, and interpolation and smoothing. The pseudo-code for a generic V-cycle Multi-grid algorithm, for a 1-D model problem [173], is shown in Figure 3.5. The algorithm in Figure 3.5 assumes $p$ levels of recursion, and mesh sizes of $h, 2h, 4h \ldots 2^p h$. In that algorithm $H$ stands for $2h$ and $h_0 = h$ represents the coarsest mesh size. A typical Multi-grid [27] method requires multiple iterations to converge to a solution depending on the initial problem (requiring multiple invocations of the V-cycle).

Several benchmark suites [10, 41, 52] include the Multi-grid algorithm. In this thesis, we use the MG CLASS A (MG-A) from the NAS benchmark suite and the SPEC CFP2000 mgrid versions of the multi-grid algorithm.
MG(A,u,f)
% A - input matrix,
% u - solution vector,
% f - right hand side,
% $I^H$ - coarsening operator,
% $I^h$ - smoothing operator,
% $u_0^h$ is initial guess;

Presmooth: $u^h = \text{smooth}(A^h, u_0^h, f^h)$

Get residual: $r^h = f^h - A^h u^h$

Coarsen: $r^H = I^H r^h$

if ($H == h_0$) then
  Solve: $A^H \delta^H = r^H$
else
  Recursion: $\delta^H = MG(A^H, 0, r^H)$
end if

Correct: $u^h = u^h + I^h \delta^H$

Postsmooth: $u^h = \text{smooth}(A^h, u^h, f^h)$

Return $u^h$

Fig. 3.5. A generic V-cycle Multi-grid algorithm
3.1.6 Sparse Cholesky Matrix Factorization

In this section, we consider solving a system of partial differential equations, discretized and described by the equation $Ax = b$ using a direct method. In parallel sparse direct solvers, a Cholesky ($A = LL^T$) or an LU ($A = LU$) factorization is first computed and then used for a triangular solution [49, 53, 68, 75, 79, 169]. Efficient implementations on distributed memory multi-processors require data and task assignments that can balance the computational load for the factorization step among the processors. The computations in the factorization step are tree-structured and are formulated bottom-up on a supernodal tree using either effectively dense panels of columns in a left-looking panel scheme [49] or using dense triangular matrices in a multifrontal scheme [79, 80, 133, 169].

Sparse matrix factorization and its incomplete variations typically require a four step process: (1) ordering to compute a fill-reducing numbering, (2) symbolic factorization to determine the nonzero structure of the factor, (3) numeric factorization, and (4) triangular solution. The first ordering step is also critical for determining the parallelism and the total computational costs for the remaining steps. A well-established practice is to compute orderings, using for example, nested dissection techniques that recursively partition the graph of $A$ using vertex separators [66, 111]. After this step, the parallelism available for the subsequent factorization and triangular solution step can be represented by a tree. This tree can be weighted to represent computation costs and the tree can be mapped to processors to enable load-balanced computation of the factorization step.

We provide a brief overview of parallel sparse Cholesky factorization using a small example to illustrate the main ideas, which are described in greater detail in the survey article by
Heath et al. [79]. Figure 3.6 concerns the sparse matrix $A$ of a five-point $7 \times 7$ finite-difference grid which is widely used as a model problem in this area. The columns of the Cholesky factor $L$ of this matrix can be grouped into supernodes [132]. A supernode is a set of consecutive columns that have nested sparsity structure, and can essentially be treated as a dense block, see for example, the last seven columns in Figure 3.6. As a consequence of sparsity in $L$, columns in a supernode need not be updated by columns in all preceding supernodes in the numeric factorization, instead columns in a supernode $v$ are updated only by columns in supernodes within the subtree rooted at $v$ [132]. The sparsity structure of $L$ can be viewed in terms of effectively dense column-blocks or alternatively, in a recursive manner in terms of submatrices as shown in Figure 3.6.

The two types of cache-efficient numeric factorizations are a column-block scheme [158] and a multifrontal scheme [53]. The two schemes differ in how they compute and apply updates to columns in a given supernode from columns in earlier supernodes. In a multifrontal scheme, dense triangular matrix operations are used to factor the columns in a supernode and to accumulate and propagate updates from these columns to those at the parent and ancestor supernodes, as shown in Figure 3.6. Multifrontal schemes typically lead to efficient parallel implementations [75, 169].

In this thesis we use DSCPACK as a representative of sparse direct solvers. DSCPACK implements the Cholesky factorization method for symmetric matrices. It uses the multifrontal scheme to obtain the linear system solution in parallel, and its underlying computational dependencies are represented by a tree.
3.2 Data Sets

In this section, we list the data sets used with our scientific compute kernels. We consider two sets of matrices:

- Sparse matrices used for evaluating effectiveness of workload distribution schemes, and workload slack to energy savings using DVFS in tree-based computations. The names and pertinent properties of these matrices are listed in Table 3.1.

- Sparse matrices used for evaluating energy and performance optimizations and trade-offs for architecture and sparse applications. The names and pertinent properties of these matrices are listed in Table 3.2.

These matrices are freely available for download from NIST Matrix Market [161] and the University of Florida matrix collection [47].

Fig. 3.6. The structure of a sparse matrix $A$ from a $7 \times 7$, 5-point finite-difference grid reordered to reduce fill (left), the structure of $L$ shown with a recursive partition (middle) and a multifrontal scheme on the binary supernodal tree (right).
Table 3.1. Sparse matrices used in evaluating effectiveness of workload balancing schemes, and opportunities for converting workload imbalance into energy savings. Table contains name of the matrix, number of processors used for evaluating opportunities for converting workload imbalance into energy savings, ranks of the matrix, number of nonzeros in the matrix, number of nonzeros in the factor L of the matrix, cost of the matrix factorization, ideal amount of work per processor, maximum workload assigned to a processor and minimum workload assigned to a processor in workload imbalance to energy savings experiments.

| Matrix | P  | Rank | $|A| (10^3)$ | $|L| (10^3)$ | Factorization Cost ($10^6$) | Ideal ($10^6$) | Max ($10^6$) | Min ($10^6$) |
|--------|----|------|------------|------------|----------------------------|---------------|-------------|-------------|
| bcsstk31 | 28 | 35588 | 608 | 10605 | 7161 | 256 | 295 | 194 |
| bcsstk32 | 110 | 44609 | 1029 | 19664 | 24787 | 225 | 241 | 207 |
| bcsstk35 | 17 | 30237 | 740 | 11335 | 11996 | 706 | 834 | 637 |
| crystk02 | 11 | 13965 | 491 | 5059 | 2890 | 263 | 406 | 158 |
| crystk03 | 24 | 24696 | 887 | 11617 | 9055 | 378 | 559 | 309 |
| finan512 | 28 | 74752 | 335 | 17905 | 13695 | 490 | 739 | 376 |
| nasasrb | 22 | 54870 | 1366 | 13084 | 6193 | 282 | 444 | 196 |
| qa8fk | 119 | 66127 | 863 | 34705 | 49393 | 415 | 523 | 387 |
| tube1 | 7 | 21498 | 459 | 5731 | 3865 | 553 | 826 | 405 |
Table 3.2. Sparse matrices used in evaluating architecture energy and performance trade-offs for sparse applications. Table contains name of the matrix, rank of the matrix, number of nonzeros in the matrix, and density, i.e., fraction of nonzeros in sparse matrix over equivalent dense representation.

| Matrix     | Rank \(10^3\) | \(|A|10^3\) | density (%) |
|------------|----------------|-------------|--------------|
| bcsstk31   | 35.6           | 1.2         | .09          |
| fdm21      | 32.1           | .16         | .01          |
| qa8fm      | 66.1           | 1.6         | .03          |
| msc23052   | 23.0           | 1.1         | .21          |

3.3 Benchmark Suites

In this section, we briefly introduce the standard benchmark suites used in this thesis in addition to the scientific compute kernels described in Section 3.1. We use these benchmarks when appropriate to evaluate the impacts and interactions of our hardware enhancements and software optimizations. We also rely on these benchmark suites to ensure that while our enhancements improve energy efficiency of scientific codes, that they do not negatively impact the performance of non-scientific codes. We do not provide detailed discussion of these benchmark suites, as it is available from other sources. In Section 3.3.1, we discuss the NAS benchmark suite. In Section 3.3.2, we discuss the SPEC benchmark suite, in Section 3.3.3, we discuss the SPEC OMP benchmark, and in Section 3.3.4, we discuss the SPLASH-2 benchmark suite.
3.3.1 NAS Benchmark Suite

NAS benchmarks [10, 156] were created at NASA as a way of comprehensively modeling and evaluating the performance of large-scale systems. The NAS benchmark suite consists of the following 7 benchmarks: CG, MG, SP, BT, FT, LU, EP. In this work we rely on the single processor, single threaded versions of NAS CG and NAS MG benchmarks. In addition to the single processor benchmark set, the NAS benchmark collection also contains the MPI and OpenMP implementations of these scientific codes.

3.3.2 SPEC

The SPEC benchmark suite [40], is well known and widely used in the architecture and compiler community to compare performance and power consumption of new compilers and hardware features against a reference system. It is a useful tool in determining the overall benefits and cost tradeoffs of new architectures, and is used to avoid introducing features that improve only one aspect of architecture while unduly hurting another. However, as noted by Murphy et al. [155], this benchmark suite does not represent well the types of workloads encountered frequently in scientific computing. In this thesis, we use the full SPEC benchmark suite only when we introduce a new hardware feature which is meant to boost significantly performance of scientific applications, and is not a feature that has been discussed or utilized in earlier works. Otherwise we focus on two scientific benchmarks from the SPEC FP benchmark set, namely equake and mgrid.
3.3.3 SPEC OMP

The SPEC OMP (OpenMP Benchmark Suite) [40], is a set of benchmarks from the Standard Performance Evaluation Corporation, aimed at evaluating the performance of shared memory multi-processor systems. The latest version of this benchmark suite (v3.2) consists of the following benchmarks: 310.wupwise_m, 312.swim_m, 314.mgrid_m, 316.applu_m, 318.galgel_m, 320.equake_m, 324.apsi_m, 326.gafort_m, 3.28.fma3d_m, 330.art_m, 332.ammp_m, 311.wupwise_l, 313.swim_l, 315.mgrid_l, 317.applu_l, 321.equake_l, 325.apsi_l, 327.gafort_l, 329.fma3d_l and 331.art_l. The benchmarks with “_m” postfix have medium input data sets, and the benchmarks with “_l” postfix have large input data sets. We use this benchmark suite in Chapter 10 to evaluate the performance of our tree-based non-uniform access latency cache architecture.

3.3.4 SPLASH-2

The SPLASH-2 benchmark suite [59, 194] is a set of parallel applications created for the study of centralized and distributed shared-address-space multi-processors. It consists of a mixture of complete scientific applications and computational kernels: barnes, cholesky, fft, fmm, lu, ocean, radiosity, radix, raytrace, volrend, water-nsquared and water-spatial. In this thesis, we focus on two of the benchmarks from this suite: lu and cholesky. We use these two kernels in Chapter 10 to evaluate the performance of our tree-based non-uniform access latency cache architecture.

3.4 Chapter Summary

In this chapter, we introduced and discussed sparse compute kernels and data sets used in this thesis for evaluating power and performance optimizations for sparse scientific applications.
We also introduced standard benchmark suites which we use to evaluate effectiveness of our new hardware features and to ensure that these new hardware features do not have negative performance impacts on non-scientific codes. In the next chapter, we begin our investigation of opportunities for entire multi-processor level power and performance optimizations. We consider the problem of balanced workload distribution among nodes of a multi-processor system in parallel tree-based applications, and its impact on performance and energy efficiency.
Chapter 4

Improving Computational Workload Distribution for Parallel Tree-Based Computations

Many modeling applications have underlying sparse computer representations that require solution of linear systems on multi-processors. In order to maintain constant performance isoefficiency, the problem size is scaled with the number of processors used, and the resulting computational workload is distributed, as evenly as possible, among all the participating processors. In addition to maintaining isoefficiency, a balanced workload distribution reduces parallel execution time and lowers the network overheads.

In Chapters 1 and 2, we identified energy consumption as a performance limiter of large parallel machines. In this chapter, we consider the problem of balanced workload distribution and its impacts on energy efficiency for scientific applications. We consider the systems where the processor dynamic energy or leakage energy dominates the overall system energy costs and where no energy reduction features are present. On such systems, reducing the overall execution time translates into lower overall energy use. Consider, for example, a parallel application running on the Earth Simulator, which consumes 7 MW of power on average [178]. Also consider a case where by better workload distribution, we reduce execution time by 20%. We can then expect the energy consumed by this application to reduce on average by 20% as well.

In this chapter, we consider the problem of balanced workload distribution and energy efficiency for scientific applications where the computation dependencies are represented by a tree. We introduce a multi-pass workload distribution scheme in light of its application to the
sparse Cholesky matrix factorization in DSCPACK [169], which we described in Section 3.1.6. Our multi-pass scheme significantly improves performance of the test application. Furthermore, our scheme can be effectively utilized for balancing workloads in incomplete Cholesky preconditioners and other codes whose underlying computational dependencies are tree-based.

The remainder of this chapter is organized as follows. In Section 4.1, we provide a short introduction to sparse matrix factorization. In Section 4.2, we present our multi-pass mapping scheme and the well known proportional mapping scheme. Finally, in Section 4.3, we compare the effectiveness of our scheme against the proportional mapping scheme and show that our scheme results in significantly better workload distributions.

Research presented in this chapter was published in a paper: “Multi-Pass Mapping Schemes For Parallel Sparse Matrix Computations” [143], co-authored by Konrad Malkowski and Padma Raghavan, in the Proceedings of ICCS 2005: 5th International Conference on Computational Science, Lecture Notes in Computer Science.

4.1 Sparse Matrix Factorization

Consider solving a system of partial differential equations, discretized and described by the equation $Ax = b$. A solution to $Ax = b$, where $A$ is sparse, can be achieved using either direct methods such as DSCPACK [169] or preconditioned iterative methods. Efficient implementations of these methods on distributed memory multi-processors require data and task assignments that can balance the computational load for the factorization step among the processors. The computations in the factorization step are tree-structured and are formulated bottom-up on a supernodal tree using either effectively dense panels of columns in a left-looking panel scheme [49] or using dense triangular matrices in a multifrontal scheme [79, 80, 133, 169].
Parallel implementations of both left-looking or multifrontal factorization depend on the supernodal tree which can be weighted to represent the corresponding computation and communication costs [72,75,169]. For illustrative purposes, view this tree as a complete binary tree with more leaves than the number of processors $P$ and with all nodes having the same computational cost. Now, at some level $l = \log_2 P$, $P$ disjoint nodes can be identified and the subtrees rooted at these nodes can be assigned to distinct processors. These subtrees represent disjoint local computations at processors with ideal task-parallelism. At a level higher than this one, disjoint processor groups of size 2 can cooperate to perform data-parallel computations at the supernode and so on, until all processors participate at the root. This is known as the balanced subtree to processor mapping. The proportional mapping scheme is a generalization of this scheme to derive assignments for practical problems where the supernodal tree can be highly irregular and the computation at a node and total computations in subtrees can vary dramatically.

4.2 Workload to Processor Mapping Schemes

In this section, we begin with an overview of the original proportional mapping scheme [166] and continue with the formulation of our multi-pass assignment schemes. The latter seek to refine and improve an assignment obtained from the proportional mapping, which is used in the first step. We use the DSCPACK direct solver, which we introduced in Section 3.1.6, for evaluation of the original proportional mapping scheme and our multi-pass mapping schemes.

To provide a precise statement of our mapping schemes, we start with a definition of the weighted supernodal tree and a valid mapping, i.e., an assignment of computations represented by the tree to a set of processors. Consider a supernodal tree $T(r) = (V, E, NW, SW)$ with $V$ vertices, $E$ edges, rooted at $r \in V$ with two weighting functions $SW$ and $NW$ representing a
suitable measure of computational costs. For each vertex $v \in V$, $NW(v)$ is the nodal weight, corresponding to the cost of computations at the node $v$. The subtree weight of $v$, $SW(v)$ is defined as the sum of nodal weights of all vertices in $T(v)$, the subtree rooted at $v$. A mapping $M = (T, P)$ indicates an assignment of a set of $P$ disjoint subtrees $T(v_0), T(v_1), \cdots T(v_{P-1})$ (including all leaf vertices in $V$) to processors $0, 1, \cdots (P - 1)$. These disjoint subtrees represent local task parallel computations; computations at an interior vertex $v$ are shared equally among all processors assigned subtrees in $T(v)$. However, such computation cannot proceed until all processors can synchronize at the internal node. Consequently, load imbalances among processors along different paths leading to a vertex $v$, result in some processors remaining idle until all can synchronize and proceed with the computations at $v$.

The proportional mapping [166] is a recursive scheme with an initial assignment of $P$ processors to the root $r$ and thus $T(r)$. Consider $T(v)$ the subtree at $v$, which has been assigned $p$ processors. If $p = 1$, the recursion terminates; otherwise for $p > 1$, for each child $c$ of $v$ assign $p_c$ to $T(c)$ where $p_c = p \times \frac{SW(c)}{SW(v) - NW(v)}$. Some rounding scheme must be used to ensure that $p_c$ is an integer number of processors and that $\sum_{c,(c,v) \in E} p_c = p$. We experimented with several rounding schemes to select the one that leads to the best mappings for our test collection.

In this scheme, at a vertex $v$ with $T(v)$ assigned $p$ processors, we compute for each child $c$, $\hat{p}_c = \lfloor p \times \frac{SW(c)}{SW(v) - NW(v)} \rfloor$, and the projected load $\hat{W}(c) = \frac{SW(c)}{\hat{p}_c}$. Next, we compute $\tilde{p} = p - \hat{p}$ where $\hat{p} = \sum_{c,(c,v) \in E} \hat{p}_c$. Let $c_1, c_2, \cdots c_{\tilde{p}}$ be the children vertices of $v$ with the $\tilde{p}$ highest values of the projected load $\hat{W}(c_i)$; for these vertices we set $p_c \leftarrow \hat{p}_c + 1$ while for the others, we set $p_c \leftarrow \hat{p}_c$. Our multi-pass assignment scheme uses this proportional mapping as the first step.
Rounding effects, incurred to ensure an integer number of processors at each node, can be exaggerated by irregular subtree weights resulting in assignments with processor loads that are not well-balanced. Consequently, the highest load at a processor, i.e., the critical path weight, can be substantially higher than the ideal of $SW(r)/P$ as shown in the next section in Figure 4.1.

Our multi-pass schemes attempt to refine the assignment from the proportional mapping in order to improve the worst load at a processor. We therefore start with a precise definition of this quantity. Consider any mapping $M = (T(r), P)$ of the supernodal tree $T(r) = (V, E, NW, SW)$ with disjoint subtrees $T(v_0), T(v_1), \ldots T(v_{P-1})$ assigned to processors $0, 1, \ldots (P-1)$. Let $\pi(v)$ denote the number of processors assigned to a node $v$; note that $\pi(r) = P$. Let $\text{path}(i)$ denote the intermediate vertices in $T$ from the parent of $v_i$ to the root $r$. Now the workload of a processor $p_i, 0 \leq i \leq (P-1)$ is given

$$W(p_i) = SW(T(v_i)) + \sum_{v \in \text{path}(v_i)} NW(v)/\pi(v).$$

An ideal assignment would lead to the ideal load $I(M) = SW(r)/P$. The heaviest load at a processor, which corresponds to a critical path is given by $H(M) = \max_{p_i, 0 \leq i \leq P-1} \{W(p_i)\}$; likewise, we define the lightest load $L(M) = \min_{p_i, 0 \leq i \leq P-1} \{W(p_i)\}$; Our goal is compute mappings where $H(M)$ is close to $I(M)$ for a given $T$ and $P$.

We next present two refinement schemes, followed by our final multi-pass scheme which enables their effective combination.

**Robin Hood Refinement Scheme.** Assume that an assignment $M$ has been provided either from proportional mapping or the application of one or more refinement schemes. First, determine $H(M)$ and $L(M)$ corresponding to the heaviest and the lightest loads at a processor. Let $p_h$ and $p_l$ denote the corresponding processors assigned to subtrees $T(h)$ and $T(l)$, rooted at vertices $h$ and $l$ respectively. We consider $p_h$, the overworked processor to be “poor” and the lightly loaded $p_m$ to “rich.” Our Robin Hood scheme removes a processor assigned along the
lightest path (path(l)), and uses it to refine the mapping of the local subtree T(h) and thus reduce the load along critical path (path(h)). The Robin Hood scheme is typically applied four times to allow refinement of paths with weights close to the weight of the critical path, and the best assignment is retained.

**Iterative Correction with Processors in Reserve.** Assume we are seeking an assignment $M = (T(r), P)$ with $P$ processors. We first obtain a mapping with $\tilde{M} = (T(r), \tilde{P})$ with $\tilde{P}$ processors where $\tilde{P} < P$. Let $\hat{P} = P - \tilde{P}$ denote the remaining processors “held in reserve.” Our correction algorithm proceeds thus in $\hat{P}$ iterations, starting with a mapping $M_1$ initialized to $\tilde{M}$ with $P_1 = \tilde{P}$ processors. At iteration $i$, compute $H(M_i)$ and let $p_h$ be the corresponding processor with the heaviest load. Refine the mapping of $T(h)$ by adding another processor to vertex $h$ to obtain a new mapping $M_{i+1}$ with $P_{i+1} = P_i + 1$ processors. A disadvantage of this scheme is that improvements depend on the initial choice of $\tilde{P}$.

