OPTIMIZING THE INTERNAL MICROARCHITECTURE AND ISA OF A TRAVELING THREAD PIM SYSTEM

A Dissertation

Submitted to the Graduate School of the University of Notre Dame in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

by

Patrick Anthony La Fratta

________________________________________

Peter M. Kogge, Director

Graduate Program in Computer Science and Engineering
Notre Dame, Indiana
October 2010
Abstract
by
Patrick Anthony La Fratta

Heterogeneity, multiple on-chip processing elements, multithreading, intelligent caching mechanisms, and compiler-assisted thread-level speculation are a few of the features of emerging architectures for meeting the increasing performance demands and power constraints of future processors and workloads. In the design of future processors, two primary challenges facing computer architects are the overcoming of the memory wall and energy consumption reduction. To address these challenges, this work employs an iterative design methodology in the optimization of an innovative processor architecture that leverages the above features in the implementation of an advanced, powerful execution model called traveling threads for exploiting parallelism and data locality in tandem at multiple levels of granularity. The design of this Passive/Active Multicore (PAM) architecture and the development of mechanisms for locality-cognizant extraction of traveling threads offer insights into the benefits of utilizing computational migration at a granularity of parallelism between the conventional instruction and thread levels. Through these insights, we conclude that PAM and traveling threads are particularly well-suited for the joint exploitation of parallelism and data locality. Quantitative results support this conclusion, illustrating the architecture’s signifi-
cant potential improvements over those currently in use in terms of both execution time and energy consumption for standard benchmarks and scientific workloads.
Pro Domīnā Nostrā, \textit{Hodegetria}.
## CONTENTS

**FIGURES** .............................................................. vii

**TABLES** ............................................................... ix

**ACKNOWLEDGMENTS** .................................................. x

**CHAPTER 1: INTRODUCTION** .......................................... 1
  1.1 General Problems ............................................. 2
  1.2 Specific Questions We Address ................................ 3
  1.3 Objectives .................................................... 5
  1.4 Approach ..................................................... 6
  1.5 Contributions ................................................ 7
  1.6 Outline of Dissertation .................................... 7

**CHAPTER 2: STATE OF THE ART** ................................... 9
  2.1 Overview ..................................................... 10
  2.2 Architecture ................................................ 13
    2.2.1 PIM .................................................... 13
    2.2.2 Heterogeneous Multicore ................................ 16
    2.2.3 GPUs .................................................. 18
    2.2.4 Asymmetric Multicore ................................... 19
    2.2.5 Multithreaded Architectures ............................. 20
    2.2.6 Thread Migration ........................................ 21
    2.2.7 Memory-Side Computations ............................... 22
    2.2.8 VLIW ................................................ 23
    2.2.9 Dataflow Architectures .................................. 24
    2.2.10 Instruction Aggregation ................................. 26
    2.2.11 SIMD ................................................ 27
      2.2.11.1 VIS ............................................ 28
      2.2.11.2 AltiVec ....................................... 30
      2.2.11.3 MMX and SSE .................................... 32
CHAPTER 4: PASSIVE/ACTIVE MULTICORE ARCHITECTURE AND BLITS

4.1 Processing Cores

4.1.1 P-cores

4.1.2 A-cores

4.2 Passive/Active Multicore Architecture

4.3 Traveling Threads

4.3.1 Thread Migration on P-cores

4.4 BLITS

4.4.1 Thread Initiators

4.5 Design Parameters

CHAPTER 5: OPPORTUNITY STUDIES

5.1 Dependence Graph Analysis

5.2 Application Properties

5.2.1 Criticality

5.2.2 Memory Latency-Bounded Speculation Depth

5.3 Modeling Bounds on Migration Overhead

5.3.1 Analytical Model

5.3.2 Performance Results

5.4 Conclusions

CHAPTER 6: MODELS FOR GENERATING TRAVELING THREADS

6.1 Locality Analysis

6.1.1 Hot/Cold Cache Lines

6.2 Computational Partitioning

6.2.1 Hot/Cold WCCs

6.2.2 Singletons and Cache Bypassing

6.2.3 Window-Based Thread Extraction

6.3 Evaluation

6.3.1 Example: DAXPY

6.3.2 Methodology

6.3.3 Results

6.4 Conclusions

CHAPTER 7: ENERGY EVALUATION

7.1 Background

7.2 Methodology

7.2.1 Analytical Model

7.2.2 Instrumentation

7.2.3 Manual Partitioning
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.3</td>
<td>Results</td>
<td>159</td>
</tr>
<tr>
<td>7.3.1</td>
<td>SPEC and Sandia Suites</td>
<td>160</td>
</tr>
<tr>
<td>7.3.2</td>
<td>DAXPY Results</td>
<td>169</td>
</tr>
<tr>
<td>7.4</td>
<td>Conclusions</td>
<td>170</td>
</tr>
<tr>
<td>8.1</td>
<td>Summary of Work</td>
<td>173</td>
</tr>
<tr>
<td>8.2</td>
<td>Key Results</td>
<td>177</td>
</tr>
<tr>
<td>8.3</td>
<td>Future Work</td>
<td>179</td>
</tr>
<tr>
<td>APPENDIX A</td>
<td>INDEX OF ABBREVIATIONS AND TERMS</td>
<td>183</td>
</tr>
<tr>
<td>BIBLIOGRAPHY</td>
<td></td>
<td>186</td>
</tr>
</tbody>
</table>
FIGURES

2.1 Use of the \texttt{fpmerge} Instruction in VIS \cite{172} ............................... 29
2.2 SSE and AltiVec Data Shuffle Instructions ........................................... 31
2.3 Synchronization Between Two Processors ............................................. 51
3.1 FPIA Microarchitectural Extensions and Design Parameters .................. 58
3.2 Example of FPIA Transformation Process ............................................. 61
3.3 Number of Unique Graphs and Graph Utilization Tests ......................... 66
3.4 Graph Utilization Tests ................................................................. 67
3.5 ICC Microarchitectural Extensions ..................................................... 70
3.6 Execution of an IMO ................................................................. 72
3.7 Example of a Basic Block Taken from CTH. ........................................ 74
3.8 ICC Example. ............................................................... 77
3.9 DCP Architecture. ............................................................... 80
4.1 PAM Architecture ............................................................... 92
4.2 Thread Execution on P-cores ............................................................ 98
4.3 Control Vectors Example from \texttt{DAXPY} Code ................................. 100
4.4 PAM Design Parameters and Metrics ................................................. 101
5.1 Example of a Branch-Critical Path ..................................................... 108
5.2 Width of ADAG vs. Graph Level ....................................................... 109
5.3 Flow Chart of Sampling Methodology ................................................ 110
5.4 Tail length and \# of stages vs. subregion size for 4 traces ................. 112
5.5 Example from ALEGRA Illustrating $IPC_{MB}$ Calculation .................. 114
5.6 $IPC_{\text{max}}$ and $IPC_{MB}$ vs. subregion size for 4 traces ................... 115
5.7 $mIPM$ Results for SPEC-Int ....................................................... 120
5.8 $mIPM$ Results for SPEC-FP ....................................................... 120
5.9 $mIPM$ Results for Sandia-Int .......................... 120
5.10 $mIPM$ Results for Sandia-FP .......................... 121

6.1 Hot/Cold ADAG illustrating graph tail and WCC composition. . 128
6.2 Hot/Cold ADAGs of DAXPY Inner Loop .......................... 133
6.3 Offload Efficiency and Estimated ‘Speedup’: DAXPY ............. 134
6.4 Offload Efficiency and Estimated ‘Speedup’: SPEC-Int .......... 138
6.5 Offload Efficiency and Estimated ‘Speedup’: SPEC-FP .......... 139
6.6 Offload Efficiency and Estimated ‘Speedup’: Sandia-Int ........ 140
6.7 Offload Efficiency and Estimated ‘Speedup’: Sandia-FP ........ 141

7.1 Contrast of an ADAG with an event graph. ..................... 147
7.2 Flow of methodology for the analytical evaluation. .......... 150
7.3 Event graph generation: Conventional Single-core Architecture .. 153
7.4 Event graph generation: Conventional Multi-core Architecture .. 154
7.5 Event graph generation: PAM Architecture .................... 156
7.6 Execution of DAXPY loop on PAM. ............................ 160
7.7 Energy Results for SPEC-Int ................................. 161
7.8 Energy Results for SPEC-FP ................................. 161
7.9 Energy Results for Sandia-Int ................................. 162
7.10 Energy Results for Sandia-FP ................................ 162
7.11 Energy Results for DAXPY ................................. 169

8.1 Block Diagram of PAM Design. ............................. 175
### TABLES

3.1 Map of Parameters to Properties .............................................. 64
3.2 Summary of Graph Analysis Results ........................................... 65
3.3 ICC Content Analysis Results and Projected λ-Speedup Bound .......... 85
3.4 Baseline HWP Configuration ..................................................... 86
4.1 Cache-Active vs. Cache-Passive Cores ....................................... 91
4.2 Types of Software-level Computational Units .............................. 94
6.1 Comparison of Categorization of Cache Lines .............................. 127
6.2 Cache Line Labeling by Line Touches ....................................... 131
7.1 Latency and Energy Estimates of Events .................................... 148
7.2 Memory Access Statistics ....................................................... 167
ACKNOWLEDGMENTS

I would like to thank my advisor, Prof. Peter Kogge, for allowing me the privilege to work with him and for his guidance during my time at Notre Dame. I would also like to extend my appreciation to my committee members, Dr. Eugene Henry, Dr. Jay Brockman, and Dr. Michael Niemier, for their feedback on this work. Thank you also to the American Society for Engineering Education and the Department of Defense for supporting my graduate studies through the Science, Mathematics, and Research for Transformation (SMART) Scholarship.

Thank you to my friends, especially those who shared time with me in prayer at the Chapel of St. Catherine of Alexandria. Thank you to Fr. Oliver Williams, C.S.C., Fr. Paul Doyle, C.S.C., and Fr. John Coughlin, O.F.M. for their support. Thank you especially to Wes Mozley and Srinivas Sridharan for their invaluable friendship.

Thank you to all the members of my family, especially to my parents: without their love this work would not have been possible.

Maxime, Deo gratias: Pater, et Filius, et Spiritus Sanctus.
CHAPTER 1

INTRODUCTION

New and improved parallel processor designs, execution models, and languages continue to emerge at an astounding rate, but numerous obstacles – both existing and projected – pose enormous challenges to the field of computer architecture. In the underlying technology, the rapid increase in transistor densities has arguably become the architect’s best friend and worst enemy. Fabrication processes providing over two billion transistors on a single die have allowed the integration of 30MB of on-chip cache in Intel’s Tukwila processor \[162\] to address the problems of high DRAM access costs and limited off-chip bandwidth. On the other hand, high power densities have led to increased chip temperatures – reports indicate that Tukwila runs as high as 105°C \[162\]. At the software level, legacy codes, the non-trivial task of parallel programming, and the intrinsic challenges of producing automatic parallelization mechanisms hinder our ability to use parallel hardware universally to address the problems that accompany processors with soaring clock rates.

Multithreading and heterogeneous multicores are key players among competing designs in high performance computing, and enhanced forms of multithreaded execution models and heterogeneous processors show promising potential for next-generation machines. This work considers a novel processor design whose features crosscut many existing multithreaded and heterogeneous architectures and ex-
tends these features with progressive concepts. The execution model advances previous work in *traveling threads*. Here, the traveling thread is a lightweight unit of computation formed transparently to the programmer to exploit locality and leverage a unique feature of the microarchitecture to efficiently move computations from one core to another. The processor is a heterogeneous multicore called the *Passive/Active Multicore* (PAM), consisting of one or more out-of-order (OoO) cores and several lightweight cores within the memory hierarchy. The heterogeneity of PAM provides a powerful platform for accelerating sequential applications through the execution of locality-cognizant traveling threads that cooperatively share on-chip resources with heavy-weight threads on the OoO cores.

In this dissertation, we develop the designs of this architecture and execution model, present evaluations to illustrate its effectiveness in accelerating real applications, and offer comparative analyses to reveal its strengths over existing designs.

1.1 General Problems

Studies in academia use two primary metrics in the evaluation of a novel processor design: execution time and power consumption. In practice, processor designers must take into account other metrics, such as cost, but the vast majority of academic literature has used only time and power.

A corollary of Moore’s Law provided the prediction of the drastic increases in clock rate which drove performance increases in OoO processors during the 1990s [123]. However, the 2000s brought in a shift in design paradigms as the performance of OoO processors reached a plateau. The problems that caused this plateau include the memory wall, heightened design complexity, the power wall,
energy consumption, and limits on instruction-level parallelism (ILP).

The most widespread proposals to address these problems in recent years has been the rise of architectures and software techniques that seek to exploit parallelism at a coarser level of granularity than the instruction. With functional modules that are assigned larger tasks rather than smaller pieces of instructions, new architectures may be able to avoid many of the problems of conventional approaches. Multithreaded [78], SIMD [95], VLIW [54], and multicore [20, 59] architectures have arisen as possible alternatives.

The architectures in existence that are most relevant to this work are the class known as heterogeneous multicores [2]. The focus of this work is a novel heterogeneous multicore architecture for efficiently executing a software computational unit called a traveling thread. The architecture leverages a diverse set of design features that crosscut research in several existing processor designs. Such an architecture provides potentially high levels of parallelism at lower clock rates with lower power and energy consumption. However, these new architectures bring with them new challenges.

1.2 Specific Questions We Address

Although heterogeneous multicore processors have several advantages over other designs which offer promising potential for overcoming performance barriers, they present several challenges that are active areas of research. The first is the ongoing challenge of parallelization. The traditional method for programming is to develop applications in a sequential fashion. To transform a sequential application for execution on a parallel computer, either a programmer, compiler, microarchitecture, or a combination of the three must address several challenges.
One of these is the problem of control hazards. In a sequential application, the processor keeps one instruction counter, and a path of instructions directs the flow of branches that guides the overall execution path. In a parallel computer, multiple execution paths exist concurrently. If the programmer is to direct these paths, he must know which paths will execute on each processing element and convey these paths to the machine through the program.

A widespread technique for handling control hazards in the compiler or microarchitecture is control speculation, in which a mechanism guesses at the execution paths. A key problem with speculation is the costs of misspeculation recovery. Another problem is data hazards. Since multiple computations are executing at the same time, a problem occurs when one computation writes to a piece of data that another is reading. Again, the programmer must convey to the machine through the program which computations have independent working sets, or the compiler or microarchitecture can speculate and recover on misspeculations. Processor designs that focus on ILP extraction face these problems as well as those that use thread-level parallelism (TLP). But the problems are potentially more difficult with TLP, as the aggregate working set of the threads is larger, speculation is more aggressive, recovery costs are higher, and the processor is handling multiple levels of granularity of parallelism at one time. Further, the additional complexity of heterogeneous processing resources adds another layer of difficulty to parallel programming. The problem then becomes not just which computations to execute and which are independent, but also which resources are well-suited for executing a set of independent computations.

There are a host of old and new languages available for parallel programming, including Erlang [48], Scala [186], CUDA [131], and MPI [57], to name a few.
Experience and literature show that parallel programming has proven to be an intrinsically difficult problem. New languages and corresponding university curricula will continue to play important roles in advancing computer performance through parallel programming. However, as the extraction of TLP becomes more prevalent, the compiler designer and computer architect will have an increasingly important role in easing the burden of parallelization from the programmer.

To address all of the above issues, this work extends previous work in the traveling thread execution model and develops a novel hierarchical multi-level heterogeneous multicore processor [102]. The specific questions that this project addresses are as follows:

1. Is it feasible to design mechanisms for automatically extracting traveling threads from real sequential applications?
2. Is the traveling thread content in real applications sufficient to provide significant performance improvements?
3. What are the characteristics of a processor design that can efficiently execute these threads, and how is this processor distinguished from existing designs?
4. What level of improvements in execution time and energy consumption can a traveling thread processor offer over existing designs?

1.3 Objectives

The goal of this work is to establish a novel architecture to efficiently execute traveling threads for addressing the current challenges in high performance computing: the memory wall, the power wall, energy consumption, design complexity, limits on ILP, and others. To accomplish this goal, this work will meet the following objectives:

1. Define the traveling thread as a software-level computational unit and corresponding architecture for executing these threads.
2. Develop models for extracting these threads from sequential applications.

3. Define an execution model for the architecture for accelerating sequential applications with traveling threads.

4. Enumerate the design parameters of the architecture and thread extraction methods.

5. Perform statistical analyses to gauge traveling thread content in real applications.

6. Quantify the benefits of the design over existing processors in terms of execution time and energy consumption.

7. Illustrate whether the potential improvement differs for standard applications and scientific applications. If so, quantify this difference.

1.4 Approach

The first three objectives in the previous section involve the design of the architecture, execution model, and thread extraction methods. This work meets these objectives through an iterative design process. In the final design, we employ a design philosophy which addresses the problems of parallelization (thread partitioning) and locality exploitation in tandem. However, each of these problems is non-trivial. To be sure, each has constituted a subfield of research in computing on its own.

In the early design stages, we focus more on the computational partitioning aspects of the problem, establishing microarchitectures and execution models which advance existing processors with relatively simple modifications. In the later stages, we incorporate locality exploitation into the ideas incrementally. Each stage presents quantitative results that provide not only a gauge of how the performance of a new design compares with a baseline, but also an improvement in our understanding of the complexity and effectiveness of such principles and ideas.
Following the establishment of the final design, we present a graphical depiction of the design space for the microarchitecture, ISA, and toolchain – including parameters and metrics – to meet objective 4. The focus then shifts primarily to design evaluation to meet objectives 5 and 6. In the quantitative evaluations, we use both standard and scientific workloads, and comparative discussions of the results fulfill objective 7.

1.5 Contributions

The contributions of this work are:

- Development of a heterogeneous multicore processor and corresponding toolchain design for supporting the efficient execution of traveling threads. The project presents supporting theoretical models for extracting and executing locality-cognizant traveling threads transparently to the programmer.

- An illustration of the design space, including relevant parameters and metrics, for this processor and toolchain. In the design work, we present and employ effective methodologies for modeling, gauging performance improvement opportunities for novel concepts, and evaluating alternatives.

- Quantitative results illustrating that the processor running traveling threads can offer significant improvements over existing designs through the joint exploitation of parallelism and locality.

1.6 Outline of Dissertation

The dissertation contents are as follows. Chapter 2 presents an overview of the state of the art in computer architecture with an emphasis on topics particularly related to heterogeneous multicore, memory performance, compilers, multithreading, and low power. Chapters 3 gives the architectural designs called floating point instruction aggregates and in-cache computations that lay the foundations for the traveling thread architecture. Chapter 4 introduces the traveling thread architec-
ture called the *passive/active multicore* and an ISA called *BLITS*. Chapter 5 offers a thorough study of traveling threads’ performance improvement potential using analytical modeling. Chapter 6 develops the models for automatically extracting locality-cognizant traveling threads from applications. Chapter 7 presents an evaluation of the PAM design in terms of both performance and energy consumption. Chapter 8 gives overall conclusions of the work, and summarizes paths for future studies in this area.
CHAPTER 2

STATE OF THE ART

Computer architects face numerous obstacles that inhibit improvements to processor performance. These obstacles, or “walls” as they are popularly known, include the memory wall, power wall, and more recently the energy wall. The ideas in this work address these obstacles by building upon classic and recent concepts in the field of computer architecture.

This chapter explores previous work that has laid the foundation for this study, beginning with an overview of the major challenges in the field in the first section. The architectures this work introduces are primarily extensions of past research in processor-in-memory (PIM) designs, heterogeneous multicore, and multithreaded architectures. Of particular relevance are the studies involving processor designs featuring register files and functional units aligned to a memory macro to exploit the high bandwidth interface of embedded memory. Such a processor is well-suited for execution of wide operations in the form of Single-Instruction, Multiple-Data (SIMD) instructions or Very Long Instruction Words (VLIW). The second section describes this previous work in processor architectures.

In the third section, we offer an overview of some of the relevant compiler designs from previous projects. The fourth and fifth sections describe past work in advanced caching mechanisms and memory consistency models, while sixth
section presents common definitions of software-level threads. The final section offers background in designing for low power and energy.

2.1 Overview

This work is within the subfield of high performance computing known as computer architecture: the design aspects of the computer system that involve both the organizational details of the hardware, as well as the hardware/software interface. One can view the architecture of a machine as independent of the underlying technology, but the architectural design process should generally take into account both how it is influenced by and the influences it has upon the underlying hardware and the overlying software. By the hardware/software boundary, we are referring to the instruction set architecture (ISA), which is the specification of the behavior of the processor as seen by the program. When discussing one particular implementation of that behavioral specification from the standpoint of organizational and technological features, we are then referring to the processor’s microarchitecture. These two aspects of processor design are the focus of this work, and many important details and design points will be emphasized as they are addressed. Here, we will give a brief discussion of their current role in microprocessor system design.

ISA design was a very high profile topic during the CISC vs. RISC debate in the 1980s, and has not received as much attention since then. This is in large part due to the fact that ISA definitions tend to stay the same to avoid the need for recompilation of old software and rewriting of compilers. However, the key influences that decisions in the ISA specification have on many other aspects of microprocessor systems continue to arise as issues in modern designs. For example,
the ISA defines the placement of the dynamic-static boundary for the extraction of parallelism from codes [160], and it was this very issue that drove the emergence of VLIW and EPIC architectures in the 1990s. The presence of data parallelism in applications will most likely lead to a continuing interest in the SIMD extensions that have arisen in many ISAs. These two factors along with the increasing interest in the extraction of different forms of parallelism from applications lead to many questions that require careful tradeoff analyses in the complexity of the compiler and the hardware, as well as the usability and survivability of a design.

In recent decades, the microarchitectural features that have lead to increased performance have consisted primarily of deeper pipelines towards higher clock rates and complex hardware mechanisms to support advanced dynamic ILP-extraction techniques such as out-of-order execution and branch predication. The goal has been to extract as much ILP from the instruction stream as possible while minimizing the number of pipeline stalls. While these techniques have been quite effective in the past, their continuance is clearly questionable for several reasons.

The first of these reasons is the infamous memory wall [195]. Although both processor speeds and memory access times have improved exponentially, the former has improved much faster than the latter (by more than a factor of ten) [118]. Hence, regardless of the speed at which a centralized OoO processor is running, it will sit idle as long as it waits for data from memory that is relatively much slower. This barrier that the processor hits when reaching the limit on the memory speeds is known as the memory wall.

Arguably the most widespread solution to the memory wall problem is caches. Caches have provided a significant increase in latency avoidance that allow for increased processor speeds. However, caches have limits in their usefulness as
solutions to the memory wall problem. Previous work has shown that many applications of interest have working sets so large that the time between accesses is too great for caches of practical size to contain the data during the time that the processor needs it [125]. Further, many applications use algorithms that rely on irregular accesses to data structures that inherently rely on indirection, resulting in pointer chasing. Hence, not all applications exhibit the spatial and temporal locality characteristics from which caches benefit.

Another problem is the heightened design complexity of OoO architectures. To deal with the hazards within the instruction stream from branches and data dependencies, and delays from memory accesses, modern superscalar architectures rely on complex techniques such as speculation and data prefetching. However, as pipeline lengths increase with more and more instructions in-flight, the number of possible hazards increases, and dealing with them while also taking into account the possibilities of exceptions becomes a formidable task. Consequently, the design complexity, along with time and costs for production and verification, have increased tremendously. Also, clock rates have become so high that cross-chip clock propagation delays are no longer negligible.

Other problems have arisen at the hardware level due to high clock rates, as power consumption increases proportionally with clock rate. From this increase in power consumption along with Moore’s exponential increases in transistor counts, power density and associated heat problems have become primary concerns. Architects now face another barrier commonly known as the “power wall” [183]. More recently, energy has emerged as a primary design constraint for both portable computing and supercomputing [64]. Solutions to power and energy problems are active areas of research at both the system-level – which takes into consideration
global interconnects, non-volatile storage, and processing components – and at
the processor level – which takes into consideration on-chip networks, memories,
and processing logic.

Finally, evidence has shown that we have reached the limits on the amount
of ILP that is available in most applications [185]. Although ILP-extraction will
continue to be a key factor in high performance computing, the extraction of other
forms of parallelism of coarser granularity, such as thread-level parallelism, will
become increasingly important.

2.2 Architecture

This project draws upon many previous and ongoing studies in processor ar-
chitecture design. Here we offer a brief overview of the architectures of particular
relevance to this work.

2.2.1 PIM

Several previous research projects with processor-in-memory (PIM) have direct
relevance to this work, ranging in scope from fabrication of a PIM chip of low
design complexity to the definition and exploration of a promising execution model
for a PIM-based system.

An execution model that shows great promise for PIM-based architectures is
based on the extraction of very lightweight threads called threadlets [128]. A
traveling threadlet is a small package of instructions containing very little state
information that is tagged with an address and shipped to a memory site to
execute at the location of its operands. Threadlet migration refers to the shipment
of a threadlet to the memory site. In practice, we view threadlets as a form of
implicit multithreading, probably generated by a compiler. Threadlets should be as lightweight as possible while exploiting as much data locality as possible between migrations. They should be lightweight not only in terms of number of instructions, but also state. Previous work has presented thorough statistical analyses of applications – including the scientific applications mentioned earlier – to estimate their threadlet content.

Previous work has defined, modeled, even fabricated implementations of the PIM concept at the hardware level. Nanda selected a set of functional blocks to construct an on-chip memory and presented energy models and area estimates of them. He then developed an execution environment that assumes an at-the-sense-amp processor (ASAP) that uses the memory. An ASAP is a processor equipped with wide-word functional units that are pitch-matched to the memory macro for a design built for high on-chip bandwidth utilization, area efficiency, and other features. Nanda’s tool optimizes energy while varying the data output width. His results offer an explanation of the relationship between energy and output width.

Kang presents one architectural approach to PIM which has given strong preference to design options that offer simplicity and short development time. This PIM chip is called PIM Lite, and Kang has given specifications of the ISA and microarchitecture along with an implementation of PIM Lite in VHDL. He offers given code examples in assembly using the ISA, along with area estimates of the PIM Lite microarchitecture.

Thoziyoor has presented a VLSI implementation of another version of PIM Lite. This version, like its predecessor by Kang, gives strong preference to a design that would be relatively simple to implement and verify. It is a 16-bit
architecture that supports multithreading and SIMD operations. No registers are present, and the ISA specifies instruction operands as frames (regions of storage within memory), which can contain either scalar or vector data. Instruction and data SRAM caches are available, and his thesis gives power, performance, and area measurements of the design [169].

Zawodny has given various options for utilizing embedded DRAM technology to construct a system which integrates caching logic onto the same chip as DRAM memory. He constructed a simulator for the assumed architecture, which caches data in latches on the sense amps. Parameters of the cache configuration are aspect ratio, associativity, numbers of subarrays, and others. His results compare the new caching architecture to conventional SRAM caches, explaining the improvements or detriments to both energy consumption and performance for various applications [198].

At the architectural level, past work has extended PIM to suit particular applications and programming models and has explored execution models and design options. Murphy investigates design options of four different types, primarily at the system-level. Through simulation he shows whether a PIM chip is better used as a cache or as page space in terms of overhead and performance, options for the interconnection network, how to place data on the PIMs, and an architectural extension to support mobile threads [127].

Rodrigues has focused on finding a balanced solution between the conflicting goals in high performance computing of the need for new hardware to accelerate applications and the support of traditional languages in which these applications are written. He focuses on improving the latency and bandwidth of Light-Weight Processors (LWPs) to efficiently support the MPI and OpenMP programming
models, improving performance of process handling and communication, and presenting an algorithm for threadlet extraction. A large contribution of his work was the development of a simulation environment, SST, used for his experimentation.

Previous work has addressed many challenges that accompany a PIM-based system at the software level. Murphy defines a new execution model – based on moving small packages of instructions to the data in memory on which they will operate – for a PIM-based system and explores its potential for overcoming the von Neumann bottleneck through latency tolerance and avoidance. His results illustrate the specific advantages that this execution model offers over conventional approaches and other types of multithreading, what the packages will look like, and how a compiler might go about extracting them from applications. He includes a presentation of a theoretical machine that would contain the ideal architectural support for the execution model based on the results obtained.

Sridharan has incorporated three synchronization schemes for light-weight threads into SST and evaluated the performance of each. Vance presents an LWP architecture and uses simulation to test the performance of various options for thread management in terms of throughput by decreasing latency and increasing concurrency. Both threads and data are stored in memory frames which are cacheable. Vance evaluates different policies for swapping these frames in and out of the cache.

2.2.2 Heterogeneous Multicore

In response to the problems met with conventional architectures mentioned in section 2.1, a large amount of research in the last decade has been devoted
to a different utilization of on-chip resources. Rather than use these resources to
design one, complex processor on the chip that is equipped with reorder buffers,
branch prediction, and many functional units, the chip multiprocessor or multicore
processor \[84, 98\] generally uses a single, simple processor design that is tiled across
the chip. Such a design style has many advantages that address the issues with
conventional architectures.

Zhong et al. propose a dual-mode architecture called Voltron that features a
scalar operand network among cores’ register files \[202\]. Voltron efficiently exploits
instruction-level parallelism, fine-grained thread-level parallelism, and loop-level
parallelism in general purpose applications. The study evaluates the performance
of two- and four-core Voltron processors.

Pangaea integrates a core implementing the IA32 ISA and a number of GMA
X4500 GPU (graphics processing unit) cores on the same chip \[194\].

Various studies with STI’s Cell Broadband Engine \[81\] have dealt with issues
similar to those addressed here. The Cell consists of nine cores on a single chip:
one 64-bit, multithreaded Power Processing Element (PPE) and eight Synergistic
Processing Elements (SPE) \[65\] supporting 128-bit SIMD execution. The cores
communicate through an Element Interconnect Bus (EIB) which allows a peak
throughput of up to 307.2 GB/s \[6\]. Each SPE has a 256-KB scratchpad memory
which leads to great flexibility in functional and data decomposition. IBM has
published a tutorial presenting seven types of programming models for the Cell
\[79\]. This document can serve as an introduction for conceptualizing how to map
applications to a heterogeneous multicore architecture. Blagojevic et al. have
presented an adaptive run-time system for dynamically partitioning and schedul-
ing computations with multiple levels of granularity across the Cell’s processing
elements for bioinformatics applications [18]. The work seeks to accelerate an MPI-based implementation of a sequencing application by partitioning MPI processes at function and loop boundaries across the Cell’s cores.

2.2.3 GPUs

*Graphics Processing Units* or GPUs are computing devices that operate in conjunction with a conventional processor to accelerate sections of code with high parallelism and regularity through *stream processing*. In stream processing, cores perform multiple instances of identical operations on a set of multiple data operands. This is similar to SIMD, but with stream processing the operations form instances of identical threads, rather than isolated operations. As its name suggests, the conventional use of the GPU was to accelerate video processing, which exhibits these computational patterns with high regularity and parallelism (so-called “embarrassingly parallel”). However, in the last decade, the field of general-purpose computing on GPUs (GPGPU) has emerged to harness the power of GPUs for non-graphics applications. Recent studies in GPGPU have shown significant speedups for scientific applications, data mining, and others. Phillips et al. report up to 7X speedup for a molecular dynamics package [137], experiments by Ma et al. have shown 50X speedup on data mining applications with the use of GPUs [113]. Gupta et al. have used GPUs to accelerate virtual machines [66].

