DIAGNOSING PERFORMANCE BOTTLENECKS IN HPC APPLICATIONS

A Dissertation
Presented to
The Academic Faculty

by

Kenneth Czechowski

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computational Science & Engineering

Georgia Institute of Technology
May 2019

DIAGNOSING PERFORMANCE BOTTLENECKS IN HPC APPLICATIONS

Approved by:

Professor Richard Vuduc, Advisor
School of Computational Science and Engineering
Georgia Institute of Technology

Edmond Chow
School of Computational Science and Engineering
Georgia Institute of Technology

Hyesoon Kim
School of Computer Science
Georgia Institute of Technology

Victor W. Lee
Parallel Computing Lab
Intel

Ümit V. Çatalyürek
School of Computational Science and Engineering
Georgia Institute of Technology

Sudhakar Yalamanchili
(In Memoriam)
School of Electrical and Computer Engineering
Georgia Institute of Technology

Date Approved: 9 May 2019

ACKNOWLEDGEMENTS

First and foremost, I would like to thank Rich Vuduc for being such a supportive advisor. He has always been a patient mentor and fervent champion for his students. He believed in me and selflessly fought to give me many incredible opportunities. I am sincerely grateful for all that he has done for me.

I also thank Victor Lee for taking the time to share his vast technical expertise with me. I learned a lot from him during my time at Intel, and I have appreciated getting to continue collaborating with him even after my return to Georgia Tech. His encyclopedic knowledge of processors and creative problem solving skills still impress me.

I would also like to thank all the close friends I made during my tenure at Georgia Tech. I will always cherish the many fond memories working alongside David Noble, Jee Choi, Casey Battaglino, Marat Dukhan, Xing Liu, Aparna Chandramowlishwaran, and Chris McClanahan in the HPC lab. That crew supplied a healthy dose of motivation and an unhealthy dose of entertainment.

I have been fortunate to collaborate with many thoughtful and talented people which have helped shape the direction of my research. I am grateful to William B. March, Edmond Chow, and Alexander G. Gray, Kartik Iyer, Hyesoon Kim, Yang You, James Demmel, Le Song, Ed Grochowski, Ronny Ronen, Ronak Singhal, Pradeep Dubey and the many other people I have had the pleasure to work with. I also want to thank the CSE faculty and staff as well as my various committee members for graceiously offering their time and attention.

I am grateful to have had Sudhakar Yalamanchili serve on my committee. He was stern but warm and intensely wise. He will be missed.

A special acknowledgment is necessary for Hiroko Kenner, my muse and dear friend. Her limitless patience and constant words of encouragement have kept me going. I could not have done this without her.

Finally, I owe my deepest gratitude to my family for their unwavering support of my dreams. They have always set high expectations for me, pushed me to achieve insurmountable goals, and persevered with me when I struggled. They are the solid foundation upon which everything I have built rests. I would be lost without them.

I am especially grateful to my parents for their unconditional love and constant encouragement. They selflessly provided for me in more ways than I can even recognize. They have instilled in me both a sense of determination and appreciation for the beauty found in life's small mysteries. They enabled me to pursue my dreams.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

The software performance optimizations process is one of the most challenging aspects of developing highly performant code because underlying performance limitations are hard to diagnose. In many cases, identifying performance bottlenecks, such as latency stalls, requires a combination of fidelity and usability that existing tools do not provide: traditional performance models and runtime analysis lack the granularity necessary to uncover low-level bottlenecks; while, architectural simulations are too cumbersome and fragile to employ as a primary source of information. To address this need, we propose a performance analysis technique, called Pressure Point Analysis (PPA), which delivers the accessibility of analytical models with the precision of a simulator. The foundation of this approach is based on an autotuning-inspired technique that dynamically perturbs binary code (e.g., inserting/deleting instructions to affect utilization of functional units, altering memory access addresses to change cache hit rate, or swapping registers to alter instruction level dependencies) to then analyze the effects various perturbations have on the overall performance. When systematically applied, a battery of carefully designed perturbations, which target specific microarchitectural features, can glean valuable insight about pressure points in the code. PPA provides actionable information about hardware-software interactions that can be used by the software developer to manually tweak the application code. In some circumstances the performance bottlenecks are unavoidable, in which case this analysis can be used to establish a rigorous performance bound for the application. In other cases, this information can identify the primary performance limitations and project potential performance improvements if these bottlenecks are mitigated.

CHAPTER I

INTRODUCTION

"Computer architectures have become so complex that manually optimizing software is difficult to the point of impracticality." – M. Frigo and S. G. Johnson (Creators of FFTW), 1998

Most scientific applications run at only a fraction of the theoretical peak throughput of the system. Software performance optimizations can significantly improve the performance of most of these applications; however, the prevalence of this unfulfilled performance potential is a testament to the difficulty involved in diagnosing and fixing performance bottlenecks. The optimization process is time consuming and requires a significant amount of guesswork. Tools, such as performance counters and static code analyzers, are available for collecting performance profiles, but these passive observations are insufficient for identifying the root cause. Currently, this process is more of an art form than a science: the programmer manually tweaks the code in a guess-then-check fashion until suitable performance is achieved.

This dissertation addresses that problem by developing a better understanding of the hardware-software interactions that drive application performance. To build this intuition, we present a systematic approach for diagnosing performance limitations, called Pressure Point Analysis, that actively perturbs code through a series of automated experiments designed to illuminate performance bottlenecks. The results provide high-fidelity insights that appeal to both the hardware and software perspectives. Ultimately, this work contributes to the understanding of hardware-software codesign and improves the productivity of the software optimization process.

The key to diagnosing performance limitations is being able to isolate the application and architectural factors that are most important to performance. Currently, this is done manually by the programmer, albeit guided by performance measurements and profiling tools. Alternatively, our approach automates this processes through a battery of off-line trial-and-error experiments that perturb the code until pressure points are revealed.

Unlike the passive observations from performance-counter-based methods, Pressure Point Analysis relies upon experimentation to actively test a hypothesis about the behavior of the processor core. Instruction sequences from the "hot loops" of the application are turned into a series of micro-benchmarks, each with a minor code variation targeting a specific architectural feature. Comparing the performance of these micro-benchmarks reveals critical information about that architectural features that affect performance. It is important to note that these code perturbations do not preserve the semantic correctness of the code. The code perturbations are not intended to optimize the code but rather to identify the source of lost cycles.

This dissertation makes four primary contributions to the field of high performance computing.

1. **Analysis of proxy application workloads**: We select, organize, and analyze a set of emerging supercomputing workloads. We use this benchmark suite to identify common computational bottlenecks using traditional performance analysis techniques.

2. **An active approach to performance analysis**: We advocate for an active approach toward performance analysis that experimentally tests potential bottlenecks by manipulating specific elements of hardware and/or software in a way that will prove or disprove a potential performance bottleneck.

3. **Evolution of Energy Efficiency**: Using a longitudinal study of the Intel®

Core™ processor, we track the impact architectural innovations have had on performance and energy efficiency. This motivates a discussion about the future strategies for designing lean processors that eliminate architectural bloat without sacrificing performance.

4. **A theoretical framework for algorithm-architecture co-design**: We propose a mathematical framework designed to analyze high-level performance impacts of algorithmic and architectural design decisions. This analytical tool provides a holistic view into the performance characteristics of hypothetical supercomputers by modeling the core hardware-software interactions that drive performance. This approach can be used to inform the codesign process of future supercomputers.

This dissertation also provides three tangible contributions in the form of publicly available software:

1. **Proxy_app benchmark suite**: The proxy_app benchmark suite provides scripts for downloading, compiling, building, and running the proxy_app workloads. All of the software dependencies, such as OpenBLAS and Eigen, are downloaded, built, and managed by the benchmark suite. Chapter 2 includes a detailed description of the workloads and high-level analysis of the workload characteristics. The software is available at `https://bitbucket.org/kentcz/proxy_apps`.

2. **Pressure Point Analysis framework**: The PPA Toolkit provides tools for identifying, extracting, and analyzing loop kernels. Chapter 3 provides more details about the Pressure Point Analysis as well as a description of the process involved in conducting this type of analysis on the Intel Ivy Bridge microarchitecture. The codebase is available at `https://bitbucket.org/kentcz/code_analysis_tool2`.

3. **Corpus of loop kernels**: As part of the Pressure Point Analysis, 130 loop kernels were extracted from the proxy_app workloads and converted into standalone microbenchmarks. The loop kernels are available at `https://bitbucket.org/kentcz/code_analysis_tool2`.

1.1 Scope and Outline

Chapter 2 introduces the Proxy_app workloads as a representative sample of emerging HPC workloads then analyzes their performance characteristics. We observe the instruction mix, instruction throughput, bandwidth utilization, and program control of these workloads. For context, the results are compared with the NAS Parallel Benchmark and SPECint workloads. We also use traditional performance diagnosis techniques to identify bottlenecks in the applications.

Chapter 3 introduces the concept of passive vs active performance analysis. We first demonstrate methods for using hardware perturbations to identify performance bottlenecks. We also introduce the Pressure Point Analysis as a technique for perturbing software to identify performance bottlenecks.

Chapter 4 considers the hardware design perspective by studying the impact evolutionary improvements in the microarchitecture of recent generations of the Intel Core processor have had on both performance and energy efficiency.

Chapter 5 analyzes the inherent performance limitations of application by simultaneously abstracting both the computational requirements and the architectural design. Using an analytical framework, we study the algorithm-architecture trade-offs in future supercomputer system architectures.

Chapter 6 summarizes the findings and offers insight into future research directions.

CHAPTER II

PROXY APPLICATIONS BENCHMARKS

In this chapter we present the Proxy_app benchmark suite, a collection of workloads for evaluating the performance of processor microarchitectures across common supercomputing workloads. This workload will be used throughout this dissertation as a representation of HPC workloads relavant to supercomputing.

2.1 Proxy Apps

As part of the Exascale Supercomputing directive, the DOE has identified several high priority applications which are targeted for execution on future exascale supercomputers. To make these workloads more accessible to researchers, several proxy apps were commissioned as part of a co-design initiative tasked with enabling scientific applications to harness the potential of exascale computing. The proxy apps help hardware and software developers make coordinated design decisions.

The proxy apps provide a smaller full featured representation of particular supercomputing applications. Core computational kernels have been extracted into self-contained proxy apps in such a way that they still preserve the workload characteristics of the original application. Lessons learned from hardware or software explorations of the proxy app can be applied to the full application.

2.2 Motivation

The proxy_app benchmark suite was created to evaluate the performance of processor microarchitectures across common supercomputing workloads. Full-scale supercomputing benchmarks are too cumbersome to run repeatedly and too unwieldy to analyze

at the level of single-core-performance. To analyze the lower-level performance characteristics of the processors, this suite instead uses proxy apps that run on a standard workstation – it provides the workload characteristics of the computational kernels within a supercomputing application but has been scaled-down to the magnitude of standard processor benchmark suites.

The proxy_app benchmark suite fulfills the standard expectations of a meaningful set of benchmarks:

1. **Emerging Workloads**: The workloads in this benchmark are all DOE sanctioned proxy apps. They are based on applications that play a key role in solving many of today's most pressing problems, including producing clean energy, extending nuclear reactor lifetimes, and certifying the aging nuclear stockpile.

2. **Diverse**: The proxy apps in the suite represent a variety of supercomputing applications. They are collected from each of the three ASCR Co-design Centers: ExMatEx, CESAR, and ExaCT.

3. **Employ State-of-Art Techniques**: Over time, both the hardware and the software evolve – new algorithms, applications and techniques are always emerging. These proxy apps and the applications upon which they are based, have been the test case for many new programming models, algorithms, performance analytics, software development tools, and software optimizations techniques.

4. **Support Research**: The associated tools provided with the benchmark suite extends beyond a simple performance scoring. Each of the benchmarks have been augmented with profiling tools and parameterized for controlling compilation configurations.

2.3 Workloads

The proxy_app benchmark suite contains the eleven proxy apps.