**Multi-Pass Mapping Scheme.** We now combine the two refinement schemes with the original proportional mapping to present our final multi-pass mapping scheme. The first pass is the proportional mapping scheme. In the second pass, this mapping is refined using the Robin Hood scheme. Let this result in a mapping $\bar{M}$ with the specified number of processors, $P$. Compute $H(\bar{M})$; if this quantity is more than the ideal load $(SW(r)/P)$, then compute $\hat{P} = SW(r)/H(\bar{M})$, where $\hat{P} < P$. Apply the proportional mapping to obtain a mapping $\tilde{M}$ with $\hat{P}$ processors. Refine it using the Robin Hood scheme to obtain a new mapping $\bar{M}$ with $\tilde{P}$ processors. Finally, refine this mapping using iterative correction with $\hat{P} = P - \tilde{P}$ processors in reserve. This defines our overall multi-pass mapping scheme.
4.3 Performance Impacts of Multi-Pass Mapping

In this section, we empirically evaluate the quality of assignments for performing parallel sparse Cholesky factorization. We report on the improvements observed when our schemes are used to refine the assignments computed by the original proportional scheme. We use a collection of well-known sparse matrices from finite-element analysis of three dimensional structures and shells, and one problem from computational fluid dynamics.

Our test suite of matrices is discussed in Section 3.2. In Table 4.1, we re-list our matrix suite and present the best observed improvements. We consider the factorization of these matrices using 8-64 identical processors after ordering using a nested dissection scheme. The supernodal trees were weighted to represent computational costs (for floating point operations, and not for communication or for other integer operations) in a parallel multifrontal scheme, for example, such as the scheme in the DSCPACK software [169].

Our results concern the quality of mappings as defined by the heaviest workload at a processor, i.e., the critical path weights. Consider \( T(r) \) corresponding to a specific problem. For each mapping \( M \) of \( T(r) \) using some \( P \) processors in the range 8 — 64, we focus on heaviest load at a processor, i.e., the critical path cost, as the main metric indicative of the quality of the mapping. The closer this metric, \( H(M) \) (defined in the earlier section) is to the ideal load \( (I = SW(r)/P) \), the better is the quality of the mapping. Over the range of processors, problems and mapping the actual value of this metric can vary significantly making direct comparisons difficult. We therefore use the following two scaled forms: (i) the relative critical load, (RCL) defined as \( \frac{H(M)}{I} \times 100 \), and (ii) the critical overload, (CO) defined as \( \frac{H(M)-I}{I} \times 100 \).
Table 4.1. Description of test matrices, relative critical loads (RCL), and numeric factorization times using the original proportional scheme and the multi-pass scheme. Each matrix - processor pair corresponds to the best observed improvements of the RCL metric using our multi-pass scheme. The column labeled “Error” indicates the difference between predicted and observed execution times.

| Matrix   | Rank $|A|$ $|L|$ | Processors | Proportional Map | Multi-pass Map |
|----------|------|------|------------|------------------|-----------------|
|          |      |      |            | RCL Time | RCL | Predicted | Observed | Error |
| bmw7st1  | 141  | 374  | 9,134      | 134 97  | 114 | 82.52    | 80.72   | 2     |
| bmwcra1  | 148  | 539  | 18,924     | 146 493 | 129 | 435.6    | 454     | 4     |
| bmw3,2_l| 227  | 575  | 17,998     | 130 263 | 122 | 246      | 242     | 2     |
| augustus7| 1,060| 518  | 79,995     | 133 1,989| 117 | 1,749    | 1,653   | 6     |
| augustus5| 134  | 64   | 4,598      | 145 32.42| 117 | 26.16    | 24.97   | 1     |
| af_shell3| 504  | 904  | 11,425     | 137 43.65| 116 | 36.96    | 40.11   | 9     |
| cfd2     | 123  | 160  | 7,465      | 154 75.67| 118 | 57.98    | 57.84   | 0     |
We begin with some experiments (reported in the right half of Table 4.1) to verify that the relative critical load (and the critical overload) metric corresponds well to the actual performance of the numeric factorization step. We used the DSCPACK software [169] with two different mappings, one from the original proportional scheme and the other from our multi-pass scheme. Our experiments were performed on a cluster with 81 dual-processor compute nodes with AMD Athlon MP2200+ processors, with 71 nodes having 1GB of main memory, and 10 nodes having 2GB of main memory, and a 9 × 9 torus Scali interconnect. Table 4.1 shows the relative critical load (RCL) and CPU time for numeric factorization using the original proportional mapping. We then compute assignments using our multi-pass scheme and their corresponding values of the relative critical load (also reported in Table 4.1). Using the relative critical load metrics for the original and multi-pass mappings and the CPU time for numeric factorization with the proportional mapping, we can project the estimated numeric factorization time for the new assignment (reported in the column labeled “Predicted” time). We next performed numeric factorization with the new assignments and observed actual CPU time. As shown in Table 4.1, these observed times are in close agreement with our predicted values, thus indicating that it is valid to use the relative critical load metric and the closely related critical overload metric to evaluate the quality of assignments.

Figure 4.1 plots the critical overload metric for the original proportional mapping, the Robin Hood scheme and our final multi-pass scheme for *augustus7* on 8 through 64 processors. The plots indicate that Robin Hood scheme can improve the assignments produced by the original proportional mapping. However, as expected, these improvements are not as substantial as the improvements from our final multi-pass scheme. Consequently, in the remainder of this
Fig. 4.1. Critical overload for the original proportional heuristic (left), and the Robin Hood and multi-pass schemes (right) for augustus7 on 8 – 64 processors.

In this section, we focus on more detailed comparisons between the quality of assignments produced by the original proportional scheme and our final multi-pass scheme.

Figure 4.2 (left) indicates how our scheme can improve the worst mapping generated by the original scheme. For each matrix, we select the instance with the largest value of the critical overload metric from an assignment by the proportional scheme. For this instance, i.e., problem-processor pair, we show the value of the overload metric when our multi-pass scheme is used. Our multi-pass scheme successfully reduced the worst case overload from almost 60% to 27% for the bmw7st1 matrix. On average, the metric is halved from nearly 50% for the original to 25% for our multi-pass scheme. Figure 4.2 (right) shows the best observed improvement from our new multi-pass scheme when compared to the original mapping for each matrix in the test set. Assignments from our multi-pass scheme reduced the critical overload for bmw7st1 from approximately 60% to 27% and from 55% to 17% for cfd2. Additionally, on average over these instances, the critical overload from the proportional scheme was 46%, whereas it was reduced to 17% from our multi-pass schemes.
We next consider the overall quality of the assignments produced by the original proportional scheme and our multi-pass scheme for the 7 matrices using 16 – 64 processors. We now consider a cumulative form of the critical overload metric shown as a stacked bar in Figures 4.3 and 4.4. In Figure 4.3, each stacked bar represents the critical overload value summed over all 7 matrices for a specific number of processors. Figure 4.3 clearly indicates that our multi-pass scheme significantly improves the quality of mapping over all problems for the entire range of processors. On average, the cumulative critical overload is reduced from a value of approximately 150 to 100 through the use of our multi-pass scheme. Figure 4.4 shows the improvements for each matrix cumulatively over the range of processors; each stacked bar represents the critical overload value summed over all 49 processor sizes (from 16 — 64) for a specific matrix. Now the average value from the original scheme is approximately 1050 which is reduced to under 725 by our multi-pass mapping scheme. These results show that our multi-pass schemes are indeed effective in producing assignments that substantially improve the balance of loads among processors.

4.4 Chapter Summary

In this chapter, we introduced a multi-pass scheme that improves workload distribution among processors in parallel tree-structured computations. Our scheme can be applied with a more complex weighting scheme to take into account both computation and inter-processor communication costs. Our scheme can also be used with a weighting function to model memory requirements to produce more balanced assignments. Additionally, we can compute assignments for applications where the triangular solution costs following the factorization are dominant, i.e., one factorization is followed by solutions for a sequence of right-hand-side vectors [195]. We
Fig. 4.2. Quality of assignments for worst load instances (left) and the best observed improvement instances (right); each group of two bars indicates the critical overload from the original proportional scheme and our multi-pass scheme. Average values are indicated by horizontal lines.

Fig. 4.3. Cumulative critical overload over all matrices for each processor, from assignments using the original proportional scheme (left) and from our multi-pass scheme (right). Each patch in a stacked bar represents the critical overload for one matrix. Average values are indicated by horizontal lines.
can also extend the weighting schemes to model parallel incomplete factorization for preconditioning \cite{citation} and the subsequent application of the preconditioner using tree-structured parallel schemes. In the next chapter, we show how the supernodal tree describing computational dependencies in DSCPACK can be combined with DVFS to further reduce energy consumption of multi-processor systems.
Fig. 4.4. Cumulative critical overload for 16–64 processors for each matrix, from assignments using the original proportional scheme (left) and from our multi-pass scheme (right). Each patch in a stacked bar represents the critical overload for one processor size in the range 16–64. Average values are indicated by horizontal lines.
Chapter 5

Converting Workload Distribution Imbalances into Energy Savings

Performance of many modeling applications depends on performance of the underlying linear solver. Maintaining high isoefficiency, as the models are scaled to many processors, requires careful data and workload partitioning among all compute nodes. In Chapter 4, we discussed the problem of balanced workload distribution of sparse Cholesky factorization among the nodes of a multi-processor system and introduced the multi-pass processor mapping technique. While our method is successful in improving the execution time of tree-based computations, it does not produce a completely balanced workload distribution. In this chapter, we show how the supernodal tree used to describe computational dependencies in parallel sparse Cholesky factorization can be used with dynamic voltage and frequency scaling to reduce application energy consumption on a multi-processor system.

5.1 Using Supernodal Tree for Energy Scaling

Consider Figure 4.2, where we compared the quality of workload distribution generated by two mapping schemes: the one-pass proportional mapping method and our multi-pass mapping method. Observe that neither method generates an ideal workload distribution among processors. A sample supernodal tree and corresponding processor assignment are shown in plot (a) of Figure 5.1, where the nodes on the critical path are marked by the thick line. We see that processors assigned to the non-critical paths of the tree (with lower computational workloads assigned to them) reach the synchronization points ahead of the processors on the critical path and idle. We observe that only the processors assigned to the nodes on the critical path have to execute at full speed. The other processors can run at reduced performance levels, resulting in energy savings. The only restriction on the energy savings is that the processors on non-critical paths compute their results before the processors on the critical path.

The supernodal tree, as used by DSCPACK, contains information about computational work required by each tree node. We extend the amount of information stored in each node of the tree to include: (i) voltage, (ii) frequency and (iii) power level. We normalize each of these metrics with respect to the voltage, frequency and power consumed by the processor running at full frequency. Table 5.1 lists the voltage, frequency and power levels used in our examples. A sample tree with the additional metrics is shown in plot (b) of the Figure 5.1.
(a) Weighted supernodal tree from DSCPACK. The numbers written inside the nodes indicate the associated computational costs. Leaf nodes represent the local phase computations, and each of them is assigned to a processor (P0-P4). The bold lines represent the critical path. (b) Supernodal tree with energy related information. The three numbers inside each node, from top to bottom, represent the voltage level, the time it takes, and the energy consumption required to compute this node.

Table 5.1. Voltage, frequency and power levels used to explain the behavior of tree DVFS scaling algorithms.

<table>
<thead>
<tr>
<th>Voltage</th>
<th>Frequency</th>
<th>Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0.8</td>
<td>0.8</td>
<td>0.512</td>
</tr>
<tr>
<td>0.6</td>
<td>0.6</td>
<td>0.216</td>
</tr>
<tr>
<td>0.4</td>
<td>0.4</td>
<td>0.064</td>
</tr>
</tbody>
</table>
5.1.1 DVFS Scaling Algorithms

We consider three energy reduction schemes for converting workload imbalances in the supernodal tree to energy savings using dynamic voltage and frequency scaling. We call these algorithms: “Algorithm I”, “VS2” and “VS3”, and discuss them below.

- Algorithm I is a recursive algorithm, similar to the one-pass proportional mapping discussed in Section 4.1. In this algorithm we focus on recursively scaling the voltage levels of each subtree first. We first consider the root node and its two children subtrees. One of the subtrees is on the critical path and it cannot be scaled down, as such a scaling would increase the overall execution time. The subtree not on the critical path is scaled down to the point that guarantees that computations in the subtree finish just before the critical path finishes. We apply this algorithm recursively to each child subtree until further subtree scaling results in performance degradation. Figure 5.2 shows a few steps of this algorithm.

Fig. 5.2. Algorithm I - an example. (a) The original tree. (b)-(d) Consecutive steps of applying Algorithm I to the tree. The subtree or node scaled at each step is enclosed in a dashed circle.
• VS2 and VS3 - The VS2 and VS3 algorithms are extensions of Algorithm I. It can be shown that the Algorithm I is optimal when the processor has an unlimited number of voltage levels. However, in real systems there are a limited number of voltage levels available for scaling. Under these conditions the Algorithm I does not always generate an optimal voltage scaling. Figure 5.3 shows an example where the Algorithm I generates a voltage scaling tree that is not optimal, i.e., some nodes of the tree can be further scaled down to reduce overall energy consumption without negatively impacting the application performance. In our example, we can either scale the node G further as shown in plot (a) of Figure 5.3 or the node H as shown in plot (b) of Figure 5.3. The VS2 and VS3 methods improve on the Algorithm I method by scaling individual nodes in the subtree, in addition to scaling the subtrees as a whole. For each subtree, the VS2 method performs the subtree scaling first, and then when no whole tree scaling is further possible it attempts to scale the root of the subtree before being applied recursively to its children (Figure 5.4). The VS3 method is a modification of the VS2 method that is applied recursively to the children subtrees first, and then once the leaf nodes are reached, it scales the root node of each subtree as it exits the recursion (Figure 5.5).
Fig. 5.3. Two examples where the results of Algorithm I in Figure 5.2(d) can be improved further. The nodes enclosed in the dashed circle have been scaled further.

Fig. 5.4. Algorithm VS2 - example. The original tree is given in Figure 5.1(b). (a)-(d) Consecutive steps of the algorithm. The subtree or node scaled at a given step is enclosed in a dashed circle.
Fig. 5.5. Algorithm VS3 - example. The original tree is given in Figure 5.1(b). (a)-(d) Consecutive steps of the algorithm. The subtree or node scaled at a given step is enclosed in a dashed circle.

5.2 Empirical Results

In this section, we discuss the effectiveness of our three voltage scaling schemes: “Algorithm I”, “VS2” and “VS3”. We use data sets obtained from the DSCPACK [169] sparse solver. The input matrices to DSCPACK are listed in Chapter 3 in Table 3.1. We use the power numbers and voltage levels for the Transmeta Crusoe processor [42] to simulate the limited number of voltage levels present in processors. The power levels, voltage and corresponding frequencies are listed in Table 5.2. The maximum frequency in our experiments is 600MHz. We use rough calculations to estimate the amount of energy consumed by DSCPACK at each node of the tree by multiplying the average power consumed by processors at each node by the estimated execution time required to finish the computations at each node. We obtain the execution times for
Table 5.2. Voltage, frequency, and power values used in our experiments evaluating tree DVFS scaling algorithms when 5 processor voltage levels are available.

<table>
<thead>
<tr>
<th>Volt (V)</th>
<th>Freq (MHz)</th>
<th>Power (W)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.6</td>
<td>667</td>
<td>5.3</td>
</tr>
<tr>
<td>1.5</td>
<td>600</td>
<td>4.2</td>
</tr>
<tr>
<td>1.35</td>
<td>533</td>
<td>3.0</td>
</tr>
<tr>
<td>1.225</td>
<td>400</td>
<td>1.9</td>
</tr>
<tr>
<td>1.2</td>
<td>300</td>
<td>1.3</td>
</tr>
</tbody>
</table>

each node from the workload estimates provided to us by DSCPACK. We plot our results in Figure 5.6, and observe that on average Algorithm I, VS2 and VS3 reduce the energy consumption by 16%, 21% and 21% respectively.

Fig. 5.6. Percentage energy savings resulting from applying: Algorithm I, VS2 and VS3.
5.3 Chapter Summary

In this chapter, we considered the problem of improving energy efficiency of scientific applications in light of workload imbalances. We introduced three methods for converting workload imbalances in applications with tree-based computational dependencies to energy savings. Our methods use DVFS to scale the performance of processors not on the critical computational path, to ensure “Just in Time” completion of tasks assigned to them. Our methods reduce energy consumption by 16% (Algorithm I), 21% (VS2) and 21% (VS3) without negatively impacting application performance. These energy savings can be further improved by combining DVFS with network link scaling [180]. In the next chapter, we move towards localized, single-node power and performance optimizations, and consider the interactions between software tuning, and hardware features in light of sparse matrix-vector multiplication kernel.
Chapter 6

Local Node Algorithm and Architecture Optimizations
For Fast Power-Aware Sparse Computations

The isoefficiency of sparse scientific applications on multi-processor systems is limited by the communication overhead, data partitioning, and computational workload distribution. Thus every effort is made to minimize the communication overheads and increase the fraction of time spent in local (communication-free) phases. In Chapter 4, we investigated the impacts of workload distribution on performance and energy consumption of the multi-processor computer systems running tree-based sparse scientific applications. We showed that while improved workload distribution schemes can efficiently reduce the application execution time, for certain types of applications the ideal workload distribution cannot be achieved. Then, in Chapter 5, we showed how this workload imbalance can be converted into energy savings by applying dynamic voltage and frequency scaling to non-critical computations.

In this chapter, we investigate opportunities for further improving energy and performance efficiency of sparse scientific codes, by focusing on localized single node optimizations. We consider the sparse matrix-vector multiplication kernel, which is at the heart of many iterative linear solvers, and look at interactions between software optimizations, hardware performance-enhancing features and power control modes. We investigate opportunities for the use of dynamic voltage and frequency scaling (DVFS) and drowsy caches in combination with software and hardware optimizations to reduce the overall system power consumption while maintaining or exceeding the performance of the application on an unoptimized system.
We evaluate energy-aware architectural optimizations through simulations with SimpleScalar [186] and Wattch [22]. Our goals are to enable more efficient use of the CPU and memory subsystem by sparse matrix kernels. Our results indicate that when these optimizations are used in conjunction with power control modes, such as dynamic voltage frequency scaling (DVFS), we can reduce the execution time by over 60% and the energy by over 85%. Additionally, we characterize variations in the relative impact of system optimizations on performance and energy metrics from interactions with the level of tuning of the sparse code. We demonstrate that observed relative improvements can vary by over 40% when the same combination of system optimizations is evaluated using different levels of tuning for the sparse kernel.

The remainder of this chapter is organized as follows. In Section 6.1, we discuss our methodology for evaluating performance and energy through simulation, specific memory subsystem optimizations, and our base RISC PowerPC architecture which is similar to the processor in BlueGene/L [46], the top ranked supercomputer. Section 6.2 contains our main contributions characterizing improvements in performance and energy and differences in relative improvements from the interplay between code and architectural features. Finally, Section 6.3 contains a brief chapter summary.

The research presented in this chapter was originally published as a paper titled “On Improving Performance and Energy Profiles of Sparse Scientific Applications” [141], co-authored by Konrad Malkowski, Ingyu Lee, Padma Rahavan, and Mary Jane Irwin, and published in the Proceedings of the 20th IEEE International Parallel and Distributed Symposium, IPDPS06, Next Generation Software Workshop.
6.1 Modeling Power and Performance Characteristics

We use cycle-accurate emulations of the sparse kernels using SimpleScalar3.0 [25] and Wattch1.02d [22] with extensions to model memory subsystem enhancements [144]. We model a single-core processor with some of the features of the BlueGene [184] starting from a PowerPC440 embedded core and including memory subsystem optimizations for prefetching as described in our earlier paper [144].

Our base architecture has two floating point units and two integer ALUs. Each FPU has a multiplication/division module and modules for other arithmetic and logic. We model a cache hierarchy with three levels on chip, including a 32KB data/32KB instruction level 1 cache (L1), a 2KB level 2 cache (L2), and a 4MB unified level 3 cache (L3). Starting with the base architecture, henceforth denoted by ‘B,’ we consider first the effects of doubling the width of the data paths, indicated using the label ‘W’.

We operate the SRAM L3 at system frequency and voltage levels when we consider different frequency-voltage pairs to simulate the effects of utilizing DVFS [38]. We consider eight CPU frequencies from 300MHz to 1000 GHz with with corresponding nominal $V_{dd}$ voltages in the 0.46V to 1.2V. The SRAM L3 cache may not benefit sparse kernels and energy-efficient alternative could include utilizing power control modes of caches; we simulate this by considering five L3 cache sizes of 256K, 512K, 1MB, 2MB and 4MB. Additionally, we consider the impacts of the memory subsystem optimizations including memory page policy and prefetching at the memory controller and L2 cache.

Memory page policy: open or closed labeled ‘MO’ or ‘MC’. This feature can impact performance depending on the data access pattern and its interaction with data layouts in memory.
(temporal and spatial locality). The closed page policy is more suitable for random memory accesses, when each access is preceded by an ‘activate’ operation and followed by a ‘precharge’ operation [45, 102]. On the other hand, with temporal and spatial locality of data accesses, an open page policy could reduce latencies, at the expense of greater complexity of the controller. An activated row stays active until a read/write operation to another row in the same bank. The latencies can be reduced to 8 cycles from 16 cycles for successive reads without bank conflicts [98].

Memory prefetching (stride-1) at the memory controller, labeled ‘MP’. MP can reduce the effective latency of memory access and it is emulated by adding a prefetch buffer to the memory controller. This buffer is a 16 element table, with each element holding a cache line of 64 bytes or 128 bytes for ‘W’ and it uses a a full LRU replacement policy. We model the power consumed by our prefetch buffer as the cost of operating a small 16 entry, direct mapped cache with a 64 or 128 byte cache line [144].

Level 2 cache prefetching (stride-1) labeled ‘LP’. Once again, this feature can reduce the latency of data access. The extra energy consumption is modeled as second cache access.

### 6.2 Empirical Results

In this section, we evaluate the impact of memory subsystem optimizations for sparse matrix vector multiplication kernels representing different levels of tuning. We use two implementations of sparse matrix-vector multiplication as discussed in Section 3.1.1:

- SMV-U, a natural implementation of sparse matrix-vector multiplication.
• SMV-O from Sparsity [95] with an appropriate level of loop unrolling and register blocking for the best performance on our base architecture, described in Section 6.1.

We use the four sparse matrices discussed in Section 3.2 and listed in Table 3.2. These matrices are first re-ordered using RCM before applying the two versions of the kernel. We also use the `equake` code from SPEC CFP 2000 with its application specific sparse matrix-vector kernel (Equake-A) and with a slightly tuned version of the kernel (Equake-AT). The Equake benchmark is discussed in Section 3.1.2. We start with in-depth analysis for SMV-U and SMV-O and conclude with an overview of results for `equake`.