In 2009, NVIDIA introduced the Fermi GPU which features 512 cores in groups of thirty-two grids of cores [41]. The GPU can support up to 6GB of memory and interfaces with the host processor via the standard PCI Express bus. The cores support double precision floating point and can perform 512 fused multiply-add
operations per clock at a 700MHz clock rate.

Intel has also recently introduced an architecture named Larrabee that will compete in the GPU market [158]. Larrabee features support for the x86 ISA, coherent L2 caches of 256-KB per core, and a 16-wide vector arithmetic logic unit (ALU) in each core. The cores communicate via a 1024-bit wide ring network. The cores differ from traditional GPU cores in that they have a wider range of flexibility for computations. While a conventional GPU core achieves high throughput by executing many instances of very simple threads, the Larrabee core offers backward compatibility for existing x86 applications and can even run OS kernels. Larrabee also extends flexibility by performing various video processing functions in software rather than with specialized hardware as in traditional GPU designs.

2.2.4 Asymmetric Multicore

Asymmetric multicores have emerged as a type of heterogeneous multicore that features multiple cores on the chip with different power and performance characteristics. In the general case, these different core designs implement the same ISA. Suleman et al. have considered using asymmetric multicores to accelerate multithreaded applications [164]. The focus of their study is on critical sections using a processor consisting of one large core and several small cores which send requests to the large core to execute critical sections. Balakrishnan et al. consider numerous multithreaded workloads run on both homogeneous and heterogeneous multicore processors to gain insight into how asymmetry affects performance, performance predictability, and scalability [10]. They consider whether exposing hardware asymmetry to the operating system is sufficient to eliminate
performance unpredictability, or if restructuring of the applications is necessary.

2.2.5 Multithreaded Architectures

Multithreading offers a different approach to achieving higher throughput in applications from that of conventional architectures. The use of multiple instruction streams to tolerate latency and expose parallelism is far from a new concept, yet many emerging architectures employ this approach in the form of multithreading to combat the worsening problem of memory delays that can result in an unacceptably large number of wasted valuable clock cycles. By keeping a few instruction streams active and ready to issue from, the processor can perform useful work while waiting on memory requests [78].

There exist many techniques for the extraction and execution of multiple threads at the hardware level. While in most cases the hardware provides multiple register sets and program counters to schedule concurrently existing threads into a single pipeline, these threads could be user-defined – as is the case with explicit multithreading – or generated by the compiler or hardware components – with implicit multithreading. Further options exist in the execution of these threads regarding whether the processor can schedule multiple threads to available functional units at the same time (simultaneous multithreading or SMT) or relying on some event to cause a hardware switch of the single active thread [177].

One of the most recently manufactured multithreaded architectures is the third generation system in the MTA line of architectures from Cray. *Eldorado* (now *XMT*) is a shared memory system in which all processors have unified access to all memory and threads that are ready for execution [51]. Although each processor runs at a modest 500 MHz, the system overall boasts high performance by giving
support for up to 128 instruction streams with each processor. By alternating instruction issue from different streams, the processor can continue to do useful work from ready streams while other streams wait on a memory accesses.

Previous work has considered multicore architectures supporting different forms of multithreading, including speculative multithreading\cite{32} and fine-grained multithreading\cite{114}. Chaudhry et al. propose Simultaneous Speculative Threading (SST) featured in Sun’s ROCK processor. Using a design tailored to exploit ILP and MLP (Memory-Level Parallelism), their results show that the SST on 8- and 32-core homogeneous multicores outperforms a traditional out-of-order processor for standard and commercial workloads\cite{32}. Madriles et al. present Anaphase, a threading scheme that uses compiler-generated, fine-grained speculative threads and microarchitectural extensions to the cores and caches for supporting checkpointing, memory order violation detection, and rollback. Chu et al. consider a multicore that resembles a clustered VLIW with distributed data caches and use profile-guided analysis to partition memory accesses and assign operations across cores to reduce memory stall time\cite{38}.

2.2.6 Thread Migration

Murphy has given a comprehensive study of an execution model built upon migrant threads\cite{128}, discussed in section 2.2.1. Sarkar et al. consider task migration in the context of embedded systems with scheduling constraints. They address the problem of cache warm-up in migrant tasks by proposing a form of OS-managed cache line migration\cite{155}. Strong et al. propose software-based policies for moving threads among cores by modifying thread migration mechanisms in the Linux OS\cite{163}, and evaluate these mechanisms on a homogeneous dual-core.
and a simulated heterogeneous quad-core. Rangan et al. use mobile threads on multicores that are heterogeneous in terms of the voltage and clock frequency of the cores to enable fine-grained power management [143].

2.2.7 Memory-Side Computations

Recent work has considered augmenting a conventional processor with compute logic in either the memory controllers or next to memory macros [170] using embedded DRAM technology [80]. Architectures with tight coupling between processing and memory include the Impulse Memory Controller [200], Active Memory Operations (AMOs) [49], and LWPs [111]. Impulse adds a layer of indirection between main memory and the processor at the memory controller, allowing the application or OS to reorganize data in memory to improve memory bandwidth utilization and cache performance. AMOs extend Impulse by allowing computations at the memory controllers of distributed shared memory systems, as opposed to just providing a mechanism for remapping of data addresses.

Li et al. proposed an architecture utilizing logic next to memory macros [111]. This architecture replicates an LWP that supports large numbers of simultaneous threads. The programmer creates these threads through a special language supporting this style of massive multithreading. An even simpler core is associated with each memory bank to assist in thread management and atomic operations.

The Yukon-256 by Micron replicates 256 processing elements next to a 16 MB DRAM macro [89]. The processors act as accelerators to a main host processor. The programmer writes the application in C++, using special libraries to offload computations from the host to Yukon. The Yukon microarchitecture includes components for efficient communication from the host to the DRAM chips. The
processing elements operate in a systolic fashion, shifting data in lock-step from
one column to the next.

The Data-IntensiVe Architecture or DIVA is a similar architecture that uses
embedded DRAM to act in cooperation with host processor [45]. Draper et al.
present the concept of a parcel as a unit of communication for DIVA, which refers-
ences execution sites by objects in memory rather than processor IDs. The DIVA
processor includes one datapath for 32-bit integer processing and another for 256-
bit wide-word processing. Their study evaluates DIVA using eight programs, and
results show an average improvement of 3.3X speedup over the standalone host
and up to a 95% reduction in host idle time due to memory stalls.

2.2.8 VLIW

Architectures utilizing VLIW or Very Long Instruction Words offer an al-
ternative to the superscalar approach of ILP-extraction [39]. While superscalar
architectures rely on hardware mechanisms to extract ILP dynamically, VLIW
architectures rely on static or compile-time techniques to search for ILP. Instead
of having short instruction words that contain a single operation, VLIWs contain
multiple operations within a single instruction word. The compiler then augments
the instruction with a template field, which indicates to the hardware which oper-
ations it can execute in parallel. VLIW lengths have spanned from only 120 bits
(5 operations) [179] to 1200 bits (30 operations) [53]. The primary benefit of the
VLIW approach is that the hardware design is greatly simplified. Not only is the
dependency-detection and instruction reordering hardware no longer necessary,
the decoding and issuing of each instruction is identical and simplified. Another
advantage is that the ILP-extraction is no longer limited to a fixed window size as
it is with conventional superscalar architectures. The compiler can perform global ILP-extraction and optimizations before run-time and guide the simple hardware with the use of templates. Assigning instruction scheduling to the compiler allows for other opportunities including arranging instructions in such a way as to minimize power consumption \[21, 106\]. VLIW disadvantages include increased code size and lack of hardware portability (since the compiler needs explicit knowledge of the hardware to carry out scheduling).

Intel and HP have shown efforts to transcend VLIW and overcome these challenges through a new approach to ILP-extraction called EPIC, which features advanced compile-time techniques for dealing with branches and high latency memory accesses \[157\]. More recently, clustered VLIW architectures have emerged which replicate multiple VLIW cores on a chip with a division of storage resources among them and an efficient means of communication \[60, 153, 201\].

2.2.9 Dataflow Architectures

Dataflow architectures use an execution model in which instruction initiation occurs by the availability of operands, rather than by an instruction pointer as in conventional architectures. A convenient representation of instruction execution on a dataflow machine is a depiction of instructions and dependencies in a directed acyclic graph (DAG). Each node in the DAG represents an instruction, and an edge between two nodes represents a dependence. A *token* at run-time represents an operand between two instructions.

The two primary categories of dataflow architectures are *static* and *dynamic*. In static dataflow, only one instance of a node in the DAG exists at time. Hence, the instruction corresponding to this node is unique, and the architecture can
direct tokens to this node by its address. In dynamic dataflow, more than one instance of the node may exist. A popular implementation of dynamic dataflow, called tagged-token dataflow architectures, marks tokens with tags and initiates instruction execution when matching tokens are available [105].

WaveScalar is a tagged-token dataflow architecture that provides sequential memory semantics for execution of conventional, single-threaded applications while offering the benefits of a dataflow execution model [165]. The compiler cuts the dataflow graphs of instructions into groups called waves. To provide sequential memory semantics, the authors present the novel concept of wave-ordered memory, in which the compiler marks memory accesses within waves with values called sequence numbers to prevent the machine from violating ordering constraints.

TRIPS (the Tera-op, Reliable, Intelligently adaptive Processing System) is an example of an architecture that exhibits properties of both dataflow machines and von Neumann machines [154]. The compiler groups instructions into blocks which have one entry point and commit atomically. Blocks have sequential ordering – as instructions do in a traditional von Neumann machine – but instructions execute in a dataflow fashion. The explicit data graph execution or EDGE ISA allows the compiler to encode dataflow parallelism directly into the instructions.

The Grid Processor Architecture or GPA consists of a two-dimensional array of ALUs for which the compiler assigns instructions to ALUs while the microarchitecture dynamically initiates the instructions in a dataflow fashion [129]. The compiler encodes the destination of an operation’s results directly into an instruction as the name of an ALU, and the processor transmits these results directly to the destination through a point-to-point network. An important benefit of the GPA is the reduction in the number of large, centralized structures relative to
out-of-order processors, which lowers the critical path latency through the hardware.

2.2.10 Instruction Aggregation

*Instruction aggregation* lowers the run-time overhead of individual instructions by grouping them into packages and handling the packages as a unit. Previous research has explored instruction aggregates to reduce power consumption and increase performance. Minigraphs are a type of instruction aggregate with a bounded, small number of register inputs and outputs [23, 24]. Sharkey et al. considered instruction packing – which allows pairs of instructions to share pipeline resources – to improve performance and power consumption [15]. Macro-op scheduling focuses on combining instructions of single-cycle latency [87] into serial multi-cycle operations to relax the atomicity constraints of instruction wakeup and select and allow increased pipelining and decreased instruction window overhead.

Hu and Smith implement a mechanism for fusing instructions in a dynamic binary translator for the x86 instruction set [75]. By combining pairs of instructions at run-time, the fusion mechanisms can reduce window scheduling slots and instruction storage required. Sassone and Wills present microarchitectural extensions for detecting and accelerating *strands*, which are sequential instructions that have only internal (transient) operands [156]. The extensions for detecting strands are off the critical path, and the strands execute on ALUs which feedback their outputs to their inputs, allowing the execution of two operations in a single cycle.

Part of this dissertation work considers instruction aggregation to accelerate
floating-point intensive codes. It is important to note here that previous work with custom ISAs and configurable architectures generally has not focused on floating point acceleration [58, 76, 107], although a limited number of such studies has been done [35, 47].

2.2.11 SIMD

One of the advantages of integrating logic on-chip with DRAM is the dramatic increase in bandwidth between the processor and the memory. One architectural approach to exploiting this bandwidth is to use wide-word functional units with the datapath pitch-matched to the sense amplifiers of the memory macro [26, 45, 61, 170]. Wide-word ALU’s offer a number of strengths that make it an attractive option in the context of PIM. Architectures that use wide-word ALU’s to execute one operation across multiple data elements are known as SIMD (Single-Instruction, Multiple-Data) architectures.

Past research in both architectural evaluation and compilation has explored SIMD as extensions to conventional architectures. Some examples include Intel’s MMX and SSE for IA-32 [142], Motorola’s AltiVec for the PowerPC [43], Sun’s VIS for the Sparc [172], HP’s MAX for the PA-RISC, SGI’s MDMX to MIPS, and Compaq’s MVI to Alpha [31].

SIMD instructions are defined as those that execute one operation across multiple operands. This type of parallelism, based on uniform operations, is called data-level parallelism or DLP. DLP is prevalent in many multimedia applications, such as video, modem filters, 3-D graphics, audio, and many others, but is also found in general purpose applications. In addition to performance and intrachip bandwidth benefits, other advantages of SIMD include decreased power consump-
Architects have investigated many ISA designs consisting of a conventional set of instructions along with an additional set (called SIMD extensions) for exploiting data parallelism. Intel’s i860, Motorola’s 88110, and HP’s PA-7100LC are all early examples of processors implementing an instruction set which included a few additional instructions for performing single operations across portions of a 32-bit word with the intention of speeding up video processing. These ideas eventually evolved into the more advanced SIMD extensions discussed here, such as the early VIS and the more recent AltiVec and SSE.

2.2.11.1 VIS

Sun’s Visual Instruction Set or VIS consists of about 24 instructions that operate on 64-bit data words. The architecture can subdivide words into into 8-, 16-, or 32-bit integers of fixed point values. There are five types of SIMD instructions, including conversion, arithmetic/logical, address manipulation, memory access, and motion estimation.

The conversion and arithmetic/logical instructions perform expected actions such as translating data stored as 8-bit values into 16-bit values and vice versa, or adding subword components of vector operands. Some specialized actions are included such as combining components of words into one in an interleaved format. For instance, the fpmerge instruction takes two 32-bit values formatted as four one-byte numbers and combines them into one 64-bit value in an interleaved format as shown in figure 2.1. This instruction allows conversion between band-interleaved and band-sequential format, which is important for rotation and transposition of 2-D arrays.
Address manipulation instructions attempt to improve efficiency of accesses into 3D data sets. The natural way to store a 3D data set would be to store elements with consecutive indices along a dimension at consecutive addresses. The VIS address manipulation instructions instead will provide addresses so that elements that are spatially close to each other (in the same “brick” in the data set) will fall on the same cache line. The instruction takes as an argument a vector containing the 3D coordinates of an element in the set, and returns the appropriate offset from the base address of the set.

In many cases, the software needs to store to only portions of the data at an 8-byte aligned address, rather than the entire piece of data at that address. One reason for this is that the operands to be modified are simply not aligned in memory. To handle this case, the edge instruction generates a mask for performing a partial store. The store will modify only the bytes in memory corresponding to the bits that are set in the mask. Similarly, conditionals may result in situations where an instruction should not operate on all 8 bytes. Standard comparison operators generate masks for performing partial stores in these cases. For example,
an operation may need to preserve previous values in memory for bytes where the result of the operation is zero. An equals test generates a mask so that instructions modify only the appropriate bytes.

Memory access instructions include those used in conjunction with address manipulation to store to and operate on pieces of vectors using masks. VIS also allows the movement of large blocks directly to and from memory or the frame buffer and the floating point register file without disrupting the contents of the cache. Applications with large data sets that touch many consecutive memory locations with a single operation can use these non-blocking instructions to improve performance.

The motion estimation instructions seek to improve the performance of video compression applications. The pdist instruction implements the commonly used sum-of-absolute-differences operation between two vectors.

VIS overloads the floating point register file and datapath for use with SIMD operations. In addition to being a simple solution that requires a relatively small amount of additional hardware, this approach offers a few advantages [172]:

- Integer registers are free for working on addresses and branches.
- SIMD instructions can easily use the floating point pipeline which support multicycle instructions.
- Implementing SIMD support in the hardware does not affect the critical path or number of register file ports of the integer pipeline.
- Avoids adding new architectural state, requiring modification to the OS, which would come with adding dedicated vector registers.

2.2.11.2 AltiVec

AltiVec is a set of 162 SIMD instructions that extends Motorola’s PowerPC architecture. The instructions operate on 128-bit operands and can split the
operands into 8-, 16-, or 32-bit integers or 32-bit single-precision floating point values in a 32-entry register file. The extensions seek to increase performance in multimedia and embedded applications, and include instructions that can quickly perform common matrix operations, memory copies, and string compares.

AltiVec includes many important design features that advance the state of the art to improve SIMD performance. AltiVec instructions use a four-operand format (three-source, one-destination), which is important since this allows instructions to be non-destructive and avoids unnecessary copying of results. Most of the instructions are flexible in their support for different length data types, including signed and unsigned integers and floating point, with support for both modulo and saturation arithmetic.

In addition to the mask-generating conditional instructions and special loads and stores that can work on components of vectors that are found in VIS, AltiVec also provides the vperm instruction which allows arbitrary rearrangement of components in a vector. It is difficult to achieve large performance gains with SIMD without the vperm feature, since the storage format of operands in data structures do not usually line up with the SIMD functional units and excessive time is spent shifting data to remedy this. This instruction is a four-operand instruction, with the first operand specifying the destination register, the next two specifying 128-bit registers containing source data, and the last giving a register containing a 128-bit permutation mask. The $i$th byte in the mask specifies two indices $j$ and $k$. The $j$th byte of one of the operands will be placed into position $i$ of the destination register. The byte from the second operand is used if $k=0$, and the byte from the first operand is used if $k=1$. The four-operand format also allows for instructions that perform common operations that use two actions, such
as multiply-add and multiply-sum. Figure 2.2 includes a simple example.

Altivec also includes more instructions that implement operations that are common in multimedia applications, as well as instructions for explicitly controlling cache behavior with prefetching.

This architecture is different from VIS as it operates on data in vector registers dedicated for holding vector operands. Including a separate vector register file is preferable for the following reasons [43]:

- Integer and floating point registers are free while SIMD is in use, making scalar processing independent from SIMD processing.
- Vector register file interface can be physically optimized for SIMD functional units.
- Simplifies instruction scheduling in the compiler.

2.2.11.3 MMX and SSE

Intel has produced a number of SIMD instruction set extensions for their architectures, beginning in the mid 90’s with MultiMedia eXtensions (MMX), 57 new instructions added to IA-32. MMX is like VIS in that it uses the floating point registers to store SIMD operands. The instructions operates on 64-bit values and can subdivide them into fixed-point 8-, 16-, or 32-bit integers or use them as one 64-bit integer. A primary design point for MMX was backward compatibility. Since MMX avoided using a new register set, control registers, or exceptions for executing MMX instructions, OS vendors could leave their software unmodified when running on processors equipped with the new extensions [134].

MMX is a fairly primitive form of SIMD extensions. The design includes many of the standard and essential features found in more recent versions, such as arithmetic and logic operations across vectors, support for saturation arithmetic, data
type conversion, and generation of conditional vector masks. However, arbitrary
permutes, floating point, and handling of misaligned operands are features that
the instruction set lacks, making MMX essentially a barebones design used as a
foundation for Intel’s future versions.

Streaming SIMD Extensions (SSE) were Intel’s next generation of SIMD in-
structions [168]. The first version of SSE supported 128-bit operations, a dedicated
set of vector registers, floating point support, explicit cache control instructions
for data prefetching and streaming, and special instructions for carrying out com-
mon multimedia operations. SSE began as a set of 70 additional instructions to
IA-32, and Intel has since introduced three more versions of SSE [140].

The SSE design relieves the constraint of backward compatibility used in
MMX, adding eight 128-bit registers dedicated for use with SIMD, along with
a new control/status register and interrupt vector. SSE and AltiVec have many
similarities at the ISA level. Both sets of instructions operate on 128-bit operands
and support various data types including floating point. They both provide in-
structions for explicit cache control and data movement among memory, cache
levels, and registers. Instructions for data type conversions and commonly-used
multimedia operations are provided with both architectures.

However, there are a few significant differences between the two architectures
that are noteworthy. Although both architectures use vector registers, AltiVec
has many more dedicated registers. While SSE can use eight of the 64-bit MMX
registers that overlap the 80-bit floating point registers along with the eight 128-
bit vector registers, AltiVec uses a 32-entry register file exclusively for SIMD. Also,
SSE retains the three-operand instruction format that is common to RISC-style
ISA’s, as opposed to AltiVec which uses a four-operand format.
SSE permutation instructions are limited compared to AltiVec. Recall that AltiVec allows arbitrary permutes, using a 128-bit permutation mask stored in a register. SSE, on the other hand, uses immediate values as permutation masks. Because these immediate values are limited to 8 bits in length, their flexibility is limited as well. Therefore, SSE provides a set of shuffle instructions that cover a range of common permutation operations. Figure 2.2 shows a comparison of the two alternatives.

Both assume that it is not the responsibility of the hardware to handle data alignment (data should be aligned to 16-byte boundaries). SSE signals faults to the system when an unaligned memory access occurs, but also provides a few instructions for efficiently handling unaligned vector loads and stores.
The versions of SSE that followed the initial release added various instructions that were generally application-specific operations. SSE2 provided operations for 128-bit integers and double-precision floating point, while SSE3 added instructions to improve performance of thread synchronization. Supplemental SSE3 and SSE4 have followed, extending the set even further, with the most notable features being some new instructions for text processing and others to improve code generation. More recently, Intel released Application Targeted Accelerators, which are application-specific functional units exposed to the software through special instructions [140].

2.2.11.4 SWAR

Another relevant project has focused on how to achieve the performance benefits of SIMD hardware with minimal extensions to the scalar hardware of conventional microarchitectures. Conventional SIMD hardware supports fixed data types, usually groups of 8-bit and 16-bit values, and possibly others. The insight of SIMD-within-a-register (SWAR) is that slight modifications to hardware designed for scalar operations allow the support of SIMD operations on data types of varying length [55]. SWAR views data in scalar registers as vectors containing a few smaller data components. Research has shown that SWAR can easily support many common SIMD operations, such as addition and subtraction (saturating and modulo), reductions, and conditional mask generation. Since SWAR is essentially using scalar features of conventional architectures to implement SIMD operations, a primary challenge of SWAR is how to provide a portable interface with implementations for a range of varying architectures.

The first key component of SWAR work explores options for how to imple-
ment the SIMD operations with scalar hardware. For instance, 4-operand, 8-bit addition can be executed on 32-bit hardware with no hardware modification. The challenge is in dealing with carry bits from each add without having to set aside bits exclusively as “spacers”, which result in wasted data. The paper develops a method for achieving this, along with implementations of reductions, a limited form of inter-processor communication, and enable masking.

The second part discusses various compiler optimizations when working with a conventional architecture in this special way. One important detail is that for a given architecture, SWAR handles certain data sizes efficiently, while for others it is more suitable to promote the size to the next highest standard data type. The compiler must be aware of which sizes work well for a given architecture, and how to choose which data size will yield the best performance. The study explores other aspects of the compiler further, such as how to track values in the context of variable-length data types with spacers, and how to mock enable-masking given that fields of a scalar register cannot be disabled.

2.3 Compilation

A large amount of previous work has addressed the problem of improving performance transparently to the programmer through advanced compiler designs. Traveling thread architectures rely on work presented in this section on SIMD compilation thread-level speculation.

2.3.1 SIMD Compilation

ISA extensions are useless unless the software level has some means of determining when the applications can utilize new instructions. Many projects have
explored methods for producing code that uses SIMD extensions. The obvious approach is to manually edit the assembly code, replacing groups of instructions in the sequential stream with a corresponding SIMD instruction that will produce the same result, where appropriate. Another explicit approach is to write a new high-level language that provides routines for the programmer to specify where to use SIMD extensions. Some examples of such a language are SWARC [52] (a language to target SWAR hardware – see section 2.2.11.4) and FORTRAN 8x [3]. The drawback to these explicit approaches is that previously written applications (so-called legacy codes) would have to be re-written by hand to use the new instructions.

Projects in both research and industry have investigated implicit techniques for SIMD extraction, primarily in the form of compilers that analyze sequential code (at various steps in the compilation process) and attempt to automatically select appropriate SIMD instructions. Studies into these special compilers began long ago for vector machines such as the Cray-1 [136] and Texas Instruments’ Advanced Scientific Computer [191]. Recent studies have produced new techniques – by evolving these past methods for vector machines or designing novel approaches – for achieving an auto-vectorizing compiler for architectures with SIMD support [92, 108, 145].

2.3.2 Allen-Kennedy Algorithm

One approach to auto-vectorization that has served as a foundation to many modern algorithms is the Allen-Kennedy algorithm [7]. Allen and Kennedy aimed to write an application for translating sequential FORTRAN code into FORTRAN 8x code, a dialect of FORTRAN that includes support for vector operations. To do
this, the translator focused on dependence analysis of program statements within loops. The first development of the initial steps of the algorithm use the simple case of single loops. Consider the simple loops below:

```c
for (i = 0; i < 10; i++)
    x[i] = x[i] + 1;
```

```c
for (i = 0; i < 10; i++)
    x[i+1] = x[i] + 1;
```

In the first example, the statements in the loop can easily be vectorized into a single operation that adds one to each current value in the array. However, the statements in the second example are inherently sequential, since a given calculation depends on the result of the previous iteration. This loop contains interstatement dependencies, and this type of statement is said to be a recurrence. To determine whether or not the loop statement is a recurrence in the general case is a very difficult problem. The paper first constrains the functions of loop variables used as indices into the arrays to only linear functions, which is usually the case in practice. It goes on to define a set of loop transformations that convert a loop to canonical form, along with algorithms to determine whether or not the statement in the loop is a recurrence.

Extensions to the algorithm handle multiple statements within a loop. This case is more complex because, in addition to the single-statement, cross-loop dependencies described above, dependencies can now occur between multiple statements. The two types of dependencies are loop-carried dependencies, in which one statement stores to a location that is accessed by another statement in a later loop iteration, and loop-independent dependencies, in which one statement stores to a location that is read by another statement in the same iteration. The software uses analyses on a dependence graph to reveal which types of dependencies exist in
the loop statements. The situation becomes even more complex when considering nested loops. Allen and Kennedy develop the algorithm further to consider these cases.

Their study also considers the additional topics of code generation and conditional statements. The techniques handle conditional statements using a combination of *IF-conversion*, which converts control dependencies to data dependencies, and *scalar expansion*, which is a conversion of a set of conditional scalar values into a vector.

### 2.3.3 SIMD Operations in Basic Blocks

While many autovectorization tools – such as the one described in the last section – focus on extracting SIMD operations from loops, other studies have developed algorithms for vectorizing linear code sequences that contain no loops. Kudriavtsev gives an example of such a tool \[95\]. While many vectorizers work by translating source code from one high-level language to another, this is an example of the embedding of a vectorizer into the compiler, analyzing the code in the form of an intermediate representation (IR). IR presents the code as a set of dataflow graphs (DFGs), where nodes represent operations such as addition, subtraction, loads and stores, or data such as registers or variables. To generate the assembly code, trees represent instructions in the target ISA, and the compiler matches these trees with subtrees of the DFGs. Each instruction can have various score components associated with it, such as execution time and power consumption, and a primary goal of the compiler is to optimize the assembly code generated in terms of this score.

To extract SIMD operations, the algorithm first must attempt to find *SIMD*
groups: sets of nodes in the DFG for translation into single SIMD instructions in the target ISA. The first step is to group together loads and stores of the same data type on consecutive memory addresses. These groups will make up a load or store to or from a SIMD register. An analysis of the operations on the data in these nodes representing SIMD loads and stores determines if they satisfy certain conditions and can form a SIMD operation.

A key challenge to this problem occurs when instructions do not use SIMD operands in the simplest fashion. Consider the two examples below:

\[
\begin{align*}
  x[i] &= y[i] + z[i] \\
  x[i+1] &= y[i+1] + z[i+1] \\
  x[i+2] &= y[i+2] + z[i+2] \\
  x[i+3] &= y[i+3] + z[i+3] \\
  x[i] &= y[i+1] + z[i-1] \\
  x[i+1] &= y[i-1] + z[i+1] \\
  x[i+2] &= y[i-1] + z[i-1] \\
  x[i+3] &= y[i+1] + z[i+1]
\end{align*}
\]

The conversion of the first set of operations simply entails two vector loads, a vector add, and a store of the result. In the second set, however, the operands in the addition are not consecutive memory addresses. The vectorization requires the rearrangement or even replication of some components of the vector to perform the desired operations. This transformation of a vector from its original layout in memory is called a permutation, and SIMD extensions generally provide instructions for generating these. Since permute instructions are overhead in the SIMD computational process, the goal of the compiler is to arrange the instructions in such a way that minimizes the number of permute instructions. Solving this problem turns out to be very difficult. A DFG with \( N \) nodes able to be permuted in any possible order with an architecture with SIMD functional units of width \( W \) has a total number of \( W!^N \) possibilities. Therefore, Kudriavtsev \[94\] makes the
simplifying assumption that the permutation chosen for a given node in a graph will be the same for at least one of its neighbors, which is a reasonable assumption in practice. Based on this assumption, the algorithm generates orderings and propagates them through the DFG, starting with memory operations with fixed orderings. With this new algorithm, the number of choices of orderings still grows exponentially with the size of the tree, but is no longer dependent on the width of the SIMD units: a great advantage over previously designed algorithms. Since many ISAs do not have instructions supporting arbitrary permutes, the problem remains of which permute instructions offer the most optimal sequence for achieving the chosen orderings. The paper proceeds to offer one solution to this problem.

2.3.4 Alignment of Short Vector Operands

One issue with processors that include SIMD functional units that operate on vectors of data is the alignment of vector operands in memory. Handling of unaligned data is an issue even when working with non-vector operands, and many microarchitectures require that the compiler align all data to the word, throwing exceptions or failing on an attempt to issue a load or store of an unaligned word. Other microarchitectures can extract the requested unaligned word at a large performance penalty.

Short vector operands that span 16 bytes or more cause serious performance problems in this respect, since, if unaligned, they are more likely to span cache lines or even pages. Current architectures with SIMD support have a few solutions to this problem.

Intel’s SSE operates on 16-byte operands and provides load and store instruc-
tions for the cases when the operands are known to be aligned and otherwise [40]. For instance, when loading an aligned double-quad word (128-bit value) from memory, the programmer or compiler should use the \texttt{MOVDQA} instruction. When data may be unaligned, the \texttt{MOVUPS} instruction will carry out the steps necessary to obtain the requested data in the case when it is not aligned, usually using multiple 64-bit loads. Yet another option is the \texttt{LDDQU} instruction, which is another implementation of the unaligned double-quad word load, optimized for the case when the data spans a cache line. Since \texttt{MOVUPS} and \texttt{LDDQU} may fetch excessive data, the compiler should use them only when necessary. One implementation of \texttt{MOVUPS} initiates two loads of 64-bit values, and merges the two to obtain the requested double-quad word. The \texttt{LDDQU} instruction loads an aligned 32-byte value and extracts the desired 128-bit value from this [42].