1. **LULESH**: The Livermore Unstructured Lagrange Explicit Shock Hydro (LULESH) is a based on the ALE3D hydrodynamics application, which consumes over 25% of data center resources throughout the DOD. It is a mesh based physics simulation on an unstructured grid. The most compute intensive part of the simulation is calculating the nodal forces during each time-step [61]. LULESH has been widely studied as a test case for various programming models and performance analysis tools. Highly optimized versions have been implemented in MPI, OpenMP, CUDA, OpenCL, OpenACC, Chapel, and Charm++. Related work has demonstrated that effective use of loop fusion, array contraction, and vectorization have significant impact on performance [62, 63].

2. **MiniFE**: MiniFE is an unstructured implicit finite-element application. It solves a sparse linear system using an un-preconditioned conjugate-gradient. Most of the computation is contained within level-1 BLAS operators such as AXPY, DOT, and NORM as well as sparse matrix-vector products.

3. **CoSP2**: CoSP2 calculates the density matrix for electronic structure calculations in quantum molecular dynamics (QCD). The algorithm is based on a Second Order Spectral Projection (SP2) method. Most of the computation is contained inside a series of sparse matrix-matrix multiplies, which use the ELLPACK-R (ELL) sparse matrix data format.

4. **CoHMM**: CoHMM is a multiscale simulation of elastodynamics based on the Heterogeneous Multiscale Method (CoHMM). The application uses spatial adaptive sampling to reduce the demand on fine-scale simulations. At each macro-scale time step, it reconstructs the required constitutive data by interpolating fine-scale simulations on the 3-dimensional simulation domain. A predictive approach is used to determine the important sample points at the beginning of each macro-scale, which provides an abundance of parallelism with

minimal synchronization [97].

5. **CoMD**: CoMD is a molecular dynamics (MD) simulation used for studying dynamical properties of liquids and solids. The majority of the computational is involved in N-body evaluations of the interatomic potentials and the corresponding forces. OpenMP is used to parallelize the force loop; however, in order to avoid race conditions and costly atomic operations, the application ignores force symmetry. Each atom in the pair computes F_{ij} separately and updates only the individual force term [21].

6. **CoEVP**: Embedded Visco Plasticity (CoEVP) uses scale-bridging to solve a Taylor cylinder impact test problem. The simulation uses a Lagrangian finite element model to calculate the deformations at a coarse scale and a viscoplasticity model to measure the fine-scale details of the material microstructure [43, 88].

   The application stores the results of fine-grain simulations in an embedded database. The fine-grained simulations synchronize internally using OpenMP, while the database serves as a synchronization point between simulations. The computational kernels inside this application make extensive use of costly mathematical operations such as *sqrt* and *pow* [39].

7. **XSBench**: XSBench is a proxy for the key computational kernel of the Monte Carlo neutronics application OpenMC. It uses the Hoogenboom-Martin model to compute the macroscopic neutron cross sections, a kernel which accounts for around 85% of the total runtime of OpenMC. There are 12 different materials and 355 different nuclides present in the modeled reactor, totaling 5.6 GB of input data [113].

8. **RSBench**: RSBench represents the multipole resonance representation lookup cross section algorithm. It is identical to XSBench in functionality, but utilizes an alternative algorithm. RSBench models the multipole cross section

lookup algorithm, which is more efficient at storing and moving data than XS-Bench [112].

9. **Nekbone**: Nekbone is an abridged version of the Nek5000 thermal hydraulic code designed for large eddy simulation and direct numerical simulation of turbulence. Nekbone embeds a matrix-vector product in a conjugate gradient iteration to solve the 3D Poisson equation [66].

10. **Lassen**: Lassen is a front-tracking application used to calculate the propagation of wave-fronts. It is a highly irregular application with dynamic load balancing.

11. **UMT**: UMT is a 3D, deterministic, multigroup, photon transport code for unstructured meshes. The kernel has a high compute intensity, but tasks are entangled with dependencies between upstream and downstream cells in the sweep directions [87].

2.4 Evaluation Methodology

Our goal is to analyze the workload characteristics of real-world supercomputing applications. We evaluate workloads from the proxy_app benchmark suite, NAS Parallel Benchmarks (NPB) OpenMP 3.3 [59], and the SPECint benchmarks from the SPEC CPU2006. All of the experiments were preformed on a an Intel Core i7 3770K Ivy Bridge processor running a stock Ubuntu 14.04 Server installation. For comparisons, we also tested an Intel Core i7 4770K Haswell processor with an identical setup. The processors were configured to run at factory prescribed 3.5 GHz frequency with Intel® Turbo Boost Technology and Hyperthreading features disabled. The benchmarks were compiled using the Intel® Composer XE version 15.0.0 tool chain [54], with the exception of a few proxy_app which were compiled with GCC 4.8.4 because of entrenched compiler dependencies.

2.4.1 Hierarchical Event-based Performance Analysis

Beyond high-level workload characteristics, we also conduct a Hierarchical Event-based Performance Analysis (HEPA) on each workload to identify low-level bottlenecks. This analysis consists of a systematic method for using architectural performance counters to diagnose performance limitations, formalized by engineers at Intel [118]. It is implemented in several of the most popular performance analysis tools, including VTune and the Linux Perf utility [29].

The HEPA methodology uses a hierarchical approach to identify bottlenecks. It starts with a high-level classification of CPU execution time, then drills down into categories that need further investigation. The drill-down continues recursively until a specific stall has been identified. Figure 1 is an abbreviation of the hierarchy used to analyze the Ivy Bridge microarchitecture, which is based on the methodology presented by Ahmad Yasim [118]. Root causes become leaf nodes in the hierarchy tree. Unfortunately, since performance counter events track different pipeline stages, many of the events are not directly comparable, preventing a single flat break-down of performance bottlenecks. In the hierarchical approach, the hierarchy is intentionally constructed in such a way that all sibling nodes measure the same pipeline stage and can be measured simultaneously, so that all sibling node are comparable.

On the Ivy Bridge microarchitecture, the top level breakdown classifies each pipeline slot as either Retiring, Frontend-Bound, Backend-Bound, or Bad Speculation. The flowchart in Figure 2 depicts the logic used to classify the pipeline slots. The Retiring label indicates that the pipeline slot was used to retire a uop successfully without stalls. The percentage of Retiring pipeline slots represents the percentage of maximum uop throughput achieved, it is highly correlated with the Instructions Per Cycle (IPC) metric. The other labels identify parts of the microarchitecture responsible for the stall. Table 1 lists the performance counter events measured during the top level analysis. Equation 1 is used for the top level analysis.

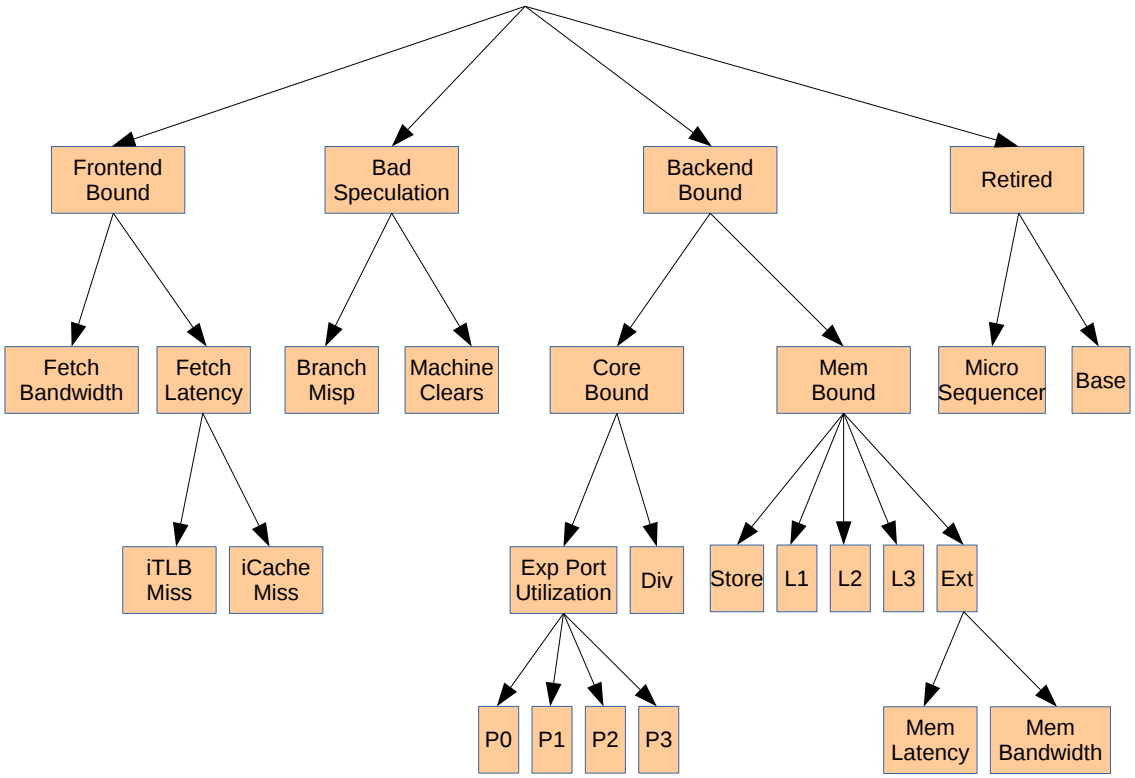

Figure 1: Hierarchy for the Ivy Bridge microarchitecture.

$$SLOTS = 4 \times CPU\_CLK\_UNHALTED.THREAD \quad (1)$$

$$RETIRED = UOPS\_RETIRED.RETIRE\_SLOTS \,/\, SLOTS \quad (2)$$

$$\begin{aligned} BAD\_SPECULATION = {} & (UOPS\_ISSUED.ANY \\ & - UOPS\_RETIRED.RETIRE\_SLOTS \\ & + 4 \times INT\_MISC.RECOVERY\_CYCLES) \,/\, SLOTS \end{aligned} \quad (3)$$

$$FRONTEND\_BOUND = IDQ\_UOPS\_NOT\_DELIVERED.CORE/SLOTS \quad (4)$$

$$\begin{aligned} BACKEND\_BOUND = {} & 1 - (FRONTEND\_BOUND \\ & + RETIRED + BAD\_SPECULATION) \end{aligned} \quad (5)$$

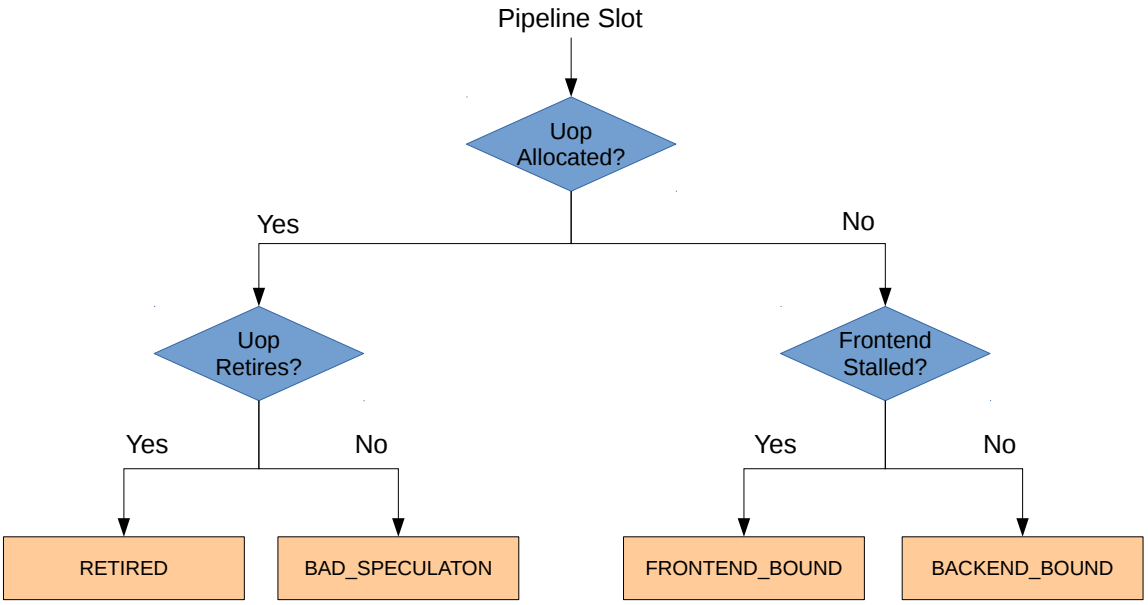

Figure 2: Top level flowchart.