We consider metrics such as execution time and energy, where energy is computed as the system power \( \times \) time. Our contributions include the following.

• Characterizing improvements in time and energy when memory subsystem optimizations are used in conjunction with DVFS and low power modes of caches.

• Characterizing relative improvements (RI) starting from the base system for fixed feature sets relative to a fixed base line.

• Modeling relative incremental improvements (RII) from adding a feature to the system after a sequence of earlier optimizations. For example, evaluating the incremental impact of adding memory prefetching (MP) for the base system with wider data paths (W) and an open page policy (MO).

We use several plots in this section with the following general format.

• The X-axis indicates 40 configurations corresponding to distinct frequency and L3 cache size pairs. The X-axis value 1 represents a CPU at 300 MHz with 256 KB L3, the value 2
represents a 300 MHz CPU with a 512KB L3, and so on with 40 representing the 1GHz, 4MB L3 configuration.

- The Y-axis shows either absolute or relative values of metrics such as time and energy and other derived metrics to capture relative improvements.

- Relative values show scaling with respect to a certain fixed point for the same kernel. Metrics for a kernel are not shown relative to values for a different kernel.

- Plots for base architecture are labeled ‘B’ and the features include wider data paths (W), open page memory policy (MO), a memory prefetcher (MP), and an L2-prefetcher (LP). When these features are added incrementally starting from the base ‘B’, the order is shown using labels of the form ‘B+W+MO+MP’ for base with wider data paths followed by adding an open page memory policy and a memory prefetcher.

We also use stacked and grouped bars to summarize results for a fixed L3 size across frequencies.

### 6.2.1 Performance and Energy Metrics: Profiles and Summary

Figures 6.1 and 6.2 show the execution times and energy for SMV-U (left) and SMV-O (right) when the features are added incrementally in the order ‘B+W+MO+MP+LP’. Both sets of plots show reductions in execution time from the optimizations and it is easy to see that both codes could benefit from significant energy savings by using DVFS, at improved execution times. Furthermore, at a given frequency for a specific memory subsystem optimization, the L3 cache size has negligible impact on execution time for both codes, thus allowing further energy savings if power saving modes of caches can be utilized. The plots indicate that at wider data paths (W) and the memory open page policy (MO) are particularly useful in reducing both time
and energy. It is also interesting to note that SMV-O on the system with all optimizations is
faster at even the lowest frequency (300MHz) than for the base configuration at 1GHz.

These plots, in Figures 6.1 and 6.2, are useful for identifying general trends but they do
not give insights into the relative effectiveness of different optimizations. We therefore use the
data presented in these plots to define and compute additional metrics. Consider a specific code
such as SMV-U. Let \( T_{f,c,q} \) denote the observed execution time, at frequency \( f \), cache size \( c \) and
feature set \( q \). We define relative improvement (RI) with respect to \( T_{1G,4M,B} \) as:

\[
RI(T)_{f,c,q} = \frac{T_{1G,4M,B} - T_{f,c,q}}{T_{1G,4M,B}}.
\]

Using the conventional definition of speedup as \( S(T)_{f,c,q} = \frac{T_{1G,4M,B}}{T_{f,c,q}} \), the metric \( RI(T)_{f,c,q} = 1 - \frac{1}{S(T)_{f,c,q}} \). Some observations regarding this metric include:

- Values greater than 0 indicate improvements (for example, decreases in time) while nega-
tive values indicate degradation (for example, increases in time).
- This metric can also be used to study improvements in energy; we indicate it as \( RI(E)_{f,c,q} \).
- RI values are defined with respect to a specific kernel and hence its base performance or
energy values.

Figure 6.3 shows the RI values for time, and energy for SMV-U at all frequencies for
the smallest L3 cache size of 256KB. Observe that in all cases, the improvements for SMV-O
are higher than for SMV-U for same configuration. On average with all optimizations, exec-
ution time for SMV-O is improved by 48% at average system energy improvements of 85%.
Corresponding average improvements for SMV-U are 35% in time and 83% in energy.
Fig. 6.1. Execution time in seconds for SMV-U and SMV-O when features are added in the order ‘B+W+MO+MP+LP’.

Fig. 6.2. Energy consumption (in Joules) for SMV-U and SMV-O when features are added in the order ‘B+W+MO+MP+LP’.
Fig. 6.3. Relative improvements (RI) in execution time and energy for SMV-U (left bar in pair) and SMV-O (right bar in pair). Values are relative to time and energy values for the base configuration at 1GHz with a 4MB L3 cache. The lines indicate average improvements for the final configuration ‘B+W+MO+MP+LP’ across frequencies (solid SMV-O, dotted SMV-U).

6.2.2 Relative Incremental Improvements (RII): Measuring Incremental Impact per Feature Addition

The RI values represent speedups for specific configurations relative to a fixed base (at 1GHz, 4MB L3). Thus, they model improvements from the specific set of optimizations as well as from other factors including frequency related scaling and the effect of cache sizes. It would be appropriate to devise a metric that removes the effects of frequency and cache sizes for modeling incremental improvements when the existing configuration is augmented by one more optimization. Otherwise, effects of frequencies and caches may dominate over
improvements strictly from the new feature. To model the speedup for adding one optimization \( r \) to an existing configuration \( q \) (at a specific frequency and cache size), we define the relative incremental improvement (RII) as

\[
\text{RII}(T)_{q+r} = \frac{T_{q+r} - T_q}{T_q}.
\]

If we use the conventional definition of speedup

\[
S_{q+r} = \frac{T_q}{T_{q+r}},
\]

then it can be seen that

\[
S_{q+r} = \frac{1}{1 - \text{RII}(T)_{q+r}}.
\]

For the sequence of optimizations given by \( B + W + MO + MP + LP \), it can be easily seen that the time with all features, namely \( T_{B+W+MO+MP+LP} \) is given by subtracting from \( T_B \), the time at the base configuration, values of \( \text{RII}(T)_{B+W+MO+MP+LP} \), \( \text{RII}(T)_{B+W+MO} \), \( \text{RII}(T)_{B+W+MO+MP} \), \( \text{RII}(T)_{B+W+MO+MP+LP} \), and \( \text{RII}(T)_{B+W+MO+MP+LP} \).

In the RII definition above, incremental optimizations to the system in a given order are seen as providing incremental speedups; RII values for energy can be defined similarly. Note that these RII values are sensitive to the ordering and they are shown in Figure 6.4 for SMV-U and SMV-O. These RII values indicate the wider data paths have greater impact for SMV-O than for SMV-U and they are the most energy-efficient. The open page policy (MO) improves energy efficiency at all frequencies while improving time significantly at higher frequencies. Additionally, the relative impact of MP is more marked in certain frequency ranges than in others. More significantly, all optimizations benefit SMV-O more than SMV-U.
Fig. 6.4. Relative incremental improvements (RII) in execution time (top) and energy (bottom) for SMV-U (left) and SMV-O (right). Plots correspond to the incremental addition of memory subsystem optimizations in the order: the base configuration (B), adding wider data paths (B+W), an open page policy (B+W+MO), a memory prefetcher (B+W+MO+MP), and an L2-prefetcher (B+W+MO+MP+LP).
Observe that the plots in Figure 6.4 indicate significant differences in RII values between SMV-U and SMV-O when the same feature is added to same configuration. These differences are clearly from the difference in the tuning levels of the two codes. Consequently, if RII values are used to select features, then the outcome can be impacted by the choice of kernel. For example, if only features yielding execution time RII values greater than .1 are to selected, then values in Figure 6.4, indicate that ‘LP’ will not be selected if SMV-U is used. Likewise, the RII values indicate greater improvements in time and energy with MO for SMV-O. Consequently, design choices could be different depending on the choice of the kernel used in the evaluation.

6.2.3 Results for Equake-A and Equake-AT

We now summarize performance and energy results when evaluations are performed using Equake-A and Equake-AT. Figure 6.5 shows the plots for execution time and energy for on the base configuration and when all optimizations are applied, i.e., for the configuration $B + W + MO + MP + LP$. Observe that the optimizations result in very small improvements in the execution time of Equake-A while Equake-AT benefits to a larger degree. As mentioned earlier, the SMV kernel could potentially be tuned further to include features for increasing data-reuse and locality of access. Nonetheless, the slight tuning does enable the code to utilize the memory subsystem optimizations in larger measure than Equake-A.

Next, we compute RII values for execution time using Equake-A and Equake-AT; these are shown in Figure 6.2.3. There is significant divergence in the RII values for the two codes; for example, LP has a negative impact for Equake-A while Equake-AT benefits to a small degree. Once again, these differences arise from the differences in the level of tuning of the SMV kernel representing the dominant computation in equake.
Fig. 6.5. Execution time (left) and energy (right) for equake on the base configuration ‘B’ and with all optimizations ‘all’. Equake-A indicates equake with its original SMV and Equake-AT is equake with a slightly tuned SMV.
Fig. 6.6. Relative incremental improvements (RII) in execution time for Equake-A with original SMV (left) Equake-AT with a slightly tuned SMV (right). Different plots correspond to the addition of memory subsystem optimizations, starting with the base configuration (B), adding wider data paths (B+W), an open page policy (B+W+MO), a memory prefetcher (B+W+MO+MP), and an L2-prefetcher (B+W+MO+MP+LP).

6.3 Chapter Summary

In this chapter, we have considered several memory subsystem enhancements for energy-aware high performance sparse computations. These optimizations benefit both the tuned and natural forms of sparse matrix-vector multiplication, a function common to many codes in an emerging class of scientific applications. For the untuned kernel, SMV-U, optimizations improve time relative to the base configuration at 1GHz by over 30% starting at frequencies as low as 500MHz with energy reductions of over by over 80%. Corresponding figures for the tuned
form, SMV-O, are in excess of 40% for time and 85% for energy. Similarly, the tuned form of equake also benefits more from the optimizations. Thus considerable savings in energy are possible with improvements in execution time if memory subsystem optimizations are used in conjunction with DVFS and low-power modes of caches. Not surprisingly, tuned kernels realize greater benefits from the memory subsystem optimizations. However, the differences in relative incremental improvements from the same set of optimizations, independent of frequency or cache size effects, are considerable depending on the level of code tuning. These differences indicate a distinct interplay between code and system optimizations.

There is increasing interest in such sparse applications because they allow scaling to solve larger and more refined models. However, tuned implementations representing the types of optimized codes found in high-performance scientific software [70, 112, 157, 189] are typically not available in current benchmark suites. We conjecture that performance analysis with such tuned codes in addition to more traditional benchmarks, will enable a more comprehensive assessment of architectural optimizations for future high end systems. In the next chapter, we consider interactions between code tuning, matrix properties, and hardware attributes towards energy-efficient scientific computing in light of Conjugate Gradient solver.
Chapter 7

Memory Subsystem Impacts on the Energy Efficiency of Conjugate Gradient Algorithm

Performance and efficiency of parallel sparse scientific solvers involved in modeling and simulations depends to a large extent on their local single-node performance and optimizations [48, 50, 53, 119, 187, 191, 194]. In Chapter 4, we discussed the impacts of workload distribution on performance of a parallel direct sparse Cholesky solver. Then, in Chapter 6, we considered interactions between power, performance, and software optimizations for the sparse matrix-vector multiplication kernel, which is a part of many Krylov subspace solvers. We showed that with proper hardware features and software tuning, energy consumption of the sparse matrix-vector multiplication kernel can be significantly reduced and its performance maintained or improved.

In this chapter, we consider the performance, power, and energy characteristics of a widely used sparse solver in scientific applications, namely the Conjugate Gradient (CG) sparse solver. Our goal is to increase the power and energy efficiency of the CPU and the memory subsystem for a CG-based sparse solver without degrading the performance and potentially even improving it. Toward this goal, we begin by specifying a processor and memory system similar to that in the BlueGene/L, the first supercomputer developed for power-aware scientific computing [184]. We use cycle-accurate simulations with SimpleScalar [186] and Wattch [22] to study performance and power characteristics when low power modes of caches are used with Dynamic
Voltage and Frequency Scaling (DVFS) [38]. Towards improving performance, we explore the impact of memory subsystem optimizations in the form of level 2 cache prefetching (LP) and memory prefetching (MP). Finally, towards the improved evaluations of architectural features of future systems, we consider how interactions between the level of code tuning, matrix properties, and memory subsystem optimizations affect performance and power profiles.

Our overall results are indeed promising. They indicate that tuned CG codes on matrices representative of the majority of PDE-based scientific simulations can benefit from memory subsystem optimizations to execute faster at substantially lower levels of system power and energy. Our results, at the level of a specific solver and architectural features, further the insights gained at a higher system level through earlier studies on the impact of DVFS on scientific computing workloads on clusters [56, 108].

The remaining part of this chapter is organized as follows. In Section 7.1, we describe our base RISC PowerPC architecture, our methodology for evaluating performance, power, and energy attributes through cycle-accurate simulation, and our memory subsystem optimizations. Section 7.2 contains our main contributions characterizing improvements in performance and energy and differences in improvements resulting from the interplay between code and architectural features. Section 7.3 contains concluding remarks. The CG algorithm is described in more detail in Section 3.1.3.

The results in this chapter have been published in the following paper: “Conjugate Gradient Sparse Solvers: Performance-Power Characteristics” [140], co-authored by Konrad Malkowski, Ingyu Lee, Padma Raghavan, and Mary Jane Irwin, and published in the Proceedings of the 20th IEEE International Parallel and Distributed Symposium, IPDPS06, Second High-Performance, Power-Aware Computing Workshop.
7.1 Modeling Performance and Power Characteristics

In this section, we describe our methodology for emulating architectural features and evaluating their performance and energy characteristics. The power and performance of sparse kernels are emulated by SimpleScalar3.0 [25] and Wattch1.02d [22] with extensions to model memory subsystem enhancements. We use Wattch [22] to calculate the power consumption of the processor components, including the pipeline, registers, branch prediction logic. Wattch does not model the power consumed by the off-chip memory subsystem. We therefore developed a DDR2 type memory performance and power simulator for our use. More details of our power modeling through Wattch can be found in [138].
7.1.1 Base Architecture

Our base architecture has two floating point units and two integer ALUs. Each FPU has a multiplication/division module and modules for other arithmetic and logic; thus, the system is capable of issuing 4 floating point instructions each cycle. The data path width of the CPU is 4
instructions and data paths between memory and L3 cache are 128 bit wide with cache lines of 128 bytes, i.e., 16 double precision operands or 32 integer operands.

We model a cache hierarchy with three levels on chip, including a 32KB data/32KB instruction level 1 cache (L1), a 2KB level 2 cache (L2), and a 4MB unified level 3 cache (L3). Wattch is configured to model only two levels of cache, but we added new functions to model our hierarchy. We used CACTI [22] modes for a 4MB SRAM L3 cache. We operate the SRAM L3 at system frequency and voltage levels when we consider different frequency-voltage pairs to simulate the effects of utilizing DVFS.

The specifications of the off-chip memory are particularly important for characterizing sparse matrix computations. We assume a DDR2 type memory power model with our base architecture to start with a lower-power configuration than what is possible with standard DDR type memory. We model nine 256MBit x 8 chips to provide 256MB of memory with specifications obtained from Micron Technology Inc. data sheets [98] with memory operating at 266MHz. We verified our approach by observing that the timing delays for our system with the CPU clock frequency at 700MHz are similar to those reported in the literature [7, 184].

Our base architecture is similar to the configuration in the BlueGene/L [184] based on the PowerPC440 embedded core. The main difference is that we only model a single-core because Wattch currently does not allow for the simulation of more than one thread of execution. This is still realistic, because the BlueGene/L processor is not designed to work in the SMP mode; Level 1 caches are not coherent and the recommended usage for the second core is as a communication co-processor. Furthermore, if both cores are fully utilized for computation, the demands on the memory subsystem will only increase, potentially magnifying the interactions between code
tuning and memory subsystem optimizations. More details of our system and its relation to the processor in the BlueGene/L can be found in [138].

7.1.2 Memory Subsystem Optimizations

Starting with the base architecture, henceforth denoted by ‘B,’ we consider memory subsystem optimizations that can potentially benefit sparse applications. These optimizations focus on prefetching to mask the latencies of memory access; such prefetchers have been considered in other contexts [44, 104, 129, 146]. Details include:

- Memory prefetching (on or off), stride-1, at the memory controller, labeled ‘MP’; this feature reduces the effective latency of memory access depending on data access patterns. It is modeled by adding a prefetch buffer to the memory controller in the form of a 16 element table where each element holds a cache line of 128 bytes. On a load, if data is not available in the prefetch buffer, a new entry is allocated in the table using a full LRU replacement policy, and the memory controller does a prefetch for the next cache line upon completion of the original load; see Figure 7.2. We model the power consumed by our prefetch buffer as the cost of operating a small 16 entry, direct mapped cache with a 128 byte cache line due to problem with CACTI.

- Level 2 cache prefetching (on or off), stride-1, denoted by ‘LP’; this feature can benefit codes with locality of data access but poor data re-use in caches. When a read at address \( a \) completes, a read is scheduled for address \( a + L2 \) cache-line size. The extra energy consumption is modeled as second cache access.
7.1.3 Emulating Low-Power Modes of Caches and Processors

Future architectures are expected to exploit power-saving modes of caches and processors. These include sleepy or low-power modes of caches [101] where significant fractions of the cache can be put into a low power mode to reduce power. Additionally, power can be decreased by Dynamic Voltage and Frequency Scaling (DVFS) [38] where the frequency and the voltage are scaled down. We now model how these features can be utilized by sparse scientific applications if they are exposed in the Instruction Set Architecture (ISA).

Fig. 7.2. A memory prefetcher implementation.
The 4MB SRAM L3 cache in our base architecture, typical of many high-end processors, accounts for a large fraction of the on-chip power. To simulate the potential impact of cache low power modes of future architectures on CG codes, we perform evaluations by considering the following L3 cache sizes: 256Kb, 512KB, 1MB, 2MB and 4MB.

To model DVFS [38], we consider eight CPU frequencies with corresponding nominal $V_{dd}$ voltages. The frequency-$V_{dd}$ pairs we used are: 300MHz-0.46V; 400MHz-0.66V; 500MHz-0.77V; 600MHz-0.84V; 700MHz-0.88V; 800MHz-0.93V; 900MHz-1.03V; and 1000MHz-1.20V.

7.2 Empirical Results

In this section, we evaluate the impact of using low power modes of caches, and low power modes of the processor using DVFS, along with memory subsystem optimizations for sparse CG solvers with different levels of tuning typical of scientific applications. We are interested in both the effect of memory subsystem optimizations and software optimizations.

To demonstrate the interplay between code optimizations, matrix properties, and architectural optimizations, we consider four forms of the CG algorithm. The first corresponds to a natural un-optimized implementation (CG-U) of a plain conjugate gradient algorithm. The second corresponds to its optimized version (CG-O) using SPARSITY [95]. For each, we consider a sparse matrix typical of scientific applications, namely bcsstk31 (Properties of this matrix are listed in Table 3.2.) from a well known test collection [73]. The matrix is considered with two different orderings (i) using RCM, with corresponding CG forms labeled CG-O RCM, CG-U RCM, and (ii) using a random ordering (CG-O Random, CG-U Random). The latter are used primarily to discuss how our CG codes relate to a well known benchmark code, the CG code from the NAS benchmark (CG-NAS) [10] and to thus characterize how features of
CG-NAS impact power and performance evaluations. In all cases, we evaluate the performance and power characteristics for a set of 25 complete CG iterations; i.e., 25 iterations of the for loop in Figure 3.3. This is consistent with the number of CG iterations used in CG-NAS. This is sufficient to capture the behavior of typical applications which require several hundred iterations for convergence to a solution of desired accuracy [14].

To evaluate performance, we use execution time (in seconds) as our metric. Other metrics include the average system power consumed the processor and memory (in Watts) and energy (in Joules), computed as the product of the system power and execution time.

In figures in this section, plots for the base architecture are labeled ‘B’, plots for base architecture with either a memory or an L2 prefetcher are labeled ‘MP’ and ‘LP’ respectively, and plots for the base architecture enhanced by both features are labeled ‘LP+MP’. The x-axis shows some or all frequencies in the range 300MHz to 1000 MHz (1GHz) while the y-axis is used to show execution time, power and energy values.

### 7.2.1 Power and Performance Tradeoffs Using DVFS and Low Power Modes of Caches

We begin by evaluating the power and performance tradeoffs for CG when DVFS and the low power modes of caches can be utilized. Figure 7.3 shows execution time and average power consumption of a CG based solver for the base architecture at the original frequency of 1 GHz and when the frequency is scaled down to 600 MHz for a series of cache sizes 256KB, 512KB, 1MB, 2MB and 4MB. The plots on the left are for the unoptimized version of the CG (CG-U) algorithm. The plots on the right show results for an optimized implementation of the CG solver (CG-O), utilizing register blocking and loop unrolling. In both cases, we consider the bcsstk31 matrix with an RCM ordering typical of scientific applications.
Fig. 7.3. Execution time (top) and average system power (bottom) for CG-U (left) and CG-O (right) for bcsstk31 with RCM. Values are shown at two frequencies, 600MHz and 1000MHz with 5 cache sizes: 256KB, 512KB, 1MB, 2MB and 4MB.

Observe that the cache size has very little impact on the performance of both CG-U and CG-O while significantly reducing the power. This is primarily because SMV-U, SMV-O and CG inherently have low cache reuse. Consequently, utilizing low power modes of caches can lead to significant decreases power consumption without adverse impacts on performance. This effect is more pronounced at higher frequencies. Observe also that as DVFS is utilized and both
frequency and voltage are scaled down, the execution time increases and the power decreases for both CG-U and CG-O. Observe also that power reductions are significantly more than the increase in execution time when both are measured relative to their values at 1000MHz. This is as expected; a linear scaling down of the frequency and voltage results in cubic improvements in power, while the CPU cycle time (and hence the execution time) is in inverse linear proportion to the frequency.