Another approach to the alignment issue is to allow the microarchitecture (rather than the compiler or programmer as above) to control instruction execution based on whether or not the operands are aligned. The ICLG (Intra-Cache Line Gathers) mechanism takes this approach [176].

2.3.5 Thread-Level Speculation

Previous work has focused particularly on partitioning sequential code into multiple threads of execution, and studies in the formation of speculative threads by the compiler are highly relevant. Bhowmik and Franklin present a framework for speculative thread compilation, addressing several challenges to speculative multithreading [16]. The Control Flow Graph (CFG) and thread dependence analysis techniques have similarities to the graph analyses used in this work, but their work focuses on relatively coarse-grained threads (>30 dynamic instructions)
and does not consider data locality. Dou and Cintra offer an extensive cost model for speculative thread compilation, enumerating five overheads incurred by speculative threads and using probabilistic analysis to estimate execution times [44].

2.4 Advanced Caching Techniques

Caches have played a key role in addressing the memory wall problem through the exploitation of locality. Research continues to investigate new cache designs to offer further improvements. This section presents three advancements to basic caches.

2.4.1 Non-Blocking Caches

_Non-blocking or lockup-free caches_, an architectural concept that dates back at least two decades [93], use the principles that the processor can still do useful work while waiting on a memory request, and that multiple memory requests can take place at that same time. Rather than locking the processor while a memory request completes, specialized hardware keeps a record of pending memory requests and which processor components are waiting on that data. The effectiveness of non-blocking caches relies on large _non-blocking distance_, which is simply the number of instructions occurring after the memory reference and before the actual use of the data. Compilers can assist in increasing the non-blocking distance, but this problem becomes much more complex when taking into account out-of-order execution and branch prediction. Non-blocking caches have been an interesting topic of study in principle, but it seems that most research in advanced cache-based memory-latency tolerance recently focuses on data prefetching [15] [33].
2.4.2 Data Prefetching

A well known and widely studied technique for increasing tolerance of high memory latencies is data prefetching. Data prefetching techniques have taken many forms, but the fundamental idea behind the techniques is generally the same: bring data into the cache before it is explicitly requested by the processor so that it is available in the cache in case such a request occurs.

Prefetching in its early stages could be found in hardware as simply exploitation of spatial locality through long cache lines and automatic retrieval of adjacent data words, and in software as compile-time insertions of cache load instructions during inner loops for data structures referenced with an offset based on the loop variable \[30, 83\]. While these older techniques primarily dealt with high memory latencies as a result of primitive data structures such as arrays, data prefetching techniques have advanced, with new approaches attempting to deal with the problems discussed in this study, such as pointer chasing. The key fundamental challenge with pointer chasing in linked data structures is the inherent sequential nature of consecutive steps of indirection involved, thereby limiting the amount of parallelism available in the memory operations \[37\].

There have been many approaches to advanced data prefetching, including some of the more heavily explored techniques such as jump-pointer \[150\] and dependence-based \[151\], and also more recently developed techniques such as push prefetching \[197\], guided region prefetching \[190\], and multi-chain prefetching \[37\]. Although shown to be quite effective for various applications and data structures, each approach has its own set of weaknesses. Specific techniques might work well for certain data structures, but fail in the general case. Software-based prefetchers, while shown to be quite successful at uncovering memory parallelism, can add
non-negligible overhead into the application and substantial complexity to the compiler design \[37\]. Hardware-based techniques, while improving execution time, can potentially increase memory traffic a great deal, incurring unacceptable power costs \[190\].

2.4.3 Caching on Multicore

Multicore architectures have changed the problem of cache design in important ways. As multiple cores at different locations on the chip compete for resources, it is important that the designer considers that poor cache allocation can be very detrimental to performance. Numerous projects have considered multicore cache design to improve performance and power. Hackenburg et al. present a comprehensive comparative evaluation of the memory subsystems of AMD’s Opteron with Intel’s Xeon quad-core processors in terms of latency and bandwidth \[67\]. The differences that the study highlights are the inclusiveness of the cache hierarchies and the cache coherence protocols. They test the processors’ memory performance using the BenchIT toolset and custom benchmarks and conclude that Xeon’s inclusive cache generally performs better than the Opteron’s non-inclusive cache. In some cases, the Opteron’s MOESI protocol shows better performance, while in others the MESIF protocol of the Xeon is superior.

The increase in transistors on the chip has led to both greater on-chip cache sizes and higher core counts. Previous studies have considered intelligent use of caches on multicore systems by exploiting computational regularities. Biswas et al. consider using many-core processors to execute multiple instantiations of the same application and merging data on cache lines to decrease off-chip accesses and improve execution time \[17\]. Past work has considered whether to use shared or
private caches and explored the strengths and weaknesses of each. More recently, the designers of the *Reactive Non-Uniform Cache Architecture* (R-NUCA) have leveraged benefits from both private and shared upper-level caches for multicore processors [70]. A common problem with shared, large caches is that accesses are non-uniform, meaning that the latencies for different cache blocks from a core may be different. One method for mitigating this problem is to allow the placement of cache blocks anywhere in the cache – that is, the address no longer determines the physical location of the block. Two challenges with this approach are defining policies for determining the where to place cache blocks and determining the location of cache block after placement. R-NUCA addresses the placement problem by grouping block accesses into three categories during the TLB miss of the block’s page. The results show that R-NUCA can offer an average of 17% speedup for scientific and multiprogrammed workloads.

In multicore processors, it is common for the cores to share upper level caches and a memory controller, and Takagi et al. address the problems arising from increased power consumption due to resource conflicts among these components [166]. The authors derive an optimal clock frequency for the cores to minimize power consumption using analytical modeling and then target this frequency (using dynamic voltage and frequency scaling (DVFS) techniques) by ordering the memory accesses on the memory bus. The work also presents a cache partitioning scheme that, through profiling, breaks the program into phases that exhibit different L2 cache miss characteristics. The results illustrate that the design reduces power consumption by up to 46.8% for a dual-core processor and up to 24% for a quad-core.
2.5 Memory Consistency Models

In a shared-memory multiprocessing system, the memory consistency model is the set of rules that specifies how the contents of memory will appear to the programmer. In other words, when a processor issues a read from a location in memory, the programmer should be able to determine the value (or possible values) that might return based on the criteria set forth in the consistency model. The rules of the consistency model have very important implications on software and hardware design, especially on what methods the compiler can use for code optimization [4].

One important aspect of memory consistency models is the order of execution of the memory operations that the programmer can expect. The most intuitive approach is to specify that programmer can expect the operations to complete in program order, which is the ordering in which the instructions would execute if executed sequentially one a time as specified by the source code [71]. This means that for a memory operation A that appears before another memory operation B in the code, the programmer is guaranteed that the results of any execution will be such that A finishes before B.

2.5.1 Sequential Consistency

In a uniprocessor system, the memory consistency model is simple: a read from memory location X should return the value of the last write to location X, where the memory operations are given in program order. The most natural extension of this idea to multiprocessors is called sequential consistency [104]. To be sequentially consistent, a system must satisfy two criteria. The first is that the system must ensure that the results of a program’s execution are the same as if
the operations executed by all processors were executed in some sequential order. The second, an extension of the first, is that the operations from a program for a given processor must appear in the total ordering in the same order as they do in that program.

In the context of a shared memory multiprocessor, the first criterion can be met by guaranteeing that all memory operations be atomic. This means that reads and writes happen instantaneously from the processors’ perspectives and hence cannot be interrupted. In a system without caches, it is assumed that the bus provides a mechanism for arbitrating memory operations and ensuring atomicity. Hence, one method for achieving sequential consistency in this case is for each processor to execute its operations in program order.

It is more difficult to achieve sequential consistency in a system with caches. The main reason for this is that all processors must receive an indication of a write to a cached copy of a piece of data. The coherence protocol defines how this takes place. Atomicity is more difficult here, since there isn’t a central point of arbitration as there is in a non-cached system. To solve this problem, the memory system and the processors can exchange messages indicating reads or writes to memory locations. Rules specify how the memory system handles arbitration between multiple requests to the same location to ensure consistency.

The relationship between cache coherence and sequential consistency is important. The cache coherence protocol, unlike the consistency model, is not concerned with the ordering of operations in the program. When a store writes to a location, the coherence protocol attempts to ensure that all caches have the same value for that location. On the other hand, to verify the sequential consistency of the results, we must know program order. The program order requirement in the
definition of sequential consistency is an issue whether or not caches are present.

2.5.2 Relaxed Consistency

The previous section outlined two rules for achieving sequential consistency on shared memory multiprocessors: maintain program order for individual processors and make memory operations atomic. However this approach places severe restrictions on the hardware and software, preventing the usage of many optimization techniques that improve speedups on multiprocessors. As a simple example, if a core wants to hoist a load instruction above another store (not necessarily to the same address) so that it can do useful work while waiting on a cache miss, the previous approach does not allow this, since this violates the program ordering constraint.

An alternative is to loosen the rules of sequential consistency, producing memory consistency models that allow such optimizations, called relaxed consistency models [204]. A simple way to categorize relaxed consistency models is by whether they relax restrictions involving program ordering or atomicity of memory operations. (This is an informal categorization, in part because, strictly speaking, there is no atomicity requirement in the definition of sequential consistency, but this categorization provides an easy way to determine what optimizations a given model allows.)

For example, one way to improve performance on multiprocessors is to buffer stores, loads (prefetching), and invalidation messages at each processor, so that it can do work while these operations are pending. To define a new consistency model which allows for buffering, new terminology is introduced. Memory operations in a multiprocessor system are strongly ordered if they satisfy the following two
conditions. First, all memory accesses on a given processor must occur in program order. Secondly, once a processor B sees a store by processor A, all other accesses by processor A before that store must also have completed from B’s perspective.

The *weak consistency model* further relaxes the rules of consistency [46]. A system follows *weak consistency* if

- Accesses to synchronization variables are strongly ordered.
- No access to a synchronizing variable is issued in a processor before all previous data accesses have completed.
- No access to data is issued by a processor before a previous access to a synchronizing variable has completed.

This definition requires distinctions between values that are *data* and *synchronizing variables*, and also between the *issuing* and *performing* of a memory access. Dubois et al. offer a set of such definitions [46].

A system that utilizes only a relaxed consistency model cannot ensure that it will produce sequentially consistent results. Applications can exhibit unpredictable behavior when run on such a system. For this reason, most systems that implement relaxed consistency models will provide special mechanisms that the programmer can use to ensure sequential consistency when desired for specific blocks of code.

As a simple example, consider figure 2.3. The programmer is attempting to send the value 10 from P1 to P2 through the variable Y. However, if the operations on P2 are reordered and the read of Y occurs before the read of C, the value of variable X may be 1 rather than the desired value of 10. Hence, the use of only a relaxed consistency model for such code can produce unintended results [4].
2.5.3 Cache Coherence

Hennessy and Patterson provide a broad overview of both directory-based and snooping protocols for both SMPs and distributed shared memory systems [72]. The basic write-invalidate protocol for both snooping and directory-based coherence schemes are covered in detail.

With snooping protocols, the caches maintain that state of a memory block with each copy of the block. Extra bits on each memory block indicate the block’s state: e.g. is being shared by other processors, is invalid, etc. All processors must keep these status bits of cached data updated, usually by tracking the transactions from all processors to memory. Hence, it is assumed that the nodes will “snoop” all memory requests for this coherence mechanism to work properly. Snooping protocols are popular among multiprocessors with a bus-based memory interface, since each node is already connected to the bus and can easily watch the bus transactions.

In the directory-based approach, a record located in a directory maintains which processors’ caches have copies of each cache block. When a processor makes a memory transaction through its memory controller, the controller first goes to

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y = 10</td>
<td>Y = 1</td>
</tr>
<tr>
<td>C = 1</td>
<td>C = 0</td>
</tr>
<tr>
<td></td>
<td>while (C = 0) {}</td>
</tr>
<tr>
<td></td>
<td>X = Y</td>
</tr>
</tbody>
</table>
the directory to check the status of the block. The memory controller will take
the necessary steps based on the block status to maintain data coherence. Note
that it is frequently the case that the directory itself is distributed. In distributed
shared memory systems or CC-NUMA (*Cache Coherent Non-Uniform Memory
Access*) architectures, each node owns a portion of the memory to which all nodes
have access, and each is responsible for maintaining a directory for the memory it
owns.

2.6 Software-level Threads

The increase in the density of resources on processing chips has ushered in a
new wave of programming models for the development of applications for these
chips. Software-level computational units exposed to the compiler and program-
mer for parallel execution are evolving to allow better exploitation of available
microarchitectural resources. The conventional options for software-level comput-
tational units available to the compiler and programmer are the thread and the
instruction.

A good example of the classic notion of a thread is the pthread [19]. The
pthread API gives the programmer control over thread creation and synchroniza-
tion through extensions to a conventional programming language. Now, with the
rising popularity of accelerating general purpose applications with graphical pro-
cessing units (GPGPU), there has been an increase in the requirements of the
programmer’s knowledge of the hardware. As pthreads hide the details of the
underlying machine from the programmer, new programming languages such as
CUDA have exposed details of the functional units and memory structures to the
programmer. CUDA threads are small, data-parallel computational units which
the programmer forms explicitly for exploiting the GPU’s SIMD functional units and memory bandwidth \[131\].

For embarrassingly parallel applications that map well to the streaming model of computation, GPUs with CUDA are a good solution. However, because of the relatively high level of difficulty of programming the GPU and the restricted set of applications to which the streaming model of execution applies, we would like to be able to embed parallelization mechanisms that apply to a wider range of applications into the compiler or microarchitecture – hidden away from the programmer – whenever possible. Traditionally, the compiler has a more detailed model of the underlying microarchitecture than the programmer, and can leverage these details to provide performance improvements that would be too time consuming for the programmer to deal with. Recent research has already proposed new programming models that allow varying degrees of programmer intervention depending on the desired level of performance and properties of the application \[54\].

2.7 Designing for Low Power and Energy Consumption

Literature dating back as early as 1984 has focused on the design of a logic circuit with low power dissipation as a primary design objective \[181\]. By the early 1990’s, the widespread use of portable electronic devices led to a greater focus on energy consumption. Energy consumption has become a primary design constraint in digital electronics. Primary objectives are increased battery life and reliability \[205\], with accompanying benefits of decreased size and weight of the final product \[119\]. These issues, along with rising concerns due to heat dissipation, lead to the establishment in 1993 of a major IEEE/ACM conference, ISLPED, whose focus is
entirely on the design of low power electronics. By 2001, computer architects have declared power dissipation a “first-class architectural design constraint” for not only portable computing devices but also high performance processors [124]. More recently, energy consumption has emerged as a major concern in supercomputing, as reports have indicated that in 2008, the annual energy costs associated with high performance computing surpassed that of the costs of the compute nodes [50]. Since 2006, the Green500 has ranked supercomputers in terms of both energy efficiency and execution time, and the metric of performance per watt has steadily gained prominence [1].

Previous work in energy efficient computing related to this study falls into the areas of compilers [68, 178, 187], microarchitecture [28, 29, 68, 187], memory hierarchy design [56, 68, 116, 138, 203], multithreading [29, 62, 63, 133, 145, 146, 175, 189], and network-on-chip (NoC) [12, 34, 117, 141, 173, 184, 188, 196].

Power dissipation in CMOS-based processors consists of two components. The first, dynamic power, is dissipated during the switching of transistors, and has a component that is proportional to the capacitive load on a gate’s output and a component proportional to the “short-circuit” current which flows momentarily while the gate is switching. The second, static power, is dissipated when the gate is not switching, and is proportional to the leakage current that flows through the device [124]. While the dynamic power has traditionally been the component of primary concern in processor design [124], static power has become an increasingly significant concern over the last decade [27, 36, 88], especially in the memory hierarchy [56, 116, 138, 203] and NoCs [34, 173, 184].

Many recent projects have focused specifically on circuit techniques for reduc-

---

1Studies using migrant computations to address power issues have generally focused on addressing heat issues rather than energy consumption [73, 121, 144].
ing leakage energy in the cache hierarchy. Two widespread approaches switching the power modes of portions of the cache when not in use [99], by turning them off (with techniques such as gated-\(V_{dd}\)) or switching them to a low-power state called *drowsy mode*. Flautner et al. introduce the idea of gated-\(V_{dd}\) by inserting a transistor between ground and the SRAM cell to allow turning off the cell [138]. Flautner et al. leverage dynamic voltage scaling (DVS) to place inactive SRAM cells into a *drowsy state* that reduces leakage energy significantly [56]. The drowsy state has the benefit that the contents of the cell are not lost as they are with gating, but there is a performance penalty of switching the cell in and out of drowsy mode. There have been various proposals recently for lowering leakage energy of caches – such as the snug-set associative cache [77] and the slumberous cache [122] – that build upon the idea of drowsy caches. Meng et al. give a thorough overview of techniques for reducing leakage energy in caches, along with a comparison of such techniques [120].
CHAPTER 3

PRELIMINARY WORK

A key paradigm shift in this work is the consideration of computational partitioning and locality analysis in tandem with architecture design, execution model design, and thread generation. To bridge the gap resulting from this shift, we have completed preliminary studies in architectures with incremental design adaptations. These studies have established toolsets and early results for use in later, more advanced studies.

The preliminary studies consist of three stages. The first stage considers one type of computational partitioning which is a form of instruction aggregation to generate heavyweight instruction packages called floating point instruction aggregates (FPIAs). A conventional core executes these operations using microarchitectural extensions. The chapter includes an analysis of the data locality characteristics of these operations.

The second stage considers an unconventional system design which augments a conventional processing node by embedding simple cores, called Local Cache Processors (LCPs) throughout the memory hierarchy. The chapter presents the design of tools for forming small computations, which we term in-cache computations (ICCs), for execution on the LCPs. The final stage presents several enhancements to the ICC design, including an improved execution model, coprocessor architecture, and ICC code generation techniques.
The distinctions between the ideas in this chapter and those in later chapters relating to PAM and BLITS stem from a difference in design philosophy. While FPIAs and ICCs have close resemblances to the threads in chapter 4, the purpose of FPIAs and ICCs is to develop an understanding of code partitioning for offloading work from the host processor. Traveling threads then build on these concepts and delve into practical details of implementation. We place restrictions on the execution of FPIAs and ICCs – for example, the design supports the execution of only one such computation at a time. We then lift these restrictions with traveling threads. The distinction between LCPs and P-cores is similar. For example, with LCPs we avoid complications due to control speculation through the use of ICC-compatible Cache Lines. P-cores make no such assumption, and also add the ability to migrate computations from one location to another. We tailor the LCP design for executing isolated computations in the caches. On the other hand, with P-cores, we are considering the benefits and difficulties of scalable parallelism in the cache hierarchy.

3.1 Floating Point Instruction Aggregates

This section presents the design, evaluation, and characterization of floating point instruction aggregates.

3.1.1 Architecture and Execution Model Design

Instruction aggregation is a technique that seeks to improve performance by decreasing the single-instruction overheads within chains of dependent operations by grouping them into one package. A FPIA (Floating Point Instruction Aggregate) is an instruction aggregate that consists of a group of floating point instructions.
in a basic block [103]. The benefits of FPIAs include exposure of parallelism and a decrease in pressure on microarchitectural resources.

A special organization of floating point units (FPUs) supports the efficient execution of these multi-operation instruction packages. The organization considered here is a 2-dimensional array of FPUs with the interconnect necessary to route operands among them and distribute required register values to and from them. However, the design details of such an organization will require some exploration. For example, the number of FPUs in the array, the dimensions of the array, and the interconnect and bandwidth requirements are a few of the options.
Figure 3.1 shows parameterized FPIA microarchitectural extensions to a conventional processor. The conventional processing components shown are the reservation stations, the integer register file, and the pipeline. The design adds components for supporting FPIA execution. The graph table is a dedicated memory for storing the instruction packets that encode the floating point operations as FPIAs. Crossbars interconnect rows of floating point functional units that execute the operations in the FPIA. This array takes floating point operands and writes results to and from the FPIA registers. The figure shows the design parameters in shaded ovals. The dotted lines indicate that a component is optional in the design.

One important point is to determine mechanisms for moving values between the FPIA register file and the conventional register file. Microarchitectural mechanisms that move values transparently to the software is one option. An alternative is to provide special instructions for allowing the compiler to move the values. Another design point is caching of FPIA instructions. The design in the figure shows the graph table for this purpose. We evaluate the feasibility of this graph table in section 3.1.3.

3.1.2 FPIA Extraction

FPIA extraction is the process of taking a program for a conventional processor and identifying sets of floating point instructions for conversion to FPIAs. The process considers instructions grouped by basic block to avoid complications arising due to branch prediction. A basic block is a sequence of instructions at run-time between two conditional branches as executed in program order. We specify that this sequence is at run-time because an instruction in the middle of a sequence of static instructions between two branches may be the target of a branch, causing
basic blocks to split these static groups. Note that as a conservative assumption, we consider a conditional branch to end a basic block whether or not it is taken, since in practice the processor will predict the branch outcome.

The analysis tool constructs dependence graphs from the basic blocks. A dependence graph is a representation of the instructions that includes nodes and edges, where nodes represent instructions, and an edge between node $i$ and $j$ (denoted $e_{ij}$) represents that $i$ writes to a register that $j$ reads. Here, we say that node $i$ produces a register value that node $j$ consumes. For a dependence graph $G$, a subgraph is a subset of nodes and edges of $G$. If edge $e_{ij}$ is in the subgraph, both $i$ and $j$ must also be in the subgraph.

After generating the dependence graph for a basic block, the process extracts each maximal subgraph of FP operations – where such a subgraph consists only of FP operations. In this work, a trace translator implements the FPIA extraction process.

**FPIA insertion** is the process of converting FP subgraphs from the FPIA extraction process into FPIA packets, and inserting instructions into the code for initiating FPIA execution. The translator replaces an FP subgraph with a single node. This node contains all information necessary to produce the register values resulting from its operations.

The next step is to find an ordering of instructions for the transformed code, as shown in figure 3.2. Consider a FP subgraph $G$ that is being transformed into an FPIA which figure 3.2 represents as node $agg1$ in the dependence graph. The process will produce a new dependence graph, $B$, for the instructions in the basic block, and $B$ will include $agg1$. To produce this graph, it first labels the nodes with integers indicating the order in which the instructions will execute. When
In the figure above, agg1, and the instructions from which it is formed, are shown as rectangles. agg2 is an example of an FP subgraph with the instruction reordering problem. agg2, and the instructions from which it would be formed, are shown as pentagons.

the instructions execute in this order, it is necessary that the resulting register state is the same as the case in which the instructions have executed in original program order.

A problem occurs when FP instruction $g_1 \in G$ produces a register value for instruction $i_1 \notin G$, and instruction $i_2 \notin G$ produces a register value for FP instruction $g_2 \in G$, where $i_1$ appears before $i_2$ in the original program order. Hence, an ordering for the new set of instructions would require reordering of the instructions not in the FP subgraphs.

Figure 3.2 shows an example of a FP subgraph (labeled agg2) with this problem. Node 16 produces a value for node 17, while node 15 consumes a value from node 14. Since nodes 14 and 17 are both in the same FP subgraph, the insertion
of $agg2$ would require reordering of the instructions so that node 16 will produce the value before the execution of $agg2$, and $agg2$ will produce a value for node 15. This reordering leads to complications such as register renaming that are outside the scope of this work, so the translator does not transform these subgraphs.

3.1.2.1 Properties of FP Subgraphs

To characterize the graphs, we developed a tool which tracks the statistics of various properties of FP subgraphs. The first property is graph size, which is the number of nodes in a graph. Next are the width and height (or depth) of the graph. We determine these by scheduling the nodes in the subgraph using the As Soon as Possible (ASAP) scheduling. An ASAP scheduler steps through a sequence of instructions iteratively. In the $i^{th}$ iteration, the scheduler inserts all vertices with no incoming edges into the schedule and removes them from the graph. The iteration in which the scheduler removes a vertex is called the vertex’s stage. Note that the average width of an ASAP-scheduled graph is approximately the maximum IPC of that section of code. ASAP scheduling is discussed further in [12].

The width of the graph is the greatest number of nodes in a given stage of the ASAP-scheduled graph, and the height is the number of stages. Register inputs and outputs are the number of unique registers indices$^1$ read and written by all instructions in the graph. The number of wide memory inputs is an estimate of the number of unique cache lines that are touched with loads, and the number of wide memory outputs is an estimate of the number of unique cache lines touched.

---

$^1$When counting the register inputs (outputs), a unique register index is one that has not yet been counted as an input (output). That is, for a given subgraph, we count each register index at most once as an input and at most once as an output.
with stores.

The number of unique FP subgraphs in a typical application has implications on the design, such as whether or not storing the FPIAs in a table is feasible, and the size of the table. To determine the number of unique FP subgraphs, we use the following steps during the trace analysis. As the analysis runs, we build a list of graphs. Upon generation of a new graph $G$, the toolset checks for the isomorphism of $G$ with all graphs in the list. If there is no graph in the list with which $G$ is isomorphic, then $G$ is unique. If there is, the code is common and may be “reused.” We use two definitions of graph isomorphism. In the first, two graphs are isomorphic if they have the same structure (the results in section 3.1.3.2 refer to this case with the label No Op Comparison. See figure 3.3(a)). In the second, the two graphs must not only have the same structure, but there must also exist a vertex map matching operations between the graphs (the results in section 3.1.3.2 refer to this case with the label w/ Op comparison. See figure 3.3(a)).

The characteristics of typical FPIA content in applications will provide estimates of values for the design parameters in figure 3.1. Table 3.1 shows how these properties correspond to the design parameters in figure 3.1. For example, consider the properties of height and width of the graph. Table 3.1 shows that these properties provide estimates of good values for the design parameters of the height and width of the FPU array.

3.1.3 Graph Characterization

This section gives the results of application analyses to characterize the potential FPIA content in applications. The following section presents the methodology, and the final section gives the results along with discussion.


3.1.3.1 Methodology

To characterize the graph content in applications, we constructed a toolset based on the Structural Simulation Toolkit (SST) [149], which partitions simulation infrastructure between frontend and backend components. We modified the frontend for performing FPIA extraction and insertion. The tool gathers statistics on FPIA characteristics.

We analyzed execution traces of runs of a suite of eight of Sandia’s floating point-intensive scientific applications. The applications cover a wide range of purposes, including molecular dynamics (LAMMPS), shock mechanics of solids (CTH), radiation transport (ITS), and circuit simulation (Xyce). We use eighteen traces of 100 million instructions generated from the eight applications, run with varied inputs. Previous work by Murphy et al. includes more details of these applications [126].

To determine the effectiveness of a design parameter value, we use the metric of graph coverage or graph utilization. Given a minimum threshold value for a graph property, graph coverage is the percentage of FP subgraphs in a trace whose property value falls below this threshold. Graph utilization provides a model from
which we can estimate values for design parameters in figure 3.1.

### 3.1.3.2 Results

Table 3.2 summarizes the results from these analyses. This summary shows that if we design the hardware to execute graphs with a property value, this will allow the execution of some minimum percentage of the graphs for sixteen out of eighteen applications, given no other hardware limitations. For instance, if we design for graphs of size 16, this will allow the execution of about 95% of the graphs in cth.4B.2gas.

From the results of the analyses, we see that the graphs usually have no more than 16 operations, and are narrow and deep. An FPU array that has dimensions 16x1 (FPUs arranged 16 vertically by 1 horizontally) will allow execution of over 75% of the graphs for most applications. Increasing the dimensions to 32x2 (FPUs arranged 32 vertically by 2 horizontally) will allow execution of almost all the graphs, or alternatively a loopback path could be added to reduce the number of FPUs and conserve area. From this, we assume that the typical FPIA we will execute has 32 operations. Using the number of register inputs and outputs from the table, we can estimate the size of one of these instructions, assuming all operations

---

**Table 3.2**

<table>
<thead>
<tr>
<th>Min Graph Coverage (16/18)</th>
<th># of ops</th>
<th>Width</th>
<th>Depth</th>
<th>Register Inputs</th>
<th>Register Outputs</th>
<th>Memory Inputs</th>
<th>Memory Outputs</th>
<th>Wide Loads</th>
<th>Wide Stores</th>
</tr>
</thead>
<tbody>
<tr>
<td>25%</td>
<td>16</td>
<td>1</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>50%</td>
<td>16</td>
<td>1</td>
<td>16</td>
<td>32</td>
<td>16</td>
<td>8</td>
<td>2</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>75%</td>
<td>16</td>
<td>1</td>
<td>16</td>
<td>32</td>
<td>16</td>
<td>8</td>
<td>4</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>90%</td>
<td>32</td>
<td>2</td>
<td>32</td>
<td>64</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>16</td>
<td>2</td>
</tr>
</tbody>
</table>
Figure 3.3. Number of Unique Graphs and Graph Utilization Tests

The figures show graph utilization for various properties. Graph utilization tells us the percentage of FPIAs in the trace that have a property less than or equal to a given value. For example, figure 3.3(b) shows that just over half of the FPIAs in cube3.4B.vbr have at most 16 operations.

Information is encoded into a single package. Assuming no more than eight types of FP operations, the encoding of operations requires 3 bits. Routing the results among the FPU array also requires bits in the instruction. For this approximation, we will assume that each of the 32 operations needs 4 bits to route its results. If we assume 64 entries in the FPIA register file, and design for 32 register inputs, the instruction will need 192 bits for input register values, and then 96 bits for the 16 output register values. As shown in figure 3.3(a), the number of unique
graphs is generally no more than 150 with op comparison. Conservatively using the maximum value and 304 from `alegra.4B.hemipen` in figure 3.3(a), we can calculate an estimate of the total size of a table for storing all FPIAs:

\[
[ 32 \times 3 \text{ (ops)} + 32 \times 4 \text{ (routing)} + 192 \text{ (inputs)} + 96 \text{ (outputs)} ] \text{ bits/inst.} = 512 \text{ bits/inst.}
\]

\[
512 \text{ bits/inst.} \times 304 \text{ insts.} = 19456 \text{ bytes. (with op comparison in isomorphism)}
\]

A memory dedicated for storing 19-KB of FPIA instructions certainly seems feasible, given that many modern processors have many megabytes of on-chip cache. Note that this assumes no immediate values (the current design assumes that all values come from registers).
One other simplifying assumption for these calculations is that in order for two graphs to be isomorphic, they do not necessarily need the same register indices as inputs and outputs. This factor may affect the table size approximation.

3.2 In-Cache Computations

In this section, we present the next stage of design in the preliminary studies, which introduces a new computational unit called the In-Cache Computation (ICC). The section opens with a definition of ICCs, a corresponding architecture, and an execution model. This is followed by a description of algorithms for generating two types of ICCs.

3.2.1 ICC Design

For the purposes of ICC design, the processor organization we consider consists of two types of processing elements: a host or heavy-weight processor (HWP); and a set of distributed co-processors (DCPs) embedded throughout the memory hierarchy. The HWP is a processing element of high complexity resembling the core in a conventional uniprocessor that supports out-of-order execution and branch prediction, while the DCPs are of a much simpler design. DCPs may reside on the same chip as the HWP next to the cache banks, and we refer to these DCPs as Local Cache Processors (LCPs).