2.5 Results

This section examines the workload characteristics of the proxy_app benchmark suite. For context, resulting workload characteristics of the proxy_app workloads are compared with the NPB and SPECint workloads.

2.5.1 Traditional metrics

Performance counters were used to measure several common performance metrics. Figure 3 lists the Instructions Per Second (IPC). The Ivy Bridge microarchitecure core is capable of sustaining up to four uops per cycle, which roughly translates to an ideal throughput of four instruction per cycle. Overall, the average IPC is a consistent 1.3 instructions per cycle across the proxy_app, NPB, and SPECint benchmark suites. Compared to NPB and SPECint, the proxy_app workloads has more variability, with MiniFE averaging 0.8 IPC and Nekbone averaging 3.4 IPC.

Figure 4 lists the effective memory bandwidth utilization. The test system has a dual channel 1600MHz DDR3 memory with a peak memory throughput of 25.6

Table 1: Performance events measured in the Top Level analysis

Event	Description
CPU_CLK_UNHALTED.THREAD	Core cycles when the core is not in halt state
IDQ_UOPS_NOT_DELIVERED.CORE	Uops not delivered to resources Allocation Table (RAT) per THREAD when backend of the machine is not stalled
UOPS_ISSUED.ANY	Uops that Resource Allocation Table (RAT) issues to Reservation Station (RS)
UOPS_RETIRED.RETIRE_SLOTS	Retirement slots used
INT_MISC.RECOVERY_CYCLES	Number of cycles waiting for the checkpoints in Resource Allocation Table (RAT) to be recovered after Nuke due to all other cases except JEClear

GB/s. Only five of the twenty-nine (proxy_app.MiniFE, NPB.cg, NPB.lu, NPB.mg, and NPB.sp) utilize more than half of the available memory throughput. The NPB workloads average 12 GB/s whereas the proxy_app workloads average only 4 GB/s, which is only 15% of the peak throughput.

2.5.2 Loop analysis

The program control flow and the underlying loop structure of a workload have a significant impact the runtime behavior of the application. Many architectural features such as the branch predictors, loop buffer, and instruction decode unit, are sensitive to the size and complexity of loop structure.

To compare the program control flow of different workloads, we measure several high-level loop characteristics: number of loop nests, execution time spent in loop nests, average number of instructions per loop nest, and the average stack depth of a loop nest.

For this data to be relevant, we only analyze "hot" loop nests and filter out auxiliary loop nests that have negligible impact on the runtime of the application. Optimizing compilers often create multiple binary loops from a single source loop

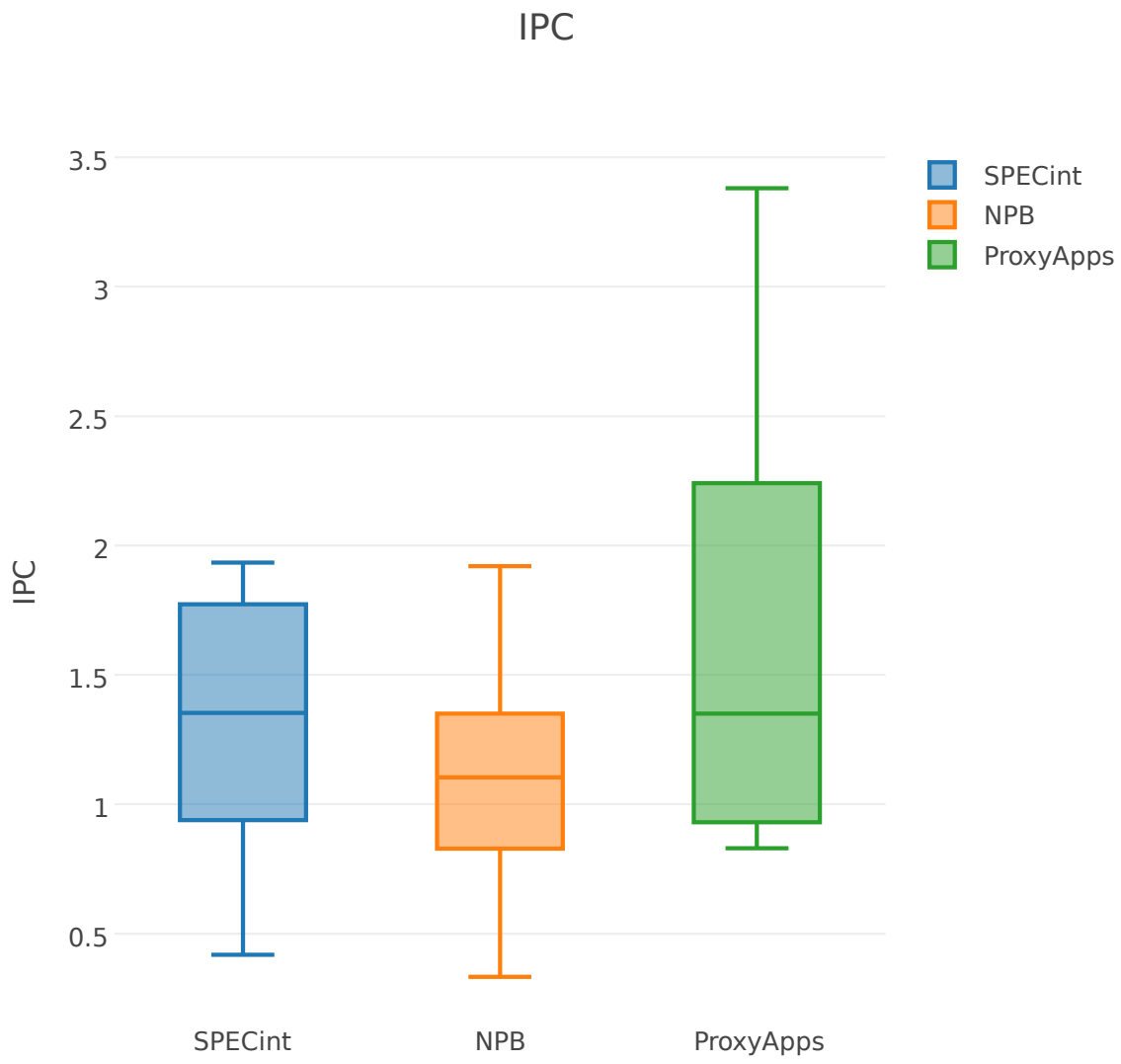

Figure 3: Box plot of Instructions Per Second (IPC).

to deal with the corner case of optimizations. For example, unrolling a loop often requires a tail loop to handle left-over iterations. Applications also contain a variety of initialization code that does not have material impact on the performance of the application. For the purposes of this study we chose 1% of execution time as our threshold and ignore all loop nests that do not consume enough execution time, which is consistent with the similar studies [64, 82]. Execution time of the loop nest is estimated by aggregating profile time across each of the instructions in the basic block.

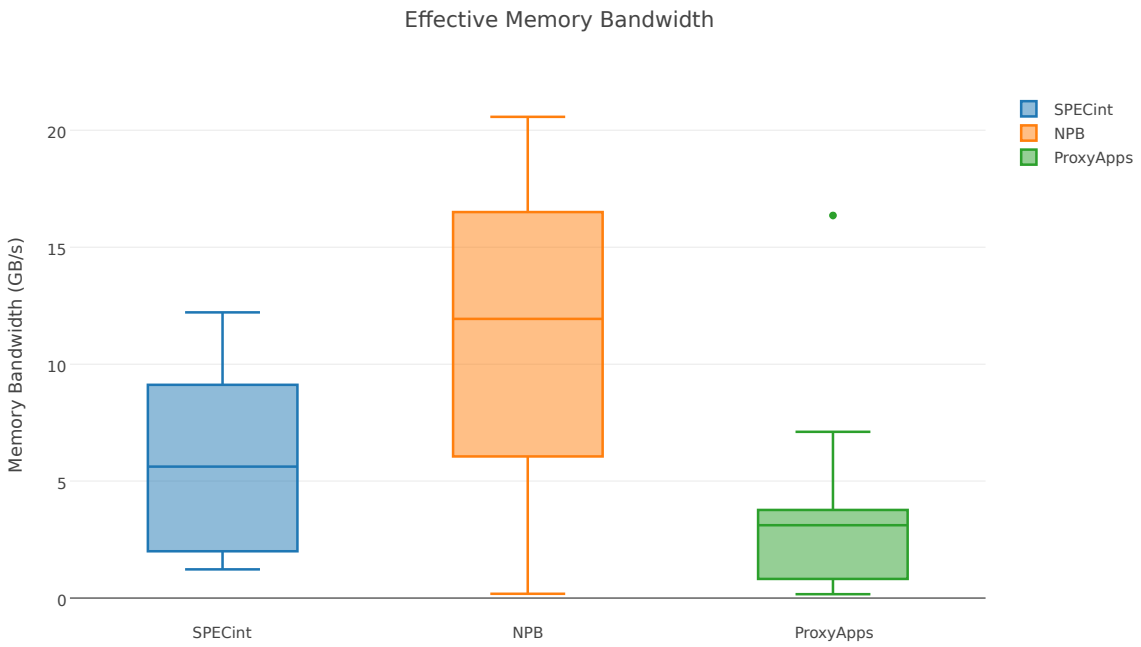

Figure 4: Box plot of memory bandwidth utilization.

Table 2 lists the loop characteristics of the proxy_app workloads. The "LN Count" column lists the number of hot loop nests and the "LN Time" lists percentage of execution time in the hot loop nests. Exception for CoEVP, Lassen, and UMT, the vast majority of execution time is spent inside loop nests. There is an average of eight hot loop nests across the proxy_app workloads, which is lower than NPB (17) and much lower than SPECint (49).

Table 2: Loop characteristics of the proxy_app workloads

	LN Count	LN Time
LULESH	11	73.7%
MiniFE	12	83.3%
CoSP2	8	91.0%
CoHMM	5	94.6%
CoMD	5	93.3%
CoEVP	11	17.2%
XSBench	3	93.1%
RSBench	3	90.5%
Nekbone	10	93.1%
Lassen	4	07.1%
UMT	12	47.6%

2.5.3 Instruction mix results

Pin was used to dynamically count the instructions executed in the proxy_apps workloads [81]. The frequencies are shown in Table 3. The percentages are based on the frequency of executed instructions based on the application simulation, and does not necessarily represent the number of instructions in the compiled binary or the faction of time spent executing the instructions. Each instruction is categorized as

1. **AVX**: part of either the AVX or SSE instruction set
2. **Vec**: a vectorized (non-scalar) instruction,
3. **FP**: a floating-point arithmetic instruction,
4. **Mem**: a memory load or store instruction,
5. **Rds**: a memory read instruction,
6. **Rds16**: a vectorized read instruction that load 16-bytes or more,
7. **Brnch**: a branch instruction,
8. **Stck**: a stack instruction.

A single instruction can qualify for multiple labels. For example, "vmulpd ymm5, ymm4, YMMWORD PTR [r14+r8*8]" loads a 32-byte value then computes a vectorized double-precision multiply. It is counted as AVX, Vec, FP, Mem, Rds, and Rds16.

Memory operations are extremely common. Across the proxy_app workloads, 30-50% of instructions executed include a memory access, except for XSBench. The vast majority of the memory accesses are loads. Only RSBench and UMT utilize vectorized loads effectively. Branch instructions range from 3.7-19.9% with an average of 12.7%. Stack instructions are slightly more common with a range of 3.1-28.4% and an average of 13.7%.

The instruction mix of the NPB and SPECint workloads are listed in Table 4 and Table 5. As expected, the SPECint workloads do not contain floating-point instruction or utilize the AVX instruction unit. Instead, the Specint workloads have more memory operations, branch instructions, and stack instructions. The NPB workloads are more similar to the proxy_app workloads. Both the NPB and proxy_app workloads have similar distributions of memory operations and floating-point operations.