Henceforth, we will only report results for system configurations with the smallest cache size of 256KB. This will represent the impact of using low power modes of caches as we continue with for CG-U (left) and CG-O (right) run on RCM ordered bcsstk31. Results relative to CG-U run on bcsstk31 ordered with RCM at 1000MHz, 4MB cache.

### 7.2.2 Impact of L2-Cache and Memory Prefetchers

We now consider in detail the impact on performance and power when memory and L2 prefetchers are used with DVFS and sleepy modes of the level 3 cache. Figure 7.4 shows relative values of the execution time, power, and energy for unoptimized CG-U (top) and optimized CG-O (bottom). The values are relative to those observed for CG-U for the base architecture, ‘B’ at 1000MHz with a 4MB level 3 cache, set at 1. Thus, presented values capture the impacts on power, performance and energy from the combined effects of hardware and software optimizations. Observe that values smaller than 1 indicate improvements in each metric while values greater than 1 indicate degradations. We provide relative values for frequencies of 400, 500, 600 and 700 MHz for the smallest level 3 cache size of 256KB and values at the reference point, i.e., ‘B’ at 1000 MHz and 4MB L3 cache. At each frequency, we provide grouped bars representing the addition of either memory prefetching (MP), or L2 prefetching (LP), or their combination
(LP+MP). In all cases, we consider 25 CG iterations for the bcsstk31 matrix with an RCM ordering, typical of scientific applications as discussed earlier.

Observe that for all frequencies reported in this figure, the optimized version CG-O has significantly better performance than CG-U. We will report results for the optimized version first, and then return to the unoptimized version.
Consider the performance and power metrics for CG-O along the bottom half of Figure 7.4. At 400MHz with either LP or both LP and MP active, we achieve performance better than that of CG-U at the reference point of 1000 MHz with over 60% savings in power. At 500MHz, we improve the performance by 23% for LP + MP with approximately 55% saving in power and over 62% savings in energy. More importantly, at all frequencies with both LP and MP, executions times for CG-O are better or equal to values observed not only with respect to CG-U at the reference point but also with respect to CG-O at the reference configuration.

Similar improvements are also observed for the unoptimized CG (CG-U) shown at the top half of Figure 7.4. However, because register blocking and loop unrolling are not present in CG-U, performance improvements from the use of LP and MP are smaller and the frequencies at which performance is better than or equal to that at the reference point are higher. For example, at 600MHz, with LP + MP, there is a 10% reduction in execution time, with approximately 51% reduction in power and over 55% for energy savings.

These results show that significant power and energy savings are possible along with performance improvements when relatively simple memory subsystem optimizations are used with DVFS and power saving modes of the L3 cache.

7.2.3 Impact of Sparse Matrix Properties

Our results so far indicate that the level of tuning of the CG codes affects the degree to which the same set of architectural features, including low power modes of caches, DVFS, and memory subsystem optimizations impact the performance and power metrics.
Fig. 7.5. Relative execution time (left column) and relative power (right column) for (i) CG-O on bcsstk31 with RCM, (ii) CG-O bcsstk31 with a random ordering, (iii) CG-U on bcsstk31 with a random ordering and (iv) the NAS benchmark CG (CG NAS) on a matrix with dimensions similar to bcsstk31. Values are relative to those of CG-U on bcsstk31 with RCM for the base architecture at 1000MHz and a 4MB L3 cache. At frequencies lower than 1000MHz, values are shown for the base and LP, MP, and LP+MP configurations with a 256KB L3 cache.
In all our experiments thus far, CG-U and CG-O were executed on the same matrix which had been reordered using RCM to increase locality of access in the source vector during matrix-vector multiplication. It is a property of the sparse matrix which allows such reorderings to be effective in clustering nonzeros about the diagonal and it arises from the fact that PDE-based simulations based on finite-difference or finite-elements matrices give rise to matrices with localized connectivity [67]. Sparse matrices from other applications can have random connectivity. More importantly, even sparse matrices from PDE-based applications may occur with a random ordering although they can be reordered to improve locality.

One commonly used version of CG code is included in the popular NAS benchmark [10] suite. That benchmark’s default data set consists of a highly unstructured, near random sparse matrix. Using the NAS CG code together with three forms of our CG-U and CG-O codes we now evaluate the sensitivity of performance and energy optimization opportunities to the matrix properties and level of software tuning.

Figure 7.5 shows relative execution time and power for CG-O on bcsstk31 and an RCM ordering, CG-O on bcsstk31 with a random ordering, CG-U on bcsstk31 with a random ordering and the CG from NAS on a matrix with dimensions close to bccsstk31. Once again, the values are relative to CG-U on bcsstk31 with an RCM ordering on the base architecture at 1000MHz and a 4MB level 3 cache. Values at lower frequencies than 1000MHz are shown for B, LP, MP, and LP+MP configurations with a 256KB level 3 cache.

Even from a cursory look at Figure 7.5, it is apparent that the sparse matrix structure is critical. When it is near random, there are significant performance degradations despite memory subsystem optimizations at all frequencies. Observe that performance and power profiles of the
NAS CG is very similar to that of CG-U on \texttt{bcsttk31} with a random ordering and in dramatic contrast to that for CG-O on \texttt{bcsttk31} with RCM. The best performing configuration for CG-O RCM, namely (LP + MP), significantly degrades performance for all instances operating on random matrices. For CG-O Random, CG-U Random and CG-NAS, the (LP + MP) configuration performs approximately 20% worse than the base configuration operating at the same frequency. Not surprisingly, CG-O performs worse than CG-U with a random ordering for \texttt{bcsttk31} because explicit zeroes added for 2x1 blocking simply increase the number of arithmetic operations without benefiting from the locality of access in the source vector. Although power savings are possible from DVFS and smaller level 3 cache sizes, these come at significant performance degradations for randomly ordered matrices. Consequently, in evaluating the impact of architectural features on power and performance, it is important to use codes representative of the application space of high-performance scientific computing.
7.2.4 Improving Performance and Power

Fig. 7.6. Relative values of Time, Power and Energy are shown with respect to CG-U for bcsstk31 with RCM on the base architecture at 1000MHz with 4MB level 3 cache. Values for CG-U are shown to the left for all frequencies with 256KB level 3 cache on the base architecture, except for the reference point at 1000MHz, shown with 4MB level 3 cache. Values for CG-O are shown to the right for the optimized system with LP + MP at all frequencies with a 256KB level 3 cache. Values smaller than 1 indicate improvements.

In the scientific computing community, the primary focus traditionally has been on performance to enable scaling to larger simulations. However, trends in the microprocessor design community indicate that scaling to future architectures will necessarily involve an emphasis on power-aware optimizations. Consequently, it is important to consider sparse computations which occur in a large fraction of scientific computing codes. Such sparse computations
significantly different from dense benchmarks [50, 124] which can effectively utilize deep cache hierarchies and high CPU frequencies. We use CG as an example of such sparse computations to demonstrate that it can benefit from low power modes of the processors and caches and simple memory subsystem optimizations to reduce power and improve performance.

Figure 7.6 shows relative values of execution time, power and energy for CG-U and CG-O on bcsstk31 with RCM. The reference point corresponds to the highest frequency 1000MHz for the base architecture with a 4MB level 3 cache, with CG-U; this is the rightmost point in the plots to the left in Figure 7.6. The plots for CG-U (to the left) correspond to the base architecture without any memory subsystem optimizations, while the plots for CG-O (to the right) correspond to the LP+MP configuration, at different frequencies with a 256KB level 3 cache.

The plots in Figure 7.6 show that power savings from DVFS and low power modes of caches can be realized only at the expense of performance degradations for CG-U on the base architecture. However, the plots for CG-O clearly indicate that a tuned code with memory optimizations can utilize low power modes of caches and DVFS for significant improvements in both execution time and power. Starting at 400MHz CG-O with LP + MP shows performance improvements while reducing power by approximately 60%. Maximum energy savings of 64% are observed at 500MHz with 22% improvements in execution time and 55% improvements in power. Energy savings close to this value can also be observed at 700MHz, with greater improvements (62%) in execution time.
7.3 Chapter Summary

In this chapter, we showed how low power modes of processors and caches can be used with simple memory subsystem enhancements to improve the performance and energy characteristics of a sparse CG solver. Furthermore, we showed that a tuned code can benefit to a greater degree than an untuned code from the same set of architectural enhancements. For example, the best improvement in execution time observed for the CG-U code with both prefetchers active (LP + MP configuration) is 45% at 1000MHz. The tuned code CG-O shows a 55% reduction of execution time at the same configuration and processor frequency. Likewise with respect to power, CG-U shows reductions of 52% at 600MHz with a 256KB level 3 cache, with and both prefetchers active (LP + MP); the corresponding value for the CG-O at the same configuration is 60%.

Our results indicate that power-aware scientific computing with sparse matrices can be achieved without performance degradation. However, interactions between the matrix properties, level of tuning in the code and architectural optimizations impact performance and power profiles. Consequently, we conjecture that using representative tuned codes in addition to more traditional benchmarks will enable a more comprehensive assessment of architectural optimizations for future high end systems. In the next chapter, we introduce the load miss predictor, and evaluate its impacts on energy efficiency and performance of sparse scientific codes.
Chapter 8

Hiding Cache Miss Penalties with the Load Miss Predictor

In the previous chapters we observed that memory accesses latency [164] presents a serious performance bottleneck for many sparse scientific applications. These applications have limited data locality and reuse, and typically do not benefit from deep cache hierarchies and high clock frequencies. In this chapter, we develop a unique memory architecture with a load miss predictor (LMP) for improving the performance of sparse scientific applications by reducing effective load latencies. We also demonstrate an LMP design that benefits sparse scientific applications without adversely impacting the performance of applications with significant data reuse and locality. Finally, we show how our LMP can be combined with Dynamic Voltage and Frequency Scaling (DVFS) [38] to reduce power while improving the performance of sparse scientific applications yielding significant reductions in energy consumption.

The remainder of this chapter is organized as follows. Section 8.1 introduces sparse applications and codes used in this chapter. Sections 8.2, 8.3 and 8.4 contain our main contributions. We develop our LMP architecture and discuss its implementation in Section 8.2. We present our methodology and empirical results respectively in Sections 8.3 and 8.4. In Section 8.5, we discuss related work. We end with concluding remarks in Section 8.6.

This chapter is based on a research paper published in the Proceeding of of the 21st IEEE International Parallel and Distributed Symposium, IPDPS’07, High-Performance, Power-Aware Computing Workshop, titled “Load Miss Prediction for Energy-Aware High Performance
8.1 Sparse Scientific Computing Applications

In this chapter, we consider the following representative sparse codes to evaluate effectiveness of our LMP. We use the NAS benchmarks MG and CG with a “W” (workstation) size workload [10]. Both benchmarks use a matrix-vector multiplication routine that accounts for more than 90% of their execution time. Therefore, in our tests, we additionally use a sparse matrix-vector multiplication kernel from Sparsity (SMV) [96]. In our experiments we consider two versions of SMV, namely SMV-U and SMV-O, which are described in Section 3.1.1. We report on the performance of SMV-U and SMV-O on the following four representative sparse matrices from structured mechanics: bcsstk31, fdm2, qa8fm, and msc23052. These matrices were introduced in Section 3.2, and their properties are listed in Table 3.2. In our experiments, these matrices were reordered using the Reverse Cuthill McKee [68] scheme to improve the locality of access in the source vector [140] as is commonly done for tuned scientific codes.

8.2 Designing a LMP

Modern CPUs have deep cache hierarchies to mask the large latencies of accessing the main memory for codes with data reuse. However, when there is poor data reuse, as is the case in many sparse scientific codes, these elaborate cache systems actually add to the access latency as the caches are inevitably and unproductively accessed before each main memory access. As an example, for level 1 cache, level 2 cache, and main memory latencies of 1, 19 and 122 cycles respectively, bypassing the L2 cache on a miss could save 16% of the total latency. This effect
will be further magnified with deeper cache hierarchies, especially with NUCA [113] caches where the longest bank miss latency can be significantly greater than the average cache latency. The rest of this section presents the design and implementation of our Load Miss Predictor to decrease penalties of such inevitable load misses.

![Load Miss Predictor design](image)

Fig. 8.1. The Load Miss Predictor design.
8.2.1 LMP design and operation

Figure 8.1 shows our proposed design, where the 32KB level-1 data cache (L1) is split into two portions, labeled regular and bypass. The regular portion is a low latency, highly associative cache typical of modern processors, 16KB in size, with a 32 byte long cache line. This portion of the L1 data cache has a direct connection to the level-2 (L2) cache. The bypass portion of the data L1 is also 16KB, and is directly connected to both the L2 cache and to the memory controller. We call it the Bypass L1 cache. Data is provided to the bypass cache from the main memory directly, and the L2 cache serves as its victim cache. The Bypass L1 cache has a cache line width of 128 bytes, which is the cache-line width of the L2 cache as well as the amount of data transferred from the memory in one request. The cache-line width is based on the 128-bit wide memory bus and SDRAM transfer protocol [97, 98]. A prediction mechanism is added to this structure (the LMP Table in Figure 8.1). A detailed discussion and experimental justification for these design choices is presented in Section 8.4.

When a load instruction is executed, its data request is sent to both the Regular L1 and Bypass L1 caches. When both caches miss the LMP table is consulted. The PC address of the load instruction is looked up in the LMP table much as in branch prediction technology. The LMP then determines whether the load should bypass (predicted miss), or proceed through the main cache hierarchy (predicted hit). When there is insufficient history for prediction, such as in newly encountered loads, they are directed through the main cache hierarchy, i.e., are predicted hits. Our predictor tracks only the history of loads that miss in both Regular L1 and Bypass L1.

On predicted hits the load proceeds through the normal cache hierarchy. If the data is not found, the predictor encodes an incorrect prediction (predicted hit, cache miss) in the table.
Otherwise, the predictor encodes a correct prediction in the table (predicted hit, cache hit). As long as the data is found anywhere in the cache hierarchy it will not be brought into the *Bypass L1*.

When data requested by a load instruction at a particular PC address is repeatedly not found in the regular cache hierarchy, that load instruction will be marked as suitable for cache bypassing (predicted miss). On a predicted miss, a parallel request for an early load will be issued to the main memory by the *Bypass L1* and to the L2 cache by the *Regular L1*. If the L2 cache responds with data before the main memory does, the memory access will be cancelled, the load will be marked (predicted miss, cache hit), and the data will be stored in the *Regular L1* cache. Any data returned by the main memory as a result of the early load will be ignored. On the other hand, if the L2 does not respond with data, the main memory read access will be already underway, thus reducing the memory latency. The corresponding data will be stored in the *Bypass L1* cache, and the load will be marked (predicted miss, cache miss). Both portions of L1 cache are checked for data on a store instruction to ensure cache consistency. As a result, a store will always proceed through the path used for loading the corresponding address.

Additional modifications to the processor are required to add the LMP table and bypass cache. Each load instruction has to be accompanied by either the complete PC address or a unique identifier in order to correctly map it into the predictor table. The bus connecting the L1 cache to L2 cache requires additional wires to communicate hits and misses to the load predictor. An additional wire is also required to control the tag and data look-up schemes in the cache hierarchy. In addition we need a mechanism such that during bypassed loads only cache tags are accessed, and data arrays are accessed only upon a hit. Finally, an additional bus is required to connect the *Bypass L1* to the main memory controller to initiate an early load.
Ideally, the prediction mechanism should be 100% accurate. However, in practice the design of the LMP must reflect a trade-off between accuracy, speed and hardware complexity. We therefore utilize a small 2-level predictor (which we call L2H8) tracking 1024 independent load instructions, with an 8 bit hit/miss history per instruction (requiring 256 entries in the second level table). We believe this configuration offers a good trade-off between accuracy and hardware costs.

An important question is what to do with data displaced from the Bypass L1 cache. We considered many replacement policies for data displaced from Bypass L1 and chose a policy, where all cache lines that are displaced from the Bypass L1 are written to the L2 cache whether clean or dirty (“all replaced victim”). Other possible policies are to write displaced dirty data directly to main memory (“no victim”), or to place only dirty displaced cache lines in the victim cache (“only dirty victim”). LMP design tradeoffs are discussed further in Section 8.4.

We expect that our LMP design will have no negative impacts on clock cycle time, because our modifications to the CPU pipeline only require carrying additional bits, and thus a wider instruction. Our LMP could possibly introduce a 1 cycle penalty to L1 cache misses followed by access to L2 or main memory. However, we do not consider this penalty to be significant given the long latencies of main memory access.

8.3 Methodology

In this section, we describe our methodology for obtaining cycle accurate performance and power numbers through simulation. We utilize SimpleScalar3.0d [25] and Wattch1.02d [22] with modifications to model power consumption and performance impacts of the LMP. Additionally, we developed an accurate model for the power and performance of DRAM type main
memory by using data from Micron [97, 98] to simulate the DDR2 DRAM memory behavior and power consumption at a 333MHz bus clock frequency.

<table>
<thead>
<tr>
<th>Setting</th>
<th>Value</th>
<th>Setting</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>-fetch:ifqsize</td>
<td>4</td>
<td>-cache:dl1</td>
<td>128:32:8:1</td>
</tr>
<tr>
<td>-fetch:mplat</td>
<td>3</td>
<td>-cache:dl1lat</td>
<td>1</td>
</tr>
<tr>
<td>-fetch:speed</td>
<td>1</td>
<td>-cache:dl2</td>
<td>1024:128:4:1</td>
</tr>
<tr>
<td>-bpred</td>
<td>bimod</td>
<td>-cache:dl2lat</td>
<td>19</td>
</tr>
<tr>
<td>-bpred:bimod</td>
<td>2048</td>
<td>-cache:il1</td>
<td>128:32:8:1</td>
</tr>
<tr>
<td>-decode:width</td>
<td>4</td>
<td>-cache:il1lat</td>
<td>1</td>
</tr>
<tr>
<td>-issue:width</td>
<td>4</td>
<td>-tlb:itlb</td>
<td>16:4096:4:1</td>
</tr>
<tr>
<td>-issue:inorder</td>
<td>false</td>
<td>-tlb:dtlb</td>
<td>32:4096:4:1</td>
</tr>
<tr>
<td>-issue:wrongpath</td>
<td>true</td>
<td>tlb:lat</td>
<td>30</td>
</tr>
<tr>
<td>-commit:width</td>
<td>4</td>
<td>-res:ialu</td>
<td>3</td>
</tr>
<tr>
<td>-ruu:size</td>
<td>32</td>
<td>-res:imult</td>
<td>3</td>
</tr>
<tr>
<td>-lsq:size</td>
<td>8</td>
<td>-res:memport</td>
<td>1</td>
</tr>
<tr>
<td>-res:fpalu</td>
<td>3</td>
<td>-res:fpmlt</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 8.1. Wattch configuration for base system representative of a modern 4-way issue out-of-order processor.

We start with a base processor architecture (henceforth referred to as B) that is similar to the processor and memory subsystem in one node of a modern superscalar cluster. Table 8.3 shows the configuration of our SimpleScalar processor. This is representative of modern processors, in particular RISC type processors like the PowerPC which are typically capable of issuing 4 instructions per cycle.

We model our processor power consumption in 130 nanometer technology. We obtain the 130 nanometer technology numbers by scaling [19] the power numbers already present in Wattch. We assume a DVFS processor running in the frequency range of 1300–2000MHz with
$V_{dd}$ between 1.07 – 1.5 Volts. The L1 cache latency is 1 cycle, L2 cache latency is 19 cycles, and memory latency is based on Micron, Inc. datasheets, and varies with CPU frequency [97, 98]. We modified Wattch to accommodate the additional functionality required by our architecture.

We use both PISA and Alpha configurations of SimpleScalar to enable the study of a larger set of codes. We use the Alpha configuration to evaluate the performance of SPEC benchmarks and PISA for the NAS benchmarks and sparse codes discussed in Section 8.1. We ran all pre-compiled benchmarks from SPEC [20], for which we had a compiler and whose code ran on SimpleScalar. We also used SimPoints [26] to reduce the execution time of these simulations. The NAS benchmarks and SMV kernels were executed until completion, after being fast-forwarded past the initialization stage.

### 8.4 Experimental Results

In this section, we first discuss the LMP design trade-offs and their impacts on performance, power and energy consumption for the SPEC 2000 and sparse benchmarks. We then consider the impact on performance and power of using our final LMP design on the SPEC 2000 benchmarks. We next consider these metrics in greater detail for the sparse codes described earlier in Section 8.1. Unless otherwise stated, we report values of time, power and energy for each code relative to the base configuration B running at 2000MHz with a 512KB L2 cache. Values at the base configuration B are set to 1. Thus, for configurations with LMP (denoted as B+LMP), values that are less than 1 indicate relative improvements while values greater than 1 indicate degradations.
8.4.1 LMP design trade-offs

Key trade-offs during the LMP design stage included the sizes of the Regular L1 and Bypass L1 caches, the L2 cache size, the predictor mechanism type and size, and the Bypass L1 data replacement policy.

To address the Bypass L1 data replacement policy trade-offs we created an “ideal” predictor. Our “ideal” predictor was a 2-level predictor, with 1024 16-bit hit/misss history entries in the first table. The second table consisted of 65536 bimodal counters. Instead of splitting the 32KB L1 data cache of the base configuration in two 16KB halves (half for the Bypass L1 and half for the Regular L1), in our “ideal” LMP predictor we made both parts 32KB in size. As mentioned in Section 8.2, we considered three options for replaced data movement: “no victim”, “only dirty victim” and “all dirty victim”. The “all replaced victim” policy showed no performance degradations for SPEC 2000 benchmarks, as shown in Figure 8.2, and was chosen as the data replacement policy for our LMP design.
Fig. 8.2. Impact of LMP bypass cache replacement policy on SPEC2000 performance with “Ideal” LMP. “No victim” (left bar), “Only dirty victim” (middle) and “All replaced victim” (right). Values are relative to base configuration B at 2000MHz set to 1.