An In-Cache Computation (ICC) that executes on an LCP is a computational unit that uses data in memory. A few instructions from the HWP’s code are the ICC’s operations. There are two primary objectives to ICC execution. The first is to increase parallelism – both system-wide and at the HWP – by offloading computations from the HWP to the DCPs. The second is to improve memory per-
formance by reducing traffic in the memory hierarchy with better data placement. Reducing traffic also has the attribute of potentially reducing energy associated with data movement.

3.2.1.1 Baseline Microarchitecture

To achieve the parallelism benefits from ICC execution, the HWP will have the ability to launch an ICC efficiently. To achieve this, we assume the HWP has microarchitectural extensions for packaging, issuing, and committing ICCs. Much of this ICC-oriented activity taking place at the HWP will likely occur along with the normal HWP actions.

Figure 3.5 shows an example of such microarchitectural extensions. ICC formation may take place in the compiler or at run-time. In the figure, the ICC Filter partitions instructions for execution at the HWP and DCPs. The ICC Constructor then packages the ICCs before passing them to the memory interface. The register update unit (RUU) has extensions for handling the return of values from the ICCs.

3.2.1.2 ICC Properties and Execution

DCPs execute small units of computation in the form of ICCs. The compiler, microarchitecture, or a hybrid of the two perform ICC generation. The ICC generator takes groups of instructions, called the ICC’s Source Instruction Group (SIG) from the HWP’s task for offloading to the DCPs. In this work, the SIG has three requirements that apply to the formation of ICCs:

- All register inputs from instructions outside the SIG are ready at the time of issue.
Figure 3.5. Microarchitectural extensions to a conventional pipeline for constructing and issuing ICCs.

The top figure shows a pipeline for a conventional processor. The bottom figure shows the extensions (highlighted in red) to this pipeline for supporting ICCs. Although the critical path will likely increase some (due to certain components such as the filter and the additional RUU logic for receiving values from the DCPs), it should be possible to perform much of the work in parallel with the conventional components.
• Their use does not require reordering of instructions in the code\(^2\).

• The SIG must include at least one memory access designated as the target.

To execute an ICC, the HWP first issues a special instruction called the *ICC creation instruction* or \texttt{iccci} to a functional unit (called the *ICC constructor*) for building the ICC. The ICC constructor bundles the SIG and any required register values into a package containing the memory addresses and data the ICC will use and then launches the ICC. The ICC then travels up the memory hierarchy, similar to a load or a store, for execution on a DCP. At each cache level, the ICC interrogates the cache for the target address. On a miss, the ICC continues up to the next level. If the ICC misses every cache level, the top-level cache controller forwards the ICC to the memory controller which passes the ICC to the DCP next to the macro holding the target address. The destination DCP unpackages the ICC and executes the computations.

Two types of ICCs that we introduce here are the In-Memory Operation and the Graphlet. An *In-Memory Operation* (IMO) is an ICC whose SIG contains exactly one store and no return values. The *Target Memory Address* or TMA is the address to which the IMO’s store writes a value. The SIG may include a load only if it reads from the TMA. Hence, an IMO requires no communication due to data dependencies, and produces a single value for storage.

The execution of an IMO, pictured in figure\(^3.6\) is a multi-step process that the HWP initiates with the \texttt{iccci} instruction. When the HWP reaches an \texttt{iccci} in the code, it issues the instruction to the ICC constructor which packages the IMO and ships it out to the memory hierarchy. The IMO travels up the hierarchy, and

\(^2\)The requirement is analogous to the ordering constraint on FPIAs. Refer to figure\(^3.2\) in section \[3.1.2\]
when the TMA hits in the cache, the DCP that is nearest the physical memory holding this address unpackages the IMO for execution.

A graphlet is an ICC whose SIG is a small weakly connected component with at least one memory access in the data dependence graph of a basic block. Since the SIG is the entire WCC, it has no input or output register dependencies from or to instructions in the basic block outside the SIG. The execution of graphlets is more complex than that of IMOs. One reason for this is that the SIG of a graphlet may contain more than one memory access. These accesses may have multiple unique addresses, implying that the graphlet may require data residing at more than one physical location in the memory hierarchy. Our design assumes

---

3A directed graph $G$ is weakly connected if for each pair of vertices $u, v \in G$, $\exists$ a path from $u$ to $v$ when directions of edges are ignored. A weakly connected component (WCC) $S$ is a maximal weakly connected subgraph of $G$ (i.e. $\exists$ a vertex $x \in G$ when $x$ is added to $S$, $S$ is still weakly connected).
that the graphlet executes on a DCP at the lowest level of the memory hierarchy holding any of the graphlet’s operands. Any additional memory values required are retrieved via a fetch message to remote DCPs.

3.2.2 ICC Extraction

The following sections present algorithms for extracting the two types of ICCs from conventional codes.

3.2.2.1 In-Memory Operations

The goal of IMO SIG extraction is to maximize the number of instructions in the IMO without violating the IMO SIG criteria. The translator groups instructions by basic block for the extraction process. It constructs a dependence graph, $G$, from the instructions in this group. Algorithm 1 shows the steps for forming the SIG for an IMO.

A compiler or translator contains an implementation of this algorithm which forms the IMOs. As part of this work, we have implemented this algorithm in a trace translator. The translator first forms the dependence graph of instructions in a basic block. It then cuts away nodes using algorithm 1 and compresses the remaining nodes into a single node that represents the iccci node in the HWP’s executable. This new node contains three types of information: the TMA, the instructions taken from the SIG, and the set of register indices (marked $RVS$ in figure 3.6) of external register inputs for the SIG.
Algorithm 1: IMO SIG Extraction

Input: $G, s$

/* $G$: dependence graph of a basic block. Each instruction is represented as a vertex $u$ in $G$. $v$ is a child of $u$ if and only if $v$ has an input dependence from $u$. $v$ is an ancestor of $u$ if and only if $\exists$ a path from $v$ to $u$. $s$: A store in $G$. */

1. Remove loads to addresses other than the TMA.
2. Remove conditional control flow instructions.
3. Create a topological sort, $T$, of the remaining vertices. /* If $j > i$, then $T(j)$ is not an ancestor of $T(i)$. */
4. Mark the node containing $s$.
5. $i = |T|$.
6. While $i > 0$ do
   7. if $T(i)$ has at least one child and all of $T(i)$’s children are marked then
      8. Mark $T(i)$.
   9. end
10. $i = i - 1$.
11. end
12. Remove all unmarked vertices, call the remaining subgraph $H$.
13. Replace all nodes in $G$ corresponding to the nodes in $H$ with a single 1ccci node.

Algorithm 2: Graphlet SIG Extraction

Input: $G, SR$

/* $G$: dependence graph of a basic block. $SR$: an integer used in the construction of graphlets. */

1. Split the graph into a set, $S$, of WCCs.
2. Let $B$ be the set of graphs with maximal number of nodes in $S$.
3. Let $M = |V(b)|$, for some $b \in B$.
4. For each $s \in (S - B)$ \ such \ that \ $s$ \ contains \ at \ least \ one \ load \ or \ store \ do
   5. if $|V(s)| \leq (M/SR)$ then
      6. Replace nodes of $s$ in $G$ with a single 1ccci node.
   7. end
8. end

Figure 3.7. Example of a Basic Block Taken from CTH.

In this example, of the five WCCs containing memory accesses, the four smallest are the graphlets.
3.2.2.2 Graphlets

Figure 3.7 shows an example of a basic block containing graphlets, taken from Sandia’s CTH application. Algorithm 2 gives the steps for graphlet SIG extraction. The algorithm transforms each WCC in the basic block containing a load or a store that has size $\leq M/SR$ into a graphlet. $M$ is the size of a WCC of maximal size in the basic block, and $SR$ is the size ratio, a variable parameter.

3.3 ICC Enhancements

In this section, we present several improvements to the ICC design. These improvements include an advanced execution model that allows for increased parallelism between the host and the DCPs and enhanced extraction mechanisms which leverage the new concept of ICC-compatible data cache lines.

3.3.1 Design

Here, we give the definition of the improved ICC execution model and DCP architecture along with the design of the ICC code translation tools.

3.3.1.1 Execution Model

The improved ICC execution model consists of three phases: the prefetch phase, the setup phase, and the execution phase. In the prefetch phase, the host issues a special directive called the Prefetch Directive out to the memory hierarchy for increasing the probability that the unified L2 cache contains both instruction and data lines in locations that will enable efficient ICC execution. This directive seeks to fulfill two conditions. The first is that copies of the instruction cache lines containing the code for an ICC are in either the cache or the instruction cache
line overflow buffer (IOB) next to the DCP on which the ICC will execute. The second is that the L1 cache does not contain lines that ICCs will modify. The purpose of this requirement is to ease the task of maintaining consistency with the host. In the ideal case, the L1 requires no invalidation during the prefetch phase\(^4\).

During the setup phase, the host constructs the ICC *activator*, (similar to the *continuation* described in previous work \([26, 170]\)) which it will inject into the memory hierarchy for initiating the execution of one or more ICCs. The ICC activator contains the register values, data addresses, starting PC of each ICC, and *control vectors* needed to initiate and carry out ICC execution. The control vectors are bit vectors that indicate which instructions following the starting IP the DCP will execute. This assumes fixed-length instructions.

When constructing the ICC, the host accumulates the register values along with the ICC data and instruction line addresses. The main thread of execution accumulates and produces these values. The ICC filter removes instructions for insertion into the ICCs from the host’s instruction stream.

Once the activator construction is complete, the execution phase begins. In the execution phase, the host injects the activator into the memory hierarchy. The L2 cache controller executes the activator and routes the register values and addresses to the appropriate DCPs based on the data addresses of each ICC. The ICCs communicate across a bus internal to the cache via special instructions which enforce a static ordering, similar to techniques used in previous VLIW architectures \([153]\). The instructions in the ICCs may have dependencies at the

---

\(^4\)The frequent occurrence of L1 cache line invalidation during the prefetch phase is an indication of poor ICC formation. In this work, we take steps to avoid this case, and leave the implementation of an exception handler as future work.
Figure 3.8. ICC Example.
host, and the ICCs will return register values as necessary.

Consider an example of ICC execution, as figure 3.8 illustrates. Instruction lines $X$, $Y$, and $Z$ contain the ICC’s instructions. These instructions include a load and a store to line $L$, which maps to a physical block next to DCP 0 in L2, and a reference to data on line $M$ which maps to a remote physical block. Prior to the prefetch phase, L1 contains a copy of part of line $L$.

When the host reaches the prefetch instruction, it issues the prefetch directive to the cache hierarchy. The L1 controller invalidates its partial copy of line $L$, and the L2 controller fetches the copy of $L$ from L3 (we assume no multi-level inclusion here to serve the purpose of the example). The L2 controller fetches data line $M$ from L3 to remote DCP $n$. It then fetches instruction line $Z$ from DCP $n$ into DCP 0’s IOB, moves line $Y$ from L3 into IOB 0, and fetches line $X$ from main memory into IOB 0.

During the setup phase, the host accumulates the register values needed for the execution of the ICC. When the host issues a memory access to a line in L3 that conflicts with $L$, the L2 controller bypasses the access to avoid the eviction of line $L$. This will require new tags at the L2 controller for indicating the locking of a line in place for an ICC. When the ICC’s register values are ready, the host issues the ICC activator to the hierarchy.

The execution phase begins, and the activator reaches DCP 0. The DCP begins executing instructions from line $W$ in its local memory, and then from lines $X$, $Y$, and $Z$ in its IOB. Instructions from line $Z$ contain loads from line $L$ (in local memory) and line $M$ (located at DCP $n$). DCP 0 then places a request on the intra-cache bus to retrieve the remote data from line $M$. It then writes a value to line $L$ using the store from line $Z$. Upon completion, the ICC returns a set of
register values back to the host\textsuperscript{5}.

3.3.1.2 DCP Architecture

The DCP executes ICCs and has its roots in previous work in the design of cores for processing-in-memory systems \cite{26,170} – especially those using dense memory caches implemented with eDRAMs \cite{80}. Figure 3.9 shows the organization of the components at the DCP node. Each DCP in the L2 has a number of cache sets associated with it. The design will maintain the functionality of a conventional L2 cache, while at the same time providing an interface to the extended processing capabilities of the DCPs. Hence, the L2 operates in a dual-mode fashion which the L2 cache controller governs (labeled \textit{Master BIU/Controller} in figure 3.9) using commands from the host.

During the prefetch phase, the controller ensures that the data lines are present in L2. The controller then asserts that the instruction lines containing the ICC code are also present at the DCPs that will modify the data lines. In the case that the sets at the DCP do not include the necessary instruction lines, the controller distributes them to a special buffer (labeled \textit{IS Line Overflow Buffer} in figure 3.9) near the DCP. The slave BIU records which instruction lines are in the IOB, as this information is necessary during the execution phase.

During the execution phase, the controller distributes the control vectors that specify which instructions each DCP will execute\textsuperscript{6}. The cache controller and

\footnotesize{\textsuperscript{5}Note that there is the possibility of deadlock here in the case where an ICC requires reordering of instructions, as described in section 3.1.2 with the example in figure 3.2. If the compiler or microarchitecture does not account for this problem during instruction scheduling, the processor may deadlock. When considering candidates for ICC creation, we eliminate those whose SIGs would require reordering of instructions in the original code (see the second item in the bulleted list in section 3.2.1.2).

\textsuperscript{6}Section 4.4.1 describes the details of control vector construction.}
Figure 3.9. DCP Architecture.
compiler can uniquely identify an ICC by the addresses of the data it uses. After
fetching a line, the DCP uses the control vectors to direct data from the cache
sets and guide the flow of execution. ICC communication occurs over a special
communication bus local to the L2 cache (labeled Intra-Cache Bus in figure 3.9).
The ICC generator enforces consistency through this bus, as guided by the scheme
described in the next section.

3.3.2 Extraction

Computational partitioning towards increased parallelism and reduced data
communication is a key challenge of ICCs. Here, we analyze the potential of
data-centric ICCs through simulation. The key concept is ICC-compatible data
cache lines (ICLs). An ICL is defined as a line that the host has touched with
memory accesses in only one basic block at the time the L1 evicts that line. This
type of line has the properties that lend itself well to both parallelism and locality
suitable for ICCs. Since the instructions do not cross basic blocks, we avoid
the complications of control speculation. Implementing misspeculation recovery
would both complicate the DCP design and require increased traffic in the cache
hierarchy, resulting in overheads in power and performance. Furthermore, such
lines will tend to hurt performance on a system using conventional caching, since
it is using a physical block for a cache line with limited temporal locality.

Through trace analysis, we determine which lines are ICC-compatible. This
information is then given to a code translator which it uses for computational
partitioning. Algorithm 3 gives the translation scheme. An initial trace analysis
phase selects a set of memory accesses for each ICC using trace analysis. The
purpose of the algorithm is two-fold. The first objective is to partition the de-
Algorithm 3: ICC Computational Partitioning

Input: \( G, S, L \)

\( G \): dependence graph of a basic block.
Each instruction is represented as a vertex \( u \) in \( G \).
\( v \) is a child of \( u \) if and only if \( v \) has an input
dependence from \( u \). \( v \) is an ancestor of \( u \) if and
only if \( \exists \) a path from \( v \) to \( u \).
\( S \): A set of ICC IDs of ICCs in \( G \).
\( L \): A set of \(|S|\) sets of vertices in \( G \).
Each set \( l \in L \) corresponds to an ICC ID \( s \in S \).
\( l \) contains the vertices corresponding to memory
accesses in \( G \) assigned to the ICC with ID \( s \).

1  index=0.
2  foreach \( l \in L \) do
3    foreach \( g \in G \) do
4      if \( g \) is in \( l \) then
5        Label \( g \) with index.
6        Depth-first search on \( g \), labeling all
7        unlabeled descendants of \( g \) with index.
8      end
9    end
10   index=index+1.
11 end
12 Remove labels of branch critical ICCs.
13 curCommIndex=0.
14 foreach \( g \in G \) do
15    if \( g \) has label \( x \) then
16      foreach \( \text{Incoming edge} \ (i,g) \) do
17        if \( i \) is an instruction outside the basic block then
18          Mark \( i \) as an external ICC input.
19        end
20      end
21      foreach \( \text{Outgoing edge} \ (g,o) \) do
22        Mark \( o \) as an external ICC output.
23        if \( o \) has label \( y \) and \( y \neq x \) then
24          Schedule bus event \( \text{curCommIndex} \) from \( g \) to \( o \).
25          curCommIndex=curCommIndex+1.
26        end
27    end
28  end
29 end

Dependence graph of the basic block into sets of instructions for inclusion in each
ICC. After labeling each ICC memory access, the algorithm removes branch critical ICCs (BCIs). A BCI is defined as an ICC whose memory accesses include a load with a branch as a descendant. One objective of ICC execution is to increase system-wide parallelism, and this implies increasing host utilization. The

\footnote{Future work will consider an optimization for including certain types of BCIs. As an example, consider a BCI leading to conditional basic blocks that each return to a common PC. However, in the applications studied here, the results in the next section show that BCIs are generally very uncommon.}
partitioning scheme assigns all branches to the host, and we want to avoid causing processor stalls or speculation due to missed branch critical loads. Following BCI removal, the current scheme simply gives priority to the first instructions in program order, using a depth-first search from each ICC memory access and labeling every unlabeled instruction that is reached.

The second objective is to determine the register values for each ICC for inclusion in the ICC activator that the host launches. In the current design, the DCP returns all modified registers to the host. Load balancing and live register analysis to further eliminate traffic are left for future work.

3.3.3 Evaluation

For the performance evaluations, we use four sets of application traces. The first two sets of traces are from applications in the standard SPEC-Int and SPEC-FP benchmark suites, and the second two are from the Sandia-Int and Sandia-FP suites from scientific application sets from Sandia National Laboratories (SNL). The Sandia-Int suite is a set of traces of seven commonly used integer-intensive applications run at SNL. Sandia-FP (described in more detail in section 3.1.3.1) is a set of eighteen traces from eight of SNL’s floating point-intensive scientific applications run with varied inputs.

Table 3.3 presents the first set of analysis results using a conventional 16KB direct-mapped L1 cache with a 64B line size. The second column of table 3.3 shows the number of L1 evictions that occur and the percentage of these evictions that are ICLs. The next column, % ICC Content, shows the percentage of instructions from the original trace offloaded from the host to the DCPs. The algorithm partitions each basic block’s instructions among the ICCs and the host. The total
number of instructions inserted into ICCs throughout the trace divided by the length of the trace (100,000,000) gives the % ICC Content. The fifth column gives the percentage of ICLs that are branch critical.

The percentage of ICLs is significant, with a minimum of 5.3% (sppm.2B), a maximum of 93% (sp.4B), and a geometric mean of 43%. A high percentage of ICLs and a high number of evictions means that there is large opportunity for both an increase in L1 performance at the host and increased parallelism through the use of the DCPs. Although the amount of ICC content is low across all the traces (1.5% on average), improved partitioning algorithms will likely improve these numbers, resulting in increased parallelism through increased DCP utilization. Several avenues exist for possible improvements to the partitioning algorithm. One option is to increase the average number of instructions between branches through the use of compiler techniques (loop unrolling), or speculatively at run-time with branch prediction. Another option is to adjust the L1 cache parameters to increase the number of ICLs. The results in the fifth column are quite encouraging, revealing that branch-criticality is a rare property among ICCs. We found the average percentage of ICCs that are branch-critical across all traces to be much less than 1%.

Consider the first trace in the table, 164.gzip.1.b. There are 1,661,259 L1 evictions, 70% (1,162,881) of which are ICLs. Intuitively, the system can achieve performance improvement by leaving these lines in L2 and working on them with the DCPs. Although it is likely that the ICCs will still result in traffic back to the host, the host can probably hide this latency since virtually none of these accesses is branch critical (as given by the right-most column). Furthermore, it is likely that this traffic will be less than that required by the fetch-modify-flush
### TABLE 3.3

**ICC CONTENT ANALYSIS RESULTS AND PROJECTED \(\lambda\)-SPEEDUP BOUND**

<table>
<thead>
<tr>
<th>Application</th>
<th># of L1 Evictions</th>
<th>% ICLs</th>
<th>% ICC Content</th>
<th>% BCIs</th>
<th>Est. Projected (\lambda)-Speedup Bound</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SPEC-Int</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>164.gzip.1.b</td>
<td>1661259</td>
<td>70%</td>
<td>2%</td>
<td>0%</td>
<td>1.16</td>
</tr>
<tr>
<td>164.gzip.2.b</td>
<td>880566</td>
<td>47%</td>
<td>1%</td>
<td>0%</td>
<td>1.33</td>
</tr>
<tr>
<td>164.gzip.3.b</td>
<td>1605856</td>
<td>57%</td>
<td>1%</td>
<td>0%</td>
<td>1.15</td>
</tr>
<tr>
<td>164.gzip.4.b</td>
<td>2139275</td>
<td>74%</td>
<td>2%</td>
<td>0%</td>
<td>1.23</td>
</tr>
<tr>
<td>164.gzip.5.b</td>
<td>1465704</td>
<td>60%</td>
<td>1%</td>
<td>0%</td>
<td>1.19</td>
</tr>
<tr>
<td>175.vpr.1.b</td>
<td>2964109</td>
<td>23%</td>
<td>1%</td>
<td>0%</td>
<td>1.04</td>
</tr>
<tr>
<td>176.gcc.1.b</td>
<td>97623</td>
<td>14%</td>
<td>0%</td>
<td>6%</td>
<td>1.08</td>
</tr>
<tr>
<td>176.gcc.3.b</td>
<td>2272139</td>
<td>37%</td>
<td>2%</td>
<td>8%</td>
<td>1.03</td>
</tr>
<tr>
<td><strong>SPEC-FP</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>168.wupwise.1.b</td>
<td>506879</td>
<td>12%</td>
<td>0%</td>
<td>0%</td>
<td>1.05</td>
</tr>
<tr>
<td>171.swim.1.b</td>
<td>8139834</td>
<td>55%</td>
<td>5%</td>
<td>0%</td>
<td>3.75</td>
</tr>
<tr>
<td>172.mgrid.1.b</td>
<td>3579451</td>
<td>49%</td>
<td>3%</td>
<td>0%</td>
<td>1.01</td>
</tr>
<tr>
<td>173.appiu.1.b</td>
<td>3729154</td>
<td>68%</td>
<td>17%</td>
<td>0%</td>
<td>2.45</td>
</tr>
<tr>
<td>177.mesa.1.b</td>
<td>1175265</td>
<td>21%</td>
<td>1%</td>
<td>1%</td>
<td>1.00</td>
</tr>
<tr>
<td>178.galgel.1.b</td>
<td>10924676</td>
<td>69%</td>
<td>8%</td>
<td>0%</td>
<td>1.01</td>
</tr>
<tr>
<td>179.art.1.b</td>
<td>8515087</td>
<td>82%</td>
<td>8%</td>
<td>0%</td>
<td>2.23</td>
</tr>
<tr>
<td>179.art.2.b</td>
<td>8499246</td>
<td>82%</td>
<td>8%</td>
<td>0%</td>
<td>3.66</td>
</tr>
<tr>
<td>183.eqake.1.b</td>
<td>2368176</td>
<td>45%</td>
<td>2%</td>
<td>0%</td>
<td>1.03</td>
</tr>
<tr>
<td>187.facevec.1.b</td>
<td>4106654</td>
<td>77%</td>
<td>5%</td>
<td>0%</td>
<td>1.22</td>
</tr>
<tr>
<td>189.lucas.1.b</td>
<td>3711088</td>
<td>54%</td>
<td>6%</td>
<td>0%</td>
<td>1.08</td>
</tr>
<tr>
<td>191.fma2d.1.b</td>
<td>2239950</td>
<td>36%</td>
<td>2%</td>
<td>2%</td>
<td>1.97</td>
</tr>
<tr>
<td>200.sixtrack.1.b</td>
<td>2542373</td>
<td>44%</td>
<td>1%</td>
<td>0%</td>
<td>1.04</td>
</tr>
<tr>
<td>301.apsi.1.b</td>
<td>4131489</td>
<td>78%</td>
<td>4%</td>
<td>0%</td>
<td>1.90</td>
</tr>
<tr>
<td><strong>Sandia-Int</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>blast.4B</td>
<td>2107204</td>
<td>25%</td>
<td>1%</td>
<td>0%</td>
<td>1.02</td>
</tr>
<tr>
<td>chaco.4B</td>
<td>11807395</td>
<td>78%</td>
<td>10%</td>
<td>0%</td>
<td>1.06</td>
</tr>
<tr>
<td>dm.4B</td>
<td>3181081</td>
<td>65%</td>
<td>2%</td>
<td>16%</td>
<td>1.77</td>
</tr>
<tr>
<td>iso.4B</td>
<td>7817602</td>
<td>47%</td>
<td>4%</td>
<td>0%</td>
<td>5.58</td>
</tr>
<tr>
<td>kmetis.4B</td>
<td>2534815</td>
<td>38%</td>
<td>1%</td>
<td>1%</td>
<td>1.01</td>
</tr>
<tr>
<td>sp.4B</td>
<td>4557051</td>
<td>90%</td>
<td>6%</td>
<td>0%</td>
<td>6.82</td>
</tr>
<tr>
<td>zchaft.4B</td>
<td>2195427</td>
<td>35%</td>
<td>1%</td>
<td>1%</td>
<td>1.06</td>
</tr>
<tr>
<td><strong>Sandia-FP</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>aegra.4B.WES94</td>
<td>4563413</td>
<td>61%</td>
<td>8%</td>
<td>7%</td>
<td>1.57</td>
</tr>
<tr>
<td>aegra.4B.hemipen</td>
<td>3019218</td>
<td>54%</td>
<td>6%</td>
<td>2%</td>
<td>1.29</td>
</tr>
<tr>
<td>cth.4B.2gas</td>
<td>4316113</td>
<td>74%</td>
<td>4%</td>
<td>0%</td>
<td>1.18</td>
</tr>
<tr>
<td>cth.4B.amr-2d-cust4</td>
<td>458864</td>
<td>40%</td>
<td>0%</td>
<td>0%</td>
<td>1.04</td>
</tr>
<tr>
<td>cth.4B.efp</td>
<td>724509</td>
<td>38%</td>
<td>0%</td>
<td>0%</td>
<td>1.01</td>
</tr>
<tr>
<td>cth.4B.sc</td>
<td>1419197</td>
<td>40%</td>
<td>1%</td>
<td>0%</td>
<td>1.01</td>
</tr>
<tr>
<td>cube3.4B.crs</td>
<td>3291162</td>
<td>19%</td>
<td>1%</td>
<td>0%</td>
<td>1.00</td>
</tr>
<tr>
<td>cube3.4B.vbr</td>
<td>1884920</td>
<td>21%</td>
<td>1%</td>
<td>0%</td>
<td>1.17</td>
</tr>
<tr>
<td>its.4B.cad.brems</td>
<td>497149</td>
<td>41%</td>
<td>0%</td>
<td>12%</td>
<td>1.05</td>
</tr>
<tr>
<td>its.4B.cg.brems</td>
<td>348060</td>
<td>46%</td>
<td>0%</td>
<td>25%</td>
<td>1.04</td>
</tr>
<tr>
<td>lmp.4B.flow.langevin</td>
<td>2074179</td>
<td>67%</td>
<td>2%</td>
<td>0%</td>
<td>1.05</td>
</tr>
<tr>
<td>lmp.4B.lj.fix</td>
<td>1491885</td>
<td>56%</td>
<td>2%</td>
<td>0%</td>
<td>1.04</td>
</tr>
<tr>
<td>lmp.4B.poly.chain.a</td>
<td>1310525</td>
<td>55%</td>
<td>2%</td>
<td>0%</td>
<td>1.04</td>
</tr>
<tr>
<td>mpsalsa.4B.airport3d_3</td>
<td>3713702</td>
<td>40%</td>
<td>10%</td>
<td>0%</td>
<td>1.02</td>
</tr>
<tr>
<td>mpsalsa.4B.thermal...</td>
<td>1466939</td>
<td>12%</td>
<td>0%</td>
<td>5%</td>
<td>1.04</td>
</tr>
<tr>
<td>sppm.2B</td>
<td>1445981</td>
<td>5%</td>
<td>0%</td>
<td>0%</td>
<td>1.01</td>
</tr>
<tr>
<td>xyce.4B.invertgen75</td>
<td>4383999</td>
<td>49%</td>
<td>3%</td>
<td>2%</td>
<td>1.02</td>
</tr>
<tr>
<td>xyce.4B.invertgen_v3_75</td>
<td>4120789</td>
<td>53%</td>
<td>3%</td>
<td>2%</td>
<td>1.02</td>
</tr>
</tbody>
</table>
TABLE 3.4
BASELINE HWP CONFIGURATION

<table>
<thead>
<tr>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Update Unit Size</td>
<td>64 entries</td>
</tr>
<tr>
<td>Load-Store Queue Size</td>
<td>32 entries</td>
</tr>
<tr>
<td>Fetch Queue Size</td>
<td>4 entries</td>
</tr>
<tr>
<td>Decode Width</td>
<td>4</td>
</tr>
<tr>
<td>Issue Width</td>
<td>8</td>
</tr>
<tr>
<td>Functional Units</td>
<td></td>
</tr>
<tr>
<td>Integer ALU - 3</td>
<td></td>
</tr>
<tr>
<td>Integer Mult/Div - 1</td>
<td></td>
</tr>
<tr>
<td>FP Add - 2</td>
<td></td>
</tr>
<tr>
<td>FP Mult/Div - 1</td>
<td></td>
</tr>
<tr>
<td>Branch Predictor</td>
<td>Bimodal, 2048 entries</td>
</tr>
<tr>
<td>Branch Target Buffer</td>
<td>512 entries, 4-way</td>
</tr>
<tr>
<td>Branch Mispredict Penalty</td>
<td>3 cycles</td>
</tr>
<tr>
<td>Caches</td>
<td>DL1 - 1024 sets, 64-byte blocks, direct-mapped, 4-cycle latency</td>
</tr>
<tr>
<td></td>
<td>DL2 - 2048 sets, 64-byte blocks, 4-way, 27-cycle latency</td>
</tr>
<tr>
<td>TLBs</td>
<td>TLLB - 16 entries, 4-way</td>
</tr>
<tr>
<td></td>
<td>DTLB - 32 entries, 4-way</td>
</tr>
<tr>
<td></td>
<td>30-cycle miss penalty</td>
</tr>
<tr>
<td>Main Memory</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Latency - 150 cycles</td>
</tr>
<tr>
<td></td>
<td>Bandwidth - 64 bits/cycle</td>
</tr>
</tbody>
</table>

approach of conventional caches. One optimization for consideration in future work is to bundle the accesses to specific words across multiple lines together after the execution phase of a group of ICCs to mitigate latency costs.