Table 3: Instruction mix of proxy_app workloads

	AVX	Vec	FP	Mem	Rds	Rds16	Brnch	Stack
LULESH	78.1%	11.2%	42.0%	50.1%	36.6%	03.9%	03.7%	28.4%
MiniFE	34.7%	12.8%	16.1%	38.2%	34.1%	02.6%	11.4%	08.0%
CoSP2	26.3%	01.6%	11.7%	46.0%	36.8%	00.0%	15.9%	03.1%
CoHMM	09.1%	01.2%	04.1%	39.9%	35.7%	00.8%	17.3%	07.2%
CoMD	58.3%	01.1%	30.3%	48.5%	38.1%	00.0%	06.5%	19.9%
CoEVP	18.7%	02.0%	07.7%	33.4%	22.1%	00.7%	14.2%	18.6%
XSBench	45.6%	09.9%	10.6%	15.0%	13.1%	01.2%	13.6%	03.1%
RSBench	68.8%	55.4%	36.3%	43.3%	34.3%	20.5%	05.7%	19.1%
Nekbone	00.0%	00.0%	00.0%	33.0%	24.1%	00.2%	19.8%	12.3%
Lassen	00.0%	00.0%	00.0%	31.4%	21.0%	02.7%	17.4%	13.4%
UMT	35.7%	18.8%	15.4%	42.4%	31.1%	08.7%	09.4%	18.7%

Table 4: Instruction mix of NPB workloads

	AVX	Vec	FP	Mem	Rds	Rds16	Brnch	Stack
bt	43.3%	00.1%	43.3%	51.6%	33.6%	00.1%	00.3%	28.9%
cg	25.0%	00.1%	25.0%	39.9%	39.0%	00.0%	15.8%	00.1%
dc	00.0%	00.0%	00.0%	34.6%	26.0%	00.0%	20.1%	12.3%
ep	46.3%	09.4%	23.6%	24.6%	18.2%	01.3%	11.6%	09.6%
ft	30.6%	00.2%	30.1%	31.6%	16.0%	00.0%	05.2%	00.7%
is	14.5%	00.0%	14.5%	36.3%	25.6%	00.0%	08.2%	06.5%
lu	51.3%	00.1%	51.3%	46.4%	33.9%	00.0%	00.7%	15.2%
mg	23.0%	19.3%	23.0%	46.8%	40.2%	02.1%	03.7%	05.5%
sp	41.3%	00.2%	41.3%	43.5%	29.3%	00.0%	02.4%	05.8%
ua	31.8%	03.8%	29.3%	55.9%	39.3%	00.2%	07.0%	21.8%

Table 5: Instruction mix of SpecInt workloads

	AVX	Vec	FP	Mem	Rds	Rds16	Brnch	Stack
perlbench	00.0%	00.0%	00.0%	43.0%	29.0%	00.2%	14.4%	22.4%
bzip2	00.0%	00.0%	00.0%	35.8%	25.4%	00.4%	08.8%	23.5%
gcc	00.0%	00.0%	00.0%	42.2%	28.4%	00.2%	15.2%	21.5%
mcf	00.0%	00.0%	00.0%	44.3%	29.4%	00.2%	17.6%	20.6%
hmmer	00.0%	00.0%	00.0%	38.8%	26.8%	00.3%	11.8%	22.5%
sjeng	00.0%	00.0%	00.0%	44.8%	29.7%	00.2%	18.0%	20.5%
libquantum	00.0%	00.0%	00.0%	44.6%	29.6%	00.2%	17.9%	20.5%
astar	00.0%	00.0%	00.0%	44.2%	29.4%	00.2%	17.3%	20.7%

2.5.4 HEPA results

Figure 5 shows the results from the HEPA analysis (see Section 2.4.1) on the proxy_app workloads. For comparisons, Figure 6 and Figure 7 shows the results for the NPB and SPECint workloads.

Instruction throughput ranges dramatically across the proxy_app workloads. Nekbone utilizes over 85% of pipeline slots whereas XSBench uses only 15%. The average pipeline utilization across the proxy_app workloads is 43% which is higher than the average across the NPB (32%) and SPECint (35%) workloads.

Scientific applications typically avoid the penalty of incorrect speculation because the regular program control flow is effectively managed by the branch predictor. In the proxy_app workloads speculation costs are only significant in the XSBench and CoMD, of which it only accounts for 13% of pipeline slots. The NBP workloads are similarly unimpacted by speculation. The more irregular branching patterns of the SPECint workloads account for an average of 10% of pipeline slots lost to bad speculation.

Overall, most the workloads are Backend Bound. Across the proxy_app workloads an average of 47% of pipeline slots are consumed by execution stalls in the Backend. The average jumps to 58% across the NPB workloads. The average is only 36% across the SPECint workloads because they are more susceptible to speculation and

Frontend issues.

The MicroOp Cache and Loop Cache have dramatically reduced the number of frontend bottlenecks in the Ivy Bridge microarchitecture. Of the proxy_app workloads, only CoSP2, CoEVP, and Lassen have significant bottlenecks in the Frontend. The SPECint workloads are more sensitive to Frontend issues because they tend to have larger code footprints. Frontend issues dominate the Perlbench and Sjeng workloads.

The Backend Level plots classify Backend Bound workloads as either Core or Memory bound by accounting for execution stalls in the backend. Seven of the eleven proxy_apps are Core Bound – only MiniFE, CoSP2, XSBench, and RSBench have more stalls attributed to the memory subsystem than the execution core. This is indicative of loops with tight data-dependent arithmetic instructions. The NPB and SPECint workloads are more Memory Bound.

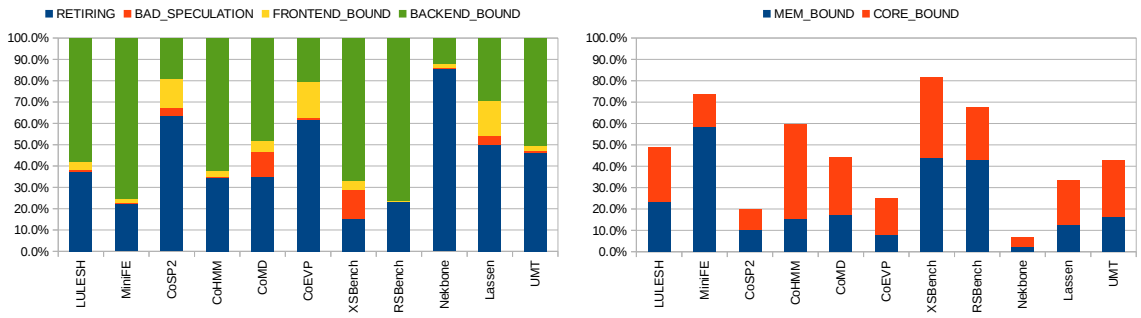

Figure 5: Top level (left) and Backend level (right) analysis of the proxy_app workloads

2.6 Discussion

Overall, the consistently low instruction throughput scores indicates a severe mismatch between the software demands and the hardware capabilities. On average, the proxy_app workloads only utilize 43% of the capabilities of the processor. The vector units are also under utilized, most floating-point operations and memory accesses did not take advantage of the SIMD capabilities. Most of the inefficiencies arise from

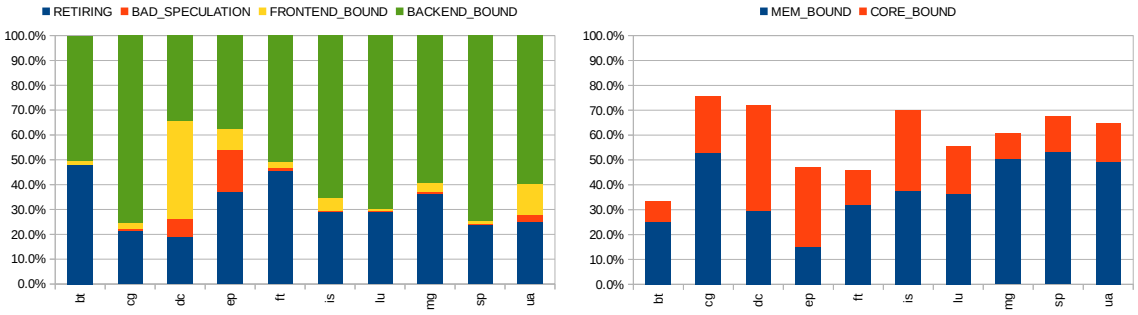

Figure 6: Top level (left) and Backend level (right) analysis of the NPB workloads

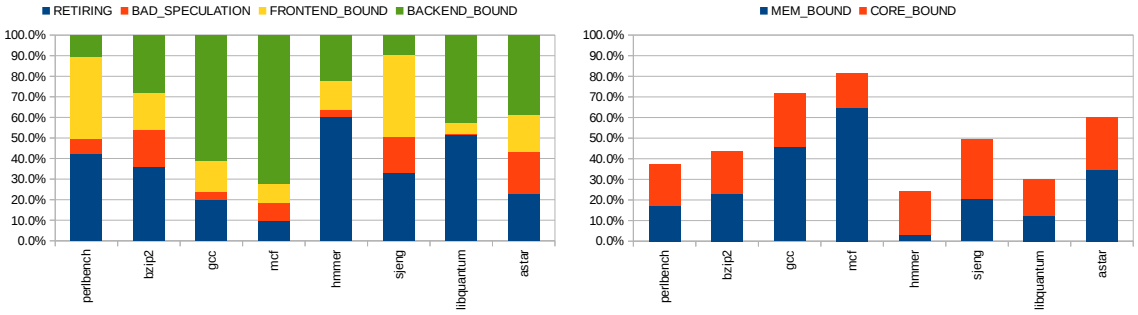

Figure 7: Top level (left) and Backend level (right) analysis of the SPECint workloads

issues in the backend of the processor, which consists of stalls in the core as well as the memory hierarchy.

The low memory bandwidth utilization in the proxy_app, NPB, and SPECint workloads suggest some degree of "memory bandwidth headroom". In particular, the emerging HPC workloads of the proxy_app suite are not as sensitive to memory bandwidth and latency as traditional performance evaluations might indicate.

Despite the reputation for being floating-point heavy, the HPC workloads in the proxy_app benchmark suite contained a lot of non-floating-point operations. A similar study of proxy app workloads by Vetter et al. found that integer and memory operations are more common that float-point operations in most HPC applications [115], which is consistent with the findings in this study. Almost a third of the instructions involved memory accesses. Branch and stack instructions are also common.

2.7 Related Work

Several performance-counter-based tools have been developed to diagnose bottlenecks and visualize profiling data. Tau [104], HPCToolkit [3], and PerfSuite [72] are among some of the most powerful. Each of these tools provides a variety of metrics to evaluate the profile information along with heuristics for identifying bottlenecks based on these metrics. Unfortunately, performance counters only provide a limited view of the internal state of the processor, limiting direct access to several critical architectural factors. Also, since these techniques only observe performance effects, more information is needed pinpoint the underlying cause.

Some of these tools also incorporate trace-based simulations. Cache simulators, such as SLO [12], can use memory traces to identify potential bottlenecks due to cache hit rates. Tools for distributed systems, such as Tau [104], can instrument MPI functions to trace potential communication bottlenecks. Of course, the accuracy of the trace-based simulations are limited by the ability to properly simulate the underlying hardware. For example, the eviction policy of the caches in the Intel Core i7 microarchitectures are considered proprietary information, making cache simulators of these microarchitectures purely speculative.

Static code analysis can supplement run-time performance analysis. The Intel Architecture Code Analyzer (IACA) [2] evaluates instruction mix and instruction-level dependencies to predict port pressure and estimate computational latencies.

Beyond diagnosing performance limitations, more sophisticated tools focus on the performance optimization task. AutoSCOPE [108] uses diagnostic information collected from performance-counter-based data, trace-based simulations, and static analyzers to select source-code transformations and compiler flags to optimize program performance. The PERI Autotuner uses performance-counter-based information for performance analysis to inform the autotuner and similarly uses the autotuner history to inform the performance analysis [8].