The “ideal” predictor described earlier has a significant hardware footprint, large power consumption, and it would add a significant delay to the memory subsystem. Thus we consider the performance of simpler predictor designs, including the “bimodal” predictor consisting of a 2048-entry table of bimodal counter, and the “L2H8” predictor. The “L2H8” predictor consists of a small 2-level predictor tracking 1024 independent load instructions, with an 8 bit hit/miss history per instruction (requiring 256 entries in the second level table). Our experiments indicate that both designs consume less power than the “ideal” predictor. Their performance is compared
against the “ideal” predictor in Figure 8.3. The “L2H8” predictor closely follows the performance of the “ideal” predictor at much lower design complexity and it is thus our choice for the LMP.

Fig. 8.3. Impact of “ideal” (left bar), “bimodal” (middle bar) and “2 level 8 bit (L2H8)”(right bar) predictors on execution time of SPEC 2000 benchmark suite at 2000MHz. Values are relative to base configuration B at 2000MHz set to 1.

Changing the Bypass L1 and Regular L1 cache sizes from 32KB to 16KB, so that the total number of data cache bytes (Bypass L1 + Regular L1) is equal to the base configuration, does not have a significant impact on the average performance of the SPEC 2000 benchmarks. The geometric mean of performance improvements, as shown in Figure 8.4, are between 2.5%
to 4% for 32-1 (32KB regular, 1KB bypass) and 32-32 (32KB regular, 32KB bypass) cache configurations respectively, with the smaller cache configurations consuming less power and energy. For example, the 16-16 cache configuration consumes 7% less power and 5% less energy, than the 32-32 configuration, relative to a base configuration.

The observed performance fluctuations for the mgrid, crafty and gzip benchmarks were investigated further. The crafty and gzip benchmarks are cache bound codes, and decreasing the amount of Regular L1 available resulted in more L1 cache misses. The SPEC mgrid code follows the same algorithm as the NAS MG benchmark, which is investigated together with our sparse codes in the following paragraphs.

Fig. 8.4. Impact of the Regular L1 and Bypass L1 data cache sizes on execution of spec2000 benchmarks. 16-16KB means 16KB Regular L1 and 16KB Bypass L1. Values are relative to unoptimized base configuration B at 2000MHz set to 1.
The change in the LMP cache configuration has a varied impact on performance of our selected sparse codes. Results in Figure 8.5 (top subfigure) show that NAS MG is very sensitive to the cache size of the Bypass L1 cache. As the size decreases from 32KB to 16KB the performance improvement reduces from 18.3% to 13.1%. Further reduction of Bypass L1 cache size to 4KB and 1KB results in significantly smaller performance gains. The CG and SMV kernels are not as affected by the Bypass L1 cache size changes (less than 2%). The average power consumption for the sparse codes is shown in Figure 8.5 (bottom subfigure). The 32-32 cache configuration consumes nearly 5% more power on average than the 16-16 configuration, and 2% more energy on average, thus making the 16-16 configuration a good compromise for both SPEC and sparse applications.
Fig. 8.5. Impact of Regular L1 and Bypass L1 cache size on performance (top subplot) and power (bottom subplot) of sparse benchmarks. X axis labels can be decoded in the following way: 16-16 means 16KB Regular L1 and 16KB Bypass L1. Values are relative to base configuration B at 2000MHz set to 1.
Figure 8.6 shows the impacts of our final LMP configuration (“L2H8” predictor, 16-16 cache configuration, “all replaced victim” replacement policy) on performance (left), power (middle) and energy (right) consumption of SPEC 2000 benchmarks. Relative values are shown for $(B + LMP)$ compared to base $B$ at 2000MHz. The addition of the LMP increases the performance by an average of 2.9%. *Mgrid*, which is representative our sparse matrix application domain, benefits at significantly higher level of 12%.

The gains in performance come at the expense of increased hardware cost and thus system power. Observe that adding the LMP results in a 5% increase in energy and 7% increase in average system power. These increases are the result of more frequent memory accesses, the addition of the 16KB *Bypass L1* cache, prediction hardware, concurrent accesses to both *Bypass L1* and *Regular L1* caches, and concurrent accesses to L2 cache tag arrays during early memory reads.

Figure 8.7 shows the impact of using the LMP on performance, power and energy for the NAS CG, MG codes and SMV-U, SMV-O. Observe that SMV-U benefits most with a 16.7% reduction in time, with SMV-O a close second with 15.6%. These improvements in execution time are primarily from decreases in the average load latencies (labeled as LSQ lat.) by 14.2%. On average, the performance of these four codes improves by 14%. However, power increases substantially by 13.6% on average, while the energy decreases slightly by an average of 2.3%, due to shortened running time but higher power consumption.
Fig. 8.6. Impact of our final LMP configuration on execution time, average power and energy consumption of SPEC 2000 benchmark suite at 2000MHz. Values are relative to base configuration B at 2000MHz set to 1.

8.4.2 Co-optimizing performance, power and energy for scientific computing

We now consider in detail how our final LMP can be used to improve the performance of sparse scientific codes, and how power can be reduced by using dynamic voltage and frequency scaling (DVFS) [38] with the LMP. Use of DVFS with the base architecture alone results in reduced performance at significant system power and energy savings. The latter can be used to potentially offset the power and energy increases when the LMP is added while retaining its performance benefits.
In Figure 8.8, we show relative values for time, power and energy for our four sparse benchmarks. The LMP offsets the performance degradation from frequency scaling down to the extent that even at 1600MHz show performance improvements. Corresponding power and energy savings are in excess of 20%. At 1400MHz performance improvements for SMV-U and SMV-O are in the range 5 to 10% with power reductions over 35% and energy reductions of over 40%. We summarize these power performance tradeoffs in Figure 8.9 indicating the
mean relative values of time, power and energy for the base configuration $B$ with DVFS and the enhanced configuration with LMP with DVFS (denoted by $B+LMP$).

![Graph](image)

Fig. 8.8. Impact of LMP and DVFS on execution time (left subplot), power (middle subplot) and energy (right subplot) of SMV-U (1st bar), SMV-O (2nd bar), CG (3rd bar) and MG (4th bar) sparse benchmarks. Results are shown relative to the base configuration $B$ at 2000MHz set to 1.

These plots clearly indicate that without the LMP, it is not possible to save power without degrading performance. Observe that $B+LMP$ configuration at 1800MHz reflects nearly equal
improvements in performance and power for over 15% improvements in energy. Without the LMP, performance degrades by nearly 7% at equivalent energy levels.

![Graph showing average relative time, power, and energy](image)

**Fig. 8.9.** Relative average time, power and energy, using DVFS with (solid lines) and without (dashed lines) LMP for 1300MHz to 2000MHz. The average is taken over the results for SMV-U, SMV-O, MG and CG). Results are shown relative to the base configuration B at 2000MHz.

### 8.5 Related research

The importance of reducing memory access latencies is reflected in a rich set of earlier results towards faster loads [32, 148, 188, 196].
Tyson [188] et al. considered techniques for improving effective cache hit rates by using static and dynamic techniques to determine whether data should be cached or not, thus reducing cache pollution. This work is similar to our approach in the sense that they use prediction mechanisms to avoid caching data that is not being reused. A key difference is that we use the prediction technology to lower the effective load latency by bypassing the cache, however the bypassed data can stay resident in the *Bypass L1* cache for future reuse.

Memik et al. [148] propose schemes to predict quickly if data at a given address is already stored in cache or not. Thus, they provide a faster but possibly less accurate look-up mechanism at every cache level. If the data is determined to not exist at a particular level a full access is avoided at that level, which results in reduced memory access latencies. Our approach is different in that it is not data address centric, but rather instruction centric to predict a complete bypass of the cache hierarchy based on past hit/miss history.

Yehia et al. [196] present a scheme which is similar in the sense that they use branch prediction for load instructions. However they focus on pairs of load instructions representing indirect memory references. Upon detection of such a pair and a miss prediction, they replace (through hardware) the second load by a new “load squared” instruction in which indirect address resolution is performed by memory-side logic. Such an approach may not be effective for the sparse scientific codes of interest to us. This is primarily because such codes avoid indirect memory references and most loads are performed to consecutive memory locations as discussed in the introduction of this chapter.

Our work is also peripherally related to the following. Software cache bypassing schemes were discussed by Chi [32]. Energy savings for scientific applications were considered by Choi et al. [38] and Freeh, et al. [60]. Additionally, prefetching techniques were discussed by Lin et
al. [129]. Effects of prefetchers on performance and power of sparse applications were investigated by authors in [140].

8.6 Chapter Summary

In this chapter, we developed the load miss predictor (LMP), and showed that it is effective at reducing memory access latencies for sparse scientific applications with limited data locality and reuse. On average, the LMP reduces the execution time by 14% at a cost of 13.6% increase in system power. A fraction of the improvements in time can be traded off for substantial power and energy savings by using DVFS. For example, by decreasing frequency to 1800MHz from 2000MHz system power is reduced by approximately 7.3% and energy is reduced by 17.3%, while maintaining 8.7% improvements in time. These results are promising and they indicate the potential for power-aware design optimizations suitable for high performance scientific computing. A natural extension of this work would be to evaluate the impact of combining our LMP with other static and dynamic schemes [148,188]. In the next chapter, we consider a phase adaptive, power-aware hardware feature selection scheme for sparse scientific applications.
Chapter 9

Phase-Aware Adaptive Hardware Selection
for Energy-efficient Scientific Computing

Scientific applications used in modeling and computational discovery typically consist of many independent “building” blocks: input/output routines, pre-conditioners, solvers, and others [8, 11, 86, 160]. Each of these components can have different hardware requirements and corresponding energy demands. In Chapters 6, 7 and 8, we investigated various single processor power and performance optimizations for sparse scientific applications. However, our work to this point has been focused on power and performance optimizations for a single component of a typical large-scale parallel application: the main compute kernel. We also focused only on manually selecting configurations for the architecture to match the application requirements.

In this chapter, we consider an adaptive hardware selection scheme for applications consisting of multiple phases. Our scheme uses dynamic voltage and frequency scaling (DVFS) and data prefetchers to adapt architecture performance and energy consumption to application requirements. Our scheme relies on the fact that certain freely available hardware counters are good indicators of performance and application phases. For example, the lowest level cache misses are good indicators of whether a section of code is memory or cache bound. Our scheme automatically activates prefetchers whenever an application enters a memory bound phase. DVFS is then used to perform power optimizations based on number of instructions issued.
To quantify the benefits of our approach, we use cycle-accurate simulation using SimpleScalar [25] and Wattch [22]. The results collected indicate that our adaptive hardware selection heuristic is effective at reducing the power consumption of sparse scientific applications while maintaining the performance of an unoptimized system. Resulting power and energy savings depend on underlying application properties. For codes exhibiting significant memory bound phases, our technique results in 39% power and 37% energy savings.

The rest of this chapter is organized as follows. Section 9.1 describes our adaptive hardware selection technique. Section 9.2 presents our methodology, and Section 9.3 gives experimental data. Related work is discussed in Section 9.4, and Section 9.5 contains our concluding remarks. The scientific applications used in this chapter are described in detail in Chapter 3.

This chapter is an extended version of a poster paper titled “Phase-aware Adaptive Hardware Selection for Power-efficient Scientific Computations” [139], co-authored by Konrad Malkowski, Padma Raghavan, Mahmut Kandemir, and Mary Jane Irwin, published in the Proceedings of International Symposium on Low Power Electronics and Design 2007 (ISLPED 2007).

9.1 Phase Adaptive Switching Scheme

Many applications can be divided into distinct hardware phases, an observation that had been utilized in prior studies [78, 109]. For scientific applications, we typically recognize communication and computation phases. This partitioning can be successfully used to reduce power consumption in certain parallel applications without significant performance reductions [109]. For certain applications, the computational phase can be further divided into two broad categories: cache bound phases and memory bound phases.
The Preconditioned Conjugate Gradient (PCG) solver, which we introduced in Section 3.1.4, is a good example of a scientific application that exhibits both cache bound and memory bound behavior. The PCG solver consists of two general phases: preconditioner generation and system solution. When incomplete Cholesky fill drop level 0 preconditioner [173] is used, the preconditioner construction stage of the PCG solver is cache bound. On the other hand, the right hand side solution is a memory bound phase, with significant performance benefits from prefetchers and opportunities for power/performance optimizations. Figure 9.1 shows plots of CPI, L3 cache miss rates and average memory latency for the PCG solver. The distinction between the cache bound and memory bound phases of the PCG solver is clear in all three metrics. The preconditioner construction is cache bound and has low CPI, L3 cache miss rate and low memory latency. The right hand side solution is memory bound, with high CPI, L3 miss rate and high memory latency.
Fig. 9.1. Impact of PCG phases on hardware metrics. CPI (top), L3 cache miss rate (middle) and memory latency (bottom) traces show that the preconditioner construction and system solution have different hardware profiles. Metrics indicate that preconditioner construction is cache bound and the system solution is memory bound. These two phases are clearly identifiable in each plot.

Memory bound phases, where the execution time is dominated by the access latency to the main memory, offer ample opportunities for power and performance optimizations and trade-offs. Data prefetchers can be utilized in such phases to reduce the average memory access latency by prefetching data into cache hierarchy, at the cost of a higher overall system power consumption. At the same time DVFS can be used to reduce both CPU supply voltage and frequency for
significant power and energy reductions while maintaining or exceeding the performance of the unoptimized system [140, 141].

We take advantage of this observation, and propose a mechanism for phase-aware adaptive hardware selection featuring DVFS and prefetchers, where DVFS and prefetchers are applied selectively to memory bound phases. We use two prefetchers, a level 2 cache prefetcher (LP) and memory controller data prefetcher (MP). Both are implemented as stride-one prefetchers, with software configurable prefetch direction. We base our switching heuristic on the observation that the lowest level cache misses (in case of this work, the L3 cache) are a good indicator of whether the present application phase is memory or cache bound. Additionally, we use the fact that CPI is a good approximation of the system performance. Our proposed switching heuristic consists of 4 states labeled B, B RND, MP + LP, Power Optimization (300 · · · 1000 MHz), where:

- **B** – Cache bound phase - Base configuration, prefetchers are deactivated, and the system is running at 1000MHz.

- **B RND** – Memory bound phase, random memory access pattern, no prefetch, 1000MHz - Random access pattern is determined by the increase in average memory access latency (for streaming codes prefetchers reduce the average memory latency).

- **LP + MP** – Memory bound phase, non-random memory access pattern - Prefetchers on, processor is running at 1000MHz.

- **Power optimization** – Memory bound phase. Prefetchers are active and appropriate DVFS gear is selected. Frequency ranges from 300 · · · 1000MHz.
Figure 9.2 has the diagram of the finite state machine representing our model, with switching conditions as annotations on the edges.

We partition the execution time of applications into $10^7$ cycle profiling windows. At the end of each profiling window, our finite state machine (fsm) determines whether a change in application phase occurred, and takes the appropriate action. We use the L3 miss rate of .4 (which worked well in our experiments) as the threshold for determining whether we are in cache bound (L3 miss rate < .4) or memory bound phase (L3 miss rate > .4). Detailed description of switching heuristic now follows.

The fsm is initialized to the B state. If, at the end of the current profiling window, the lowest level cache miss rate increases above the threshold, then the fsm activates the prefetchers and transitions to the LP + MP state. Additionally, a flag is raised indicating that the system just switched in order to avoid oscillation. At the end of the next profiling window, the fsm compares the average memory latency in current and previous windows. If the average memory latency increases, then we transition to the B RND state (prefetchers result in performance degradation for codes exhibiting random memory access pattern [140]). Otherwise, our heuristic performs power optimization using DVFS.

During power optimization the switching heuristic considers the number of instructions executed during current and previous profiling windows (both windows were executed at the same frequency), and selects the new DVFS setting using Equation (9.1). In current implementation of our switching heuristic, the power optimization is performed once per transition from a cache bound phase to a memory bound phase. After the power optimization, the system does not adjust DVFS setting until the code becomes cache bound again. We made this decision based on the observation that scientific application phases tend to be long and show consistent memory
Fig. 9.2. Finite state machine (fsm) implemented by our adaptive hardware. Switching conditions are annotated on the edges. Only one transition is made per profiling window. The only exception is the power optimization state which sets the DVFS gear. Most of the state transitions are determined by whether the L3 miss rate falls below the “threshold”, which is set to .4. However, transition to B RND from LP + MP state is determined by the average load latency. If the load latency in LP + MP is greater than in B, then random access pattern is predicted. When power optimization is performed the new DVFS gear is determined by the number of instructions issued in current and previous windows.
access behavior (a natural extension of this idea would be to perform DVFS adjustments at the end of each profiling window as necessary). Finally, when the application data access pattern becomes cache bound and the lowest level cache miss rate falls below the threshold, the system switches to the B configuration and the CPU frequency is set to 1000MHz.

\[
F_{new} = \left[ F_{old} \times \frac{\text{Inst. executed in current window}}{\text{Inst. executed in last window} \times 100} \right] \times 100
\]  

(9.1)

When a random access memory pattern is detected, our switching heuristic enters the B RND state, where the system CPU frequency is set to 1000MHz and the prefetchers are deactivated. The fsm detects the random access pattern based on the increase in the average memory latency after activating prefetchers.

The system switches into the B configuration once the lowest level cache miss rate falls below the designated threshold and it is determined that the data access pattern is cache bound again.

9.2 Methodology

We use SimpleScalar [25] and Wattch [22] cycle-accurate simulators to simulate and test our phase adaptive hardware selection heuristic. We model a quad-issue out-of-order PowerPC like RISC processor with 2 floating point arithmetic units, 2 integer arithmetic units, 3 level cache hierarchy, 128 bit memory bus, and default SimpleScalar settings for remaining processor parameters. We configured Wattch to simulate power consumption of a processor in 130nm technology, using scaling numbers based on [19]. We assume a processor that can run between 300 and 1000MHz, with corresponding supply voltage of .47V for 300MHz and 1.2V for 1000MHz.
Fig. 9.3. High level view of the proposed hardware implementation of the switching heuristic. Our implementation requires only a few additional hardware counters and the switching logic for the control hardware.
To simulate the impact of varying CPU frequencies, our memory subsystem latency is estimated based on Micron, Inc. datasheets [97] for 266MHz DRAM chips. As a result, our main memory latency varies with the change of the CPU frequency.

Our processor cache configuration shows resemblance to BlueGene/L [184] processor which we used as a reference point due to its low power design. The L2 cache is a 2KB (smaller than the L1 cache) and serves as a prefetch buffer for the L1 cache in order to mitigate the access latencies to the larger L3 cache. The L1 cache has an access latency of 1 cycle, L2 cache has an access latency of 6 cycles and the L3 cache 19 cycles. The latencies are cumulative and so the total cost of accessing the L3 cache is $26 = 19 + 6 + 1$ including L3 cache hit, L2 cache miss and L1 cache miss.

We have additionally modified Wattch and SimpleScalar to include memory controller data prefetcher (MP), a level 2 cache prefetcher (LP), processor DVFS scaling and memory subsystem power estimator (based on Micron data sheets). We model both prefetchers as stride-one prefetchers, with the default stride direction towards higher memory addresses. The direction of the prefetch can be changed at runtime with a system call placed inside the user application.

Our adaptive scheme can be implemented in both software and hardware. Figure 9.3 shows a simplified design diagram for the proposed hardware implementation. We estimate that required modification to the hardware would involve the addition of only 10 counter registers and two integer dividers, as well as a comparator to perform the necessary calculations. Software implementation would involve fewer additional hardware counters and could utilize processor ALU; however, it could also potentially have negative performance impacts.

We assume no penalties for switching the DVFS gears in our simulations. This is because such switchings would not happen frequently in scientific codes due to their long lasting phases.
We further assume that our switching heuristic (when implemented in hardware) would cause no additional increase in the cycle length of the processor. We believe that the phase adaptive hardware selection could happen in parallel with the normal execution of the processor, the exception being the switching of the DVFS gears when the processor would have to be stalled.
9.3 Empirical Results

Fig. 9.4. Impacts of adaptive hardware and automatic phase detection on hardware metrics. The left figure in first row shows effects of PCG phases on CPI and L3 miss rate without adaptive hardware, while the right figure shows the impacts of PCG phases on CPI and L3 miss rate with adaptive hardware. The left and right figures in the second row show CPI and L3 miss rates for NAS MG without and with adaptive hardware selection present. For both applications, the L3 miss rate is unaffected by the use of prefetchers and DVFS, while the CPI for the memory bound phases decreases when the phase adaptive hardware selection is present. Thus the L3 miss rate is a reliable indicator of whether phase is memory or cache bound.
For evaluating our adaptive hardware architecture and impacts of the finite state switching machine, we use the MG CLASS A (MG-A) from the NAS benchmark suite and an in-house Preconditioned Conjugate Gradient (PCG) solver which we wrote in C. Both codes are discussed in more detail in Chapter 3.

We consider three variations of the PCG solver. The first is a standalone PCG with one preconditioner construction and one right hand solution (rhs). For this variation the preconditioner construction dominates. The second variation represents applications where each preconditioner construction is followed by many right hand side solutions and these solutions dominate the execution time of the solver. We use PCG with solutions for 20 rhs. The third variation represents applications where the underlying system is evolving and involves multiple preconditioner construction, each followed by multiple right hand side solutions. We simulate these applications and their interaction with our adaptive hardware using PCG with 4 matrices, with 1 preconditioner construction and 20 rhs solutions for each matrix.