Next, we modified SimpleScalar to project a bound on the speedup due to the improvement in the host’s L1 hit rate (called the $\lambda$-speedup). The $\lambda$-speedup does not include the performance improvement due to the offloading of instructions from the host to the DCPs. Table 3.4 shows the baseline SimpleScalar configuration, and table 3.3 shows the speedup numbers in the right-most column. These speedup numbers are an optimistic estimate for $\lambda$-speedup in that we free L1 blocks from ICLs by converting ICLs’ memory accesses to single-cycle non-memory access ops. The speedup estimates vary widely from negligible speedup ($cube3.4B.crs$) to 6.82 ($sp.4B$). As expected, high speedups tend to correlate with a high percentage of ICLs and a high number of evictions as given in table 3.3.
(sp.4B has the highest percentage of ICLs at 94%). However, note that this analysis covers shorter traces than those used for the experiments to obtain the results in the other columns of the table, which may cause disparity. The averages for SPEC-Int, SPEC-FP, Sandia-Int, and Sandia-FP are 1.15, 1.54, 1.87, and 1.08, respectively, and the average across all traces is 1.32.

3.4 Conclusions

This section has presented preliminary work that serves as a foundation for later studies in the design of advanced parallel architectures for addressing performance and energy challenges. From the application characterization studies with FPIAs, we found that floating point subgraphs in Sandia’s floating point applications are generally narrow and deep – 75% of the subgraphs for most applications are at least 16 operations deep and at most 2 operations wide. The studies of the memory access patterns of these subgraphs showed that 75% of the subgraphs touch an estimated 10 or fewer unique cache lines.

The early results for ICCs are quite promising, revealing high percentages of potential ICC content across standard and scientific workloads. From these results, we have reached the following conclusions:

- The problem of branch-criticality in ICCs occurs very rarely – less than 1% of ICCs are branch critical on average for the traces considered.
- A naive computational partitioning algorithm results in 1.5% of instructions offloaded from the host to the DCPs.
- Potential IPC improvement observed due to improved cache miss rates was 1.32 on average.
- In other studies, we have found that ICCs significantly decrease the number of host L1 misses – up to 61% when used with a column-associative L1 [100].
These improvements will provide for more flexibility in the design space that will likely allow for decreased L1 cache size and reduction of traffic in the memory hierarchy: both highly desirable characteristics in the design space of high-performance, low-power architectures.
CHAPTER 4
PASSIVE/ACTIVE MULTICORE ARCHITECTURE AND BLITS

In this chapter, we offer an introduction to an innovative architecture for a new class of heterogeneous multicores which we term Passive/Active Multicores (PAM) [100,101], and a corresponding execution model based on migrant computations [90]. PAM consists of two sets of cores: a set of one or more conventional cores – called the active cores or A-cores – which actively govern the contents of caches; and a set of simpler cores – termed the passive cores or P-cores – which work on data in the caches with traveling threads. The primary differences between PAM and the HWP/DCP architecture in chapter 3 stem from our tailoring PAM for efficient execution of these traveling threads. The P-controller, SIMD execution, and exposure of locality through the ISA are a few of the new features.

The contributions of this chapter are:

- The introduction of the cache-passive and cache-active cores as building blocks for a Passive/Active Multicore (PAM) architecture.
- The definition of the locality-tuned traveling thread as a software-level computational unit for PAM, and an illustration of how thread migration occurs on PAM.
- The definition of an ISA called BLITS for use on the cache-passive cores for constructing threads.
- A presentation of the parameters and metrics for the design and evaluation of the PAM architecture.
4.1 Processing Cores

In modern processors, a core can span a wide range of possible designs in terms of complexity, energy and power consumption, and performance. All cores generally have execution units for performing logical and arithmetic operations on data, logic for fetching and storing data to and from storage structures, control logic for guiding the execution of instructions, and datapath for routing bits among the components.

4.1.1 P-cores

The cache-passive core – or passive core or P-core – is a core of low-cost and low-complexity which has a high bandwidth path between memory and execution units and is optimized for compiler-extracted fine-grained thread execution. The P-core is cache-passive is the sense that it does not include hardware logic with its memory that implements a traditional cache replacement policy and cache coherence logic. This is similar to the Synergistic Processing Elements (SPEs) in the Cell B.E. which use software-controlled scratchpad memories \[79\]. Although software may control the P-cores’ storage contents in this manner, logic that is external to the P-cores may alternatively control their contents (such as the case where P-cores’ memory acts as L2 cache slots for another core \[100\] \[101\]).

The tailoring of the P-core for the execution of traveling threads has important implications which distinguish it from the DCP in the previous chapter. First, the DCP executes ICCs for improving cache performance, which are spatially static – that is, their execution begins and ends on the same core. On the other hand, P-cores improve not only cache performance but also parallelism through the execution of traveling threads. As such, the P-cores will provide a microar-
TABLE 4.1  
CACHE-ACTIVE VS. CACHE-PASSIVE CORES  

<table>
<thead>
<tr>
<th></th>
<th>Cache-Active</th>
<th>Cache-Passive</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>On Hit</strong></td>
<td>* Data may be copied into core’s registers for a long time (&gt;&gt;1 basic block).*</td>
<td>* Same, except no “long-term” copies.</td>
</tr>
<tr>
<td></td>
<td>* Synchronization of register value copies is a software problem.</td>
<td>Never use a copy for &gt; basic block.</td>
</tr>
<tr>
<td></td>
<td>* Synchronization of cache lines within hierarchy is a hardware problem.</td>
<td></td>
</tr>
<tr>
<td><strong>On Miss</strong></td>
<td>* The lines (and tags) stored in the cache change and move all the way up and down through the hierarchy.</td>
<td>* Lines change in-place at P-core.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>* Typically does not cause a line to move down in the hierarchy.</td>
</tr>
</tbody>
</table>

4.1.2 A-cores

We can contrast the P-core with the conventional concept of a core, which we refer to as a *cache-active core* or *A-core*. The A-core controls which lines are resident in its caches, either through loads and stores and a conventional cache replacement policy implemented in hardware, or via special instructions as with scratchpad memories \[110\]. Table 4.1 compares the activities that occur on a memory reference of a cache-active core to a cache-passive core.
Figure 4.1. PAM Architecture

An example of a PAM architecture. The processor consists of two types of cores: the A-core which actively governs the contents of the cache, and a set of P-cores which process on contents of the L2 cache but do not move cache lines.

4.2 Passive/Active Multicore Architecture

The Passive/Active Multicore Architecture or PAM is made up of one or more A-cores and one or more P-cores. The A-cores sit at the bottom of the memory hierarchy, while each P-core sits in L2 and has some number of physical cache slots associated with it. These slots double as both the P-cores’ local memory and as a section of storage for an upper level cache (in this work, the L2) for the A-cores.

Figure 4.1 gives an example of a PAM architecture. For the PAM architecture, we assume that the extraction of parallelism occurs through a combination of compiler-generated speculative threads and aggressive branch prediction at runtime. There is one A-core present with an L1 cache and a number of P-cores arranged in what is termed the P-core Mesh, whose memory blocks form the multi-bank L2 cache for the A-cores. The A-core is a conventional core which can efficiently execute sections of code with low ILP. During the execution of these
sections, the A-core is responsible for managing branch prediction and initiating computations on the P-cores by sending initiation requests to the P-controller. The P-controller serves the dual purpose of acting as the L2 cache controller for the A-cores, and also for governing the execution of computations running on the P-cores. The P-controller’s responsibilities include servicing tag lookups from both the A-core and the P-cores. The P-cores consist of the memory slots, an interface to the P-controller, functional units, and a special structure called the Thread I$ which can store P-core code for threads. The concept of the P-core is based on the idea of PIM Lite presented in prior work [111, 170], which pitch-matches functional units to the memory macro to provide a high-bandwidth path from storage to the execution units.

4.3 Traveling Threads

The definition of a thread in this work differs from prior definitions in important ways. Our thread shares some properties with the traditional notion of a thread – such as a pthread – but then also shares some properties of the traditional notion of the instruction. It also has some properties that are not traditionally found in either the thread or the instruction. Table ?? enumerates the differences in various properties of different types of software-level computational units.

The thread for our execution model is a software-level computational unit that contains instructions and data necessary to initiate these instructions, exploits parallelism at the instruction-level, locality in both register files and memory, and microarchitectural mechanisms to efficiently move the thread from one site of execution to another\(^1\). These threads have three important characteristics:

\(^1\)Logically, a site of execution is a processing element that has associated with it storage which is some portion of a larger address space.
TABLE 4.2

TYPES OF SOFTWARE-LEVEL COMPUTATIONAL UNITS

<table>
<thead>
<tr>
<th></th>
<th># of Instrs</th>
<th>Created By</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process</td>
<td>&gt;&gt;100</td>
<td>Programmer</td>
</tr>
<tr>
<td>Pthread</td>
<td>~100</td>
<td>Programmer</td>
</tr>
<tr>
<td>CUDA Threads</td>
<td>~10–100</td>
<td>Annotation from programmer through function, compiler-constructed. Match streaming SIMD model</td>
</tr>
<tr>
<td>PAM Thread</td>
<td>~10</td>
<td>No programmer intervention. Compiler or Microarchitecture-derived or both</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Interface</th>
<th>Scheduled By</th>
<th>Resources Needed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process</td>
<td>Function</td>
<td>OS</td>
<td>Memory: Heap, Stack, Code</td>
</tr>
<tr>
<td>Pthread</td>
<td>API</td>
<td>OS</td>
<td>Stack, subset of registers</td>
</tr>
<tr>
<td>CUDA Threads</td>
<td>Warps, Thread Blocks, Threads</td>
<td>Compiler-dictated policy executed in H/W</td>
<td>Memory, Registers</td>
</tr>
<tr>
<td>PAM Thread</td>
<td>ISA</td>
<td>At run-time by A-core</td>
<td>Cache Line, Registers, Small piece of code</td>
</tr>
</tbody>
</table>

- **Migrant**: These threads differ from conventional threads of execution in that their formation takes into account an alternative mechanism available for remote data access. A load or store in a conventional thread causes data to move from a remote location to a cache that is local to the thread. In contrast, the PAM architecture provides a mechanism for packaging the thread’s instructions and its state and moving it for execution at the core closest to the cache containing the remote data.

- **Locality-Tuned**: The assignment of instructions from the sequential code to threads takes into account locality of data operands and favors the assignment of instructions that have operands that are near each other in memory to the same thread. Thread assignment also favors less communication of register values among threads.

- **Fine-grained**: Unlike conventional threads, which tend to have 30 instructions or more [44], these threads are small in size – generally 20 instructions or less. Primary benefits of fine-grained threads are the reduction of per-thread overhead (especially network overhead due to thread migration) and increased numbers of threads leading to higher levels of parallelism.

This definition of threads has important implications for the compiler and microarchitecture:
• The software level will have access, via a special instruction, to run-time information regarding whether a memory address is local and present (low in time and energy costs) OR remote or not present (expensive in time and energy costs). An address \( A \) is local to P-core \( p \) if it falls on a cache line that maps to a set associated with \( p \). Otherwise, it is remote. Address \( A \) is present at \( p \) if it is local to \( p \) and \( p \)'s cache currently contains the line holding \( A \). Otherwise, it is not present. Note that local vs. remote is a static property (it doesn’t change), while present vs. not present is a dynamic property (may change at run-time). If address \( A \) is local and present, the instruction causes the controller to lock the line containing the address \( A \) in the cache until the thread accesses it. If address \( A \) is local but not present, there is no guarantee that it will remain not present.

• The ISA will provide an instruction that will cause the thread from which the instruction executes to move from one site of execution to another. An efficient microarchitectural implementation of this instruction is a primary design objective.

• To use these instructions for forming such threads, the compiler will have knowledge of storage past the register file, and also make predictions about run-time locality.

• The thread exploits a high bandwidth path to memory\(^2\).

4.3.1 Thread Migration on P-cores

The primary means for bringing together computations and data on P-cores is thread migration. Thread migration circumvents the need for an expensive (in terms of hardware complexity, time, and energy) cache coherence protocol (among P-cores). Migration also allows the compiler designer or programmer to think of computational partitioning and data placement together as tightly coupled problems, rather than the traditionally separated problems as in von Neumann architectures. Hence, a primary feature of the P-core is logic for efficiently moving

\(^2\)Per thread memory bandwidth will be high relative to conventional architectures for two reasons. First, the computation will be spread out among multiple execution sites, relieving pressure from what would be a centralized storage structure. Second, the microarchitecture is based on a previously designed and fabricated core called PIM Lite [170], which aligns functional units to memory to achieve a high bandwidth path between them.
a thread from one P-core to another. Computations for P-cores will leverage the feature of efficient thread migration mechanisms as a primary means for achieving high performance and low power consumption with low hardware complexity over a wide range of applications.

The primary benefits of traveling threads and PAM over centralized architectures are:

- Reduced per core bandwidth requirements to local storage.
- Reduced average latency of memory accesses across A-cores and P-cores (relative to the A-core alone) with good computational partitioning.
- Reduced cache pollution through avoidance of conflicts in the local storage, resulting in increased hit rates.
- Increased parallelism due to offload and remote, concurrent execution of instructions.
- Reduced power density by distributing the computations.

The four primary tradeoffs in a system optimized for migrant computations are:

- Overhead of migrating threads among cores.
- Thread management overhead, including initiation and commit.
- Area costs of extra cores.
- Mechanisms for integration/combination of results from different threads.

Figure 4.2 illustrates the execution of a parallel section of code on PAM. The compiler generates the code required for the execution of P-core threads. The L2 fetches cache lines containing this code from the memory and upper level caches into the P-cores’ memory via requests from the A-core. The A-core initiates P-core thread execution with a message sent to the P-controller. Execution of the thread begins at the P-core whose memory contains the thread’s code. When a
computation on a P-core $s$ requires data residing in the cache bank of a remote P-core $t$, the computation is moved from $s$ to a special storage location at $t$, labeled Thread $I^s$ in figure 4.1. To determine the destination of the migration, the P-core must send a tag lookup to the P-controller. The network package sent from $s$ to $t$ contains the thread’s code and any register state required to resume execution of the thread.

The lookup, packaging, and network events associated with migration are a primary source of overhead in this execution model. It is desirable to execute many of a thread’s instructions at each core before migrating the thread – in other words, to raise the computation-to-communication ratio of P-core execution.

4.4 BLITS

The ISA for the P-cores provides an interface to the compiler or programmer for its microarchitectural features, including efficient wide SIMD execution and thread migration. We name this ISA Byte-addressed Long Instructions for Thread Synthesis, or BLITS, to emphasize the features of the ISA. The instructions are byte-addressed to support widely used SIMD operations. The ISA is a set of long instructions so that the compiler can encode parallelism directly into the instructions. This is similar to a VLIW architecture, and in the P-core design we seek to achieve the VLIW’s strengths of simplified microarchitecture and low energy consumption. BLITS allow the construction of traveling threads introduced previously.
Figure 4.2. Thread Execution on P-cores

The figure illustrates the execution of a phase of an application with high parallelism using thread execution on P-cores.
4.4.1 Thread Initiators

The A-core sends commands to the P-cores to govern thread execution. To begin the execution of a thread, the A-core passes a command called a thread initiator, which is an advancement of the ICC Activator presented earlier. The controllers for the upper level caches fetch instruction cache lines as the host requests them. Along with the lines containing A-core code, the controller also brings in lines containing BLITS code for execution on the P-cores, so it is likely that the L2 contains the instruction cache lines prior to the time of thread initiation. Hence, rather than shipping code out from the host during initiation, it requires much less cache bus bandwidth – and therefore less time and energy – to use a control vector\(^3\). The A-core moves addresses into registers to indicate which cache lines contain instructions for threads. It sets bits in the control vectors to indicate which instructions the P-cores should execute, as shown in figure 4.3.

4.5 Design Parameters

Figure 4.4 illustrates the parameters and metrics for the design and evaluation of PAM and traveling thread generation. In addition to the enumeration of the parameters and metrics, the graph shows the relationships among them and at which level of the software-hardware stack each falls. The rectangular boxes represent parameters that we can vary in the design optimization. There are two levels in the stack which include parameters: the compiler-level – which includes parameters that effect computational partitioning and locality analysis of thread generation – and the chip-level – which includes parameters that effect microarchitectural components. This set of parameters is not exhaustive. Although other

\(^3\)The control vector was introduced briefly in section 3.3.1.1
At A-Core

<table>
<thead>
<tr>
<th>Addr</th>
<th>Op</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x9019717c</td>
<td>ALU</td>
</tr>
<tr>
<td>0x9004059c</td>
<td>LOAD</td>
</tr>
<tr>
<td>0x900405a0</td>
<td>ALU</td>
</tr>
<tr>
<td>0x900405a4</td>
<td>ALU</td>
</tr>
<tr>
<td>0x900405af</td>
<td>ALU</td>
</tr>
<tr>
<td>0x900405ac</td>
<td>ALU</td>
</tr>
<tr>
<td>0x90040560</td>
<td>ALU</td>
</tr>
<tr>
<td>0x900405bd</td>
<td>ALU</td>
</tr>
<tr>
<td>0x90040588</td>
<td>ALU</td>
</tr>
<tr>
<td>0x9004059c</td>
<td>JMP</td>
</tr>
<tr>
<td>0x90197160</td>
<td>ALU</td>
</tr>
<tr>
<td>0x90197164</td>
<td>BRA</td>
</tr>
</tbody>
</table>

New Cache Line Addrs

Control Vectors

At L2 Cache

I-Cache Line: 0x2401016

- ALU
- BRA
- X
- X
- X
- X
- X
- X
- ALU
- LOAD
- ALU
- ALU
- ALU
- ALU
- ALU
- JMP

I-Cache Line: 0x24065c5

- ALU
- BRA
- X
- X
- X
- X
- X
- X
- X
- X
- X
- X
- X
- X

Figure 4.3. Control Vectors Example from DAXPY Code
Figure 4.4: PAM Design Parameters and Metrics
parameters, such as technology, are available to the architect, figure 4.4 highlights the parameters that are most relevant to our design space exploration. Note that although we do not evaluate variations on each of these parameters, this diagram provides a platform for future explorations.

The ovals represent the metrics that may change as we adjust the design parameters. The metrics fall into four layers of the stack. We synthesize metrics at the “Low-Level Chip Metrics” level to obtain the more abstract metrics of the “High-Level Chip Metrics” level. For example, the costs from L1 lookups, L1 accesses, L2 lookups, and L2 accesses are all stand-alone, low-level metrics of interest, but they will also factor into the equation for calculating the higher-level metric of overall costs of A-core memory accesses.

The usefulness of these metrics will depend on the context of usage. Our tools allow reporting of metrics from every layer of the stack, but the most significant metrics fall at the bottom layer labeled “Deliverable Metrics.” The node labeled “Cumulative Execution Costs” represents the overall time and energy costs of execution for a given design. Note that there is also an “adjustable” aspect to the metrics, but in a different manner from that of parameters. Although the overall costs attributed to some class of events is calculated, we can adjust the cost of an individual event of that class. For instance, based on technology, the cost of a single L1 tag lookup may change considerably, so we can vary it as needed. On the other hand, parameters are, for example, thresholds and physical dimensions of microarchitectural structures.

The remaining nodes are parallelograms, which represent complex dimensions of the design space and warrant a significant amount of study past what is given in this work. For example, the “Interconnect Design” represents the design ex-
ploration of the on-chip network for the PAM architecture. Networks-on-chips or NoCs are an entire subfield of research in computer architecture in and of themselves. Design exploration of the PAM NoC is an avenue for future work.

The edges between nodes represent that one parameter or metric has direct effects on the other. For instance, a change in the size of the L2 cache has a direct effect on the costs of an L2 tag lookup. The figure offers some insight into how some parameters indirectly effect costs where such a relationship is not immediately apparent. For instance, L2 line size effects the Tlet I$ R/W costs directly, which in turn effects the cost of a thread migration. Hence, we can expect an increase in L2 line size to have some effect on the cost of a thread migration.
CHAPTER 5

OPPORTUNITY STUDIES

In the design of a novel architecture, it is important to understand and quantify the performance bottlenecks that this new design will overcome. This chapter presents empirical results illustrating fundamental application- and architecture-level challenges of aggressive branch prediction which the PAM design addresses. The first section characterizes properties of the applications and presents bottlenecks in the exploitation of parallelism. The material includes empirical results from both a suite of scientific applications from Sandia National Laboratories and the standard SPEC benchmarks. The final section offers a performance comparison of PAM with conventional architectures.

The specific objectives of this chapter include:

• Characterize the parallelism available through aggressive branch prediction in a set of applications.

• Introduce the new concept of graph tails and explain how they limit performance.

• Estimate the bounds on the migration overhead of one PAM architecture in order to surpass the performance of a conventional processor.

• Present an analytical model for determining these estimates and define the metric of instructions per migration (IPM). Use IPM in the analytical model to estimate the effective instructions per cycle (IPC) of the PAM.
5.1 Dependence Graph Analysis

A fundamental concept of our thread formation is the ASAP-scheduled Directed Acyclic Graph (DAG), or ADAG\(^1\). We use the following techniques in the implementation of a tool for forming threads, which we refer to as the thread partitioner.

For the DAGs in this chapter, a vertex represents an instruction, and an edge between two vertices \(u\) and \(v\) represents a register or memory dependence between instructions \(u\) and \(v\). A register dependence is a write-read pair to the same architectural register, and a memory dependence is a store-load pair with the same target memory address. When building the ADAG, we consider instructions sequences across multiple basic blocks (recall from section 3.1.2 that a basic block here is a dynamic group of instructions between two branches) by using an oracle branch predictor with 100% accuracy. Thus, a dependence chain of instructions ends at a store to some address \(X\) with no load from \(X\) later in program order within the segment of the program under consideration.

If we know the branch outcomes from an execution of the program consisting of \(n\) basic blocks, we then have a total ordering of those basic blocks \(\{bb_0, bb_1, \ldots, bb_{n-1}\}\) – representing the “program order” execution. A sequence of basic blocks, then, is a set of consecutive basic blocks \(\{bb_i, bb_{i+1}, \ldots, bb_j\}\) from the total ordering where \(i<j\).

In the construction of threads, the thread partitioner considers a sequence of basic blocks called a region. A region is a basic block sequence of some specific size that we set as a parameter (called region size, in number of basic blocks) to the thread partitioner. A sequence of basic blocks within a region is a subregion.

\(^1\)See section 3.1.2.1 for the definition of ASAP scheduling.
thread partitioner varies the subregion size from 1 to step\_depth\_limit by powers of two, where step\_depth\_limit is a parameter.

Because the thread partitioner considers instruction sequences that cross basic block boundaries, an important technique in the implementation of these techniques in practice is speculation. Many modern architectures employ speculation of two types: data speculation and control speculation. With data speculation, the compiler or microarchitecture can reorder the memory accesses, such that some load $l$ occurring after some store $s$ in program order executes before $s$ – even though $l$ may read from the address to which $s$ writes. The technique requires mechanisms for detecting when the hoisted load is actually dependent on one of these stores, and how to recover from an ordering violation [69].

In this work, we are primarily concerned with control speculation. Control speculation involves the concept of conditional branches for which a processor must perform operations to determine whether the branch is taken or not taken [8]. Once the processor determines the outcome of the branch, the branch is said to be resolved. Control speculation occurs when the processor uses a prediction mechanism to execute instructions from one way of a branch before resolving that branch. Speculation depth is the number of unresolved conditional branches between two instructions $u$ and $v$ in the code along some valid instruction sequence, where the processor has resolved the last branch before $u$ in program order and $v$ occurs after $u$.

5.2 Application Properties

This section presents various bottlenecks that conventional architectures face in the exploitation of parallelism. Empirical results illustrate the prevalence of
these bottlenecks in real applications. The discussion includes an explanation of
how the PAM architecture is well-suited to address these bottlenecks.

5.2.1 Criticality

The ADAGs are useful in modeling bounds on the parallelism available in a
region. However, in the construction of ADAGs from execution traces, we will
notice that there is generally a set of critical paths that place a lower bound on
the execution time. The first is the branch-critical path – the minimal path of
instructions through the ADAG of the application that contains the conditional
control-flow instructions. The length of this path then places a lower-bound on
the execution time of an application in that even if the processor predicts the
branches perfectly, it must verify the correctness of the predictions by executing
the branch-critical path. Figure 5.1 shows an example of a branch-critical path in
an ADAG using a subregion from ALEGRA.

Next is the concept of an ADAG tail. In an ADAG with \( n \) instructions and
\( m \) stages, we will observe that the first \( i \) stages of the graph have a high number
of instructions per stage (width), but the maximum width of the remaining \( m-i \)
stages never goes above some value \( t \), called the tail width threshold. The cause for
the tail is that the ASAP scheduling brings all of the parallelism available across
the basic blocks to the top of the ADAG, so there is little parallelism left in the
final stages.

Figure 5.2 shows the width of each stage of the ADAG for an execution trace
from \texttt{blast} for 3 different trace lengths over the last 1200 stages of the trace.
Assuming that \( t=5 \), the figure shows that \( m-i \) increases substantially as \( m \)
increases. Hence, the time spent on the serial portion of the code will increase as
Figure 5.1. Example of an ADAG from ALEGRA Illustrating a Branch-Critical Path

The branch-critical path is enclosed in brackets.

subregion size (or speculation depth\(^2\)) increases. At some point in the execution of the application, the processor must execute this critical path, which will place a lower bound on execution time. We refer to the last \(m-i\) stages of the ADAG as the tail.

To further illustrate this trend, we have performed a more extensive set of experiments using the methodology shown in figure 5.3. The flow chart illustrates the methodology for producing figures 5.4 and 5.6. The algorithm takes 2 parameters: \(i_{\text{max}}\) and \(\text{step\_depth\_limit}\). \(i_{\text{max}}\) represents the number of samples that we will take for each subregion size, and \(\text{step\_depth\_limit}\) is the maximum subregion size.

\(^2\)Note that the speculation depth is the subregion size minus one.
Width of ADAG vs. stage with ASAP schedule for blast. SimPoint \cite{135} was used in the generation of an execution trace of blast that approximates overall execution, as described in \cite{125}. Here, we have taken a random sample of this execution trace to generate three ADAGs. Scheduling is not constrained by control dependencies, as we assume an oracle branch predictor with 100% accuracy. \( n \) is the number of instructions in the ADAG and \( m \) is the number of stages. The figure shows a set from the last stages of each ADAG, illustrating how tail length increases with number of instructions. The plots for \( n=10000 \) and \( n=5000 \) are offset vertically by +1 and +2, respectively, to improve visibility of graph tails.

size. The algorithm begins by selecting a random basic block, which we will call \( BB \), of any size from a trace of 100,000,000 instructions. The algorithm initializes the subregion size to 1, and constructs an ADAG from instructions in the subregion starting at \( BB \). It then updates accumulators for various properties – such as number of instructions, number of memory accesses, number of stages, and tail length\(^3\) – of the graph. The next step is to increment \( i \) (the index of the current sample), and compare it to \( i_{\text{max}} \). If they are not equal, the algorithm selects another random basic block for the next sample and continues. Otherwise, it reports

\(^3\)We calculate the tail length by choosing a value, \( t \), for the tail width threshold, and counting the number of stages from the last stage before reaching a stage of width \( \geq t \). Here, we use a value of 5 for \( t \). Section 6.3.2 discusses how we chose this value.
average values across the samples – for example, for figure 5.4, we gather average number of stages and average tail length. The next step is to increase the subregion size by a factor of two. If the subregion size is greater than \textit{step\_depth\_limit}, then the algorithm is complete. Otherwise, we take another set of samples for the new subregion size, starting at a different random basic block.

For this study, \(i_{\text{max}}\) is set to 16, and the \textit{step\_depth\_limit} is set to 32. The primary reason for choosing these values is that they resulted in run times that permitted the analysis of all traces. Run times increase with both \(i_{\text{max}}\) and \textit{step\_depth\_limit}. In the latter case, this is primarily because ADAG construction time increases significantly, and the value of 32 resulted in acceptable run times. We did perform some study of variation of results versus \(i_{\text{max}}\). For the
cases we considered, using the value of $i_{max}=16$ did not result in high variation between runs.

The charts in figure 5.4 show the average number of stages in the ADAGs versus subregion size for four applications. The green portion of the graph shows the average number of stages in the graph tail, while the red portion shows the average number of stages not in the graph tail. Emerging architectures supporting aggressive branch prediction, such as PAMs, should take this trend into account, as the graph tails will place different demands on the processor than the wide portions of the ADAGs.

These results illustrate that using aggressive branch prediction to extract parallelism will lead to an execution graph that exhibits two phases which have drastically different levels of parallelism: the graph tail phase, which is a sequential section of code that will execute well on a conventional core that cannot take advantage of parallelism; and the non-tail phase which would execute efficiently on many small cores. The results form the basis for the PAM architecture design.

5.2.2 Memory Latency-Bounded Speculation Depth

In parallel architectures – such as multicores with shared caches or SMT processors – that execute multiple threads on centralized processing hardware, the average access time to the shared storage will place a bound on the parallel execution of threads. This is because as the number of threads increases, the number of unique values the threads use (read or write) will also increase. Even if each thread is not reading unique data, it will certainly be writing unique data (if not, then there is no need to execute the thread). The increasing data usage associated with increasing thread count will cause more cache misses to the shared storage,
Figure 5.4. Tail length and # of stages vs. subregion size for 4 traces.

Each figure shows the observed tail length and number of stages in the ADAGs. Each point shows ADAG tail length and stage count averaged across multiple trace samples – as shown in the methodology given in figure 5.3 - from each application trace. Note that as subregion size increases, the tail length also increases, revealing that the serial portion of the code becomes an increasing problem as subregion size increases. We verified that this trend was present among most of the traces studied.
which will raise average access time. At some point, the access time will be on
the critical path, causing threads to stall while waiting for data.

To consider this problem from a mathematical perspective, say the average
access time\(^4\) to a shared cache is \(s\) ns and we have a total of \(i\) instructions in \(t\)
threads. So, assuming that each instruction needs 0.34 operands on average\(^5\) from
the shared cache to execute, then the limit, \(t_{\text{max}}\), on the number of threads per
unit time for which the storage can provide data is:

\[
t_{\text{max}} = \frac{1}{s} \frac{\text{access}}{\text{ns}} \times 1 \frac{\text{operand}}{\text{access}} \times 1 \frac{\text{instruction}}{0.34} \frac{t}{\text{threads}} = \frac{2.94 \times t}{s \times i} \frac{\text{threads}}{\text{ns}}
\]

(5.1)

So, given some fixed number of threads \(t\) with \(i\) total instructions, we can see that
\(t_{\text{max}}\) will decrease as the access time \((s)\) increases.

In the context of speculative thread extraction mechanisms, this means that
the average memory access time (\(AMAT\)) will place a bound on the maximum
subregion size that will decrease execution time. In other words, even if the
architecture speculates perfectly and attempts to execute the instructions of the
ADAG in parallel, the AMAT will limit the number of executable instructions per
stage.