CHAPTER III

PRESSURE POINT ANALYSIS OF PERFORMANCE

Diagnosing performance limitations in HPC applications can be difficult. Complex interactions between the hardware and software of large-scale parallel systems make performance diagnosis difficult. For example, if a turbulence simulations is run on a Craxy XE6 supercomputer and it takes four hours to complete, how do we evaluate its performance? How do we know if investing more time in software optimizations are likely to yield significant performance improvements? How do we identify the performance limitations so that we can improve either the software or the hardware to get better performance?

The key to diagnosing performance limitations is being able to isolate application and architectural factors that impact performance. Unfortunately, due to the complexity of low-level interactions, existing performance analysis tools —which analyze data collected from analytical models, performance models, and profiling tools— do not have enough information to isolate all relevant interactions to make a conclusive assessment.

Instead, this chapter advocates for an active approach toward performance analysis. Traditional performance analysis techniques are passive: they infer potential bottlenecks from observations of performance characteristics. Instead, an active approach experimentally tests potential bottlenecks by manipulating specific elements of hardware and/or software in a way that will prove or disprove a potential performance bottleneck. For example, if we believe that a given application is memory-bandwidth bound, we can test this hypothesis by generating a version of the application with all memory accesses removed then compare its performance with the original application.

3.1 Hardware perturbations

Incremental changes in the underlying execution hardware can be used to experimentally test potential performance bottlenecks. This works by analyzing the change in the runtime performance of an application when a specific hardware feature is disabled. A common example of hardware defeaturing is disabling hardware prefetching. If disabling the hardware prefetchers has a significant impact on the runtime performance, we know that the application is highly sensitive to cache access latencies. Unfortunately, there are only a limited number of hardware modifications available, not enough to defeature pipeline depth, branch prediction, reorder buffer size, or cache latency.

3.1.1 Throttling core and memory clock frequency

Throttling the memory clock frequency can be used to quantify the impact memory bandwidth has on the runtime performance of an application. If throttling memory bandwidth does not have a significant impact on performance, we can conclude that the memory bandwidth is not the primary performance bottleneck.

Previous work by Malladi et al. has suggested that many so called "memory bandwidth bound" applications are actually mischaracterized [83]. These characterizations are refuted by measuring performance penalties incurred by executing the application with the DRAM channel frequency throttled down. Applications that are not sensitive to degradation in the memory bandwidth are therefore not memory bandwidth bound. For example MemcacheD, a common data center workload, intuitively should be either memory or network bound, but close inspection has revealed it to be limited by the instruction fetch rate [77].

The proliferation of the overclocking community, which consists of hardware enthusiasts interested in maximizing the performance of their desktop machines by boosting CPU clock frequency, has motivated hardware manufacturer to provide mechanisms

for manually controlling the voltage and frequency of various hardware components. The "unlocked" line of Intel desktop processors provide knobs for controlling the clock frequencies of the cores, bus, uncore, and DRAM. Our Haswell test system has dual-channel 32GB overclocked DDR3-2133 memory. Using the motherboard overclocking settings, we are able to manually throttle the DRAM clock frequency from 2133 MHz down to 800 MHz, which throttles the system memory bandwidth from 34.1 GB/s to 12.8 GB/s. Experiments with the GUPS and Stream benchmarks have shown that clock frequency throttling does in fact scale both memory access latency and memory bandwidth appropriately. The test system is also capable of scaling the core clock frequency from 800 MHz to 3.5 GHz.

Our experiments with the proxy_app workloads show that these applications are highly sensitive to clock frequency scaling, but only the MiniFE workload is sensitive to memory frequency scaling. Figure 8 shows the performance speedup attributed to clock frequency throttling. The Core frequency scaling plot on the left shows the performance improvement when throttling the clock frequency from 800 MHz to 3.5 GHz, a 4.375x increase. The different series represent different memory clock frequencies: blue is when the DRAM frequency was set to 800 MHz, orange is when the DRAM clock frequency was set to 1600 MHz, and yellow is when the DRAM frequency was set to 2133 MHz. The CoHMM, CoEVP, and Nekbone workloads had nearly perfect 4.375x speedup to match the 4.375x increase in clock frequency and had virtually no performance degradation with the lower DRAM frequency, which clearly rules out performance bottlenecks related to the memory bandwidth. A couple of the workloads, for example RSBench, had higher performance improvements when the DRAM frequency was higher, but the impact of the memory clock frequency is overshadowed by the impact of the core clock frequency. The MiniFE workload is the exception, although it is sensitive core frequency, it is more sensitive to the DRAM frequency. The Memory Frequency Scaling plot on the right shows the performance

improvement when throttling the memory clock frequency from 800 MHz to 2133 MHz, a 2.667x increase. The different series represent different core clock frequencies: blue is when the core frequency was set to 800 MHz, orange is when the core clock frequency was set to 2.1 GHz, and yellow is when the core clock frequency was set to 3.5 GHz.

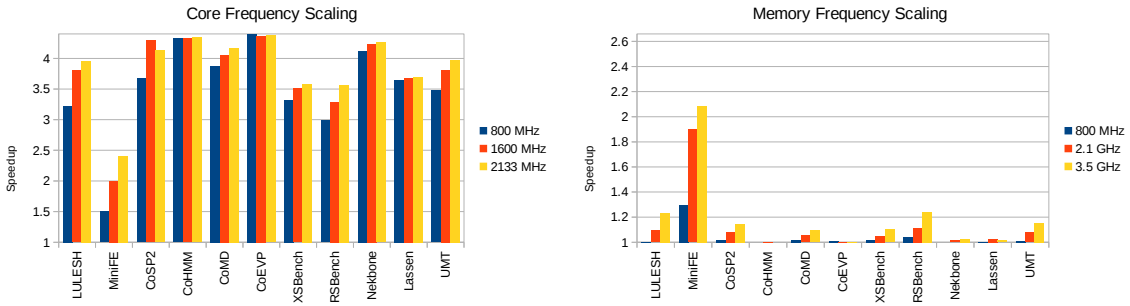

Figure 8: Core (left) and memory (right) clock frequency scaling.

Most applications have a balance point, a ratio between the core frequency and the memory frequency where the core and memory are both saturated. Figure 9 shows the performance of the NPB.lu workload across various combinations of core and memory clock frequencies. The balance point for NPU.lu is when the core and memory frequency are at a 3:1 ratio. When the memory clock frequency is 800 MHz, the performance scales linearly with the core clock until the core frequency reaches 2.4 GHz, at which point the memory bandwidth has been saturated and the performance can no longer scale with the core clock frequency. When the memory frequency is 1066 MHz, the memory bandwidth become saturated at 3.0 GHz, as seen in Figure 9. When the memory frequency is 2133 MHz, the core frequency would need to reach 6.4 GHz to saturate the memory bandwidth.

3.1.2 Throttling cache capacity

Data caches are crucial for hiding the long latencies associated with fetching data from memory. Effectively managing temporal locality, spacial locality, and cache pollution

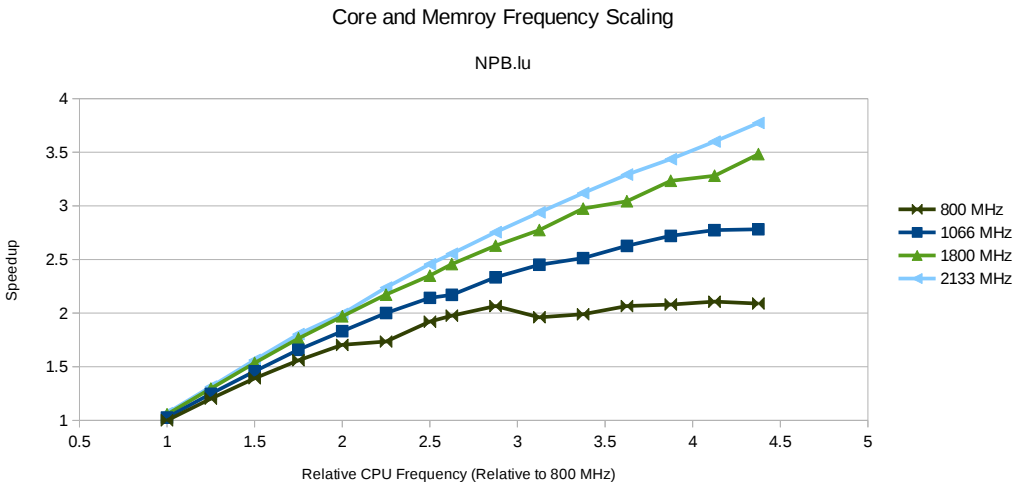

Figure 9: Runtime performance as a function of core frequency on NPB.lu workload. The different series represent different DRAM clock frequencies.

can have a significant impact on the performance of memory-intensive workloads. Satish et al. have found that optimizing the memory access patterns is one of the most productive software optimization techniques for improving runtime performance.

Traditionally, cache utilization is studied with either trace-driven or execution-driven simulations which feed memory accesses into a cache simulator. Unfortunately, simulation based methods are extremely slow, require substantial resources to develop an accurate simulator, and depend on intimate knowledge of the underlying hardware to implement.

Hardware mechanisms for throttling cache capacity offer an alternative way for measuring the relationship between cache capacity and runtime performance. Throttling cache capacity, even to the extreme of disabling cache utilization entirely, can be used to quantify the performance contributions of the cache. The next generation of Intel server processors will include Cache Allocation Technology (CAT) which is capable of throttling L3 cache capacity available to specific cores or processes [55]. This is implemented by controlling access to individual ways in the 16-way associate L3 cache. For example, a user can simulate the performance with a half-sized cache

by restricting the application from caching data in eight of the sixteen ways in the L3 cache.

3.1.3 Avoiding vector units

Vector processing units enable a single instruction to operate on multiple data elements simultaneously. Collapsing several scalar instructions into a single vector instruction can promote code compactness and improve performance as well as energy efficiency by increasing execution parallelism.

The Ivy Bridge processor used in this study supports MMX, Intel Steaming SIMD Extensions (SSE), and the Advanced Vector Extensions (AVX) instruction sets. The SSE instructions operate on up to 128-bit wide data elements (either four single precision or two double precision floating point numbers). The AVX data elements are two times wider than the SSE data elements, operating on up to eight 32-bit values or four 64-bit values at a time.

Although it is not possible to defeature Ivy Bridge's vector processing units, it is possible to quantify the performance benefits of the SIMD widths by limiting the ISA extensions permitted by the compiler. For example, recompiling the proxy_apps with the "–xSSE4.2" compiler flag will prevent the compiler from generating code with 256-bit wide SIMD instructions.

Under ideal conditions, doubling the SIMD width can double the runtime performance of an application; however, the larger SIMD width has only a modest impact on the performance on the prox_app workloads.

3.1.4 Disabling hyperthreading

Simultaneous Multi-Threading (SMT) technology improves runtime performance by filling pipeline bubbles with independent instructions from other hardware threads resident on the same core. These instructions can come from the same application (multi-threading) or from other applications (multi-processing). From a performance

perspective, SMT is beneficial because it can increase instruction throughput (IPC) by using an extra thread context to expose additional instruction level parallelism (ILP). In this case, over-provisioning the number of threads can increase utilization.

3.2 Software perturbations

Like hardware perturbation, small modifications to the software can be used to identify the source of performance limitations. In many cases, monitoring the change in performance after applying minor perturbations can be used to prove or disprove the existence of specific performance bottlenecks. We refer to this diagnostic technique as Pressure Point Analysis (PPA).

PPA constructs a series of experiments to isolate the contribution of a specific architectural feature through carefully designed code perturbations. Ideally, a battery of PPA experiments could be used to exhaustively test every possible performance bottleneck; however, the full battery of PPA tests are beyond the scope of this work. Instead, we only provide a couple examples as a proof of concept.

Section 3.2.1 and Section 3.2.2 provide examples of how PPA is used to diagnose common performance bottlenecks. Section 3.2.3 describes a strategy for studying the bottlenecks caused by cache utilization. Details of the machinery used to perform PPA are found in Section 3.3.