We begin by analyzing the impact of phase aware adaptive hardware selection on CPI and L3 cache miss profiles for the PCG and NAS MG. Plots in Figure 9.4 show these profiles for the unoptimized system (plots on the left in rows one and two) and the system with phase aware adaptive hardware selection (plots on the right in rows onw and two). All plots show that the L3 cache miss rate is not significantly affected by the presence of adaptive hardware selection, and thus a good base for our selection heuristic. The CPI is a good indicator of performance for a given configuration when the adaptive hardware selection is not present. It can also be used to measure performance gains due to the addition of prefetchers. However, combined use of DVFS and prefetchers significantly impacts the CPI profile of memory bound phases, making this metric unsuitable for standalone phase detection.
The plots in Figure 9.5 show that our phase aware adaptive hardware selection technique adapts well to the changing requirements of our applications. Each of these applications has a varying mix of cache bound and memory bound phases. In all cases our phase aware adaptive hardware successfully uses opportunities for power and performance optimizations and adjusts the CPU frequency and prefetchers accordingly, as can be seen from Figure 9.5. We also note that our approach adapts well to the application behavior. For example, usage patterns of prefetchers change, depending on the application being executed. Beginning with the top row, the left plot represents the standalone case with 1 preconditioner construction and 1 right hand side solution. The second plot shows the case with 1 preconditioner construction and 20 right hand side solutions, while the third plot (bottom row, left) shows the effects of solving four matrices requiring 4 preconditioner constructions and 20 right hand solutions for each matrix. Finally, the fourth plot shows how the adaptive hardware adapts to NAS MG.
Fig. 9.5. Response of adaptive hardware to changes in application phases. Starting with the top row, the leftmost figure shows prefetchers and frequency settings for 1 preconditioner construction and 1 rhs. The second figure shows prefetchers and frequency settings for 1 preconditioner construction and 20 rhs. Third figure (bottom row, left) shows prefetchers and frequency settings for 4 problem solutions involving 1 preconditioner construction and 20 rhs solutions each. The fourth figure shows prefetchers and frequency settings for NAS MG class A benchmark. In all figures our adaptive hardware selection, adjusts well to changing application profiles.
Figure 9.6 presents the performance, power and energy results obtained using our approach, normalized with respect to the case where the entire application is executed in the B state (i.e., 1000MHz, no prefetchers). We see from these results that these sparse codes run with our phase aware adaptive hardware selection technique exhibit competitive performance behavior to the unoptimized system while reducing overall power and energy consumption. We observe that the amount of power savings is dependent on the fraction of the code that is memory bound and can be power optimized. For codes where the memory bound phase dominates we observe significant power and energy savings approaching 39% and 37%, respectively. For the PCG code with 1 rhs and 1 preconditioner construction, the cache bound preconditioner construction dominates and we observe only 14% power and 14% energy savings. On average, the performance of the system is not negatively affected, and in some cases even improves beyond the performance of the unoptimized system. These performance gains are caused by the way our scheme selects the new frequency setting in Equation (9.1). There are only 8 DVFS gears in our design, and our adaptive scheme selects the lowest power gear that will also maintain the performance of the unoptimized system.

Our adaptive hardware selection technique is also energy-efficient when compared to a performance-optimized configuration with the processor at 1000MHz with both the prefetchers turned on. As shown in Figure 9.7 energy savings approach 30%, power savings approach 50%. As expected our adaptive technique does not perform as well as the performance-optimized configuration. Again, energy and power savings vary depending on the workload phase composition. To sum up, we can conclude that our adaptive approach leads to both power and energy savings, over both the base case and fully performance-optimized case. These settings are due to the
fact that our approach selectively uses prefetchers and DVFS, considering the characteristics of individual program phases.

Fig. 9.6. Our adaptive hardware and switching heuristic have no significant impact on performance of test sparse applications (left bar group). The resulting power (middle bar group) and energy (right bar group) reductions range from 10% for PCG to 34% for NAS MG to 39% for PCG 20 rhs.
Fig. 9.7. Our adaptive hardware technique is more energy-efficient than performance optimized processor at 1000MHz with prefetchers turned on. However, the significant power and energy savings come at lowered performance (left bar group). The resulting power (middle bar group) and energy (right bar group) reductions approach 50% and 30% respectively.

9.4 Related Work

The majority of research on power aware scientific computation has focused on utilizing low power modes of caches, utilizing DVFS and load imbalances or link shut-down in network interfaces.

Freeh et al. [109] focused on adaptively using low power DVFS gears for power aware clusters based on communication and workload imbalances. Hsu et al. [89] proposed a single
processor DVFS scaling model based on sustained MIPS rate and least squares analysis. Ge et al. [63] explored various methods for DVFS control. Gulin et al. [31] demonstrated that DVFS can be successfully applied to tree-based scientific applications based workload estimates. Son et al. [180] further expanded on [31] by combining power savings resulting from DVFS with the power savings associated with link power scaling. Choi et al. [37] proposed an on-chip DVFS scaling technique based on workload decomposition into on-chip and off-chip parts, and utilizing the CPI metric. Cho et al. [36] investigated optimal DVFS settings for minimizing both CPU and memory energy consumption.

Detection of phase changes in applications, phase length prediction, application of DVFS low power gears to appropriate phases, and associated tradeoffs were investigated by Isci et al. [100]. They observed that switching to low power DVFS modes made sense only when the predicted length of phase and power savings outweighed the costs of switching in terms of power and time penalties.

To our knowledge, this is the first work to consider optimizing for power and performance by adaptively selecting hardware enhancements such as prefetchers and utilizing them with DVFS to match the demands of the application, while improving power efficiency. We believe that further enhancements to this work can be achieved by considering additional hardware enhancements.

9.5 Chapter Summary

In this chapter, we proposed a phase aware adaptive hardware selection technique which employs DVFS and prefetchers, with the goal of reducing power consumption of scientific applications while not degrading the performance. Our switching technique relies on metrics that can
be easily obtained from performance counters available in many modern processors: lowest level cache miss rate and number of instructions issued. We expect the hardware implementation of our adaptive heuristic to have a small hardware cost. Furthermore, we expect it not to impact the processor cycle length. Our technique successfully reduces the power consumption of the test applications by as much as 39% and energy consumption by as much as 37%, without negatively impacting the application performance. When compared against performance-optimized processor our technique reduced power consumption by nearly 50% with energy savings of 30%. In the next chapter, we move to the multi-core processor design space, and propose a new tree-based non-uniform access latency cache architecture.
Chapter 10

Tree Cache - A Novel Approach to Non-Uniform Cache Architectures for CMPs

Increasing process miniaturization, wire delay and power dissipation have brought about a shift from a single super-pipelined, super-scalar processor design to a multi-core processor design where performance is provided by multiple smaller, simpler cores [2, 4, 105]. As the technology evolves, chips with as many as 80 cores are prototyped [1, 3], and we expect the number of cores in the future chip multi-processors (CMPs) to grow even higher. The increase in the number of cores results in high demand for the limited off-chip memory bandwidth. To offset the impact of these demands, designers are considering CMPs with larger on-chip caches. However, as the cache capacity grows larger, and the transistors become smaller, the access latencies from cores to data locations grow proportionally to their distance due to increasing wire delays [12, 15–17, 33, 34, 114].

The problem of long wire delays in large L2 caches and increasing uniform cache access latencies has been studied extensively in the context of single core and multi-core processor architectures. A promising approach to reducing the average access latency is to divide caches into smaller blocks (called banks), each with different access latency, from a given processor core’s perspective. This type of cache architecture is called the non-uniform access cache architecture (NUCA) [114]. The key challenges in the NUCA design are in locating the data and ensuring that the frequently accessed data is always close to the requesting processor, while at
the same time facilitating capacity sharing. Various mechanisms such as migration [114], replication [16, 198, 199], cache-cooperation [28, 29], and others [33, 34, 93, 103, 168, 190, 197] are proposed to address these challenges. However, most of these designs have so far focused on a relatively small number of cores (4 to 8), and it is not clear if they are scalable to the CMP systems consisting of tens or hundreds of cores.

We consider a multi-core system where each core is connected to the L2 cache banks via a NoC (see Figure 10.1 in Section 10.3). The key challenges in this architecture are data placement and data lookup. The two best known NUCA implementations are the static NUCA (S-NUCA), where data is placed and located in a bank based on a subset of address bits, and the dynamic NUCA (D-NUCA), where the data can be located and placed in one of many banks [114]. In this chapter, we focus mainly on the problem of data lookup in NUCA type on-chip caches, and propose a novel cache architecture with a mix of best properties of S-NUCA and D-NUCA with broadcast, i.e. limited use of links and banks to access the data, flexibility and adaptivity to application needs, high performance, low energy costs and low network contention.

We propose and evaluate a new cache architecture with an inclusive, tree-based, hierarchical directory (T-NUCA). In this design, the cache structure is divided into the hierarchical directory and the data storage. The data storage cells are laid out in the standard mesh layout and each bank is associated with a local directory node. The hierarchical directory is laid out as a tree. At each higher level of the hierarchical directory the number of directory nodes is cut in half, their capacity is doubled, and each parent directory node contains all the data tags of its children. Finally, at root level there is only one directory node, that contains tags for all data resident in the cache. As a result, the root level has a global view of the cache contents.
In this chapter, we also present an evaluation of the trade-offs between the S-NUCA, D-NUCA and T-NUCA designs. We consider two run-time execution scenarios: single-program and multi-program. In the single-program scenario our proposed T-NUCA architecture outperforms the D-NUCA by 27%, at a 35% overhead in number of transistors used, while reducing the network link use by 48%, energy consumption by 93%, and EDP by 95%. Relative to S-NUCA, our T-NUCA reduces the execution time by 38%, increases the link use by 161% and energy consumption by 32% due to directory overheads, and reduces EDP by 19%. In the multi-program scenario, our T-NUCA design reduces the execution time by 17%, energy by 93% and EDP by 94% over D-NUCA. Relative to S-NUCA, our T-NUCA design reduces the execution time by 71%, energy by 58% and EDP by 88%.

The remainder of this chapter is organized as follows. Section 10.1 provides background on S-NUCA and D-NUCA. In Section 10.2, we describe S-NUCA and D-NUCA cache configurations used as reference points in evaluating the performance and energy efficiency of our T-NUCA. In Section 10.3, we provide details of our T-NUCA design, qualitatively compare our T-NUCA design to S-NUCA and D-NUCA reference configurations. Section 10.4 presents our experimental setup and tools used to evaluate S-NUCA, D-NUCA and T-NUCA, and Section 10.5 presents and discusses our results. Section 10.6 discusses related work on the NUCA design. Finally, in Section 10.7, we summarize our results and provide concluding remarks.

10.1 Non-Uniform Cache Architecture

The S-NUCA cache architecture was originally proposed by Kim et al. [114] as a way of handling the problem of increasing wire latencies in large cache structures for single-core processors. The main idea was to divide the large cache structure into multiple smaller banks,
and allow the access latency to depend on the distance between the bank that contains the data, and the processor requesting the data. The banks were selected based on the address of the requested data. On average, this design decreased the cache access latency over the uniform cache access architectures.

Kim et al. further enhanced the S-NUCA design by dividing the cache into bank-ways, and each bank-way into several banks [114]. In this design (the D-NUCA), the bank-way is selected based on a subset of data address bits. Since data can anywhere inside of a bank-way, a lookup method is required. Kim et al. considered three lookup approaches: (i) sequential, where each bank further away from the core is accessed only after previous bank was checked for data; (ii) broadcast, where all banks in the bank-way are accessed at the same time to look for data; and (iii) hybrid, where certain banks are accessed in parallel, while the remainder of the banks are accessed in sequential fashion. The trade-off between these three methods is one between energy efficiency versus performance. These dynamic placement methods can be further enhanced by introducing data migration, where on each access data is brought closer to the requesting processor (data always stays within the bank-way).

The challenges of long wire delays, data placement and data lookup in NUCA caches become more complex in the chip multi-processor (CMP) design space. Now, in addition to deciding on whether the cache should employ (i) static mapping vs. dynamic mapping; (ii) migration vs. no migration or (iii) replication vs. no replication, we have an additional design choice of whether the L2 cache should be shared or private. The consequences of these design choices have been investigated in prior CMP NUCA papers [29,33,77,131,198], with a majority of proposed designs using a shared L2 cache, with variations of dynamic placement and lookup. This is a trend we follow in our design as well.
10.2 The Reference S-NUCA and D-NUCA Configurations

In this section, we discuss our two reference NUCA configurations: the S-NUCA, and the D-NUCA with broadcast and data migration. These two architectures represent the opposing ends of the NUCA design spectrum. S-NUCA is a conservative design which reduces the energy consumption and the network use at the cost of performance, while D-NUCA is an aggressive design which trades energy consumption and network use for high performance.

S-NUCA uses the static data placement and lookup method, where data banks are chosen based on a subset of bits in the data address. In our S-NUCA design, the selected bank also handles coherency when data is shared between multiple cores. We employ the MESI protocol [162], and use either direct invalidations to, and writebacks from specific L1 caches, or use broadcast invalidations sent to all L1 caches. We do not implement any migration in the S-NUCA design. Due to static data placement in cache, the S-NUCA design is energy efficient, and has a low network link usage. However, it does not guarantee high performance due to lack of migration (data can be far away from the requesting core, many cores can be competing for the same bank or cache-way set).

Our D-NUCA design is based on Kim et al.’s D-NUCA design [114]. In our D-NUCA design we focus on achieving maximum performance. Therefore, we make the following design choices. Our L2 cache consists of one bank-way, which is shared among all the cores. We use dynamic mapping of data to banks. In particular, on a cache miss, a bank is randomly selected to hold the data. On each access to the L2 cache, a broadcast is issued to locate the bank in which the data is located. The miss is determined after a time-out equal to twice the longest access latency. During the broadcast all banks are activated and checked for the requested data.
On each read access, the data is migrated one bank closer to the core that requested it. This migration is performed in the X direction first and then in the Y direction. We do not implement replication in order to keep the coherence protocol simple, though our approach can be extended to accommodate replication as well. Finally, we employ the MESI protocol [162] for L1 and L2 cache coherency. All decisions regarding the MESI protocol are handled at the level of the L2 bank containing the data.

In our base D-NUCA configuration, we rely on a network broadcast to determine the location of data. While this design choice is simple and performance efficient, it has a significant cost in terms of energy as well as link utilization. On each L2 cache access, we utilize majority of links in the network and access each bank of the cache. In our simulation environment we idealize the hardware design and assume no network contention. An alternative design choice uses on a sequential lookup scheme. In such a scheme the banks are activated one at a time in order of increasing distance from the core requesting the data. A bank is accessed in this order only after a miss in the previously accessed bank is declared. We do not consider this scheme, because while it does result in lower energy consumption on average it does so at the cost of decreased performance [114].

### 10.3 A Hierarchical, Tree-based Directory NUCA Architecture (T-NUCA)

In this section, we present the main contribution of this chapter, the T-NUCA architecture. Figure 10.1 depicts our target NUCA configuration on a CMP system with 64 cores. In this design, the L2 cache is divided into 64 banks, and each core is directly connected via a router to one of the banks. The only way to access data located in other banks is via the mesh-based NoC interconnect. The challenge in this design is in determining where to place the data in
cache and how to locate the data on an L1 cache miss. One approach is to use static mapping of data to banks (S-NUCA), another approach is to have a random data placement in banks, and a broadcast lookup (D-NUCA). Other approaches to solving this problem limited the number of processors considered to 4 - 16 cores [16, 29, 34, 77, 93, 103, 199].

![Fig. 10.1. Conceptual design of a CMP with D-NUCA. P = processor core, and C = L2 cache bank.](image)

We propose a novel approach to solving the data lookup problem in a D-NUCA. Instead of relying on the NoC to locate the data by broadcasting requests to all banks, we consider adding an inclusive, tree-based, hierarchical directory for data lookups and updates. In our design, we separate the data lookup stage of the cache access from the data access stage. As a result, when we search for the required data, only directory tags are accessed. Consequently, in our design,
data accesses and tag lookups do not contend for the same network resources, thereby helping on-chip parallelism as well. The conceptual design of our T-NUCA is shown in Figure 10.2.

Fig. 10.2. **Left:** Conceptual 2D T-NUCA cache design showing the additional tree directory network laid out in the chip. **Right:** Conceptual 3D T-NUCA design showing how directories are laid out and connected to each other creating a tree. This virtual tree captures how a data lookup proceeds in the L2 cache. At each level the total number of bits used by directories is constant for the total directory cost of $C \times N \times \log_2 N$. 
10.3.1 T-NUCA Design

In our design the data arrays are arranged as a 2-D mesh and connected via a NoC. Each data array is associated with one local directory node located at the lowest level of the tree (level 0). At each successive level up in the tree, the number of separate directory nodes is cut in half, and their capacity is doubled. Since our hierarchical directory is inclusive, all addresses at the lower levels of the tree are replicated at the higher levels of the tree. Each parent directory node in the tree contains all the tags stored in its children. This replication continues recursively until the root directory node is reached, at which point there is only one directory node containing information about every address currently resident in the L2 cache. In our implementation, we double the capacity of higher level directory nodes by doubling their associativity.

10.3.2 T-NUCA Bit Overheads

In this section, we discuss the bit overheads associated with our T-NUCA design. Consider an L2 cache with $N$ banks laid out in a $\sqrt{N}$ by $\sqrt{N}$ grid, where each bank of the cache has $C$ bits dedicated to address tag storage. Our hierarchical directory is built as a binary tree and thus requires $\log_2 N$ levels ($\log_2 \sqrt{N}$ in each grid direction). Each level of the tree requires $C \times N$ bits to store all address tags present in cache. Thus, the bit cost of our hierarchical directory is $C \times N \times (\log_2 N + 1)$, compared to a cost of $C \times N$ for storing all address tags in an S-NUCA or D-NUCA. The cost of storing the data arrays in all three architectures is $A \times N$, where $A$ is the number of bits dedicated to storing all cache lines. Both $A$ and $C$ are functions of $N$, and we expect them to grow at least linearly with the number of cores on chip. We also expect the ratio of $A$ and $C$ to stay fixed as the numbers of cores and banks increase.
The bit overhead associated with the hierarchical directory in T-NUCA is expressed by
\( \Phi = \frac{C \times N \times (\log_2 N + 1) + A \times N - C \times N - A \times N}{C \times N + A \times N} \), which simplifies to: \( \Phi = \frac{C \times \log_2 N}{C + A} \); can be approximated by \( \Phi = \frac{C \times \log_2 N}{A^2} \). More specifically, consider a 16MB cache with 64 banks, 64 byte wide cache lines, 4 way set-associativity and 48-bit addresses. In such a design, each cache set consists of 2,048 bits for data storage, and 32 bits for each address tag (128 bits total). Thus a 16MB D-NUCA (S-NUCA) requires 134 million bits for data and 8 million bits for address tags. Our T-NUCA design requires the same number of bits for data storage as D-NUCA, and 59 million for the directory (due to directory overheads). The total bit overhead of our T-NUCA is 35% over D-NUCA with comparable cache data capacity. As we expect the ratio of \( A \) and \( C \) to remain fixed as the cache capacity grows to accommodate more cores, we observe that the tree directory overheads will slowly increase with each doubling of the number of banks in cache.

We expect the tree directory to grow in height by one level with each doubling of number of banks, thus growing at the rate of \( \log_2 N \), where \( N \) is the number of banks.

The growth of bit overhead of T-NUCA can be reduced by changing the design of the hierarchical tree directory. For example, instead of using a binary tree we could use a quad tree. With a quad tree design, our 16MB cache would require only 25 million bits for the hierarchical directory (a 18% bit overhead over tag storage requirements for S-NUCA), compared to 59 million bits required for a binary tree-based directory, and 8 million bits required to store the address tags in S-NUCA. Furthermore, the growth of the directory overheads would now be on the order of \( \log_4 N \). The costs could be further reduced by use of a partial-tree to “flatten” the tree and reduce the number of directory levels.
10.3.3 T-NUCA Access

In this section, we discuss how data accesses are handled in T-NUCA in detail. On an L1 cache miss, the directory node associated with local L2 bank is checked first. On a miss there, the node’s parent is queried, and so forth, until either the requested address is found in cache, or the root directory node is reached. If the data is located, the request is sent via the directory tree to the data’s owner bank. The owner bank then sends the data to the requesting core over the mesh NoC, and handles the coherency issues. The data line then migrates to a bank closer to the requesting core over the mesh network, and all the information in the hierarchical directory is updated. If the data is not found at the root directory node, a cache line is randomly selected as a victim and evicted.

Our design performs the hierarchical directory status updates in the background (without involving the mesh NoC). The only exception to this rule are MESI coherence operations, which stall the cache request until the coherence protocol requirements are met. In addition, all invalidations, write-backs and MESI status updates in L1 caches take place over the mesh network. Lastly, writes from L1 to L2 require an ACK message to be sent to the writer upon completion of the write.

10.3.4 Qualitative Comparison of S-NUCA, D-NUCA and T-NUCA

In this section, we qualitatively discuss the trade-offs among the S-NUCA, D-NUCA and T-NUCA designs. In particular, we focus on trade-offs related to performance energy metrics, network use, and complexity of data lookup. We also consider the impacts of thread-to-core mapping inside of the CMP. In Section 10.5, we evaluate S-NUCA, D-NUCA and T-NUCA impacts on these metrics.
S-NUCA design has a low energy consumption at the expense of a potential performance degradation. On each access to the L2 cache, data is looked up and placed in banks based on data address bits. Because of static placement, we have no influence on where data is placed in cache (and main memory), and consequently data can reside far from a core at each access. Furthermore, significant bank contention can develop if applications map their data sets into a single bank, or even worse, a single cache-line set within a bank. However, the S-NUCA design is very conservative with respect to energy consumption, as only one bank, and a limited number of routers and links are activated on each L2 cache access.

The D-NUCA represents a high performance cache design, at an expense of higher active energy consumption. The use of network broadcast and migration of frequently used data closer to requesting core, results in a shorter average access latency than for S-NUCA. However the dynamic energy costs of D-NUCA are higher than S-NUCA because of the broadcast operation at each access. Additionally, in the event of cache becoming flooded with data requests, network contention can develop resulting in performance degradation.

Our T-NUCA design tries to find a better energy and performance trade-off for NUCA in CMPs. In this design, we aim at achieving high performance, while reducing the energy and network utilization. We expect our T-NUCA to have the following advantages over the broadcast based D-NUCA: (i) significant reduction in overall network traffic, (ii) comparable performance, especially for codes that exhibit significant levels of data locality, (iii) reduced energy consumption and EDP due to lower bank activity, and (iv) global view of the cache storage allowing for efficient implementation of cache capacity sharing and replacement protocols. Additionally, because our T-NUCA design uses data migration to bring data closer to the requesting core at each access, we expect it to have lower bank contention than S-NUCA.
The performance of applications running on our T-NUCA can be sensitive to thread-to-core mapping, and the inter-thread data sharing patterns. In addition, the performance of the applications can be affected by the hierarchical tree directory structure. For example, in certain situations, the latency of accessing the data from the neighboring bank in T-NUCA can be significantly higher than that for accessing the same bank using the broadcast method in D-NUCA. This can happen when a request needs to traverse all levels of the hierarchical directory structure. However, with data migration in place, such an event should not be a frequent occurrence.