If we have an upper bound on the AMAT (\(AMAT_{\text{lim}}\)), we can calculate the

\(^4\)By *average access time*, we are referring to a value calculated from metrics taken from the run
of an application, rather than a property (such a latency or bandwidth) of the cache hardware
itself. That is, given the set of memory accesses \(M\) in the application, the average access time is
the *cumulative execution time* of the instructions in \(M\) divided by \(|M|\). The *cumulative execution
time* is the duration of time during which any instruction in \(M\) is executing. For example, if
two memory accesses \(i\) and \(j\) both execute from time \(t_{\text{start}}\) to \(t_{\text{end}}\), the cumulative execution
time of \(i\) and \(j\) is \(t_{\text{end}} - t_{\text{start}}\) (not \(2*(t_{\text{end}}-t_{\text{start}})\)).

\(^5\)Rupnow et al. have given instruction mixes of the SPEC-FP applications, observing about
34\% of instructions on average are loads \([152]\).
Figure 5.5. Example from ALEGRA Illustrating $IPC_{MB}$ Calculation

The figure shows two models illustrating the calculation of $IPC_{MB}$. The cycle time is taken as the reciprocal of the clock rate of the execution units in a recent out-of-order processor design [74].

$IPC_{MB}$ (for Memory-Bounded IPC), which is the IPC bounded by the AMAT:

$$IPC_{MB} = \frac{T_C \times IPA}{AMAT_{lim}}$$  \hspace{1cm} (5.2)

where $T_C$ is the cycle time and $IPA$ is the average number of instructions per memory access. $IPA$ for an ADAG is the total number of instructions in the ADAG divided by the total number of memory accesses. Figure 5.5 shows an example of the calculation of $IPC_{MB}$ using a subregion from ALEGRA.

The next metric we consider is $IPC_{max}$, which is an estimate of the maximum
Figure 5.6. $IPC_{\text{max}}$ and $IPC_{MB}$ vs. subregion size for 4 traces.

To produce these figures, we used the methodology shown in figure 5.3. Each point in the figures is an average across 16 samples. $IPC_{MB}$ shows the average of the memory-bounded IPC values, using an optimistic memory access time taken from CACTI. $IPC_{\text{max}}$ shows the average IPC values bounded by instruction dependencies. Note the common trend that the fraction of parallelism available ($IPC_{\text{max}}$) that can be exploited decreases as subregion size increases due to the bound placed by memory access time.
IPC of the application for a given section of the trace, bounded by the instruction dependencies only (all instructions, including memory accesses, execute in one cycle). This assumes a perfect ASAP scheduling, such that each node (representing an instruction) finishes one cycle after the completion of the instructions on which it is dependent:

\[
IPC_{\text{max}} = \frac{I_{\text{tot}}}{L_{\text{asap}}}
\]  

(5.3)

where \(I_{\text{tot}}\) is the total number of instructions in the graph, and \(L_{\text{asap}}\) is the number of stages in the ADAG for the given section of code\(^6\). From these equations, we generate the graphs in figure 5.6. The cycle time is set at 0.25 ns (4 GHz clock rate). For IPC\(_{\text{MB}}\), we optimistically assume a 100% L1 cache hit rate, and take the value of AMAT\(_{\text{lim}}\) from CACTI [171] as the access time for a 16-KB, direct-mapped cache with 64-byte line size and 4 banks using 32-nm technology. The top edge of the green section of the figures is the memory-bounded IPC, while the top edge of the red section is the maximum IPC as given by equation 5.3. Hence, we can view the red section as parallelism that is available that the machine cannot exploit due to limits on memory latency. Notice also that as subregion size increases, IPC\(_{\text{max}}\) increases across all figures 5.6(a)-5.6(d), while IPC\(_{\text{MB}}\) generally shows negligible increase. We verified that this is the general trend for all traces studied, and show one from each suite to illustrate the trend. Hence, the memory wall problem worsens with increasingly aggressive speculation.

5.3 Modeling Bounds on Migration Overhead

To develop an understanding of the tradeoffs involved with PAM architectures versus conventional heterogeneous multicores, we have established models for the

\(^6\)Equivalently, all instructions in each stage execute concurrently in one cycle, for each stage.
overheads that set PAM apart from other architectures [97]. These migration overheads – or the costs associated with address lookups, packaging, and network events of thread migrations – are a primary inhibitor of performance for PAMs. In this section, we quantify bounds on migration overhead by determining the minimum number of instructions per migration required to match the performance of a processor bounded by $\text{IPC}_{\text{MB}}$. The next section presents the methodology used along with the applications considered. The second section gives an analytical model for migration overheads. The final section presents the performance results and discussion.

5.3.1 Analytical Model

The following bounds migration overhead:

$$\frac{T_C}{p \times \text{IPC}_{\text{lim}}} + MPI \times T_M = \frac{T_C}{\text{IPC}_{\text{lim}}}$$

(5.4)

where $T_C$ is cycle time, $MPI$ is the number of migrations per instruction, $T_M$ is the time required for one migration, $\text{IPC}_{\text{lim}}$ is the maximum IPC achievable by the conventional architecture, and $p$ is the number of P-cores in the PAM. The right-hand side of the equation gives the time per instruction for a baseline superscalar architecture, and the left-hand side gives us time per instruction for PAM. By setting these equal, we identify cases where performances are at least the same. Assuming that the machine achieves perfect parallelism across the P-cores\(^7\), the time per instruction is then divided by $p$. We then add the penalty for migrating

\(^7\)These equations do not take into account that only a fraction of the instructions execute at the P-cores due to partitioning of the tail/non-tail phases and branch-critical paths. As a simplifying assumption, the P-cores parallelize the entire application.
computations with the term MPI*T_M. With T_M fixed, we then solve for MPI, or the maximum number of migrations per instruction to match the performance of the conventional architecture:

\[ MPI = \frac{T_C}{IPC_{lim} * T_M} * \frac{p - 1}{p} \]  

(5.5)

Alternatively, we can solve for \( mIPM \), or the minimum number of instructions each P-core must execute on average between each migration to match the performance of the conventional architecture:

\[ mIPM = \frac{p * IPC_{lim} * T_M}{(p - 1) * T_C} \]  

(5.6)

Equation 5.6 serves as the basis for our performance evaluation. \( IPC_{lim} \) is set to the maximum IPC achievable by the conventional processor bounded by an optimistic memory access time when speculating aggressively. We find \( IPC_{lim} \) by taking the maximum value of \( IPC_{MB} \) from the top three points from the \( IPC_{MB} \) data sets, as shown in figures 5.6(a)-5.6(d). In other words, considering the cases of speculating across 8, 16, and 32 branches, the maximum average memory latency-bounded IPC of these is \( IPC_{lim} \). Note that this is a highly optimistic performance number in terms of cache performance for baseline comparison, as we are assuming a 100% L1 hit rate. The number of P-cores, \( p \), is scaled from 2 to 64. The cycle time of the P-cores is scaled up by a factor of 8 relative to the A-core\(^8\). Finally, the time per migration, \( T_M \), is assumed to be the L2 access time for L2 cache.

---

\(^8\)Using the 500MHz clock rate of the embedded DRAM macro given by Barth et al. [13].
sizes varying from 1 MB to 16 MB, taken from CACTI\textsuperscript{9}.

5.3.2 Performance Results

For the performance evaluations, we use execution traces of applications from SPEC-Int, SPEC-FP, Sandia-Int, and Sandia-FP (see sections 3.1.3.1 and 3.3.3). We apply the methodology in figure 5.3 running trace analyses on SST [149] to produce figures 5.7–5.10. These figures show the trends for \textit{mIPM} values required of the PAM to match the performance of the conventional architecture. \textit{mIPM} is given in the figures with respect to varying numbers of P-cores and varying L2 cache size. We generated these graphs for all 47 traces, but here the figures show 3 from each of the 4 suites as they illustrate the trends characteristic of the majority of the graphs.

Intuitively, \textit{mIPM} is the minimum computation-to-communication ratio required for the migrant threads to meet the performance of the conventional processor. In other words, \textit{mIPM} gives the minimum number of instructions that must be executed on average at each P-core before the thread is moved to another P-core to resume execution. If the actual number is higher, then migration costs are amortized over more instructions, and thus have less effect on performance. Hence, a lower value of \textit{mIPM} is more desirable, as this implies that P-cores are free to migrate threads more frequently, as less locality is required among threads’ working sets to out-perform the conventional processor.

Consider the highlighted lines in figures 5.7–5.10 that that run parallel to the

\textsuperscript{9}T_M is the time for a thread to move from one P-core in the L2 to another, and we estimate this as the time to complete an L2 cache access in a conventional processor. This L2 access time, taken from CACTI, includes the time for cache bank access and routing from bank to the controller. We do not include TLB lookup time, as virtual addressing issues are left for future work.
mIPM vs. number of P-cores and L2 cache size for three application traces from the SPEC-Int suite. mIPM is the minimum number of instructions per migration required to match the performance of a conventional processor architecture whose IPC is bounded by memory access time.

Figure 5.7. mIPM vs. # of P-cores and L2 cache size, SPEC-Int applications.

Figure 5.8. mIPM vs. # of P-cores and L2 cache size, SPEC-FP applications.

Figure 5.9. mIPM vs. # of P-cores and L2 cache size, Sandia-Int applications.
axis of L2 size. These lines show an increase in \( mIPM \) as we hold the number of P-cores constant and increase L2 cache size. This is because our model increases the penalty associated with each migration as the L2 size increases, as we assume that a thread migration requires a lookup in the L2 tag array to determine the location of its operands. As L2 size increases, this tag lookup time will also increase\(^{10}\). An increase in the cost of each migration, represented as term \( T_M \) in equation 5.4, will require the execution of more instructions at each P-core for every migration to offset the increase in migration overhead.

Notice that as the number of P-cores increases, the rate of increase in \( mIPM \) with L2 size is lower. For a simple approximation of the parallel execution time with P-cores, we assume that the average number of instructions executed per cycle is equal to the number of P-cores. The average time for execution of each instruction on the P-cores is represented by the term \( \frac{T_C}{p \cdot IPC_{1m}} \) in equation 5.4. Hence, as this term decreases with the increase in \( p \), the \( MPI^*T_M \) term – which represents the average time per instruction allowed for migration time – increases. This results in a decrease in the minimum instructions per migration required to

\(^{10}\)The tag lookup time is estimated for each cache configuration using the CACTI toolset \[171\], with the same base parameters as given in section 5.2.2.
match the conventional performance on the right-hand side of equation 5.4.

As the number of P-cores increases, the effects of an increase in \( p \) decreases. The figures show this trend on the lines parallel to the P-core axis. If L2 size is kept constant and we increase the number of P-cores, the line decreases until about eight P-cores and then flattens. In equation 5.6, we see that the \( \frac{p}{p-1} \) term will approach one as \( p \) increases, so the increase in cores will have less of an effect on IPM. This implies that migration overhead (\( MPI^* T_M \) in equation 5.4) is the dominant term in the execution time, and the decrease in computation time as a result of increased P-cores has negligible effect. This point is reinforced in that we continue to see an increase in \( mIPM \) with migration time due to increasing cache size regardless of the number of P-cores. From these plots, we project that about four or eight P-cores are good design points when we are able to have good parallel efficiency, high clock rate, and a relatively low value of \( IPC_{lim} \). Also, decreasing migration time will be more important as cache size increases.

These results offer a general idea of the level of difficulty for forming computations (whether by static, dynamic, or hybrid techniques) for a PAM in order to out-perform the memory latency-bounded IPC. The graphs reveal that exceeding the \( IPC_{lim} \) should not present a difficulty for any number of P-cores using a 1 MB L2, as the \( mIPM \) is generally 1 or lower. However, since the line of \( p \) P-cores assumes perfect parallelization across the \( p \) cores, achieving this level of parallelization in practice will be difficult. This difficulty will manifest itself as increased complexity in the microarchitecture and compiler in forming and scheduling computations, and also as increased misspeculation costs. Consideration of such design details is left for future work.

Here we conclude that outperforming \( IPC_{lim} \) for a moderate number of P-cores,
such as 4 or 8, with a reasonable cache size, such as 4 or 8 MB, seems feasible, and the results confirm the benefits of distributed computations over conventional centralized processing with respect to memory-bounded parallelism. The results show that these configurations generally require a minimum of 2–4 instructions per migration to out-perform $IPC_{lim}$. Even across all configurations considered, the values of $mIPM$ appear within reasonable bounds. The maximum value of $mIPM$ observed for all traces was 7.09 instructions per migration, the minimum was 0.04, and the average was 1.88. In our ongoing work for developing automatic computational partitioning algorithms for P-cores, we have observed average thread sizes as high as 7.7 instructions on average across all traces with varied configurations in preliminary results. Experiments have shown P-core computations covering up to 24% of applications.

5.4 Conclusions

Emerging heterogeneous multicore architectures and their corresponding execution models and toolchains are under investigation for their effectiveness in accelerating single-threaded workloads using parallelization through aggressive branch prediction. This chapter has characterized the run-time properties of applications when extracting parallelism through aggressive branch prediction. We introduced the concept of graph tails in ASAP-scheduled DAGs and illustrated that applications generally exhibit two phases when aggressively speculating: a sequential section making up the graph tail phase, and a highly parallel section which we call the non-tail phase. The design of PAM and the traveling thread execution model leverages the results from the application characterization for partitioning the application into tail and non-tail phases and accelerating both.
In a PAM, the distribution of processing elements results in the desired characteristic that the memory access time does not place a tight bound on the IPC. A combination of mathematical modeling and simulation has illustrated that the minimum number of instructions per migration required to outperform an ideal processor bounded by memory access time for various PAM configurations falls within reasonable bounds when speculating aggressively.
Chapter 6

Models for Generating Traveling Threads

We next address the problem of dividing computations between the A-cores and P-cores. We build upon the previous techniques for locality analysis and WCC-based computational partitioning for ICCs. Performance evaluations show that these new techniques provide significant performance improvements from PAM with high offloads of traveling threads to P-cores. In the development of a new model for traveling thread generation, we meet the following objectives:

- Define a new model for partitioning data between the A-cores and P-cores that introduces the novel concepts of hot and cold cache lines.

- Introduce a model for partitioning computations into traveling threads that combines WCCs and hot/cold cache line analysis. This leads to the new concepts of hot and cold WCCs.

- Extend the partitioning model to identify opportunities to take advantage of cache bypassing using singletons.

- Develop a new simulator for producing quantitative results to present the effectiveness of the thread partitioning approach and estimate potential performance improvements.

6.1 Locality Analysis

In the first phase of our analysis, we characterize the locality of a cache line, especially in L1. In this discussion, a set of addresses does not uniquely identify
what we refer to as a cache line. Rather, we are interested in the copy of a cache line that is resident in some cache for some period of time. For example, the copy of the cache line whose first address is 0xFF538590 that is in L1 from time $t_i$ to $t_j$ is distinct from the copy of that line when in L1 from time $t_m$ to $t_n$.

Here, the objective is to classify one set of lines as hot which exhibit high locality of access and whose corresponding computations will execute on the A-cores. Another set of cold lines exhibits low locality of access whose corresponding computations will be the candidates for the threads that will execute on the P-cores. Using this characterization, our mechanisms for cache line insertion then become similar to those conventionally used for characterizing values during register allocation in a compiler. Previous work has applied this analogy, albeit in a more direct fashion, for use with scratchpad memories [110]. Table 6.1 compares the terminology for locality classification in the contexts of this work, for scratchpads and register allocation, and for a conventional cache.

6.1.1 Hot/Cold Cache Lines

The parameters to the analysis are cache line size and hot line threshold, $T$. When analyzing a basic block $b$, we examine $T$ instructions following $b$ in program order. If a memory access in $b$ touches cache line $l$, and then an access in the $T$ instructions following $b$ touches line $l$ again, then we label $l$ as hot. Say a later instruction less than $T$ instructions after $b$ in some basic block $c$ touches $l$ again. $l$ will be hot in $b$ and in all blocks from $b$ to $c$. In block $c$, an examination of $T$ instructions following $c$ for another touch of $l$ determines whether $l$ will stay hot after $c$. If there is no other touch after $c$ within the threshold, $l$ is not hot after $c$. We then label any line that does not satisfy this definition as cold. Table 6.1
The table compares the terms in this work to conventional terms for classifying a cache line (or register value) by certainty of its presence in the cache (or register file). The static categories are terms for classifying the locations of data at compile-time, while the dynamic categories are for classifying (usually with some amount of uncertainty) the locations of data at run-time. Note that the certainty of a line’s presence in the cache for P-cores may be highly dependent on branch prediction accuracy. We use “hot”, “warm”, and “cold” for cache lines in a similar manner to “live” and “killed” for register assignment, in that a “live” value is a likely candidate for assignment to a register. (In [5], the authors note that “if registers are full and we need another register, we should favor using a register with a dead value, since that value does not have to be stored.”)

shows that some lines have the label warm. The next section will define this label.

6.2 Computational Partitioning

Part of the novelty of traveling threads is the incorporation of the results of data locality analysis into the thread partitioning mechanisms during the assignment of instructions to threads. This section uses the cache line categorizations from section 6.1.1 in the introduction of hot and cold WCCs as the basis for traveling threads. Next is a discussion of an optimization for a special case called singletons. A presentation of a practical implementation – called window-based thread extraction – of the thread partitioning methods follows.
6.2.1 Hot/Cold WCCs

In the second phase of our analysis, we partition the WCCs of the non-tail section of the DAG into hot and cold WCCs (see section 3.2.1.2). A WCC is hot if it touches a hot cache line, and is cold if it touches at least one memory access and all its memory accesses are to cold cache lines. It is now possible for a cold cache line to change to warm. If a memory access in a hot WCC touches a cold cache line, the line changes from cold to warm. The toolset will select the cold WCCs as the basis for traveling threads.

Figure 6.1 shows an ADAG taken from the instructions in a simple loop. The memory accesses to hot lines are pentagons, while the memory accesses to other
lines (cold or warm) are rectangles\textsuperscript{1}. There are four WCCs in this subregion, and the two WCCs in the middle are cold. The line through the ADAG splits the non-tail (in the upper section of the ADAG) from the tail (in the lower section), with the tail width threshold\textsuperscript{2} set to 3.

On the PAM architecture, the A-core will execute the hot WCCs, and the P-cores will execute the cold WCCs. The advantages of this partitioning include:

- Reduction in communication among threads. There are, by definition, no edges between WCCs in a subregion. The edges account for both register and memory dependencies. Hence, this thread partitioning scheme favors threads that incur less communication costs\textsuperscript{3}.

- Leveraging of the features of previous cache designs. Because the partitioning uses an analysis which considers data at the cache-line level of granularity, the mechanisms for exchanging data between P-cores and A-cores will resemble that of conventional caches. Although modifications to cache policies and extensions to the cache hardware will likely lead to improved performance, much of the cache design will stay the same.

- Natural exploitation of thread-level parallelism. A WCC of significant size, when considered in its entirety, spans vertically down the graph. By splitting the graph into WCCs, these components that span the graph vertically can execute in parallel to utilize the P-cores as extra computational resources.

6.2.2 Singletons and Cache Bypassing

A special case, called a \textit{singleton}, occurs in this scheme of graph partitioning that presents problems for thread creation. A \textit{singleton} is a WCC of size one where the only node in the WCC is a cold memory access (either a load or a store).

\textsuperscript{1}Using this representation of an ADAG, a rectangle is an access to a cold line when all other memory accesses in the WCC are rectangles. A rectangle is an access to a warm line when there is at least one other node that is a pentagon in the WCC.

\textsuperscript{2}We choose a tail width threshold of 3 here to serve the example of illustrating the graph tail.

\textsuperscript{3}Note that the current methods do not take into account loop-carried dependencies. This is a point for future work.
By the previously defined methods, a singleton will map to a traveling thread. However, the behavior of such a “thread” would be contrary to the objectives of the traveling thread execution model, as it does not offer an increase in computational parallelism. A simple solution is to ignore this case and simply treat it as a normal memory access. However, considering that the singleton, by definition, is a cold memory access, a better option is to employ a technique called cache bypassing to avoid polluting the cache with a line that does not exhibit good locality [82].

A cache bypass is the movement of a data from level $L_n$ in the memory hierarchy directly to a lower cache level $L_{n-i}$, where $i>1$. This technique also allows the bypassing of L1 to send data directly to the register file. Hence, the data skips over intermediate levels of the memory hierarchy. This is desirable in cases for which the insertion of data into the cache will likely cause pollution. In the case of singletons, the data with the isolated access would bypass the L1 cache and the A-core would fetch the data directly into the register file.

### 6.2.3 Window-Based Thread Extraction

We have built a SST-based [149] trace analyzer that implements the concepts from the previous section. The experiments use traces that are 1 billion instructions in length. Parameters to the trace analyzer include the cache line size and the hot line threshold, $T$. The trace analyzer first scans a sequence of instructions of a basic block $b$, plus an additional $T$ instructions, which we will refer to as the $T$-set. It then builds the DAG of $b$ and splits the DAG into WCCs. For each memory access in $b$ to some line $l$, we place $l$ into the eligible set. If a memory access in the $T$-set touches a line $m$ that is in the eligible set, then we insert $m$ into the hot set and all others into the cold set. A WCC that touches at least
one hot cache line, then, is *hot*, while the remaining WCCs are *cold*. If a memory access in a hot WCC touches a cold cache line, then we move that line to the *warm* set. We then gather statistics on the basic block and the cache lines, and move on to the next basic block, \( C \). The analyzer clears the warm and cold sets after each basic block, but a line \( h \) in the hot set remains until that line qualifies as cold in some future basic block.

Table 6.2 enumerates the state transitions. It shows possible labels of a cache line \( L \) in the hot/cold labeling scheme during the analysis of a basic block \( C \). \( L_{in} \) is the label of the line when the analysis enters basic block \( C \). \( L_{mid} \) is the label of \( L \) during the analysis of basic block \( C \), and \( L_{out} \) is the label of \( L \) when the analysis of basic block \( C \) ends. The *Eligible?* column indicates whether the line is in the eligible set. An asterisk in the row indicates a case which is of particular interest to our study. To summarize the table, \( L_{mid} \) is hot if and only if \( L_{in} \) is hot. For \( L_{out} \) to be hot, two conditions must be met. First, a touch to \( L \) must occur in the T-set. Second, the line must be in the eligible set or \( L_{mid} \) must be hot.
6.3 Evaluation

This section presents an evaluation of the proposed techniques implemented in a toolset within the SST framework [149]. The toolset generates “offload efficiency,” which is the percentage of instructions in the trace that the analyzer offloads from the A-core to the P-cores. We then produced estimates of the potential performance improvements offered by the design. To do so, a high-level P-core simulator generates an estimate of the execution time of the P-core sections of the code. For all experiments, the cache line size is 64B with an L1 of 16KB, an L2 of 512KB, and 16 P-cores in the L2 (32KB per P-core).

This section first gives an example using DAXPY to illustrate the overall approach. The second section presents the methodology for the final simulations, and the third section gives the results and discussion.

6.3.1 Example: DAXPY

DAXPY (double alpha x plus y) is a loop from LINPACK, a suite for performance evaluation of supercomputers. As a widely-used operation from BLAS (Basic Linear Algebra Subprograms), the DAXPY loop (shown below) is in use for many matrix computations, such as LU factorization:

\[
\text{for}\ (\ i = 1 ; \ i \leq n \times \text{incx} ; \ i = i + \text{incx}) \\
\quad \text{dy}[i] = \text{da} \times \text{dx}[i] + \text{dy}[i];
\]

We used g++ and Amber to generate a PowerPC trace from this code section, extracted the instructions in the loop from the trace, and ran these instructions through the trace analysis described above. We vary the hot line threshold from 0 to 1 to emphasize how offload efficiency decreases as this value increases, as the difference is significant between these two cases. Figure 6.2(a) shows the ADAGs
The ADAG on the left shows a subregion of the sample loop from DAXPY with the subregion size set to 1, and the ADAG on the right shows a subregion with the subregion size set to 2. The memory accesses to hot cache lines are black boxes, while the memory accesses to cold or warm cache lines are white triangles. The gray ovals are other instructions, such as those corresponding to ALU operations, floating point operations, and branches. The hot line threshold is 1 for these graphs.

Figure 6.2. Hot/Cold ADAGs of DAXPY Inner Loop
Figure 6.3. Offload Efficiency and Estimated ‘Speedup’ for the sample loop from DAXPY.
of the subregion of the basic block with the subregion size set to 1, and figure 6.2(b) shows the ADAG with the subregion size set to 2. The hot line threshold is set to 1 in these figures. Figure 6.2(a) consists of the instructions in one iteration of the loop, and the ADAG of these instructions consists of two WCCs. The WCC on the left of figure 6.2(a) has 5 stages and 6 memory accesses, 3 of which are cold. The WCC on the right of figure 6.2(b) has 8 stages and 10 memory accesses, 6 of which are cold. With the tail width threshold\(^4\) set to 5, the bottom 7 stages make up the graph tail of the entire ADAG (which consists of both WCCs). In figure 6.2(b), the ADAG consists of the instructions in two iterations of the same loop. Increasing the subregion size results in the merging of all instructions, except for the two instances of the loop branch, into a single WCC. This WCC has 11 stages and 32 memory accesses, 18 of which are cold. The bottom 6 stages make up the graph tail. Figures 6.2(a) and 6.2(b) show that when the hot line threshold is 1, there are few cold WCCs in the non-tail in either case (0 in 6.2(a) and 1 in 6.2(b), with a tail width threshold of 5).

Figure 6.3(a) gives the offload efficiency of the trace of the loop for each combination of subregion size and hot line threshold, and figure 6.3(b) shows the estimated ‘speedup’ for each of these cases. The cases with hot line threshold set to 1 reflect that there are few cold WCCs in the non-tails, as the offload efficiency is low and there is negligible speedup. However, a hot line threshold of 0 forces all memory accesses cold, so we see significant offload efficiencies, with a significant peak speedup of 2.4 with subregion size set to 2. Although the offload efficiencies increase with subregion size, the speedup decreases as subregion size increases. This is because as subregion size increases, the increase in WCC sizes

\(^4\)Section 6.3.2 discusses how we chose the value for tail width threshold.
causes small threads to combine into fewer, larger threads. Although these large threads are wide, the P-core cannot exploit this ILP due to its simple design. The result is that the P-core serializes the large threads, and we find that performance is superior when many small threads execute across the P-cores.

6.3.2 Methodology

We ran the traces of one billion instructions each through a toolset which implements the ADAG analysis from the previous two sections. The tool scans the trace with region sizes of 16 basic blocks, and for each region, the tool constructs ADAGs with the subregion size varied from 1 to 16 by powers of 2. These experiments use a tail width threshold of 5 and a hot line threshold of 50. With regard to choosing these values, we have done some explorations in varying the hot line threshold, and the value used here produced positive results while also indicating opportunity for overlap between A-core execution and data transport\(^5\). For tail width threshold, we chose a value of 5 as an estimate to provide some parallelism for the A-core during tail execution, but not enough parallelism to warrant offloading to P-cores. We performed some sensitivity analysis for this value using one of the best-performing traces (\texttt{lmp.4B.1j.fix}) and one of worst (\texttt{179.art.1.1b}). For \texttt{179.art.1.1b}, we had to lower the tail width threshold to 2 before observing significant improvements in offload efficiency for small subregion sizes (<4 basic blocks). Although this value provided reasonable results for \texttt{lmp.4B.1j.fix}, we used the value of 5 to ensure tail content was not excessive across all traces in the final experiments. We leave a comprehensive study on the tradeoffs of these parameters for future work.

\(^5\)This overlap between A-core execution and data transport increases with hot line threshold.
6.3.3 Results

Figures 6.4–6.7 show the offload efficiencies and estimated speedups for the benchmarks from the integer- and floating point-intensive SPEC and Sandia application suites. Across all cases, we observed an average of 5.8% offload of the computations to the P-cores, with a minimum of 0% and a maximum of 24%. The average (geometric mean) of the estimated speedups is 4.0% across all traces, with a minimum of 0% and a maximum of 19%. The trend for the estimated speedup values generally follows that of the offload efficiencies. However, there is a large amount of variance across the suites and across subregion sizes. We can see that results for the floating point suites are significantly better than those for the integer suites. SPEC-Int and Sandia-Int (figures 6.4 and 6.6) showed maximum offloads of 11% and 8%, and maximum speedups of 7% and 8%, respectively. The average offload efficiencies for SPEC-Int and Sandia-Int were 2.8% and 1.7%, and the average speedups were 2.3% and 1.5%. For SPEC-FP (figure 6.5), the maximum offload was 24% with a 16% speedup, and for Sandia-FP (figure 6.7) the maximum offload was 20% with a maximum speedup of 19%. The averages for SPEC-FP were 8.4% offload and 5% speedup, while the averages for Sandia-FP were 7% for offload with 5.1% speedup.

There is a clear trend across the change in values of subregion size. For most traces, there is a peak for both offload efficiency and speedup at subregion size of 2 or 4, with 11 exceptions out of the 47 traces. For subregion sizes of 2 and 4, we observed average offloads of 16% and 12.6% and average speedups of 9.3% and 8.4%, respectively. For the remaining subregion sizes of 1, 8, and 16, the average offloads were lower: 4.1%, 2.1%, and 0.5%, with average speedups of 6.2%, 3.8%, and 0.5%, respectively.
Although P-core communication traffic and scheduling conflicts can be detrimental to performance, the primary reason for low performance is low offload. The offload efficiency values result from the primary components that define threads. These primary components are tail content, memory access content of WCCs, and results of the locality analysis. The A-cores always execute the ADAG tails, so a higher tail content will result in lower offload. The A-core also executes all WCCs without at least one memory access, so a larger number of these non-memory access WCCs will also result in lower offload. Finally, if the locality analysis results in a higher number of hot cache lines, this will result in less opportunity for thread content, although not necessarily lower offload.

An analysis of the results confirms that these trends follow the model. The
Figure 6.5: SPEC-FP: Offload Efficiency (top) and Estimated ‘Speedup’ (bottom)
average percentages of the traces that are tail content in SPEC-FP is 47%, while this percentage is 41% across the Sandia-FP suite. These numbers are noticeably higher for the integer suite, as SPEC-Int has an average of 53% tail content, and Sandia-Int has an average of 50% tail content. An implication of these numbers is that there is less application content available for offload, since the A-core always executes ADAG tails. Recall that the partitioner analyzes the remaining portion of the application, which makes up the non-tail, for locality to assign instructions to threads. The results show that the percentage of cold memory accesses are noticeably higher in the floating point benchmarks than in the integer benchmarks. In SPEC-FP, cold memory accesses make up 57% of all memory accesses and 23% of the instructions, and in Sandia-FP these percentages are 44% and 18%. In SPEC-Int, the numbers drop to 39% of accesses and 12% of instructions, and in
Figure 6.7: Sandia-FP: Offload Efficiency (top) and Estimated Speedup (bottom)
Sandia-Int they are 32% and 11%. Although lower percentages of cold memory accesses do not necessarily imply lower offload, this does suggest lower opportunity for offload.

The average tail content of the traces when subregion size is set to 2 or 4 is 50%, while this average drops to 44% when subregion size is set to 1, 8, or 16. While this conflicts with the conclusion above, we observed significantly fewer cold memory accesses with a subregion size of 2 or 4 as compared to a subregion size of 1, 8, or 16. For the former case, the percentage of cold memory accesses is 49% of memory accesses and 19% of instructions, while for the latter case these numbers are 43% and 17%.