3.2.1 Bank conflicts

The Ivy Bridge microarchitecuture's L1D cache was designed to sustain two 128-bit loads and a 128-bit store every cycle. To achieve this throughput, each 64-byte line of the data cache is organized into eight 8-byte banks[1]. A bank conflict happens when two simultaneous load operations have addresses with the same 2-5 bits. When a bank conflict occurs, one of the two load operations will be delayed until the conflict

[1]The organization details of the Ivy Bridge L1D cache organization are speculated based on conclusions drawn from numerous performance experiments [37, 38, 55].

is resolved, which can degrade performance. In fact, the frequent occurrence of bank conflicts make it difficult to maintain the maximum L1D cache throughput in real-world applications. Furthermore, there are no penalties for reading and writing from misaligned memory operands, however, misaligned accesses occupy more banks which substantially increases the chances of bank conflicts.

The following example demonstrates a bank conflict. The instructions in Listing 3.1 load memory addresses that utilize the same bank because they have the same cache line offset. If these two instructions are dispatched in the same cycle, there will be a bank conflict. The instructions in Listing 3.2 can be processed simultaneously because they use different banks.

Listing 3.1: Example with a bank conflict

```
mov r8, [rsi]    ; Uses bank 0
mov r9, [rsi+64] ; Uses bank 0
```

Listing 3.2: Example without a bank conflict

```
mov r8, [rsi]    ; Uses bank 0
mov r9, [rsi+96] ; Uses bank 4
```

The Sandy Bridge performance monitoring unit tracks bank conflicts with the L1D_BLOCKS.BANK_CONFLICT_CYCLES event, which counts dispatched loads cancelled due to L1D bank conflicts with other load ports. However, the presence of bank conflicts does not imply a performance degration – some bank conflicts have no impact on performance because the stalled instruction is not part of the critical path. A more intensive approach is necessary to measure the performance degradation caused by bank conflicts.

3.2.2 Instruction decode rate deficiencies

The COSP2, CoEVP, and Lassen workloads have bottlenecks that originate in frontend of the processor. These stalls arrise from the frontend's inability to feed decoded

uops to the backend fast enough to keep the backend occupied. The frontend can fetch up to 16 bytes and decode up to four[2] instructions and produce a maximum of four uops per cycle. Although, there are several hazards that inhibit maximum throughput in the decode unit. For example, only one multi-uop instructions can be decoded each cycle.

Using a software perturbation technique called register scrambling, we can experimentally measure the instruction decode rate of code blocks to uncover potential bottleneck in the decode units. The main idea is to throttle the rate at which the backend of the core can process uops as a mechanism for revealing the rate at which the frontend is capable of supplying uops to the backend. The register scrambling transformation alters the instruction-level dependencies (ILP) of a kernel to affect the instructions per cycle (IPC) of the code. As the name suggests, instruction dependencies are artificially manipulated by changing the register numbers. The instruction mix remains constant, but the performance changes as the instruction-level parallelism changes.

Even the largest loop nests in the proxy_app microbenchmarks are small enough for the entire instruction sequence to fit comfortably within the uop cache and therefore do not depend on the frontend to supply uops to the backend; although, this is not necessarily the case when the basic block is executed within the context of full application. Loop unrolling is used to increase the number of instructions enough to exceed the capacity of the uop-cache, forcing each uop to flow through the frontend. In many cases, when the IPC is low, there is no performance benefit from utilizing the uop-cache because the uops are being supplied at a faster rate than the backend can process them. The register scrambling transformation is used to vary the rate at which the backend processes instructions. When the IPC of the scrambled kernel is less than

[2]In certain circumstances, the decoder can fuse an instruction with a subsequent branch instruction to produce a single compute-then-branch uop. This is known as Macro-op fusion. In this case, the frontend might be capable of decoding up to five instructions in a single cycle.

the instruction decode rate, the performance of the unrolled version that is fed by the L1-iCache is the same as the rolled version that utilizes the uop-cache. When the IPC of the scrambled kernel exceeds the instruction decode rate, the rolled version will outperform the unrolled version. Figure 10 shows the results from an experiment on a kernel. Ten scrambled versions of the kernel were generated to vary the IPC of the backend between 0.5 and 3 instructions per second. Rolled and Unrolled varient of each kernel were tested to compare the performance benefit of the uop-cache. As the plot shows, scrambled kernels with an IPC less than 2.4 had the same performance between the rolled and unrolled variants of the kernel. However, the performance between the rolled and unrolled variants diverged when the IPC exceeded 2.4. In the case of the unrolled kernels, the performance plateaus at 2.4 instructions per second because the instruction decode unit becomes the bottleneck.

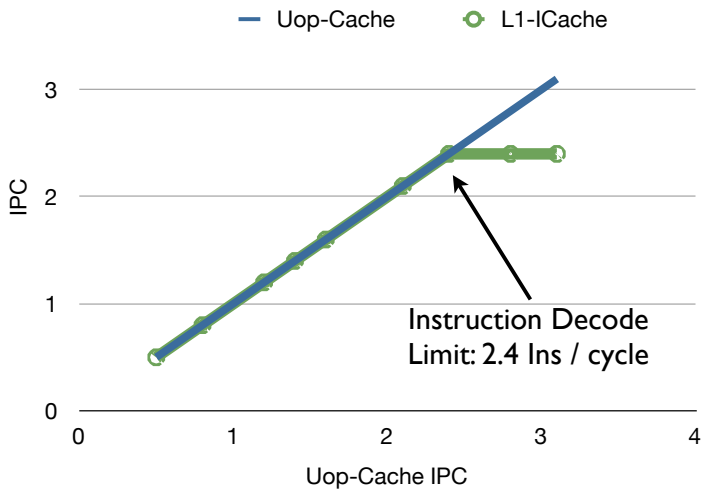

Figure 10: The decode limit.

3.2.3 Cache utilization

Over 30% of the instructions in the proxy_app workloads involve memory accesses. This level of throughput places enormous load on the memory hierarchy. High cache hit rates in the L1, L2, and L3 data caches as well as the L1-TLB and the L2-TLB are crucial to the performance of any application. To diagnose performance

limitations in the memory hierarchy, we rely on a set of experiments that use memory access rewriting to selectively control the cache hit rates of each individual memory operation.

The memory access rewriting exploits cache associativity to ensure that the memory operations of a specific instruction will always hit in a prescribed level of the cache hierarchy. The L1 data cache in the Ivy Bridge architecture is an 8-way set associative 32 KB cache with a 64-byte line size. The six least significant bits (bits 0-5) of a memory address specify the line offset. The next six bits (bits 6-11) determine which of the 64 L1 sets the memory address will be mapped into. Since the cache is 8-way set associate, only nine unique addresses are necessary to overflow the cache, if all nine of the memory addresses are mapped to the same cache set. For example, the memory accesses in the microbenchmark in Listing 3.3 will all hit in the L1 cache because there are only four unique addresses: 12+r9+8 in instruction 1, r13+r9+8 in instruction 2, r12+r9+16 in instruction 3, and r13+r9+16 in instruction 5. We can overflow the L1 cache sets by unrolling the loop nine times and increasing the offset value by an additional 4096 (2^{12}) bytes in each instance of the original loop. The memory accesses in the modified microbenchmark in Listing 3.4 will hit in the L2 data cache.

Listing 3.3: Microbenchmark for Kernel 11. All memory accesses will hit in the L1 cache.

```
0       loop:
1       inc       r10
2       vaddsd    xmm0, xmm0, QWORD PTR [r12+r9+8]
3       vmovsd    QWORD PTR [r13+r9+8], xmm0
4       vaddsd    xmm0, xmm0, QWORD PTR [r12+r9+16]
5       vmovsd    QWORD PTR [r13+r9+16], xmm0
6       add       r9, 0
7       cmp       r10, rbx
8       jb loop
```

Listing 3.4: Modified microbenchmark for Kernel 11. Memory accesses will hit in the L2 cache.

```
 0      loop:
 1      vaddsd    xmm0, xmm0, QWORD PTR [r12+r9+8]
 2      vmovsd    QWORD PTR [r13+r9+8], xmm0
 3      vaddsd    xmm0, xmm0, QWORD PTR [r12+r9+16]
 4      vmovsd    QWORD PTR [r13+r9+16], xmm0
 5      add       r9, 0
 6      vaddsd    xmm0, xmm0, QWORD PTR [r12+r9+4104]
 7      vmovsd    QWORD PTR [r13+r9+4104], xmm0
 8      vaddsd    xmm0, xmm0, QWORD PTR [r12+r9+4112]
 9      vmovsd    QWORD PTR [r13+r9+4112], xmm0
10      add       r9, 0
...
41      vaddsd    xmm0, xmm0, QWORD PTR [r12+r9+32776]
42      vmovsd    QWORD PTR [r13+r9+32776], xmm0
43      vaddsd    xmm0, xmm0, QWORD PTR [r12+r9+32784]
44      vmovsd    QWORD PTR [r13+r9+32784], xmm0
45      add       r9, 0
46      add       r10, 9
47      cmp       r10, rbx
48      jb loop
```

The ability to control cache utilization enables experiments to isolate the performance impact of each individual memory access operations. Unfortunately, interference from the Load Store Buffer in the Ivy Bridge microarchitecture have complicated the address rewriting process. So far, our attempts to rewrite the memory addresses have failed to produce the intended cache utilization behavior.

3.3 Automating software perturbations

The PPA Toolkit was developed to programmatically discover performance bottlenecks. It is an automated tool which uses PPA to detect different performance bottlenecks in a given application. It takes an x86_64 binary executable as input then runs various experiments on the application to find potential bottlenecks. Since the toolkit operates on binary x86_64 code, it is both language and compiler agnostic.

PPA Toolkit is an extensive Java application built with the Dropwizard framework and the maven package manager. The toolkit leverages several external tools including Gnu Binutils, Intel C Compiler (ICC), Intel Architecture Code Analyzer (IACA), X86 Encoder Decoded (Xed), Pin, and VTune. The source code is publicly available on bitbucket.

The PPA process can be broken into several steps: (1) identifying the hot loops from the target application, (2) extracting these loops into stand-alone microbenchmarks in assembly language, (3) generating a battery of microbenchmark variants that target specific architectural features, (4) running them in isolation while collecting various performance metrics, and (5) interpreting results to identify performance limitations.

3.3.1 Identifying hot loops

The analysis begins with two executions of the application. The first pass gathers profiling information about the application. The toolkit uses Intel's VTune Amplifier XE commandline tools to collect thousands of event samples per second, using the CPU_CLK_UNHALTED event as the trigger. The distribution of the samples are used to approximate the percent of runtime spent executing each specific instruction, although, because of "event skid" this attribution is only accurate at the granularity of code blocks [86]. In addition to collecting the instruction pointer (IP) of each event sample, the tool also collects the stack trace of each sample for tracking hot functions

and control flow.

The second pass leverages the instrumentation-driven simulation capabilities of Pin to investigate the dynamic control flow of the application. A custom-built Pin plug-in was developed to statically dissect the basic block structure of the code, collect control flow statistics, and gather the execution count for each basic block. The Pin plug-in collects three types of data.

1. **Identification of basic blocks**: Rather than relying on a purely static dissection of the application assembly to reconstruction the control-flow graph, which has difficulty handling dynamically generated addresses (e.g. indirect jump instructions), the Pin plug-in leverages Pin's data flow apparatus to dynamically track control flow during execution. The dynamic analysis also has the advantage of ignoring dead and unexercised code.

2. **Execution count of each basic block**: Pin is used to dynamically insert a hook into each basic block so that during runtime an execution-count of the basic block is incremented each time a basic block is executed. The execution-count can be combined with the profiling data to calculate the IPC of the basic block. The execution count of the basic blocks can also be used to infer the execution count of individual instructions within the block.

3. **Tracking ingress and egress of each basic block**: The Pin plug-in uses global state within Pin to maintain a pointer to the previously executed basic block. Each time a new basic block is entered, the ingress statistics of the current basic block and the egress statistics of the previous basic block are updated.