We expect that the application performance on T-NUCA will be affected by thread-to-core mapping. Careful thread-to-core mapping schemes already exist in many modern systems, and can be adapted to take advantage of T-NUCA structure. For example, thread affinity in pthreads, scheduling in batch systems, where the batch system attempts to assign contiguous blocks of processors to a task, task and node insulation in BlueGene/L [62]. Additionally, Jin et al. [103] showed how the operating system can map data structures so that they are allocated close in cache to the processors that need them.

10.4 Experimental Setup

In this section, we evaluate the performance and energy efficiency of our new cache design against the S-NUCA and D-NUCA, in both single-program and multi-program execution scenarios. We use the two Serengenti machine configurations in SIMICS [137] full system simulator to obtain the memory access traces for the SPEC OMP [106] and SPLASH-2 [194] benchmarks. We fast-forward each benchmark several million instructions beyond the start of the benchmark, and then execute each benchmark for 250 million instructions per core. The SPEC OMP benchmarks (ammp, art, equake, fma3d, gafort, galgel, mgrid and swim) are fast forwarded
4 billion instructions (on each core). The SPLASH-2 benchmarks LU and CHOLESKY are fast-forwarded 100 million instructions (on each core). The SPEC OMP benchmarks are run with the reference problem size, and the LU benchmark is run with matrix size of 1024 and CHOLESKY benchmark is run with the tk29.0 matrix.

We use the Sarek configuration running Solaris 9 with 8 processors, to collect the memory traces of the SPEC OMP benchmarks for the multi-program scenario. We use the Solaris 10 Serengenti Abisco machine to generate memory access traces for the single-program scenario. We use a 64 processor configuration of the Abisco machine (we modified the Serengenti system configuration to accommodate up to 384 processors) to obtain memory traces of two C SPEC OMP benchmarks (EQUAKE and ART) and two SPLASH-2 kernels (LU and CHOLESKY).

The SIMICS memory traces are executed by our cache simulators: one simulating the S-NUCA cache, one simulating the D-NUCA, and one simulating our T-NUCA cache. All simulators implement a 64 core processor architecture, where each core is directly connected via a router to a local bank of the L2 cache. The other (remote) banks of the L2 cache can be accessed only via a network-on-chip. For all configurations we consider a 16MB L2 cache, divided into 64 banks. Each of the banks is 256KB. The banks are four-way associative, with a 64 byte cache line. There are 1024 cache sets in each bank. In all our experiments, T-NUCA uses requires more bits than the D-NUCA and S-NUCA.

We use cache simulators to estimate latency and energy consumption of the L2 cache, and latency of the L1 caches. We obtain the cache access latency and energy estimates from CACTI6.0 [154] for the 40nm technology. We also obtain the average leakage power per bank from CACTI. We convert the leakage power values inside of the cache simulators into per cycle
<table>
<thead>
<tr>
<th>Component name</th>
<th>CACTI 6.0 result</th>
<th>Simulator input</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Router</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Buffer read energy</td>
<td>0.000759669 (nJ)</td>
<td>0.000759669 (nJ)</td>
</tr>
<tr>
<td>Buffer write energy</td>
<td>0.0011871 (nJ)</td>
<td>0.0011871 (nJ)</td>
</tr>
<tr>
<td>Cross bar access energy</td>
<td>0.00192301 (nJ)</td>
<td>0.00192301 (nJ)</td>
</tr>
<tr>
<td>Arbiter access energy</td>
<td>0.000206373 (nJ)</td>
<td>0.000206373 (nJ)</td>
</tr>
<tr>
<td>Delay</td>
<td>3 cycles</td>
<td>3 cycles</td>
</tr>
<tr>
<td><strong>Links</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Horizontal link delay</td>
<td>0.0750763 (ns)</td>
<td>1 cycle</td>
</tr>
<tr>
<td>Vertical link delay</td>
<td>0.151622 (ns)</td>
<td>1 cycle</td>
</tr>
<tr>
<td>Horizontal link energy</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- dynamic/access</td>
<td>7.24874e-05 (nJ)</td>
<td>7.24874e-05 (nJ)</td>
</tr>
<tr>
<td>- leakage</td>
<td>1.36254 (nW)</td>
<td>1.36254 (nW)</td>
</tr>
<tr>
<td>Vertical link energy</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- dynamic/access</td>
<td>0.000146397 (nJ)</td>
<td>0.000146397 (nJ)</td>
</tr>
<tr>
<td>- leakage</td>
<td>2.75176 (nW)</td>
<td>2.75176 (nW)</td>
</tr>
</tbody>
</table>

**D-NUCA and S-NUCA Banks**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Access time</td>
<td>0.462894 (ns)</td>
<td>2 cycles</td>
</tr>
<tr>
<td>Read energy/access</td>
<td>0.0741428 (nJ)</td>
<td>0.0741428 (nJ)</td>
</tr>
<tr>
<td>Total leakage power of a bank</td>
<td>484.062 (mW)</td>
<td>484.062 (mW)</td>
</tr>
</tbody>
</table>

**T-NUCA Banks**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Data array time</td>
<td>0.462894 (ns)</td>
<td>2 cycles</td>
</tr>
<tr>
<td>Data array read energy/access</td>
<td>0.0665994 (nJ)</td>
<td>0.0665994 (nJ)</td>
</tr>
<tr>
<td>Data array total leakage power</td>
<td>455 (mW)</td>
<td>455 (mW)</td>
</tr>
<tr>
<td>Data tag time</td>
<td>0.25835 (ns)</td>
<td>1 cycle</td>
</tr>
<tr>
<td>Data tag read energy/access</td>
<td>0.00754336 (nJ)</td>
<td>0.00754336 (nJ)</td>
</tr>
<tr>
<td>Data tag total leakage power</td>
<td>28.6 (mW)</td>
<td>28.6 (mW)</td>
</tr>
</tbody>
</table>

Table 10.1. L2 cache configuration parameters from CACTI6.0.

<table>
<thead>
<tr>
<th>Cache Name</th>
<th>Size (KB)</th>
<th>Associativity</th>
<th>Cache Line width (bytes)</th>
<th>Number of banks</th>
<th>Data latency (cycles)</th>
<th>Directory latency (cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>32</td>
<td>4</td>
<td>64</td>
<td>1</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>L2</td>
<td>16,384</td>
<td>4</td>
<td>64</td>
<td>64</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cache Name</th>
<th>Size (KB)</th>
<th>Router latency (cycles)</th>
<th>Hop dir. X latency (cycles)</th>
<th>Hop dir. Y latency (cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>32</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>L2</td>
<td>16,384</td>
<td>3</td>
<td>1</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 10.2. Summary of cache configuration parameters for the L1 and L2 caches for the S-NUCA, D-NUCA and T-NUCA configurations.
leakage energy constants by assuming that our processor cores run at the 3.54GHz, the maximum network frequency achievable by the NoC evaluated by CACTI. The values of all cache parameters are listed in Table 10.1. The values returned by CACTI are listed in the “CACTI 6.0 result” column. The values of all parameters in cycles (assuming 0.28 ns cycle) are listed in the “Simulator input” column.

To evaluate the energy consumption and performance of our T-NUCA design in detail, we calculate the access latencies and energy consumption values of the hierarchical directory, data array access, and mesh interconnect separately. We rely on CACTI for appropriate energy and latency values for each component. We use the tag array latency and energy numbers reported by CACTI for the hierarchical directory. The energy consumption of each directory at each successive higher level in the tree is doubled, to account for the doubling of the directory capacity. In addition to scaling the directory energy consumption at each level, we also scale the tree link delays accordingly. Our tree is laid out in such a way that directories the X direction are connected first and then the directories in the Y direction. The resulting link latencies as we go from level $i$ to level $i+1$ are 1, 1, 2, 1, 1, 2 cycles, respectively. At each level of the tree directory we use a constant access latency of 1 cycle (we increase the capacity of each directory by increasing associativity). To calculate the mesh network latency we use the network link latencies generated by CACTI and assume a three stage router (3 cycle latency, when no contention).

We consider two execution scenarios in our experiments. In the first scenario, there are eight programs running concurrently on the 64 cores. We assign each application to a contiguous block of 8 cores. This configuration is used to test the behavior of the cache system when programs are competing for the available L2 cache capacity. Within this scenario, we also consider
the impacts of thread-to-core mapping by the operating system. We consider three such mappings: (i) the row-wise mapping (our base configuration), (ii) the column-wise mapping, and (iii) the blocked mapping (where data is assigned to a rectangular block of cores and the block aspect ratio is kept as close to 1 as possible, i.e., a square). The second scenario is representative of a scientific application running on a multi-core machine. In this setup, we simulate a single program having access to all 64 cores. This configuration is used to validate the effectiveness of the proposed cache subsystem when there is no resource contention among applications. It also represents a scenario where an entire CMP is devoted to a data intensive scientific application.

10.5 Experimental Results

In this section, we evaluate the performance and energy efficiency of the T-NUCA design and compare it against S-NUCA and D-NUCA. We begin our analysis of the T-NUCA design in a single-program scenario. For our experiments we use four benchmarks: two SPEC benchmarks: (i) ART and (ii) EQUAKE, and two benchmarks from the SPLASH-2 suite: (i) LU and (ii) CHOLESKY. In the multi-program scenario, our system is executing 8 applications from the SPEC OMP benchmark suite on 64 cores, and each application is assigned 8 cores in a row of the chip. This scenario evaluates the effectiveness of our T-NUCA design in an environment where multiple applications run and compete for system resources. In both execution scenarios, we compare the performance and energy efficiency of our T-NUCA design against the S-NUCA and D-NUCA.
10.5.1 Results for Single-Program Execution Scenario

We begin by investigating the average number of links, and the average number of hops per L2 access used by S-NUCA, D-NUCA and T-NUCA in single-program execution scenario in Figure 10.3 (see left plot). When compared against S-NUCA, the T-NUCA lowers the average hop count by 33% to 72%, with a mean of 43%. However, due to directory overheads, the average number of links used per L2 access increases by 57% to 181%, on average by 148%. With respect to D-NUCA, T-NUCA reduces the average number of links used by 49% to 70% (54% on average), while maintaining the same average number of hops per L2 access.

Fig. 10.3. Single-program scenario - Average hop count and number of links used per access for S-NUCA (Left), D-NUCA (middle), and T-NUCA (right).

In Figure 10.4, we compare the energy, EDP and execution time results for S-NUCA, D-NUCA and T-NUCA. The results are normalized with respect the S-NUCA set at 1. We observe that when compared to S-NUCA and T-NUCA, the D-NUCA energy and EDP costs are a factor
of 15 to 20 times higher (middle and right plots). This is due to network and bank active energy consumed by D-NUCA on each broadcast. T-NUCA maintains or improves the performance of three of the four benchmarks used (ART, EQUAKE, LU), and lowers EDP by 19% for two of them (EQUAKE and LU). For example, for EQUAKE, our T-NUCA outperforms the D-NUCA by 27%, at a 35% overhead in number of transistors used, while reducing the network link use by 48%, energy consumption by 93%, and EDP by 95%. Relative to S-NUCA, our T-NUCA reduces the execution time of EQUAKE by 38% and EDP by 19%, while increasing the link use by 161% and energy consumption by 32% due to directory overheads. On average, relative to S-NUCA, the T-NUCA improves the performance by 19%, at 37% energy and 14% EDP increase. Relative to D-NUCA, T-NUCA improves the performance on average by 19%, with a 93% energy and 94% EDP reductions.

Fig. 10.4. Single-program scenario - Time (Left), energy (middle) and EDP (right) results for ART, EQUAKE, LU and CHOLESKY benchmarks on S-NUCA, D-NUCA, and T-NUCA. Results normalized with respect to S-NUCA.
We analyzed in detail the simulator statistics to determine why the T-NUCA performance for CHOLESKY and ART benchmarks was not improved over the S-NUCA. Our investigation revealed that even though the thread-to-core mapping is constant across the architectures, the slowest cores in S-NUCA and T-NUCA are different. We observed that the performance of the slowest core in S-NUCA was significantly improved in the T-NUCA, and that another core became the slowest core in T-NUCA. Further analysis of per core behavior for each benchmark in S-NUCA and T-NUCA revealed that the latency penalties for the two benchmarks are due to hierarchical directory overheads. While the average number of hops, and the average number of hops used by the busiest core decreased in T-NUCA, the average per access latency increased due to the hierarchical directory overheads.

The plots of Figure 10.5 show the distribution of levels where data read requests were satisfied for all 64 cores (left plot) and the busiest core (right plot). We observe that on average about 42% of requests are satisfied at the root level, indicating that for almost half of the L2 cache accesses the requested data is on the opposite half of the chip than the requesting core. An analysis of the distribution of levels where data read requests were satisfied for the busiest core reveals that for CHOLESKY benchmark (worst performance, energy and EDP loss) over 70% of read requests are satisfied at the root tree level, while for the LU benchmark (best performance and EDP improvements) less than 40% require access to the global directory.
Next, we consider changing the migration scheme from the XY-order to the YX-order in an attempt to decrease the directory lookup penalty associated with crossing a row of banks. We present our results in Figures 10.6 and 10.7. Consider Figure 10.6 first. In the left plot, we show the results for T-NUCA with YX-order routing (T-NUCA-YX) relative to S-NUCA, and in the right plot, we show our results relative to the T-NUCA with XY-order routing. We observe that while T-NUCA-YX performs better than S-NUCA (with exception of CHOLESKY benchmark), it still suffers from significant energy penalty, up to 60% for CHOLESKY. At the same time, T-NUCA-YX improves the performance of the ART benchmark by 15% (see right plot of Figure 10.6), the energy consumption by 9% and EDP by 23% over T-NUCA with XY-order routing. The performance, energy consumption, and EDP of the remaining three benchmarks is
maintained. The performance gain of ART is explained in Figure 10.7, which shows the distribution of directory hits per access for T-NUCA with YX-order routing. We observe that the number of hits at the lowest directory level for ART increases from 50% (as reported in Figure 10.5) to almost 55%.

![Bar chart](image)

**Fig. 10.6.** T-NUCA-YX - Single-program environment - Time, energy and EDP results for ART, EQUAKE, LU and CHOLESKY benchmarks run on T-NUCA-YX, relative to S-NUCA (left) and relative to T-NUCA (right).
Fig. 10.7. T-NUCA-YX - Single-program environment - **Left**: Distribution of accesses satisfied at each level of the tree directory for the whole CMP. **Right**: Distribution of accesses satisfied at each level of the tree directory for the busiest core.

### 10.5.2 Results for Multi-Program Execution Scenario

We now investigate impacts of S-NUCA, D-NUCA and T-NUCA designs on the network activity of the L2 cache and the average distance data needs to travel on an L2 cache read, in the multi-program execution scenario, in Figure 10.8 (left plot). We observe that our T-NUCA reduces the network link use by 81%, relative to D-NUCA, while maintaining the same average number of hops that data needs to travel on an L2 cache read. This is an expected result, as both schemes use the same data placement and migration policies. Additionally, when compared to S-NUCA, T-NUCA uses 57% less links on average and has a lower average hop count due to data migration.
We next consider mesh NoC and hierarchical directory activity in T-NUCA. We observe that T-NUCA reduces the network activity and energy consumption of the system by taking advantage of data locality. As shown in the middle plot of Figure 10.8, an overwhelming majority of all read accesses in the L2 cache were resolved at the local directory level (level 0), and the average directory level where data is found is 0.52. When we analyze the performance of the busiest core in the simulation (the one that executed the longest), we see that the majority of L2 reads was resolved at the root directory level. This behavior indicates that on average the data needed by the busiest core is located far from it. This is validated by the average hop count for the busiest core of 5.75 hops, while on average for the whole CMP the average hop count is just 0.63 hops.

In the multi-program scenario, our T-NUCA design improves the performance by nearly 72% when compared to S-NUCA cache, and by 17% when compared to the D-NUCA cache (see the rightmost graph in Figure 10.8). Furthermore, our T-NUCA cache offers nearly 58% energy reduction over S-NUCA and 93% over D-NUCA respectively. Also, the EDP for T-NUCA is 88% lower than EDP for S-NUCA and 94% lower than EDP for D-NUCA.
Fig. 10.8.  Multi-program scenario. **Left:** Average number of links utilized per access (top subplot), and the average distance in hops of where data is found on a L2 read (bottom subplot). **Middle:** Distribution of accesses satisfied at each level of the tree directory for the whole CMP (top subplot) and the busiest core (bottom subplot). **Right:** Execution time, energy and EDP results with S-NUCA, D-NUCA and T-NUCA.

The energy breakdowns for S-NUCA, D-NUCA and T-NUCA are shown in Figure 10.9. We observe that in the S-NUCA (bottom, left plot), the majority of the energy consumption is due to bank leakage (76%). In the D-NUCA (bottom, middle plot), the majority of power consumption is due to bank access energy (90%). For T-NUCA (bottom, right plot), the major energy components are data array leakage (48%).
Fig. 10.9. Multi-program scenario. Per component energy contribution for S-NUCA (Top Left), D-NUCA (Top Right), and T-NUCA (Bottom). Each pie is represents 100% of energy consumption for each cache architecture.

In Section 10.3.4, we observed that in our T-NUCA the thread-to-core mapping can influence the performance of applications. Thus far, we used a row-wise thread-to-core mapping,
which might unduly benefit our T-NUCA (tree is built row-wise first). We now consider T-NUCA with a column-wise and a blocked thread-to-core mapping. In the column-wise mapping, 8 cores in a column are assigned to one application. In the blocked mapping, 8 cores in a 4 by 2 rectangle are mapped to each application.

Fig. 10.10. Multi-program scenario, thread-to-core mapping impacts. **Left:** Execution time relative to S-NUCA with row-wise mapping. **Middle:** Energy consumption relative to S-NUCA with row-wise mapping. **Right:** EDP relative to S-NUCA with row-wise mapping.

In Figure 10.10, we present impacts of the thread-to-core mappings on the performance, energy and EDP results for S-NUCA, D-NUCA and T-NUCA. We observe that changing the row-wise mapping technique to column-wise mapping techniques does not significantly affect the performance, energy, and EDP values. For all of our designs, the best performance is achieved for the blocked mapping. For T-NUCA with blocked mapping, the execution time is reduced by 7%, relative to T-NUCA with row-wise mapping, energy is reduced by 5% and
EDP by 12%. We also observe that on average the row-based mapping increases the number of hits at the lowest level of the tree directory, while at the same time, increasing the number of hits satisfied at the highest level of directory for the busiest core (See Figure 10.11).

![Multi-program scenario, thread-to-core mapping impacts. Distributions of directory hits among T-NUCA tree levels for various thread-to-core mappings.](image)

Finally, we consider a scenario when the designer commits the directory budget of T-NUCA towards increased capacity of S-NUCA or D-NUCA. We considered a 24MB S-NUCA and D-NUCA caches (which in terms of bits are equivalent to a 16MB T-NUCA). Our experiments revealed that increasing the capacity of the S-NUCA and D-NUCA caches results in
no performance improvements for S-NUCA and 2% performance improvement for D-NUCA. These performance improvements for S-NUCA and D-NUCA come at a cost of increased energy consumption by 45% and 36% respectively, and increased EDP by 45% and 35%. Furthermore, our 16MB T-NUCA design outperforms the two larger S-NUCA and D-NUCA by 72% and 16%, at energy savings of 71% and 95% and EDP savings of 92% and 99%.

### 10.6 Related Work

In this section, we discuss previous work on the NUCA architectures. The NUCA cache architecture was originally proposed by Kim et al. [114] as a way of handling increasing wire latencies in large cache structures. The main idea concerned dividing the cache structure into banks, and allowing the access latency to be dependent on the distance between the bank and the processor requesting the data. Additionally, several variations of the NUCA were introduced. Huh et al. further extended the work on NUCA to accommodate the case of multiple cores sharing the overall NUCA cache [94].

While Kim et al. advocated the use of relatively small blocks, Chisthi et al. [33] provided an alternative design called NuRAPID. They separated the data from the tags, just like it is done in many of the larger monolithic L2 caches. In their approach the placement of data is performed in the directory at first, after which modifications are made to the storage banks. In addition to separating the directory lookup from data access, they advocate using a few large data arrays, instead of many small banks, for higher reliability and efficiency. As a result of larger arrays the amount of data migration is reduced over the standard NUCA design. Christi et al. [34], further extended their work to accommodate a limited number of cores.
Chang et al. [29] introduced the concept of cooperative caching in multi-core processor systems. Their design was based on each core having an L2 cache, and a network interconnect to other processors. The L2 caches of each core were cooperate by listening in on the network traffic, in order to reduce the conflicts and increase the overall cache capacity available to each core.

Zhang et al. [198] proposed a system in which reduction of wire delays was achieved by L1 victim replication in the local L2 bank. Brown et al. [24] introduced a proximity-aware directory-based coherence protocol for the multi-core processor architectures.

Separate approaches to designing multi-core and non-uniform cache architectures were proposed by Liu et al. [131] and Guz et al. [77]. Liu et al. [131] advocated a fully shared L2 cache where the most frequently shared data elements are placed in a central bank located between and closest to all cores. Guz et al. [77] on the other hand proposed the Nahalal architecture in which the L2 cache is divided into two distinct regions: (i) private L2 region for each core and (ii) shared L2 region among all cores. The architecture layout resembles that of a collective village Nahalal where the shared elements are in the center of the city (processor), the farmers (cores) live in a circle enclosing the shared elements, and private fields (private L2 caches) are located in the periphery on the outskirts of the town. While all of these approaches are very effective for a small number of processors, none of them addresses the problem of cache sharing and cache access in architectures where we have 64 or more cores.

There has also been a significant research effort dedicated to understanding the network-on-chip impacts on multi-core processors design. Muralimanohar et al. [153] investigated various interconnect designs for large NUCA caches. In particular he studied effects of designing a hybrid in cache network by using: (i) different wire types for address and data networks, (ii)
different topologies for data and address networks, and (iii) different interconnect architectures for parts of the address network. Kumar et al. [76] studied the overheads and scaling issues in various interconnects for Multi-core architectures.