6.4 Conclusions

This chapter has presented a memory access characterization model for generating locality-tuned traveling threads for the PAM architecture. After presenting methods for determining the content of locality-tuned traveling threads in applications, we described the implementation of these methods in a working toolset. The analysis of several application traces from standard and scientific workloads has shown that significant portions (up to 24%, and 5.8% on average) of application content are eligible for offload to the P-cores in the caches. Simulations offer estimates that these offloads may provide up to 19% speedup and 4.0% speedup on average. Note that these performance estimates are pessimistic in some respects. For example, these results do not take into account increased L1 cache hit ratios due to the moving of computations and corresponding memory accesses to the P-cores. Hence, there are promising opportunities for additional performance benefits which are avenues for future work.
CHAPTER 7

ENERGY EVALUATION

Although it has been a key consideration in portable computing, energy con-
sumption has emerged as a primary design constraint in high performance comput-
ing. Traveling threads and PAM offer potential reductions in energy consumption
while sustaining improvements in execution time. To incorporate this additional
constraint into the design, this chapter presents extensions to the architecture,
thread generation tools, and evaluation methodology. This chapter will meet the
following objectives:

- Enumerate the opportunities that PAM and traveling threads offer to reduce
  energy consumption arising from sources of both dynamic and static power
dissipation.

- Present a methodology and toolset for evaluating the PAM design across
  standard and scientific workloads in terms of both execution time and energy
  consumption.

- Employ the methodology to present quantitative comparisons of the PAM
  design to a conventional OoO uniprocessor system and a conventional, ho-
mogeneous multicore in terms of energy consumption.

7.1 Background

Here, we highlight two ways in which fine-grained migrant threads can reduce
the energy consumption associated with dynamic power dissipation. First is by
a reduction in clock frequency required to achieve a given IPC, which permits
reduction in supply voltage. Second is a reduction in switching activity due to
unnecessary data and code movement, which traveling threads mitigate in several
ways. First, they remove the requirements that the processor move every cache
line (both data and instruction) into the L1 and flush every dirty cache line from
L1. Second, for multicore processors, these threads avoid the overhead of pushing
and pulling data cache lines in and out of upper level caches to maintain coherence.
Third, these threads incur less traffic during migration relative to large threads.
Fourth, they allow computations to exploit locality at multiple levels of granularity
(one level at the P-cores, another at the A-cores). This may prevent false sharing
and wasted cache bandwidth due to excessive cache line size during phases of
program execution with low spatial locality\(^1\).

Migrant threads can also decrease energy consumption associated with static
power dissipation\(^2\) by exploiting code and data locality through flexible thread-
to-core mapping. This allows an increase in potential for circuit deactivation with
power gating through improved intra- and inter-thread locality. Improvements
in intra-thread locality will allow deactivation of core-to-core interconnect and
upper-level storage structures. If a thread can avoid data and code access to
remote memory structures, the processor can deactivate these structures for long
periods of time. Fine-grained threads provide opportunities – which would be
unavailable on a conventional architecture – to isolate a subset of the memory

\(^1\)For instance, in a phase of the program with high spatial locality, a large cache line size may
offer benefits. Assuming the architecture does not support a variable cache line size\(^\text{182}\) in
L1, the large cache line size may be detrimental to performance during another phase of the
program that exhibits poor spatial locality. However, since the L2 may easily have a cache line
size different from L1, P-cores naturally offer some benefits of variable cache line sizes.

\(^2\)These techniques will also provide further reductions in dynamic power dissipation.
accesses to a single P-core. Improvements in inter-thread locality will potentially reduce long-distance communications of data and code across the chip. When the thread generator can cluster computations and memory accesses to a set of threads that execute on spatially local P-cores, this will also allow the deactivation of remote cores and structures.

7.2 Methodology

We model the execution of applications by subregion with ADAGs, and employ graph analysis to quantify energy consumption and execution time. Hot/cold cache line analysis with WCC partitioning provides automatic thread extraction mechanisms. We also apply a manual thread partitioning approach to the DAXPY loop.

7.2.1 Analytical Model

The basis of the analytical model is the transformation of ADAGs to another graphical representation of an application which we term the event graph. The ADAG representation of a program is abstract in that the physical events associated with the nodes in the ADAG are unknown. For example, a memory access may hit in the cache, which will incur low time and energy costs. On the other hand, if the same memory access misses, it will incur high time and energy costs.

In contrast, the vertices in an event graph are event nodes. An event node represents the microarchitectural events occurring as a result of the execution of a node (representing an instruction) in the ADAG. Each event node has at least two values associated with it: an estimate of the duration of time required for that event to occur; and an estimate of the energy required for the event.
Figure 7.1 contrasts an ADAG with an event graph\textsuperscript{3}. In this contrast, we see differences akin to those between graphs in the intermediate representation (IR) in the optimization stage of a conventional compiler \textsuperscript{[5]} and the graphs in a runtime task scheduler in a modern embedded multiprocessor \textsuperscript{[193]}. In the former case, the graphs contain relatively little information regarding costs associated with the nodes in the dependence graphs. In the latter case, the graphs take into account details of the implementation (such as sizes of storage structures) to get more accurate time estimates to improve scheduling.

Edges between event nodes impose an ordering on the events, and the spatial layout of the nodes represents the scheduling, as in an ADAG. The purpose of the event graph is to provide overall time and energy estimates of executions of the application on different architectures. Two challenges with using the event graph to provide these estimates are: 1) estimating the run-time ordering of the events; and 2) transforming nodes in the ADAG to event nodes, which requires techniques for estimating the time and energy costs of the execution of instructions. Both the ordering and the costs will vary with architecture and static code analysis techniques. This work takes into account variability in both of these design dimensions and considers three architectures: a conventional single-core architecture supporting OoO execution; a conventional multi-core architecture with OoO cores, each with a private L1 cache; and the PAM architecture. The code analysis techniques include instruction scheduling and thread partitioning, and we build upon the techniques presented in previous chapters for these purposes.

We model the P-cores in the PAM as in-order cores\textsuperscript{4}. For in-order cores,\textsuperscript{3}

\textsuperscript{3}All examples in this chapter use sections of traces of the DAXPY loop from LINPACK (www.netlib.org/benchmark/hpl).

\textsuperscript{4}Here, an in-order core is defined as a core that executes the instructions in program order. The core cannot issue an instruction until the instruction preceding it retires.
Figure 7.1. Contrast of an ADAG with an event graph.

The figure highlights the differences between an ADAG and an event graph. The annotations in the ADAG include the internal instruction ID in the code analysis tools, the operation type, an address for memory accesses, and input and output register indices. The event graph annotations include microarchitectural events associated with each instruction. The values at the bottom of each node are the energy consumption of the event (in nJ) on the left and the latency of the event (in clock cycles) on the right. A T-Lookup indicates a thread performing the lookup of an address to its local memory, while a T-Access is a thread accessing its local cache bank.
TABLE 7.1

LATENCY AND ENERGY ESTIMATES OF EVENTS

<table>
<thead>
<tr>
<th>Component</th>
<th>Latency (cycles)</th>
<th>Energy (nJ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>12</td>
<td>1.4</td>
</tr>
<tr>
<td>Floating Point</td>
<td>37</td>
<td>8.4</td>
</tr>
<tr>
<td>L1 Lookup</td>
<td>3</td>
<td>0.0009</td>
</tr>
<tr>
<td>L1 Access</td>
<td>4</td>
<td>0.045</td>
</tr>
<tr>
<td>L2 Lookup</td>
<td>5</td>
<td>1.2</td>
</tr>
<tr>
<td>L2 Access</td>
<td>12</td>
<td>1.2</td>
</tr>
<tr>
<td>Thread Lookup</td>
<td>3</td>
<td>0.0009</td>
</tr>
<tr>
<td>Thread Access</td>
<td>4</td>
<td>0.045</td>
</tr>
<tr>
<td>Thread Initiation</td>
<td>17</td>
<td>1.2</td>
</tr>
<tr>
<td>Thread Migration</td>
<td>5</td>
<td>1.2</td>
</tr>
<tr>
<td>DRAM Access</td>
<td>196</td>
<td>14.2</td>
</tr>
</tbody>
</table>

The table shows the latency estimates for various events in the analytical model, assuming a 4-GHz clock (0.25 ns cycle time). We use the functional unit latencies from Azizi et al. [9], and derivations of the functional unit energy estimates use these latencies along with the power numbers from Kumar et al. [96]. The model uses the cache and DRAM latency and energy estimates from CACTI [171]. The L1 cache is a direct-mapped, single-bank 4KB cache with 64B line size using 65nm technology. The L2 cache is a 4-way, single-bank 2MB cache with 64B line size using 65nm technology. We estimate the latency and energy of various thread events using the costs of cache access events. For instance, the costs of a thread’s local address lookup and cache access are the same as that of the A-core’s L1.

determining the instruction ordering is straightforward – we use the PowerPC compiler’s instruction ordering and perform thread partitioning using the techniques from chapter 6. To estimate the time and energy costs of event nodes that represent non-memory accesses, the models use the latency and energy of associated functional units. For memory accesses, we perform cache simulation to determine which accesses are hits and which are misses to provide the estimates. Table 7.1 presents the costs of various events that occur in the simulations.

For OoO cores – which include the core in the conventional single-core architecture, the cores in the conventional multi-core architecture, and the A-cores in
the PAM architecture – scheduling and estimating time and energy costs are more complicated. The run-time ordering of events is dependent on many factors that are unknown without the use of simulation, such as structural hazards (full buffers, unavailable reservation stations and functional units, etc.) and branch prediction. Further, time and energy costs have greater variance with the compiler’s instruction scheduling and thread partitioning due to this run-time variability.

We address these difficulties in the analytical evaluation by making simplifying assumptions. We select a subregion size and then perform thread partitioning and event scheduling from the ASAP ordering of instructions. The selection of subregion size limits the number of instructions in-flight and the instructions which may be in flight at a given phase of program execution. These simplifications allow the event graph to more accurately model the out-of-order execution. Our time and energy estimates, then, do not take into account run-time variability such as structural hazards and branch misprediction.

Using these graphical models of in-order and OoO execution, we can perform comparisons of execution time and energy consumption of the architectural alternatives. Figure 7.2 shows the flow of the analysis for each of the architectural alternatives. The basis of the conventional architecture is the OoO model presented above, along with cache simulation to produce time and energy estimates. For PAM, we perform hot/cold cache line labeling in place of cache simulation, and perform an additional thread partitioning pass before simulation and event graph generation. Similarly, the conventional multicore uses the same thread partitioning to distribute computations across the OoO cores. Cache simulation for each core’s private L1 provides latency and energy estimates of memory accesses for event graph generation. The next section presents the details of the graph
It is important not only to understand why we use hot/cold cache line labeling for PAM evaluation, but also to make clear the distinction between line labeling and cache simulation. The objective in cache simulation is to determine which memory accesses are hits and which are misses. Cache hits and misses will depend on the pattern of past memory accesses and which lines map to which cache sets. A conventional cache hierarchy uses a cache insertion policy from which we can determine which accesses hit in the cache.

For PAM, a simple cache policy is too restrictive for the general case, since such a policy does not take into account the option of data processing in upper-level caches. Hence, more advanced cache policies will consider both data placement and computational partitioning among cores in tandem. We use line labeling for
performing data placement and computational partitioning.

Cache simulation can still play an important role in the evaluation of PAM. Note that the current model for line labeling does not take into account L1 size, so it is uncertain whether a hot line will be a hit or a miss. The following paragraphs present two approaches to translating memory accesses to cache hits and misses following line labeling.

The first option is to choose a cache size and configuration, and then perform a cache simulation. Cache simulation reports exactly which accesses are hits and misses, but requires additional time due to both simulation and the requirement of determining a good cache configuration.

The other approach is to use a cache configuration which will ensure that all accesses to hot lines – not including the first (“compulsory”) touch which causes the line to transition from cold to hot – will be hits. We can achieve this by using a fully associative cache with a size equal to the maximum number of lines that are hot at any point during the execution of an application. We can then estimate the access time of each memory access without the use of an additional cache simulation pass.

7.2.2 Instrumentation

There are two important metrics in the comparison of PAM event graphs to event graphs from conventional single-core and multicore architectures. The stage in figure 7.2 labeled “Critical Path Limits & Energy Gap” from the analysis provide these metrics. We calculate the critical path limit and energy gap values from the PAM event graph. From these values, we determine the potential for the PAM to outperform the conventional architectures. To explain these metrics
and their significance, we will first present the details of the instrumentation for generating and analyzing the event graphs for the conventional architectures.

Figure 7.3 illustrates the event graph generation for the conventional single-core architecture. We set the subregion size through a configuration parameter. The tools generate the ADAG from the PowerPC execution trace and an ASAP scheduling of the instructions. The event graph generator then inserts the time and energy costs of each non-memory access instruction. The generator estimates these costs as the execution of a single instruction on the functional unit corresponding to that instruction. A key insight in the comparison of event graphs is that the critical path through the graph represents the overall time costs, while the sum of the energy costs of all nodes gives the total energy costs. Hence, while the scheduler can mitigate time costs through parallel execution\(^5\), energy cost reductions occur only through elimination of events or replacement of high-energy events with low-energy events.

To form the event graphs for the conventional multicore and PAM, we first partition the graph into WCCs. For the conventional multi-core, an additional cache simulation pass determines the hits and misses of each core. From this, we generate the event graph for the conventional multicore, as figure 7.4 illustrates. The first stage forms the ADAG, as shown in figure 7.4(a). The symbols in this figure are the same as those in figure 7.3(a). In the second stage, the tools label the cache lines as hot and cold. From this labeling, the third stage (figure 7.4(c)) generates the hot/cold ADAG. In this figure, the white triangles represent memory accesses to cold cache lines, the black rectangles represent accesses to hot and warm cache lines, and the gray triangles represent all other instructions (such

\(^5\)The graph indicates parallel execution with nodes at the same stage in the scheduling.
The figure illustrates the stages of event graph generation for the conventional architecture. The first step is the formation of the ADAG, shown in figure 7.3(a). In this figure, black rectangles represent loads, white rectangles represent stores, gray diamonds are ALU instructions, gray ovals are FP instructions, and the gray pentagon is a branch. The second stage (figure 7.3(b)) is the cache simulation, which results in a list of events in the cache hierarchy associated with each memory access. The last stage (figure 7.3) uses the results from the first two stages to generate the event graph. The values at the top of the event nodes have the same meaning as those in figure 7.1. For each path \( p \) from a root node \( R \) to a leaf node \( L \), node \( R \) has a bracketed value that is the sum of the latency values of each node along \( p \).
Figure 7.4. Event graph generation for a conventional multi-core architecture.
as ALU and FP). Stage 4 (figure 7.4(d)) partitions the instructions between the cores. In these figures, the partitioner assigns instructions in black rectangles to core 1, and instructions in white ovals to core 0. The top section of figure 7.4(d) represents the non-tail portion of the ADAG, while the bottom represents the tail.

Stage 5 (figure 7.4(e)) performs the cache simulation for each core. Finally, stage 6 uses the information from the previous stages to generate the event graph. In this figure, the rectangles execute on core 1, and the ovals execute on core 0.

Figure 7.5 illustrates the event graph generation for PAM. Starting with the representation in figure 7.4(d), the thread generator partitions the ADAG between A-core threads and P-core threads, and assigns costs to the A-core’s memory access nodes. We will refer to the representation that results from these steps as the intermediate event graph. Now, the generator inserts thread initiation events into the A-core’s threads in the intermediate event graph, and inserts thread migration events into the P-cores’ threads. To determine the time and energy budget available to match or outperform the conventional architectures, we calculate two metrics from the PAM event graphs that we call the critical path limits and energy gap. To determine the critical path limits for a PAM event graph, we first calculate the execution time of the critical path of the corresponding event graph for a conventional architecture. The critical path limit of a path \( p \) through the event graph, then, is the difference between the critical path of the conventional event graph and the execution time of path \( p \) in the PAM intermediate event graph. The energy gap is the difference between the total energy consumption of the event graph of the conventional architecture and the total energy consumption of the PAM intermediate event graph. To match the execution time of the conventional architecture, we must ensure that the sum of the time costs of the new events
The first three stages in event graph generation for the PAM architecture closely resemble those for the conventional multicore in figures 7.4(a)–7.4(c). This figure highlights the key changes in the last three stages. In partitioning the non-tail section of the hot/cold ADAG (figure 7.5(a)), the partitioner performs the additional step of inserting thread initiators and thread migrations. The gray triangle is a thread initiator, and the gray diamond is a thread migration. Stage 5, shown in figure 7.5(c), includes the simulation of events supporting thread execution on P-cores, including thread initiation and migration, and also P-core memory accesses (T-Access in the figure) and address lookups (T-Lookup). Figure 7.5(c) shows the PAM event graph for the non-tail, which includes additional nodes for the new P-core events.
along each path of the intermediate event graph does not exceed the critical path limit of that path. Similarly, to match the energy consumption of the conventional architecture, we must ensure that the sum of the energy costs of the new events does not exceed that of the energy gap.

For the analytical evaluation in this work, we consider one assignment of thread initiation events and thread migration events to the intermediate event graph. One thread initiation event occurs at the beginning of each subregion to initiate all P-core threads in that subregion. Thread migration events occur whenever a P-core executes a memory access that is not local to the P-core on which the thread is executing. Note that thread migration event assignment is dependent on the microarchitectural design parameter of memory per P-core. The costs of thread migration events, like all events, are also dependent on microarchitectural design parameters. For thread migration events, significant microarchitectural design variables include access time of associated memory structures and interconnect latency.

7.2.3 Manual Partitioning

In addition to the automatic thread partitioning using hot/cold line labeling, we have applied a manual partitioning of traveling threads to the DAXPY loop (see section 6.3.1) in the following form:

\[ Y[j, k] = \alpha \ast Y[i, k] + Y[j, k]; j, k < i \] (7.1)

In one approach to parallelizing DAXPY on a conventional multiprocessor system [91], each processor owns a QxQ block of the matrix Y, where Q is much larger than the cache line size, C. The loop visits each element in the diagonal of the
matrix. The row containing the current element (element $Y[i,i]$) in the diagonal is the pivot row. For each element $Y[i,k]$ in row $i$ to the left of the diagonal, the loop scales $Y[i,k]$ by $\alpha$ and adds the result to every element below it in column $k$.

We manually partition the computations in DAXPY for PAM such that the A-core is responsible only for orchestrating the execution of threads. The P-cores, then, are responsible for updating the matrix rows. We modified the implementation of DAXPY described in [91] for execution on PAM as follows:

1. The nodes perform the reads of $\alpha$s from remote memory. The L2 controller distributes the $\alpha$s to the P-cores.

2. The nodes perform the reads of C of Y’s elements from the first row in the block. The cache controller distributes these rows to the P-cores.

3. The node performs reads of C elements for each row from the block in local memory. The L2 receives copies of the elements, but the L1 does not.

4. The host core initiates a thread in the P-core containing the first element. The thread adds the scaled version of the first row to each line it owns. The core leaves the updated lines in its local cache.

5. The thread migrates to the P-core holding the next elements.

6. Steps 3–5 are repeated for each P-core holding elements for the current iteration.

7. As the node brings in new lines on each row, the P-cores flush the updated lines back to local memory.

This translates into a loop of the following steps, which figure 7.6 illustrates:

1. Read in C $\alpha$s from remote memory, one at a time.

2. Distribute the $\alpha$s to the P-cores.

3. Read in C of Y’s elements from remote memory.

4. Distribute a copy of these to each P-core. These copies will stay at the P-cores for a long time for reuse.
5. Read in Q lines of C elements of Y from local memory. Only one copy of each line exists at the node: one copy for the P-core assigned to the L2 bank holding the line.

6. The host core sends out a thread initiator to begin thread execution on the P-core with the first element.

7. The core executes a BLITS for each line, which reads and writes each element of the line.

8. The thread migrates to the next P-core. Steps 7 and 8 are repeated for each P-core holding rows for the current iteration.

9. As new lines are brought into L2, flush the updated lines out to local memory.

7.3 Results

This section presents the results of the evaluations from the experiments that use the methodology from the previous sections. The first part gives the results
from the SPEC and Sandia suites using the automatic thread partitioning tools. In
the second part, we discuss the results from applying manual thread partitioning
to the DAXPY loop.

7.3.1 SPEC and Sandia Suites

Figures 7.7–7.10 show the energy consumption results from the SPEC and
Sandia applications. These figures compare the energy consumption results for a
homogeneous multicore processor and the PAM processor to a conventional, single-
core processor. We assume the core in the conventional single-core is equivalent
in performance to the cores in the homogeneous multicore as well as the A-core in
the PAM. The homogeneous multicore is a dual-core processor, where each core
has a private L1 cache and shares a single L2 cache.

The figures give two bars for each application trace of 10,000,000 instructions.
The left bar shows the energy consumption from on-chip data transport and on-
chip cache accesses for the homogeneous multicore normalized to the energy con-
sumption of the conventional single-core. This “normalized energy consumption”
is the energy consumption of the homogeneous multicore divided by the energy
consumption observed for the same trace for the conventional single-core. Hence,
a lower value indicates less energy consumption for the homogeneous multicore.
Likewise, the right bar for each trace shows the normalized on-chip energy con-
sumption from data transport and cache access for the PAM. Note that we report
energy consumption for on-chip data transport and cache access because these are
the components that we have focused on reducing in the current PAM design.

The first noticeable trend in the charts is that the energy consumption of the
homogeneous multicore is quite high for most cases. This is primarily a result
On the x-axis, we show the varying of subregion size (1, 2, and 4 basic blocks) for each application trace.

Figure 7.7. Energy Results for SPEC-Int

Figure 7.8. Energy Results for SPEC-FP
Figure 7.9. Energy Results for Sandia-Int

Figure 7.10. Energy Results for Sandia-FP
of our assigning each individual core an L1 cache that is half the size of the conventional single-core’s L1. The halving of the cache size was done for the sake of fairness, as it results in the homogeneous multicore having the same total L1 size (for the two cores combined) as the conventional single-core and the PAM.

Unfortunately, poor load balancing between the cores results in a high L1 cache miss rate with the smaller L1 cache. We observed poor load balancing because we partition the instructions between the cores using the PAM thread partitioner. All of the threads that would execute on the passive cores execute on one of the cores, while the threads that would execute on the active cores execute on the other. With the current partitioning mechanisms, most of the instructions execute on the A-core. This leads to the load balancing problem for the homogeneous multicore, which in turn leads to poor utilization of one of the L1 caches. Another reason for increased cache traffic is that we are using a simple MESI protocol which results in high energy overheads to maintain coherence between the cores. The results here highlight some of the problems that arise with energy consumption when using this type of multicore configuration.

The results show that the energy consumption of PAM is generally far less than that of the homogeneous multicore. The primary reason for this is that the PAM avoids the traffic overheads of the homogeneous multicore from cache coherency and load imbalance. While the homogeneous processor has two cores with separate L1 caches that compete for data, the thread partitioner divides computations and data intelligently among the A-core and P-cores for PAM. This results in significant energy savings: the average (geometric mean) energy consumption of PAM across all cases is 50.2% less than the average for the homogeneous multicore.

For most traces, there was little change in the PAM’s energy consumption rela-
tive to the conventional single-core. We’ll first discuss the reasons for the improvements observed in the good cases and then address the outlier, \texttt{171.swim.1.1b} with subregion size set to 1. A discussion of the reasons that most cases show only a small change in energy consumption will follow.

There were eight cases that showed significant (more than 5%) improvement over the conventional single-core, and the top three performers – \texttt{164.gzip.4.1b} with subregion size of 1 (11% improvement), \texttt{164.gzip.1.1b} with subregion size of 1 (7.3% improvement), and \texttt{176.gcc.3.1b} with subregion size of 1 (6.5% improvement) – were all from the SPEC-Int suite. We broke the on-chip energy consumption down into seven components: lookups and accesses to the L1 and L2, traveling thread initiations, thread migrations, thread cache accesses, and thread cache lookups. The conventional case did not run traveling threads at all, so all of the energy consumption due to traveling thread management is overhead for PAM. However, the PAM can potentially reduce all of the other energy components. We observed that the primary source of improvements were from reduced number of L2 accesses, although there were some small improvements due to L1 accesses and L2 lookups. For the cases that showed the most improvement, the decrease in L2 access energy were large enough to offset the thread overheads and provide additional improvements. This points to the conclusion that we have reached one goal of traveling threads: to exploit locality in a more intelligent manner. Rather than bringing cache lines all the way down to L1, we leave them out in L2 and exploit the locality with the thread. The thread’s local accesses to L2 cache banks are cheaper than the A-core’s “global” accesses to the entire L2. Hence, when the thread partitioner can convert a significant number of the L2 accesses to local thread accesses, we observe significant improvements.
However, in some cases, we observed a break-even between thread overheads and L2 accesses, while in other cases, the thread overhead costs exceeded the improvements. There are some scenarios in which the traveling threads can have very bad effects on performance. For example, if the A-core initiates a thread for a cache line that will be brought into the L1 cache soon anyway (before the cache line in its slot is touched), then we are effectively going out to the L2 an extra time (once for the thread initiator, and once for the L2 access that we would’ve done anyway). Such cases can cause high energy consumption overhead for PAM. A point of future work is to incorporate mechanisms into the thread partitioner to detect these cases.

One interesting outlier occurred with 171.swim.1.1b. With subregion sizes of 2 and 4, this trace showed improvements of 2.7% and 3.4%, respectively, while the same trace showed a 28% increase in energy consumption with subregion size set to 1. The results show that for this trace, with subregion size set to 1, there are many more threads of much smaller size than for the other two subregion sizes. With a subregion size of 1, we observed that there were 2,082,134 threads with an average size of 4.442 instructions per thread. With subregion sizes of 2 and 4, the numbers of threads were only 118,416 and 56,950, respectively, and the average thread sizes increased to 81.22 and 166.7 instructions per thread. A result of the large number of threads in the case where subregion size is set to 1 is that the overhead for thread initiation is relatively high, while most of the other metrics are about the same. The smaller threads do lower the overhead due to migration, but not enough to compensate for the initiation overhead.

For the cases where PAM shows small improvements relative to the single-core, we have found that the primary reason for this is that there is a low number
of memory accesses to cold cache lines that fall into cold WCCs. Although the number of accesses to cold cache lines is fairly significant, many of these accesses are within hot WCCs – note that these cold cache lines convert to warm cache lines in this case.

Table 7.2 gives various memory access statistics for all the traces considered. The left-most column shows the trace name, followed by the number of memory accesses in the trace. Each of the next three pairs of columns show the number of memory accesses to non-hot (either cold or warm) cache lines on the left, and the fraction of these non-hot accesses that are warm on the right. The three pairs of columns give the metrics for the subregion sizes of 1, 2, and 4 from left to right.

There are two important points to make before analyzing these results. First, a significant amount of thread coverage is necessary for there to be significant change between the PAM and the conventional single-core. In the extreme case, with no thread coverage, the metrics for the PAM will be the same as the conventional single-core. Second, a large number memory accesses to cache lines for threads to work on is necessary to generate a significant amount of thread content. The current design allocates only cold cache lines for threads. Hence, we must see a non-negligible number of accesses to cold cache lines to have the potential for noticeable change in processor behavior.

A conclusion from this line of reasoning is that we should be able to see two correlations between the values in table 7.2 and the results in figures 7.7–7.10. The first is that a low number of non-hot accesses for a trace should result in negligible change in PAM energy consumption. The second is that a high fraction of warm accesses should also result in negligible change.