Listing 3.5 shows a hot loop from the XSBench workload that consumes 78.5% of the execution time. The loop implements a binary search over an ordered set of energies to find the index of array A that matches the "quarry" value. The compiled

assembly of that loop is shown in Listing 3.6. The loop is composed of five basic blocks: 404072-4040ac, 4040ae-4040b1, 4040b3-4040b7, 4040b9-4040b9, 4040bc-4040be. We define basic blocks as a continuous sequence of instructions that are guaranteed to execute if any other instruction in the basic block is executed. Branch, jump, function call, and function return instructions terminate the continuity of the basic block. The 4040b7-4040be block might look like a basic block, although, since the jump instruction at 4040b1 jumps into the middle of that block, 4040b9-4040b9 and 4040bc-4040be are separate basic blocks. The control flow graph of this loop is depicted in Figure 11.

Listing 3.5: Hot Loop in XSBench

```
while( max >= min )
{
        mid = min + floor( (max-min) / 2.0);
        if( A[mid].energy < quarry )
                min = mid+1;
        else if( A[mid].energy > quarry )
                max = mid-1;
        else
                return mid;
}
```

Listing 3.6: Hot Loop in XSBench

```
404072:       89 c1                mov    ecx,eax
404074:       c5 e9 57 d2          vxorpd xmm2,xmm2,xmm2
404078:       2b ca                sub    ecx,edx
40407a:       c5 d1 57 ed          vxorpd xmm5,xmm5,xmm5
40407e:       c5 eb 2a d1          vcvtsi2sd xmm2,xmm2,ecx
404082:       c5 d3 2a ea          vcvtsi2sd xmm5,xmm5,edx
404086:       c5 fb 59 da          vmulsd xmm3,xmm0,xmm2
40408a:       c4 e3 61 0b e3 01    vroundsd xmm4,xmm3,xmm3,0x1
```

```
404090:       c5 db 58 f5          vaddsd xmm6,xmm4,xmm5
404094:       c5 fb 2c ce          vcvttsd2si ecx,xmm6
404098:       48 63 c9             movsxd rcx,ecx
40409b:       48 8d 34 49          lea    rsi,[rcx+rcx*2]
40409f:       48 c1 e6 04          shl    rsi,0x4
4040a3:       c5 fb 10 14 3e       vmovsd xmm2,QWORD PTR [rsi+rdi*1]
4040a8:       c5 f9 2f ca          vcomisd xmm1,xmm2
4040ac:       76 05                jbe    4040b3 <binary_search+0x73>
4040ae:       8d 51 01             lea    edx,[rcx+0x1]
4040b1:       eb 09                jmp    4040bc <binary_search+0x7c>
4040b3:       c5 f9 2f d1          vcomisd xmm2,xmm1
4040b7:       76 11                jbe    4040ca <binary_search+0x8a>
4040b9:       8d 41 ff             lea    eax,[rcx-0x1]
4040bc:       3b c2                cmp    eax,edx
4040be:       7d b2                jge    404072 <binary_search>
```

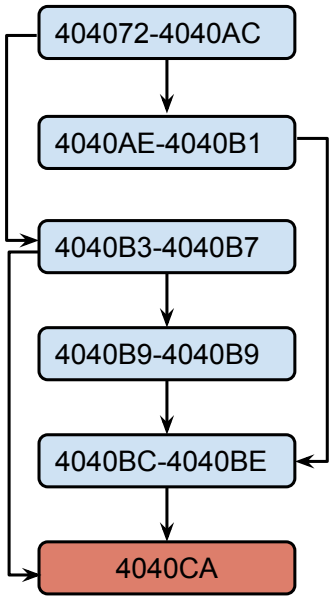

Figure 11: A diagram of the control flow through the basic blocks of the XSBench loop (See Figure 3.6).

3.3.2 Extracting loops

GNU's objdump tool is used to parse the binary object file into a sequence of instruction. This process involves identifying instruction boundaries —the x86_64 instruction set uses variable length encoding, each instruction is 1-15 bytes in size— and dissassembling the machine code into assembly instructions. The instructions are then decoded using a custom-built x86 instruction decoder built on Intel's Xed library. The Xed-based tool produces a data structure describing the opcode and operands, and flags, which form the instruction IR.

Listing 3.7 has an example of a the internal representation of the `MOV RCX,QWORD PTR [rip+0x2ae41a]` instruction. This IR contains information about the size, operation class, the type of the operands, the word size of the operands, the access mode (read versus write) of the operands, the data type of the operands, the visibility of the operands, the flag registers modified, the number of memory bytes accessed, as well as the base, index, offset, and scale of the memory operands.

Listing 3.7: Decoded representation of `MOV RCX,QWORD PTR [rip+0x2ae41a]`


```
{
  "Decode": "48 8b 0d 1a e4 2a 00 ",
  "Class":"MOV",
  "Category":"DATAXFER",
  "ISAExt":"BASE",
  "ISASet":"I86",
  "Operands": [
    {
      "OpNum":0,"Type":"REG0","Details":"RCX","VIS":"EXPLICIT",
      "RW":"W","Bits":64,"NumElem":1,"ElemSize":0,"ElemType":"INT"
    },
    {
      "OpNum":1,"Type":"MEM0","Details":"MEM","VIS":"EXPLICIT",
      "RW":"R","Bits":64,"NumElem":1,"ElemSize":64,"ElemType":"INT"
```

```
    }
  ],
  "MemoryOps":[
    {
      "MemOpNum":0,"RW":"R","Base":"RIP",
      "Displacement":2810906,"AWidth":64
    }
  ],
  "MemopBytes":8
}
```


Common register operands, flag operands, and memory accesses are used to infer dependencies between instructions. These dependencies are then used to construct a dependency graph of the instructions in a basic block.

After decoding the stream of instructions, the basic block must is passed through several instruction transformations to turn it into a stand-alone microbenchmark.

1. **Overwrite jump instruction**: The microbenchmark wraps the basic block inside a "jb" loop with a counter register that tracks the iteration count. All other jump instruction are removed. Instruction dependency graphs are consulted to identify and remove obsolete CMP/TEST instructions as well INC/DEC instructions that were associated with the jump instruction that was removed.

2. **Overwrite memory addresses**: Most of the basic blocks contain memory load and store operations, most of which calculate the memory address based on a base register, an index register, a scale value, and an offset. To keep the microbenchmark from accessing illegal memory addresses, (1) the base register needs to be initialized to a preallocated block of memory, (2) the index register needs to be initialized to zero, and (3) the base register value needs to be updated to negate the offset and scale values. To keep the memory address

stationary across each iteration, any other instructions in the basic block that write to the index register must be modified and consequently all instructions that indirecly impact the base register or index register must be modified. In the case of aligned load instructions (e.g. VMOVAPD), the base address, scale, and offset values must be checked to ensure the generated address is still 32-byte aligned.

3. **Recalculate RIP addresses**: Some memory addresses are listed using instruction pointer relative addressing. These instruction addresses must be rewritten so that the instruction will access a preallocated block of memory instead.

4. **Register initialization**: In most cases, data registers and memory blocks are cleared (initialized to zero), although, in some situations specific values are necessary to avoid illegal operations. For example, dividing by zero will result in a floating-point exception, to avoid this situation the divisor should be set to one instead of zero.

The mustache templating language is used to generate a C source code file with the inline assembly and initialization embedded into a "void kernel(int iterations)" function. The C code can be compiled with the Intel C Compiler (ICC), linked with a test harness, then executed.

3.3.3 Generating perturbations

The PPA Toolkit is an extensible framework capable of implementing various types of bottleneck experiments. A new set of experiments can be implemented by extending the Experiment Java class. The Toolkit API provides access to the instruction IR, the instruction dependency graph, and tools for exporting a set of instructions into a C source file.

3.3.4 Performance experiments

The toolkit generates microbenchmarks for each of the basic blocks as well as each perturbation of the associated basic block. The microbenchmarks are run within a test harness which runs the basic block loop for a billion iterations. Most of the microbenchmarks take less than a second to execute. The cycle count is measured then divided by the number of iterations to calculate the average number of cycles per iteration of the basic block. The number of cycles per iteration is used to compare the performance of the microbenchmark against each of the perturbations.

3.4 Related Work

The notion of code perturbations used by Pressure Point Analysis is inspired by stochastic optimization techniques. Seminal work by Schkufza et al. showed that by relaxing the constraint that compiler transformations must preserve semantic correctness, random code mutations enabled their superoptimizing compiler to discover radically new implementations of targeted computations [99]. Schulte et al. also use random code mutations to discover optimizations that improve energy efficiency [100].

Knights et al. pioneered the concept of Blind Optimizations by improving application performance simply by inserting random sequences of nop instructions into the code stream. They found that the Pentium 4 was highly sensitive to code alignment and that the nops shifted the code alignment enough to expose performance cliffs caused by alignment hazards. Hundt incorporated the concept of blind optimization into their Extensible Micro-Architectural Optimizer (MAO) project to semi-automatically discover microarchitectural hazards in the Intel Core-2 processor.

The GREMLINS project by the ExMatEx group employs various techniques to degrade processor resources in an effort to understand how future machine designs will impact application performance. Interference processes, called gremlins, steal compute resources from the target application. The gremlins can consume parts of

the available power budget, memory bandwidth, cache capacity, or network bandwidth resources [101]. Eklov et al. developed a similar concept, called the Bandwidth Bandit, which uses an interference process to steal memory bandwidth from a target application. They use the bandit process to model application performance as a function of available memory bandwidth [31]. The PPA methodology uses a similar approach of experimentally testing the impact of resource degradation, but PPA operates on lower-level resources and depends on more precise mechanisms for restricting resource usage.

The Camino compiler infrastructure project by Hu et al. uses sophisticated profiling and static analysis techniques, similar to the PPA Toolkit, to suggest a set of possible code placement optimizations. However, rather than directly identifying the performance bottlenecks, Camino relies on "behavior characterizations" to enumerate a set of transformation that might improve performance. Overall, the goal of the Camino is to serve as a testbed for various low-level optimizations, rather than a diagnostic tool for identifying performance issues.

CHAPTER IV

ASSESSING THE IMPACT OF THE MICROARCHITECTURE

4.1 Introduction

Power and Energy have always had a significant impact on processor design. Yet, as the demand for energy efficient computing increases and rate of improvement from process technology slows, energy efficiency has become a first-class design constraint. Prior work has shown that increasing single-threaded performance through microarchitectural advances tends to increase power [6, 46]. This increase is due to the increased complexity of the core such as a larger instruction window and more aggressive use of speculation. However, this increase can be countered by reductions in energy due to improvements in process technology, architecture, and microarchitecture. This paper examines the improvements in each of these areas. We use measurements from six generations of Intel® Core™ processors running a variety of vectorizable workloads. We show that through the combination of these techniques, both the time and the energy required to run the workloads has been reduced.

This paper makes three contributions to the understanding of energy efficiency.

1. **Methodology**: We present a methodology for empirically measuring instruction-level energy efficiency on a modern processor. This includes techniques for isolating architectural features as well as a novel technique for studying the relationship between performance and power.

The material in this chapter has appeared in a separate refereed publication [24].

2. **Attribution**: By isolating key architectural features, we account for many of the variable costs which impact energy efficiency. This provides valuable insight into the driving forces behind energy efficiency on a real-world processor.

3. **Evolution of Energy Efficiency**: Using a longitudinal study of the Intel® Core™ processor, we track the impact architectural innovations have had on energy efficiency. This motivates a discussion about the future strategies for improving energy efficiency on a Big Core.

4.2 Methodology

Our goal is to understand the impact architectural features have on performance and energy efficiency of real-world processors. Ideally, a properly controlled experiment would involve designing and manufacturing processors, with different permutations of features, across several process technology nodes; however, this approach is infeasible. Instead, we rely on comparisons of existing processors. To control the variables we focus on successive generations of a single architecture family, where each processor has a similar base architecture with modest incremental changes in architecture features each generation. With the proper setup, these cross-generational comparisons provides a unique before-and-after evaluation of newly added architectural features.

This section provides more detailed descriptions of our experimental methodology and setup.

4.2.1 Processors

Table 6 lists the processors used in the study: Penryn (PNY), Nehalem (NHM), Westmere (WSM), Sandy Bridge (SNB), Ivy Bridge (IVB), and Haswell (HSW). In total, our data spans six years, six generations, four major microarchitecture revisions, and three process technology nodes.