Jin et al. [35] proposed operating system level L2 cache partitioning, where banks in the L2 cache would be statically assigned to applications, in such a way as to minimize conflicts between applications, reduce overheads and achieve fair sharing of capacity.

Finally, Li et al. [127] proposed and evaluated a system where the NOC latency is reduced by utilizing 3D stacking of chips, instead of spreading the processors in just two dimensions. We believe that our tree based approach could benefit from a 3D design to reduce overall access latencies.

Trees have been used successfully for a long time in scientific and high-performance computer designs. For example, many of the high speed networks for large scale systems are based on fat trees [62, 125, 128, 185]. Furthermore, many of the global operations data operations in high performance computing map very nicely to a tree network, e.g., broadcast, all-to-all, scatter/gather operations in MPI [118, 163]. Certain high-performance machines such as Blue-Gene/L [62] have a dedicated tree network to handle these operations in addition to the more standard torus/mesh interconnect designs. Tree structures have also been used successfully in shared distributed memory systems [136, 159]. In these applications, the trees are used to store the page the sharing information for large number of nodes, where full map sharing directory would be prohibitively expensive. In a more recent work, Eisley et al. [54] proposed in network virtual trees for maintaining sharing and coherency information. Finally, in many scientific applications, trees form the basis of computation. They convey information about data and computational dependancies [143], as well as allow for significant performance optimizations [181].
All of these approaches motivate us for exploring the T-NUCA cache architecture in the context of CMPs.

10.7 Chapter Summary

In this chapter, we propose a new T-NUCA cache architecture. In our design, the data and address tag bits are separated, and address tags are stored in an inclusive, hierarchical, tree-based directory. In our hierarchical directory the entries of the children nodes of the tree are replicated at the parent nodes recursively; thus the root node, which contains all address tags and has a global view of the cache contents. We evaluate the performance and energy efficiency of T-NUCA against S-NUCA and D-NUCA in two execution scenarios, a single-program scenario, where one application is assigned all cores on the chip, and a multi-program scenario, where eight applications are assigned to the chip at the same time. We show that in the single-program scenario, in the best instance, our T-NUCA architecture outperforms the D-NUCA by 27%, at a 35% overhead in number of transistors used, while reducing the network link use by 48%, energy consumption by 93%, and EDP by 95%. Relative to S-NUCA, in the best instance, our T-NUCA reduces the execution time by 38%, increases the link use by 161% and energy consumption by 32% due to directory overheads, and reduces the EDP by 19%. On average, relative to S-NUCA, T-NUCA improves performance by 19%, at 37% energy and 14% EDP increases. On average, relative to D-NUCA, T-NUCA improves the performance by 19%, with energy and EDP reductions of 93% and 94%. Similar performance improvements are observed in the multi-program scenario. In the multi-program scenario, relative to D-NUCA, our T-NUCA design reduces the execution time by 17%, energy by 93% and EDP by 94%. Relative to S-NUCA, our T-NUCA design reduces the execution time by 71%, energy by 58% and EDP by 88%. These
results indicate that T-NUCA presents an improved trade-off between performance and energy on parallel workloads, relative to S-NUCA and D-NUCA.
We conclude this thesis by discussing our major research findings, associated open problems, and possible extensions to our work on the problem of enabling energy-efficient scientific computing. Our work is motivated by the fact that many fields in science and engineering rely on computational models for design and discovery. The underlying representations of these models are typically sparse and require large multi-processor systems to guarantee sufficient accuracy and reasonable execution times. Owing to the sparse nature of these models, the computational efficiency is usually low, and the energy requirements of the multi-processor systems are high.

We approach the problem of enabling energy efficiency in scientific computing in order of decreasing scale of size. We begin with whole multi-processor system scale optimizations where we expect to achieve the greatest performance and energy improvements. We then investigate single node power and performance optimizations.

At the scale of whole multi-processor system optimizations, we discuss the impact of workload balancing on parallel application performance and energy efficiency. We examine parallel applications with computational dependencies represented by a tree where workload is distributed among processors using a proportional workload distribution scheme. We then present a new multi-pass workload distribution scheme for tree-based computations (see Chapter 4). Our multi-pass scheme performs three passes over the tree representing the computational dependencies. On the first pass, it generates the standard proportional mapping. On the second pass,
it locates the tree paths with the most and the least work associated with them, and it applies the “robin hood” scheme. On the third pass, our scheme performs iterative refinement with $k$ processors in reserve based on results from the previous two passes. To capture the quality of workload distribution by the proportional and multi-pass mappings, we introduce a metric called critical overload. Our improved scheme reduces the cumulative critical overload on the suite of test matrices by an average of 50% over the base proportional scheme.

An additional benefit of reducing the workload imbalance is the reduction of the overall energy costs associated with the computation. Consider, for example, a parallel application running on the Earth Simulator which consumes 7 MW of power on average [178]. Also consider a case where we reduce execution time by 20% through better workload distribution. We can then expect the energy consumed by this application to reduce by 20% on average as well.

Our research shows that our multi-pass mapping technique successfully reduces the workload imbalance when compared to the base proportional mapping. However, our technique does not remove the imbalance completely. Therefore, we consider opportunities for combining the tree representation of parallel computation dependencies with dynamic voltage and frequency scaling (DVFS) to convert workload imbalances into energy savings on multi-processor computer systems (see Chapter 5). We introduce three energy reduction schemes that, for our suite of test matrices, offer average energy savings of up to 21% without reducing the performance of the application. Our methods recursively traverse the computational dependency tree and apply DVFS to nodes not on the critical path of computation. An extension of this work that considers both processor DVFS and link scaling is discussed in a paper by Son et al. [180].

We observe that our multi-pass method, as described in Chapter 4, is effective at reducing workload imbalance on a relatively small number of processors (up to 64). It is an open question
whether this method is effective at reducing workload imbalances on modern petascale machines with thousands of processors. It is also an open research problem whether our “slack-to-energy” conversion methods, as discussed in Chapter 5, will be effective in petascale or exascale environments. Furthermore, changes in processor design, from single-core to multi-core, and higher contributions from leakage power to the processor power budget, introduce additional opportunities and challenges for dealing with the problem of workload imbalance and energy efficiency. We plan on investigating these problems in our future work.

At the next scale, we consider single node opportunities for power and performance optimizations where the node has either a single-core or a multi-core processor design. In Chapters 6 and 7, we discuss the interactions between software tuning (such as matrix ordering, matrix blocking, register blocking, and data structure design) and hardware features (such as hardware prefetchers, DVFS, low power drowsy caches, and memory bus width) and analyze opportunities for energy and performance trade-offs for reducing energy consumption of sparse scientific codes in light of single-core processor design. We show that energy consumption can be reduced by up to 64% with proper data structures, matrix properties, software tuning, and hardware features present in the system. At the same time, the application performance relative to an unoptimized system can be maintained or exceeded.

In Chapters 8 and 9, we introduce novel hardware features aimed at improving performance and energy efficiency of scientific applications run on single-core processors. In Chapter 8, we introduce the load miss predictor (LMP), a cache subsystem enhancement aimed at reducing the memory access latency for streaming data accesses. Our load miss predictor attempts to predict whether data will be found in the lowest level of the cache or whether an early load to main memory should be issued. The LMP improves the performance of sparse scientific
applications by up to 16.7% without negatively impacting the performance of non-scientific applications. In addition, when combined with DVFS, the load miss predictor on average reduces system power by 7.3% and energy by 17.3% while maintaining an 8.7% improvement in execution time. Next, in Chapter 9, we discuss a phase-aware adaptive hardware selection scheme. We propose a system where the prefetcher activity and processor performance automatically adapt to application needs based on available hardware counters with the goal of reducing energy consumption without degrading application performance. Our scheme is effective at adapting to application requirements and reducing energy by up to 37% for codes with significantly long memory bound phases.

In our research, presented in Chapters 6, 7, 8 and 9, we rely on the fact that memory bandwidth can be used to hide memory latency and that memory bound codes do not run at full processor speed. We show that for systems where active power dominates, we can combine memory latency hiding techniques with DVFS to lower energy consumption without degrading performance. However, there are still many open research problems that need to be addressed.

Our schemes are presented for single-core processors with relatively high levels of supply voltage and small contributions from leakage power. As the technology scales, the transistors become smaller, the supply voltage decreases and leakage power is a greater fraction of the total processor power bill. Additionally, in order to continue performance scaling at present rates and to limit the growth in energy density, processor design is shifting to multi-core design. It is not clear whether our techniques will be effective at reducing energy consumption without impacting performance in these new multi-core environments. Some of the open questions involve the effectiveness of prefetchers in serving multiple cores, the limits of memory bandwidth between
the lowest level of cache and the main memory subsystem, and the effectiveness of low power processor modes achievable via DVFS. We plan on addressing these issues in our future research.

Finally, in Chapter 10, we consider the problem of energy-efficient data lookup in non-uniform access cache architecture (NUCA) on multi-core processors. We propose a novel, tree-based directory NUCA architecture as an alternative design to the S-NUCA and D-NUCA architectures. We evaluate the performance and energy efficiency of T-NUCA against S-NUCA and D-NUCA in two execution scenarios, a single-program scenario, where one application is assigned all cores on the chip, and a multi-program scenario, where eight applications are assigned to the chip at the same time. We show that in the single-program scenario, in the best instance, our T-NUCA architecture outperforms the D-NUCA by 27%, at a 35% overhead in number of transistors used, while reducing the network link use by 48%, energy consumption by 93%, and EDP by 95%. Relative to S-NUCA, in the best instance, our T-NUCA reduces the execution time by 38%, increases the link use by 161% and energy consumption by 32% due to directory overheads, and reduces the EDP by 19%. On average, relative to S-NUCA, T-NUCA improves performance by 19%, at 37% energy and 14% EDP increases. On average, relative to D-NUCA, T-NUCA improves the performance by 19%, with energy and EDP reductions of 93% and 94%. Similar performance improvements are observed in the multi-program scenario. In the multi-program scenario, relative to D-NUCA, our T-NUCA design reduces the execution time by 17%, energy by 93% and EDP by 94%. Relative to S-NUCA, our T-NUCA design reduces the execution time by 71%, energy by 58% and EDP by 88%. These results indicate that T-NUCA presents an improved trade-off between performance and energy on parallel workloads, relative to S-NUCA and D-NUCA.
At the suggestion of Dr. Michael Perrone, of the IBM T.J. Wattson Research Laboratory, we plan on extending our study of the T-NUCA designs. In particular, we plan to investigate the scaling of T-NUCA performance and energy attributes as we scale to hundreds of cores on-chip with large variations in cache capacity and alternative tree topologies.

Over the next few years, we plan to continue doing research in power and performance optimization and instrumentation for large-scale scientific applications. We believe that power-performance optimizations will play a key role in the design of future computers especially with the move towards multi-core architectures in large computer installations used for scientific discovery at the peta-scale and beyond. Some of the more interesting and open-ended design problems we would like to investigate include multi-core programming challenges, power and performance adaptivity, performance guarantees given power and reliability constraints, and algorithmic approaches for exa-scale parallel system reliability, adaptivity, and multi-objective optimizations.
Appendix

Performance Metrics, Power Metrics and Moore’s Law

A.1 Efficiency and Speedup in Scientific Computing

The efficient solution of problems on multi-processor computers in the classical sense is understood to be the solution that involves the lowest amount of network overhead and results in shortest possible execution time. Ideally when using \( p \) processors we would like to solve the problem \( p \) times faster than on the single processor, and thus achieve the speedup \( S = p \). Gramma et al., define speedup and efficiency in the following way, as excerpted from Chapter 5 of their book [119]:

**Speedup** is a measure that captures the relative benefit of solving a problem in parallel. It is defined as the ratio of the time taken to solve a problem on a single processing element to the time required to solve the same problem on a parallel computer with \( p \) identical processing elements.

**Efficiency** is a measure of the fraction of time for which a processing element is usefully employed; it is defined as the ratio of speedup to the number of processing elements. In an ideal parallel system, speedup is equal to \( p \) and efficiency is equal to one. In practice, speedup is less than \( p \) and efficiency is between zero and one, depending on the effectiveness with which the processing elements are utilized. We denote efficiency by the symbol \( E \).
Mathematically, speedup $S$ is defined as:

$$ S = \frac{T_1}{T_p} \quad (A.1) $$

while efficiency is defined as:

$$ E = \frac{S}{p} \quad (A.2) $$

where $T_1$ is execution time of the serial version of the application, and $T_p$ is the execution time of the parallel version of the same application.

Iso-efficiency and weak scaling are two concepts which are frequently encountered in scientific computing. Typically, when the number of processors is increased and problem size remains constant (strong scaling), due to increased communication overheads the efficiency of the application decreases as the number of processors used increases. In order to maintain constant efficiency (iso-efficiency) the problem size is scaled together with the number of processors used (weak scaling) [119].

We extend the context of efficiency for scientific codes to include energy efficiency, due to the growing importance of energy consumption in large-scale computer systems, i.e., we are interested in a solution that executes fast, but also that reduces energy consumption.

### A.2 Power and Performance Metrics in Computer Architecture Design

In this section, we provide a brief overview of the metrics used to evaluate power and performance trade-offs and optimizations in this thesis. In particular we focus on: power, energy, execution time, FLOPS, MIPS and speedup. Where necessary, we introduce and describe new metrics, such as relative incremental improvement and relative improvement in Chapter 6. There
are numerous performance and architecture metrics used in the computing literature [83, 90]. These include: power, energy (power delay product), energy delay product (EDP), execution time, speedup, efficiency, FLOPS, FLOPS/Watt, MIPS, and others. These metrics should not be believed absolutely, and should always be considered in a greater context in order to avoid wrong conclusions and keep the larger picture in mind [135]. Furthermore, it is an open question which metric is the most appropriate for large-scale scientific computing [90]. A brief description of the standard metrics used is presented below.

**Performance Metrics**

- **Execution time** - run-time of application or segment of interest from start to completion.

- **MIPS** - millions of instructions per second. This metric measures the number of instructions executed by a specific architecture per second. This metric is very architecture specific, and can be reliably used to compare performance of processors with the same architecture set. It has little use in comparing architectures with different instruction sets.

- **FLOPS** - floating point operations per second. This metric is used frequently used by the scientific computing community to measure the performance of scientific applications on computers. It focuses only on floating point operations such as add, multiply and divide executed by hardware. In architectures where fused multiply add instructions are used, each such instruction is typically counted as two separate floating point instructions.

- **Relative performance** - the ratio of execution time on old architecture and execution time on new architecture. In serial performance comparisons this metric is used to compare performance of the new architecture versus some baseline architecture. The best example is
the SPEC [41] benchmark suite which provides relative performance to “Ultra Enterprise 2”, which was based on a 296 MHz UltraSPARC II processor.

- **Speedup** - This metric is commonly used to measure the improvement in parallel performance in scientific and other multi-threaded applications. It essentially calculates the relative performance between the parallel version of the application and the best serial version of the application. In parallel applications care has to be taken to compare the best serial implementation of the problem to the best available implementation of parallel version of the application. Otherwise the benchmark results will be unreliable.

### Power Metrics

- **Power** - In Newtonian physics work $W$ is defined as the product of force $F$ acting on the body and the distance $s$ during which the force is applied. Power is the average amount of work $W$ expended over time $t$ to move the object. Power in electronics is the rate at which potential energy is converted into heat by a resistor. Power is then defined as $P = I \times V = I^2R$ [57], where $I$ is the current flow through a device, $V$ is the voltage drop across the device, and $R$ is the resistance of the device. In modern processors and integrated circuits power is defined as $P = \alpha \times V_{dd}^2 \times f + \beta \times I_l \times V_{dd} [167]$, where $\alpha$ is the number of transistor switching, $V_{dd}$ is voltage supplied to the transistors, $f$ is the processor frequency, $\beta$ is the total number of transistors and $I_l$ is the leakage current in the transistors. While, ideally we would like to think of gates and transistors as perfect switches that either transmit signals or not, as shown in Figure A.1, transistors have both capacitive and resistance components [58, 85, 115], which result in power consumption when the transistors and gates switch, as shown in Figure A.2.
• Energy - Energy is defined as the product of power and time. Energy defines the total capability of one object to perform any type of work in a physical sense.

• Energy delay product (EDP) - Energy delay product is an artificial metric created by architecture developers to better portray trade-offs between energy consumption and execution time of an application in computers. EDP is defined as $EDP = E \times t$. EDP is a variation on the $ED^p$ metric [90].

• FLOPS/Watt - This metric is similar to the EDP metric and captures the cost of performing one floating point operation. The metric is application and architecture dependent.

Power and energy are typically not used alone to define power and performance characteristics of the circuit. They are usually correlated with time. Figure A.3 gives a good example why these metrics cannot be used alone and shows the difference between power and energy (Figure courtesy of Professor Mary Jane Irwin and Professor Vijay Narayanan at the Pennsylvania State University).

Fig. A.1. Idealized gate design. Transistors act as switches and there is no power loss.
Fig. A.2. A more realistic model of gate design. Observe the presence of resistor before ground and a capacitor on the output line. These two elements simulate resistive and capacitive properties of transistors and interconnect wires. They also represent the fact that it takes time for the output signal to stabilize.

To capture the trade-offs and relationship between power, energy and time, a variation of the $ED^3$ metric is typically used [90]. This artificial metric represents a trade-off where the designer considers 1% improvement in performance as worth paying $n\%$ increase in energy use [90]. Figure A.4 shows the relationships between time, energy and $ED^1$ (Figure courtesy of Professor Mary Jane Irwin and Professor Vijay Narayanan at the Pennsylvania State University). In that example $ED^1$ can be used to find optimal configuration points in terms of both energy and execution time.
Fig. A.3. Comparison between power and energy consumed by computer system. Figure courtesy of Professor Mary Jane Irwin and Professor Vijay Narayanan at the Pennsylvania State University.
Fig. A.4. Comparison between time, energy and energy delay product of a computer system. Figure courtesy of Professor Mary Jane Irwin and Professor Vijay Narayanan at the Pennsylvania State University.

A.3 Signal propagation speed

The maximum distance an electrical signal travels in global wires, in one cycle, is limited by the electro-magnetic signal propagation speed. The signal propagation speed in vacuum is equal to the speed of light of $299,792,458 \, \text{m/s}$, or approximately $30 \, \text{cm/ns}$ (11.8 $\, \text{in/ns}$). When the transmission wire is surrounded by a material other than vacuum (or air) the resulting signal
propagation speed decreases according to the Equation A.3 [23], where $\epsilon_r$ is surrounding material dielectric coefficient. For example, for the FR4 material that is used to build circuit boards the $\epsilon_r$ is approximately 4 [23], which means that signals on traces on circuit boards propagate at the speed of approximately $6 \frac{in}{ns}$. Additionally, the speed at which signals can be transmitted inside of the global wires and network interconnect is also affected by the switching speed, capacitive and resistive wire properties, and width and length of the wire [152]. Thus in general signals inside of circuit boards, processors and interconnects travel at lower speed than the speed of light.

\[
\text{Propagation Speed} = \frac{11.8 \frac{in}{}}{\sqrt{\epsilon_r \frac{ns}{}}} \quad (A.3)
\]

**A.4 Moore’s Law**

In 1965, the co-founder of Intel, Gordon Moore, made an observation that the complexity of the chip (the number of transistors on chip) will double at the minimum component cost roughly every year due to the innovations in process technology and clever circuit design and packing [150]. Moore based his predictions on integrated circuits with just 50 transistors, and predicted that by 1975 systems would contain around 65000 transistors. Moore observed that due to feature miniaturization and lithography process improvements this tendency could be maintained for at least several years beyond his prediction date.

In 1975, Moore revised this growth rate to account for packing difficulties [151]. In the revised prediction, the number of transistors would double every two years. The increase in the number of transistors on chip has followed this trend and continues to follow Moore’s curves at present time (year 2008). This growth allowed for an increase in processor capabilities and
performance as a result of addition of hardware features such as bigger caches, deeper pipelines, out-of-order execution. Thus as a side effect of Moore’s Law the resulting performance increase followed the same curve as the curve representing the number of transistors on chip\(^1\).

\(^{1}\)To date the Moore’s prediction has been so accurate that it is called a law. However, “Moore’s law” is not a physical law, but rather a psychological law, resulting from the ingenuity of engineers and scientists working on the development of new processors, and reduction of production costs.
References


http://www-flash.stanford.edu/SPLASH/.


[182] Tensilica Steve Leibson. Everything you know about microprocessors is wrong.


Vita

Konrad Malkowski was born in Świnoujście, Poland on May 17, 1979. He is the eldest son of Barbara and Sławomir Małkowski. In 1998 he graduated from High School Number 14 in Szczecin, Poland. In 2002 he received a Bachelor of Science degree in Computer Science, with a minor in Mathematics and Statistics, *summa cum laude*, from the Canisius College of Buffalo, New York. In 2002 he was accepted into the Ph.D. program in Computer Science and Engineering at the Pennsylvania State University. In Fall 2002 he was supported by the department of Computer Science and Engineering as a teaching assistant. Since spring semester of 2003 he was supported by Professor Padma Raghavan as a research assistant. During the summer of 2004 he was employed at the Lawrence Berkeley National Laboratory as a student assistant. During the academic year 2007-2008 Konrad was a recipient of the Robert M. Owens Memorial Scholarship.

Konrad’s research interests are in parallel scientific computing, including (i) scalable algorithms, software, and tools, and (ii) parallel, multi-core and single processor architectures.

Konrad is a member of IEEE Computer Society, ACM Computer Society, SIAM society, and the Alpha Sigma Nu Honor Jesuit Society. He is also a recipient of the “Who’s Who Among Students in American Universities and Colleges” award and “AmPol Eagle Citizen of the Year 1998” award in the Youth category. From 2004 to 2007 Konrad was active member of Pennsylvania State University Shotokan Karate club. In Fall 2007 semester Konrad became an active member of the Pennsylvania State Model Railroad Club.