6Note that in table 7.2 we abbreviate “Accesses” as “Accs.”
### TABLE 7.2

MEMORY ACCESS STATISTICS

<table>
<thead>
<tr>
<th>Application</th>
<th>Spec-Int</th>
<th>Spec-FP</th>
<th>Sandia-Int</th>
<th>Sandia-FP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># of Memory Acces</td>
<td># of Subregion Size</td>
<td># of Subregion Accesses</td>
<td># of Subregion Accesses</td>
</tr>
<tr>
<td>164.gzip.1.1b</td>
<td>3460715</td>
<td>98143</td>
<td>1376</td>
<td>93767</td>
</tr>
<tr>
<td>164.gzip.2.1b</td>
<td>2891446</td>
<td>12332</td>
<td>.5035</td>
<td>9602</td>
</tr>
<tr>
<td>164.gzip.3.1b</td>
<td>3329871</td>
<td>84151</td>
<td>.1525</td>
<td>77939</td>
</tr>
<tr>
<td>164.gzip.4.1b</td>
<td>3493162</td>
<td>113312</td>
<td>.0708</td>
<td>109352</td>
</tr>
<tr>
<td>164.gzip.5.1b</td>
<td>3293008</td>
<td>66689</td>
<td>.1729</td>
<td>63104</td>
</tr>
<tr>
<td>175.vpr.1.1b</td>
<td>4561879</td>
<td>135586</td>
<td>.9643</td>
<td>129222</td>
</tr>
<tr>
<td>176.gcc.1.1b</td>
<td>143054</td>
<td>414</td>
<td>.3238</td>
<td>406</td>
</tr>
<tr>
<td>176.gcc.3.1b</td>
<td>3166398</td>
<td>161372</td>
<td>.5307</td>
<td>156720</td>
</tr>
<tr>
<td>168.wupwise.1.1b</td>
<td>4085706</td>
<td>4214</td>
<td>.6665</td>
<td>4214</td>
</tr>
<tr>
<td>171.swim.1.1b</td>
<td>2784276</td>
<td>277824</td>
<td>0</td>
<td>277824</td>
</tr>
<tr>
<td>172.mgrid.1.1b</td>
<td>4101387</td>
<td>4031</td>
<td>.9555</td>
<td>3991</td>
</tr>
<tr>
<td>173.apphu.1.1b</td>
<td>4593453</td>
<td>2996</td>
<td>.9989</td>
<td>1977</td>
</tr>
<tr>
<td>177.mesa.1.1b</td>
<td>3169515</td>
<td>93256</td>
<td>.4997</td>
<td>91009</td>
</tr>
<tr>
<td>178.galgel.1.1b</td>
<td>4798691</td>
<td>847229</td>
<td>.7331</td>
<td>781885</td>
</tr>
<tr>
<td>179.art.1.1b</td>
<td>4444055</td>
<td>538584</td>
<td>.9651</td>
<td>536804</td>
</tr>
<tr>
<td>179.art.2.1b</td>
<td>4433324</td>
<td>547934</td>
<td>.9999</td>
<td>546059</td>
</tr>
<tr>
<td>183.equake.1.1b</td>
<td>3358188</td>
<td>16714</td>
<td>.9228</td>
<td>16067</td>
</tr>
<tr>
<td>187.facerec.1.1b</td>
<td>4148658</td>
<td>136143</td>
<td>.9023</td>
<td>136143</td>
</tr>
<tr>
<td>189.lucas.1.1b</td>
<td>2307548</td>
<td>45669</td>
<td>1.0000</td>
<td>45669</td>
</tr>
<tr>
<td>191.mfmd.1.1b</td>
<td>6077881</td>
<td>262</td>
<td>.4656</td>
<td>257</td>
</tr>
<tr>
<td>200.sixtrack.1.1b</td>
<td>4034963</td>
<td>494724</td>
<td>.9281</td>
<td>494709</td>
</tr>
<tr>
<td>301.apsi.1.1b</td>
<td>4103597</td>
<td>327916</td>
<td>.9347</td>
<td>326185</td>
</tr>
<tr>
<td>blast.4B</td>
<td>2563938</td>
<td>132970</td>
<td>.8657</td>
<td>131502</td>
</tr>
<tr>
<td>chaco.4B</td>
<td>3612290</td>
<td>917</td>
<td>.8462</td>
<td>911</td>
</tr>
<tr>
<td>df.4B</td>
<td>4773942</td>
<td>2</td>
<td>1.0000</td>
<td>2</td>
</tr>
<tr>
<td>iso.4B</td>
<td>6437127</td>
<td>351856</td>
<td>1.0000</td>
<td>351856</td>
</tr>
<tr>
<td>kmetis.4B</td>
<td>3762583</td>
<td>39021</td>
<td>.6547</td>
<td>37923</td>
</tr>
<tr>
<td>sp.4B</td>
<td>4761636</td>
<td>357120</td>
<td>1.0000</td>
<td>357120</td>
</tr>
<tr>
<td>zchaff.4B</td>
<td>2984256</td>
<td>91818</td>
<td>.3836</td>
<td>90793</td>
</tr>
<tr>
<td>alegra.4B.hemipen</td>
<td>4964826</td>
<td>237263</td>
<td>.8296</td>
<td>214222</td>
</tr>
<tr>
<td>alegra.4B.WES04</td>
<td>405983</td>
<td>99758</td>
<td>.8607</td>
<td>90630</td>
</tr>
<tr>
<td>cth.4B.2gas</td>
<td>4193428</td>
<td>160080</td>
<td>.9968</td>
<td>167631</td>
</tr>
<tr>
<td>cth.4B.amr2d-cust4</td>
<td>3572476</td>
<td>33334</td>
<td>.5802</td>
<td>32949</td>
</tr>
<tr>
<td>cth.4B.epf</td>
<td>4128540</td>
<td>13735</td>
<td>.8900</td>
<td>12518</td>
</tr>
<tr>
<td>cth.4B.sc</td>
<td>4248885</td>
<td>4746</td>
<td>.9346</td>
<td>4668</td>
</tr>
<tr>
<td>cube3.4B.crs</td>
<td>3732491</td>
<td>117692</td>
<td>.2812</td>
<td>115240</td>
</tr>
<tr>
<td>cube3.4B.vbr</td>
<td>3296696</td>
<td>4518</td>
<td>.0066</td>
<td>4515</td>
</tr>
<tr>
<td>its.4B.cad.brems</td>
<td>3088419</td>
<td>46826</td>
<td>.4781</td>
<td>46395</td>
</tr>
<tr>
<td>its.4B.cg.brems</td>
<td>2814577</td>
<td>50504</td>
<td>.2524</td>
<td>49859</td>
</tr>
<tr>
<td>lmp.4B.flow.langevin</td>
<td>3778018</td>
<td>37000</td>
<td>.4046</td>
<td>36630</td>
</tr>
<tr>
<td>lmp.4B.lj.fix</td>
<td>3512154</td>
<td>33852</td>
<td>.1784</td>
<td>33651</td>
</tr>
<tr>
<td>lmp.4B.poly.chain.a</td>
<td>3200017</td>
<td>54193</td>
<td>.1805</td>
<td>53238</td>
</tr>
<tr>
<td>mpsalsa.4B.airport3d_3</td>
<td>3698613</td>
<td>79893</td>
<td>.4833</td>
<td>76408</td>
</tr>
<tr>
<td>mpsalsa.4B.thermal_convet_square</td>
<td>3118843</td>
<td>27330</td>
<td>.7367</td>
<td>27126</td>
</tr>
<tr>
<td>s ppm.2B</td>
<td>2851692</td>
<td>2346</td>
<td>.1069</td>
<td>2346</td>
</tr>
<tr>
<td>xyce.4B.invertgen75</td>
<td>3877672</td>
<td>107193</td>
<td>.7527</td>
<td>99090</td>
</tr>
<tr>
<td>xyce.4B.invertgen_v3_75</td>
<td>3913209</td>
<td>104522</td>
<td>.7972</td>
<td>96041</td>
</tr>
</tbody>
</table>
For the first conclusion, consider the three traces with the lowest numbers of non-hot memory accesses – dfs.4B (average of 2 non-hot memory accesses across the subregion sizes), 191.fma3d.1.1b (average of 255.3), and chaco.4B (average of 906.7). Note that all of these cases support the conclusion, as the change for each of these cases is < .01%. The most notable of these points is that of 191.fma3d.1.1b, as this is the only case of the three that has a fraction of warm accesses less than 1. This case is notable because it satisfies the condition for the first correlation listed above, but not the second. For example, with subregion size of 1, 46.6% of non-hot accesses in this trace were warm, but only 262 non-hot accesses overall. The change in energy consumption for this case was only -.0033%.

There are several cases for which all non-hot memory accesses are warm. Among these cases, the one with the highest number of non-hot memory accesses is sp.4B (357,120 accesses), which showed no change in energy consumption and hence supports the conclusion.

To support the contrapositive (that a noticeable change in energy consumption requires a non-negligible number of cold accesses), consider the top three performers mentioned above – 164.gzip.4.1b with subregion size of 1, 164.gzip.1.1b with subregion size of 1, and 176.gcc.3.1b with subregion size of 1. Note that the number of cold accesses is the product of the number of non-hot accesses and the fraction of cold accesses (one minus the fraction of warm accesses). The numbers of cold accesses for these cases were 105,290, 84,639, and 75,732 accesses, respectively.
7.3.2 DAXPY Results

For the DAXPY loop in LINPACK, PAM offers opportunities for reduction in the energy consumption from on-chip data transport, tag accesses, and SRAM accesses. The node need not drag all data down to L1 and into the host’s register file. Since the P-cores operate on cache lines directly in L2, the node leaves some cache lines there.

The results show up to a 23% reduction in total energy consumption. The improvement increases with number of P-cores, but the improvement per core decreases also. That is, from 1 to 4 P-cores, we observed a 20% change, while from 4 to 32, the change was only 7%. The accesses to L2 data contribute 9.2% of the total, the on-chip data transport (OCT) contributes to 14.3% of the total, and the tag lookups contribute only 0.3%. Hence, for the single P-core configuration, that the tag lookup energy increases by more than a factor of 5 is insignificant. The factor of two increase in data access is offset by the decrease in transport energy. For this case, the P-cores increase overall energy consumption by 4.2%.

We can see that the improvement in on-chip transport increases slightly from 2
to 4 and 8 cores, but decreases again from 8 to 16 and 32 cores. Note that although an increase in number of P-cores decreases the energy cost per migration, the number of migrations increases with number of P-cores. Hence, there is a non-trivial tradeoff. However, the change in improvement due to transport across configurations is slight, and the major contributor in the reduction of costs is L2 data accesses. The change for this component varies from a 100% increase for 1 core to a 43% reduction with 32 cores. The key conclusion here is that the threads exploit a data locality that is intrinsic to \texttt{DAXPY}. While the conventional design also exploits this locality, it must perform the expensive data movements into and back out of L1 to do so. Although the PAM architecture must send out thread initiators and perform migrations, we can see that these overheads are offset by a reduction in cache data traffic. The improvement offered by PAM increases with number of cores, as the cost per access decreases with memory per core. As the results show, the improvement in per access cost is not offset by increased migration overheads associated with increased number of cores.

7.4 Conclusions

This chapter has presented models, methodology, and experimental results for comparing the energy consumption of a conventional uniprocessor, a homogeneous multicore, and PAM. The results of automatic thread partitioning with the SPEC and Sandia suites showed up to 11% decrease in energy consumption from on-chip data transport and cache access relative to the conventional uniprocessor. The cases that showed the most improvement were from the SPEC-Int suite and occurred with the subregion size set to 1. The results support our hypothesis that a non-negligible number of cold accesses is necessary for significant change in
energy consumption. In the case with the most improvement, 113,312 instructions from the trace of 10,000,000 instructions were memory accesses to cold lines.

We observed a significant number of accesses to non-hot lines in many cases, indicating high potential for thread coverage. However, many of these accesses fall within hot WCCs, which causes the lines to switch from cold to warm. The current thread partitioner design has no special mechanisms for handling warm lines: we treat them as hot lines. This indicates that in future designs, we will either need mechanisms for composing threads from accesses to warm lines, or we need to circumvent the frequent occurrence of warm lines. One option for pursuing the first route is to design a second phase of WCC partitioning that generates threads from hot WCCs that touch warm cache lines.

One option for taking the second route is to vary the thread partitioning parameters (hot line threshold and tail width threshold) to reduce the number of warm lines. We have already completed some experimentation with this possibility, and an important point to realize is that the roots of the problem are the limits on locality and cache technology. Circumventing the labeling of lines as warm does not address the essence of this problem. However, if we were to consider a case in which cache misses were much more expensive, we believe this would justify higher hot line thresholds, leading to larger numbers of cold lines, which would lead to higher thread coverage. A configuration that may correspond to this case in reality would be to embed the P-cores at higher levels in the cache hierarchy – such as L3 and at the DRAM controller – where misses are much more expensive. Another option is to consider a conventional processor with an L2 that is significantly smaller than the PAM’s L2. There are interesting tradeoffs here between hit rates and latencies for consideration in future work.
With manual thread partitioning on the DAXPY loop, we were able to tune the threads more accurately to better exploit locality. That is to say, while the automatic partitioner will sometimes label lines as cold that may increase energy consumption, manual thread partitioning allows us to explicitly identify which lines will be good for threads to work on. This explicit identification leads to low overhead in thread initiation and migration relative to the automatic thread partitioner along with high levels of parallelism and locality exploitation at the P-cores. Using this approach, we observed significant improvements in energy consumption in the on-chip cache hierarchy from 1 to 4 cores, but saw a plateau from 4 to 32 cores. The improvements were as high as 23% – much higher than for the automatic thread partitioner. However, the primary drawback to manual thread partitioning is drastically higher code development time. While the automatic thread partitioner ran through all suites with a variety of code patterns, coding and optimizing the single loop of DAXPY for PAM by hand took more than a day of programming for the author.
CHAPTER 8

CONCLUSIONS

This chapter first offers a summary of the work and a presentation of the final PAM design. The second section outlines the key results, and the chapter closes with a presentation of future work with traveling thread architectures.

8.1 Summary of Work

The overall objective of this work has been to offer a new alternative to addressing the memory wall. Our work has in fact found a new approach by moving computations into the memory hierarchy and using a combination of compiler and run-time code migration to utilize the new capabilities. The optimized traveling thread architecture shows up to 19% speedup and 23% reduction in energy consumption relative to a conventional uniprocessor, while also making L1 caches more effective. Further, we have shown that this is applicable across a wide range of benchmarks.

Figure 8.1 shows the final PAM design. In the center of the figure is a chip-level view of the processor, while the bottom section highlights features of the A-core and the top section shows the details of the P-core. New features of the P-core include microarchitectural components for improving speculation performance along with issue logic for efficiently handling control vectors, while the new
features of the A-core include logic for detecting graph tails to improve thread partitioning at run-time.

The compiler generates an initial partitioning of code between the A-core and P-cores, along with corresponding control vectors. The control vectors are the logical interface between the A-core and the P-cores. The A-core sends these control vectors to the P-cores, which use the vectors during instruction issue to select code for a thread from an instruction cache line.

On the A-core, the branch-critical path detector and graph tail detector track instruction dependencies in the OoO hardware and use this information – along with performance statistics from the L1 cache – to modify the control vectors in the control vector cache. These control vector updates result in run-time optimization of the thread partitions. A multilevel branch predictor guides speculation, and prior to P-core thread initiation, the predictor moves control vectors from the control vector cache to the control vector buffer. We design the A-core and P-cores for subregion sizes of 4 basic blocks per thread initiator, based on results in chapter 6 showing superior performance at this subregion size. This size corresponds to 16 64-bit instruction cache lines. The thread initiator buffer takes these control vectors and instruction cache line addresses along with register value sets from the register files to form thread initiators.

We embed 8 P-cores into the L2, with 2 MB of local cache each, using the positive results from chapter 5 with these design parameter values. Additionally, results in chapter 7 showed a good balance between number of P-cores and energy consumption with these values. Each P-core has a buffer for storing pairs of

\footnote{We’ve observed that basic blocks touch 2.6 instruction cache lines on average across all traces. We estimate, then, that a subregion of size 4 will touch about 10 instruction cache lines, and round this up to the next power of 2.}
Figure 8.1. Block Diagram of PAM Design.
control vectors. Again, we use the subregion size of 4 to set the size of this buffer, dividing the 16 control vectors from the A-core evenly among the P-cores. The vectors from the buffer feed into the issue logic, which route instructions to the 4 functional units aligned to the cache macro. We supply 2 FUs of each type (integer and floating point) as the results in chapter 5 indicate an average mIPM of 2. Two floating point units also provide sufficient width for exploiting the parallelism available in most floating point subgraphs, using the observations in chapter 3.

The functional units feed speculative state to a storage opposite the local cache. Logic for verifying correctness of speculation takes the correct branch history from the A-core and commits or discards this state based on a comparison of branch predictions to history.

It is of interest here to estimate the area remaining on a typical chip given such a design. To provide this estimate, we use a chip area budget of 210 mm$^2$, a NoC overhead factor (1.37), and per tile area estimates based on results from Balfour and Dally’s designs in 65 nm technology [11]. CACTI provides the cache area estimates for the P-cores [171], and we use the PPE from the Cell B.E. implemented in 65 nm [167] as the area of the A-core. From this, we calculate an area of 91 mm$^2$ for the A-core, P-cores, and lower level caches. Assuming we use the remainder of the chip for cache, there is room for a 62 MB L3, using the L3 cache density from the Power7 processor, which consists of embedded DRAM at 0.24 mm$^2$ per 1 Mbit macro [14]. Alternatively, we can place an additional A-core/P-core processing unit on the chip for executing multiprogrammed workloads. This option provides space for a 15 MB L3.

To compare this to a state-of-the-art processor, consider IBM’s Power7 [192],
which features 8 cores, each with a 32 KB L1 I-cache, 32 KB L1 D-cache, and a 256 KB L2. The cores share 32 MB of eDRAM in L3. The estimates of the contents of the PAM chip appear quite optimistic in comparison, as the Power7 is in 45 nm with a die size of 567 mm$^2$. However, each core on the Power7 supports up to 4 simultaneous threads with 8-way instruction fetching and 10 functional units, making them much more heavily-equipped than the P-cores. It is feasible that the design complexity of a single Power7 core may be comparable to that of the A-core. But even taking this into consideration, the proposed PAM chip may not be feasible in 65 nm with the stated L3 cache sizes. Simple ways to remedy this are to lower L3 cache size or use fewer P-cores. Our results have shown only small differences in performance between 4 and 8 P-cores, but a more extensive study on the tradeoffs in P-core count and L3 is an interesting avenue for future work.

8.2 Key Results

In-Cache Computations (ICCs) have the potential to significantly decrease L1 cache misses at the host processor – up to 61% when used with a column-associative L1 cache. This leads to potential speedups of 1.32 on average due to improved cache performance. This speedup estimate is pessimistic in that it does not include improvements from increased parallelism offered by ICCs.

Our results have confirmed that the problems of ADAG tails, memory-bounded IPC, and branch-critical paths are present in most of the application traces we considered. We tailor the PAM design to address these problems. In the break-even performance planes between PAM and a conventional processor, we observed a “sweet spot” at 4–8 P-cores with a 4–8 MB L2 cache (1–2 MB per P-core).
In other results, we observed average thread sizes of 7.7 instructions, so we conclude that each thread, on average, would be allowed 1.9–3.9 migrations on average to match the performance of the conventional processor.

Applying WCC-based thread extraction with hot/cold cache line locality analysis, simulation results show offload efficiencies of up to 24% (5.8% on average) and performance improvements of up to 19% (4.0% on average). PAM’s performance results for floating point-intensive applications were superior to integer-intensive applications for both the SPEC and Sandia suites. While SPEC-Int and Sandia-Int showed average speedups of 2.3% and 1.5%, respectively, SPEC-FP and Sandia-FP showed average speedups of 5% and 5.1%. Further, performance improvements were noticeably higher for subregion sizes of 2 and 4 (9.3% and 8.4%, respectively) than for the subregion sizes of 1, 8, and 16 (6.2%, 3.8%, and 0.5%, respectively).

The thread partitioning algorithm for PAM requires PAM’s capabilities, since if applied to a conventional, homogeneous multicore by itself, it results in significant overheads in traffic due to low cache utilization stemming from poor load balancing. PAM’s average energy consumption running the same set of threads was 50.2% less than the average for the homogeneous multicore. We reported up to 11% reduction in energy consumption relative to a conventional uniprocessor.

For the manual thread partitioning applied to DAXPY, we observed up to 23% reduction in on-chip energy consumption relative to a conventional processor. The improvements increased significantly from 1 to 4 cores, but increased marginally from 4 to 32 cores.
8.3 Future Work

The following are open questions for future research efforts:

- What are policies for sharing caches efficiently between A-cores and P-cores?
- Will conventional compiler optimizations (e.g., live register analysis) offer significant reductions in A-core/P-core traffic?
- Are there applications for which branch-critical ICCs (BCIs) are significantly detrimental to performance? If so, what are mechanisms for converting BCIs to executable ICCs?
- Are there opportunities for scheduling cache line migrations between L1 and L2 to reduce latency costs?
- What are efficient mechanisms for providing virtual address translation at the P-cores? Are conventional translation mechanisms suitable for the A-core?
- Do large variations in the tail width threshold and hot line threshold have significant effects on the design and its performance?
- How can we modify the design to decrease the number of warm lines? Is moving P-cores to the L3 effective for increasing hot line threshold?
- Are there benefits to embedding P-cores in the L3 or main memory? If so, are the benefits additive with P-cores in L2?
- Are the benefits of advanced architectural features\(^2\) additive with PAM?

A primary objective of future work should be the design of a next generation PAM system, which we will refer to as PAM-1 and will draw from and improve upon this dissertation work.

One point of consideration for PAM-1 is whether the traditional primitives of loads and stores for memory access are suitable. With scratchpad memories and

\(^2\)Examples are scratchpad memories\(^\text{[109]}\), cache bypassing\(^\text{[82, 86]}\), advanced cache insertion policies\(^\text{[139]}\), variable line sizes\(^\text{[132]}\), new cache controller designs (e.g., programmable decoders\(^\text{[199]}\), and lock-up free caches (especially MSHR-based cache architectures\(^\text{[174]}\)).
migrant computations, do we really need to spend a significant amount of computational resources calculating virtual addresses, translating them, and moving replicated data across the chip? The answers seem to be no, because the physical location of the computation itself is spatial information. A core should never, in the ideal case, attempt to access a piece of data that may be on the other side of the chip, so the conventional load-store model does not seem to be appropriate.

Our studies with speculation have raised additional questions of interest for PAM-1. Further reflection on the nature of speculation reveals that there are fundamental differences that are of highest importance between how a machine should handle a computation that is not speculative and one that is – whether we have 99% or 1% certainty that its execution is on the correct path. Should we be using the increasing amounts of on-chip memory as traditional caches or for storing speculative state? Should we optimize for performance or energy efficiency? Clearly, the answers depend on whether we are executing speculatively. PAM-1, then, might address this issue by providing two ISAs – one for non-speculative computations and the other for those that are speculative.

The importance of time- and energy-efficient speculation for such a design calls for other features in the ISA. Can a PAM-1 compiler utilize instructions for communicating probabilities of branch outcomes through the ISA (as in PaCo [115]) in a way that decreases misspeculation costs? Can it utilize instruction extensions for modifying control vectors at run-time? Data longevity is another important metric for which compiler hints may be useful. As a datum’s lifetime increases, it will tend to move farther from the microarchitectural component that produced it, and this distance translates into energy costs.
Another important theme throughout this work has been the emphasis on processor heterogeneity. Above the ISA, such heterogeneity has the advantage that it offers the compiler options for increasing performance, but the disadvantage of increased compiler complexity. Under the ISA, heterogeneity is bountiful, not only in the core designs, but also in many other aspects. Bower et al. divide sources of heterogeneity into the categories of static and dynamic [22]. The use of different core designs is an example of static heterogeneity, as is the use of multiple voltage and frequency domains. Examples of dynamic heterogeneity include DVFS, process variability, and physical faults. The questions for the architect are which features of the microarchitecture will be heterogeneous, and whether and how to expose the different types of heterogeneity to the compiler.

One avenue for future work is the consideration of a heterogeneous core design that can be both active and passive. At different periods of execution, it may be desirable to have a different core executing non-speculatively. Further, it may be beneficial to have multiple cores executing non-speculatively at the same time. Hence, key points in the microarchitectural design are: which cores will implement both A-core and P-core functionality; how to implement these functionalities; and how to switch between the two efficiently. P-core mode would be a low-power mode for executing speculative computations. Are existing techniques such as power gating and drowsy caches effective for achieving low power consumption in this mode? A-core mode, on the other hand, would be designed for performance. Is it feasible to provide dedicated logic for finding and executing branch-critical paths? The increased performance demands of the A-core will likely require larger cache size relative to the P-core. If the core had been using part of its local store for speculative state, it should switch to using it as cache. Can the microarchitecture
leverage hints from the ISA regarding data longevity to estimate cache needs and switch the power modes of its cache?

A promising direction for evaluating the energy consumption of such a parallel processing system is to break the system into levels of spatial decompositions. This decomposition would consist of a partitioning of the system into $L$ levels of spatial granularity, corresponding to processing components such as processors, cores, ALUs, etc. Each level, then, has a spatial decomposition of $S$ components which constitute a component of the next higher level. We can then measure the dataflow among components within a level and the dataflow crossing the boundaries of a level at run-time to estimate the energy consumption of the system. Such a decomposition would provide a framework that is elegant, recursive, and canonical for the comparative energy evaluation of multiple designs.

This framework, in conjunction with the preceding open questions, provides a foundation for future progress in heterogeneous multicore processors supporting computational migration. Consideration of such designs will become increasingly important in meeting performance demands and energy constraints as cores and memory on the chip scale.
## APPENDIX A

### INDEX OF ABBREVIATIONS AND TERMS

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>A-core</td>
<td>Cache-Active Core</td>
<td>p. 91</td>
</tr>
<tr>
<td>ADAG</td>
<td>ASAP-scheduled Directed Acyclic Graph</td>
<td>p. 105</td>
</tr>
<tr>
<td>ALU</td>
<td>Arithmetic Logic Unit</td>
<td>p. 19</td>
</tr>
<tr>
<td>AMAT</td>
<td>Average Memory Access Time</td>
<td>p. 113</td>
</tr>
<tr>
<td>AMO</td>
<td>Active Memory Operation</td>
<td>p. 22</td>
</tr>
<tr>
<td>ASAP</td>
<td>At-the-Sense-Amp Processor</td>
<td>p. 14</td>
</tr>
<tr>
<td>ASAP</td>
<td>As Soon As Possible</td>
<td>p. 62</td>
</tr>
<tr>
<td>BCI</td>
<td>Branch-Critical ICC</td>
<td>p. 82</td>
</tr>
<tr>
<td>BLAS</td>
<td>Basic Linear Algebra Subprograms</td>
<td>p. 132</td>
</tr>
<tr>
<td>BLITS</td>
<td>Byte-Address Long Instructions for Thread Synthesis</td>
<td>p. 97</td>
</tr>
<tr>
<td>CC-NUMA</td>
<td>Cache Coherent Non-Uniform Memory Access</td>
<td>p. 52</td>
</tr>
<tr>
<td>CFG</td>
<td>Control Flow Graph</td>
<td>p. 42</td>
</tr>
<tr>
<td>DAG</td>
<td>Directed Acyclic Graph</td>
<td>p. 21</td>
</tr>
<tr>
<td>DAXPY</td>
<td>Double Alpha X Plus Y</td>
<td>p. 132</td>
</tr>
<tr>
<td>DCP</td>
<td>Distributed Co-Processor</td>
<td>p. 68</td>
</tr>
<tr>
<td>DFG</td>
<td>Dataflow Graph</td>
<td>p. 39</td>
</tr>
<tr>
<td>DIVA</td>
<td>Data-Intensive Architecture</td>
<td>p. 23</td>
</tr>
<tr>
<td>DLP</td>
<td>Data-Level Parallelism</td>
<td>p. 27</td>
</tr>
<tr>
<td>DVFS</td>
<td>Dynamic Voltage and Frequency Scaling</td>
<td>p. 46</td>
</tr>
<tr>
<td>DVS</td>
<td>Dynamic Voltage Scaling</td>
<td>p. 55</td>
</tr>
<tr>
<td>EDGE</td>
<td>Explicit Data Graph Execution</td>
<td>p. 25</td>
</tr>
</tbody>
</table>

(continued on next page)
Index of Abbreviations and Terms (continued from previous page)

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>EIB</td>
<td>Element Interconnect Bus</td>
<td>p. 17</td>
</tr>
<tr>
<td>FPIA</td>
<td>Floating Point Instruction Aggregate</td>
<td>p. 57</td>
</tr>
<tr>
<td>FPU</td>
<td>Floating Point Unit</td>
<td>p. 58</td>
</tr>
<tr>
<td>GPA</td>
<td>Grid Processor Architecture</td>
<td>p. 25</td>
</tr>
<tr>
<td>GPGPU</td>
<td>General-Purpose computing on GPUs</td>
<td>p. 18</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphics Processing Unit</td>
<td>p. 18</td>
</tr>
<tr>
<td>HWP</td>
<td>Heavy-Weight Processor</td>
<td>p. 68</td>
</tr>
<tr>
<td>ICC</td>
<td>In-Cache Computation</td>
<td>p. 68</td>
</tr>
<tr>
<td>ICCCI</td>
<td>ICC Creation Instruction</td>
<td>p. 71</td>
</tr>
<tr>
<td>ICL</td>
<td>ICC-compatible Cache Line</td>
<td>p. 81</td>
</tr>
<tr>
<td>ICLG</td>
<td>Intra-Cache Line Gathers</td>
<td>p. 42</td>
</tr>
<tr>
<td>ILP</td>
<td>Instruction-Level Parallelism</td>
<td>p. 3</td>
</tr>
<tr>
<td>IMO</td>
<td>In-Memory Operation</td>
<td>p. 71</td>
</tr>
<tr>
<td>IOB</td>
<td>Instruction Cache Line Overflow Buffer</td>
<td>p. 76</td>
</tr>
<tr>
<td>IPC</td>
<td>Instructions Per Cycle</td>
<td>p. 104</td>
</tr>
<tr>
<td>IR</td>
<td>Intermediate Representation</td>
<td>p. 39</td>
</tr>
<tr>
<td>ISA</td>
<td>Instruction Set Architecture</td>
<td>p. 10</td>
</tr>
<tr>
<td>LCP</td>
<td>Local Cache Processor</td>
<td>p. 68</td>
</tr>
<tr>
<td>LWP</td>
<td>Light-Weight Processor</td>
<td>p. 15</td>
</tr>
<tr>
<td>mIPM</td>
<td>Minimum Instructions Per Migration</td>
<td>p. 118</td>
</tr>
<tr>
<td>MLP</td>
<td>Memory-Level Parallelism</td>
<td>p. 21</td>
</tr>
<tr>
<td>MMX</td>
<td>MultiMedia eXtensions</td>
<td>p. 32</td>
</tr>
<tr>
<td>NoC</td>
<td>Network-on-Chip</td>
<td>p. 54</td>
</tr>
<tr>
<td>OCT</td>
<td>On-Chip Transport</td>
<td>p. 169</td>
</tr>
<tr>
<td>OoO</td>
<td>Out-of-Order</td>
<td>p. 2</td>
</tr>
<tr>
<td>P-core</td>
<td>Cache-Passive Core</td>
<td>p. 90</td>
</tr>
<tr>
<td>PAM</td>
<td>Passive/Active Multicore</td>
<td>p. 92</td>
</tr>
</tbody>
</table>

(continued on next page)
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIM</td>
<td>Processor-in-Memory</td>
<td>p. 13</td>
</tr>
<tr>
<td>PPE</td>
<td>Power Processing Element</td>
<td>p. 17</td>
</tr>
<tr>
<td>R-NUCA</td>
<td>Reactive Non Uniform Cache Architecture</td>
<td>p. 46</td>
</tr>
<tr>
<td>RUU</td>
<td>Register Update Unit</td>
<td>p. 69</td>
</tr>
<tr>
<td>SIG</td>
<td>Source Instruction Group</td>
<td>p. 69</td>
</tr>
<tr>
<td>SIMD</td>
<td>Single-Instruction, Multiple-Data</td>
<td>p. 27</td>
</tr>
<tr>
<td>SMT</td>
<td>Simultaneous Multithreading</td>
<td>p. 20</td>
</tr>
<tr>
<td>SNL</td>
<td>Sandia National Laboratories</td>
<td>p. 83</td>
</tr>
<tr>
<td>SPE</td>
<td>Synergistic Processing Element</td>
<td>p. 17</td>
</tr>
<tr>
<td>SR</td>
<td>Size Ratio</td>
<td>p. 75</td>
</tr>
<tr>
<td>SSE</td>
<td>Streaming SIMD Extensions</td>
<td>p. 33</td>
</tr>
<tr>
<td>SST</td>
<td>Simultaneous Speculative Threading</td>
<td>p. 21</td>
</tr>
<tr>
<td>SST</td>
<td>Structural Simulation Toolkit</td>
<td>p. 64</td>
</tr>
<tr>
<td>SWAR</td>
<td>SIMD Within a Register</td>
<td>p. 35</td>
</tr>
<tr>
<td>TLP</td>
<td>Thread-Level Parallelism</td>
<td>p. 4</td>
</tr>
<tr>
<td>TMA</td>
<td>Target Memory Address</td>
<td>p. 71</td>
</tr>
<tr>
<td>TRIPS</td>
<td>Tera-op, Reliable, Intelligently adaptive Processing System</td>
<td>p. 25</td>
</tr>
<tr>
<td>VIS</td>
<td>Visual Instruction Set</td>
<td>p. 28</td>
</tr>
<tr>
<td>VLIW</td>
<td>Very Long Instruction Word</td>
<td>p. 23</td>
</tr>
<tr>
<td>WCC</td>
<td>Weakly Connected Component</td>
<td>p. 72</td>
</tr>
</tbody>
</table>
BIBLIOGRAPHY


97. P. La Fratta and P. Kogge. Design enhancements for in-cache computations. In Workshop on Chip Multiprocessor Memory Systems and Interconnects, Austin, TX, June 2009.


127. R. C. Murphy. Design parameters for distributed pim memory systems. Ms, University of Notre Dame, Notre Dame, IN, April 2000.


161. S. Sridharan. Implementing scalable locks and barriers on large-scale lightweight multithreaded systems. Ms, University of Notre Dame, Notre Dame, IN, July 2006.


202