We selected flagship processor models from the high-end enthusiast segment rather

than trying to match features exactly. Fortunately, the clock speed, core count, and last level cache capacity are relatively similar. The L1 data cache is a constant 32 KB across all processors. The largest discrepancy is WSM, which, for business reasons, was designed with two extra cores. For consistency, we disabled those cores and treated WSM as a four-core processor. To account for the minor clock frequency difference, performance is measured in cycles rather than time. Finally, since all memory accesses in these experiments are confined to the L1 cache, slight differences in the rest of the cache hierarchy, mid-level cache (MLC), last-level cache (LLC), memory controller, and the rest of the uncore should not have a significant impact on results.

4.2.2 Architecture Features

The core features we studied include (i) the frontend caches, which include the loop cache and the micro-op cache (ii) the out-of-order resources, which include the execution units, issue port, and the number of reorder buffer entries and (iii) the SIMD execution unit and ISA extension. Details of these features are listed in Table 7.

4.2.3 Kernels

We use the Livermore Loops benchmark suite [90] (see Table 8), a collection of compute intensive kernels extracted from scientific applications used by the Lawrence Livermore National Laboratory, to evaluate the processors.[1] The Livermore Loops derive from actual high-performance computing applications, yet are small enough to instrument, study, and control manually. We also acknowledge that the kernels are not necessarily the optimal C code implementation of the computation, but are instead canonical examples of typical scientific code. Note, the original Livermore Loops benchmark set contains 24 kernels, however, we had to omit five kernels

[1]Note, all floating-point calculations within these kernels are based on double-precision, not single-precision, floating-point types.

Table 6: The six processor models used in this study.

Model	Core X9650	Core i7 975	Core i7 980X	Core i7 2700K	Core i7 3770K	Core i7 4770K
Code Name	Penryn (PNY)	Nehalem (NHM)	Westmere (WSM)	Sandy Bridge (SNB)	Ivy Bridge (IVB)	Haswell (HSW)
Release Cycle	Tick	Tock	Tick	Tock	Tick	Tock
Core Microarchitecture	Core	Nehalem	Nehalem	Sandy Bridge	Sandy Bridge	Haswell
Process Node	45 nm	45 nm	32 nm	32 nm	22 nm	22 nm
Frequency	3.00 GHz	3.33 GHz	3.33 GHz	3.5 GHz	3.5 GHz	3.5 GHz
Cores	4	4	6	4	4	4
LLC	12 MB	8 MB	12 MB	8 MB	8 MB	8 MB
TDP	130 W	130 W	130 W	95 W	77 W	84 W
Release Date	Q4'07	Q2'09	Q1'10	Q1'11	Q2'12	Q2'13

because they do not fit basic loop structure used in our experimental framework.

For our experiments, we compiled each of the loops with the Intel® C Compiler 13.0.0.079, extracted the assembly instructions from each loop nest, then made minor modifications to remove any dependence of the loop body on any explicit loop iteration variables. These modifications provide additional control over execution by ensuring all memory accesses hit in the L1 cache, allowing us to run the loop for an unlimited number of iterations, and controlling the input values of all operations. Figure 12 shows the original Livermore Loops C code, the compiled assembly, and the modified version of the assembly for Kernel 7. The lines in red highlight the modifications to the compiled assembly, which confine the memory access pattern to a smaller working set by keeping register *rdi* constant. Although it is not shown in the figure, we also initialize all of the registers and memory locations that are touched by the kernel to ensure that dynamic behavior remains constant each iteration. Figure 12 also contains a table with a sample of the performance, power, and energy data collected

Table 7: Architectural Features

	Core	Nehalem	Sandy Bridge	Haswell
L1 Bandwidth (load, store)	16, 16 Bytes per cycle	16, 16 Bytes per cycle	32, 16 Bytes per cycle	64, 32 Bytes per cycle
Instruction Cache	32KB L1 Icache	32KB L1 Icache	32KB L1 Icache, 1.5K uop cache	32KB L1 Icache, 1.5K uop cache
Reorder Buffer	96 entries	128 entries	168 entries	192 entries
Ins/Uop Queue	32 ins	28 uops	28 uops (56 on IVB)	56 uops
SIMD Extensions	SSE	SSE	AVX	AVX2

for Kernel 7.

The resulting kernels have between 12 and 183 instructions inside a single for loop, which we run for several billion iterations. This enables us to make very precise measurements of performance and power. We also use performance counters and simulators to analyze runtime behavior.

For illustration purposes, we also added an additional kernel to the original nineteen kernels: Kernel 20 – Peak floating-point throughput. This kernel is designed to attain the theoretical peak floating-point throughput. Both a vector multiply (mulpd) and vector add instruction (addpd) are issued each cycle. There are no memory accesses or auxiliary instructions in this kernel. We do not include Kernel 20 when calculating averages in the results section.

4.2.4 Experimental Platform

Our results are based on empirical measurements of CPU power, which are measured by instrumenting the 12 volt rail that feeds the voltage regulator which in turn feeds the processor. To account for minor voltage fluctuations, both voltage and current are measured to calculate power. A National Instruments® cDAQ-9174 with a 9229 module is used to sample power at a rate of $2KHz$. This device is managed by a

Table 8: The 20 kernels evaluated in this study.

Kernel	Description
1	Hydro fragment
2	Incomplete Cholesky Conjugate Gradient
3	Inner product
4	Banded linear equations
5	Tri-diagonal elimination, below diagonal
6	General linear recurrence equations
7	Equation of state fragment
8	Integrate predictors
9	Difference predictors
10	First sum
11	First difference
12	2-D PIC (Particle In Cell)
13	1-D PIC (Particle In Cell)
14	ADI integration
15	2-D explicit hydrodynamics fragment
16	General linear recurrence equations
17	Discrete ordinates transport
18	Matrix-matrix product
19	2-D implicit hydrodynamics fragment
20	Peak floating-point throughput

Kernel #7 (Equation of State)

Livermore Loops C Code:

```
for ( k=0 ; k<n ; k++ ) {
        x[k] = u[k] + r*( z[k] + r*y[k] ) +
             t*( u[k+3] + r*( u[k+2] + r*u[k+1] ) +
             t*( u[k+6] + r*( u[k+5] + r*u[k+4] ) ) );
}
```

	Cycles Per Iteration	CPU Power (Watts)	Energy Per Iteration
PNY	23.0	60.5	116 nJ
NHM	22.5	73.2	124 nJ
WSM	22.5	50.5	85 nJ
SNB	18.3	69.0	90 nJ
IVB	18.3	34.3	45 nJ
HSW	17.3	40.1	50 nJ

Compiled Assembly:

```
innerloop:
vmulpd   ymm15, ymm2, [32+r15+rdi*8]
vmovupd  xmm7,  [8+r15+rdi*8]
vmovupd  xmm13,  [40+r15+rdi*8]
vmulpd   ymm3, ymm2,  [r12+rdi*8]
vmovupd  xmm6,  [24+r15+rdi*8]
vaddpd   ymm4, ymm3, [r10+rdi*8]
vmulpd   ymm5, ymm2, ymm4
vaddpd   ymm0, ymm5, [r15+rdi*8]
vinsertf128 ymm8, ymm7, [24+r15+rdi*8], 1
vinsertf128 ymm14, ymm13, [56+r15+rdi*8], 1
vmulpd   ymm9, ymm2, ymm8
vaddpd   ymm13, ymm14, ymm15
vaddpd   ymm10, ymm9, [16+r15+rdi*8]
vmulpd   ymm14, ymm2, ymm13
vmulpd   ymm12, ymm2, ymm10
vaddpd   ymm15, ymm14, [48+r15+rdi*8]
vmulpd   ymm4, ymm1, ymm15
vinsertf128 ymm11, ymm6, [40+r15+rdi*8], 1
vaddpd   ymm3, ymm11, ymm12
vaddpd   ymm5, ymm3, ymm4
vmulpd   ymm6, ymm1, ymm5
vaddpd   ymm0, ymm0, ymm6
vmovupd  [r13+rdi*8], ymm0
add   rdi, 1
cmp   rdi, r9
jb    innerloop
```

Modified Assembly:

```
innerloop:
vmulpd   ymm15, ymm2, [32+r15+rdi*8]
vmovupd  xmm7,  [8+r15+rdi*8]
vmovupd  xmm13,  [40+r15+rdi*8]
vmulpd   ymm3, ymm2,  [r12+rdi*8]
vmovupd  xmm6,  [24+r15+rdi*8]
vaddpd   ymm4, ymm3, [r10+rdi*8]
vmulpd   ymm5, ymm2, ymm4
vaddpd   ymm0, ymm5, [r15+rdi*8]
vinsertf128 ymm8, ymm7, [24+r15+rdi*8], 1
vinsertf128 ymm14, ymm13, [56+r15+rdi*8], 1
vmulpd   ymm9, ymm2, ymm8
vaddpd   ymm13, ymm14, ymm15
vaddpd   ymm10, ymm9, [16+r15+rdi*8]
vmulpd   ymm14, ymm2, ymm13
vmulpd   ymm12, ymm2, ymm10
vaddpd   ymm15, ymm14, [48+r15+rdi*8]
vmulpd   ymm4, ymm1, ymm15
vinsertf128 ymm11, ymm6, [40+r15+rdi*8], 1
vaddpd   ymm3, ymm11, ymm12
vaddpd   ymm5, ymm3, ymm4
vmulpd   ymm6, ymm1, ymm5
vaddpd   ymm0, ymm0, ymm6
vmovupd  [r13+rdi*8], ymm0
add      r8, 1
cmp      r8, r9
jb innerloop
```

Figure 12: This figure shows Kernel 7 and a sample of the data collected for this kernel. To conserve space, the assembly code listings contains the AVX version (26 instructions) instead of the longer SSE version (51 instructions); however, the data table contains values collected from the SSE version.

second workstation to ensure that data acquisition does not affect the target machine.

Each system runs a stock Ubuntu 12.04 Server installation – no additional attempts were made to optimize the operating system. The processors are set to run at factory prescribed frequency with Intel® Turbo Boost Technology and Hyperthreading features disabled. The benchmarks were compiled using the Intel® Composer XE version 2013.0.079 tool chain [54].[2]

Preliminary experiments have shown that a rise in processor temperature can increase power by as much as five watts. To minimize this impact, benchmarks are run

[2]Intels compilers may or may not optimize to the same degree for non-Intel processors for optimizations that are not unique to Intel processors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Processor-dependent optimizations in this product are intended for use with Intel processors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel processors. Please refer to the applicable product User Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804.

continuously for a five minute warmup period to allow the temperature to stabilize before starting the experiment. Tests are conducted by running the benchmark continuously for five minutes, immediately following the five minute warmup period. Power and performance values are averaged across the entire five minute trial. Repeated runs of the same experiment show the experimental precision of power measurements to be within 0.05 watts.

When measuring power, we run an instance of the kernel on each core to amplify the dynamic power consumption. Therefore, the measured power values presented in this paper represent the total power of the processor when each of the four cores are running an instance of the kernel. However, when we refer to IPC we are referring to statistics of an individual core.

4.2.5 Definition of Energy Efficiency

This paper is primarily focused on energy efficiency. Broadly speaking, the objective is to minimize the total amount of energy consumed by the processor to complete a particular computation. We calculate energy consumption (measured in joules) by measuring the average power of the processor (measured in watts; note, 1 $watt = 1\frac{joule}{sec}$) and multiplying by the duration of the computation (measured in seconds but reported in clock cycles). For example, the average energy consumed per iteration of Kernel 7 (see Figure 12) on HSW is[3]

$$(17.3\ cycles)(\frac{1sec}{3.5 \times 10^9\ cycles})(10.0\ W)\ =\ 50\ nJ. \tag{6}$$

Energy efficiency is inversely proportional to energy consumption. It is generally expressed as performance over power,

$$\frac{1}{energy}\ =\ \frac{1}{duration \times power}\ =\ \frac{performance}{power}. \tag{7}$$

[3]To get the cost of a single iteration, we must also divide processor power by four since the benchmark is running simultaneously on each of the four cores (see Section 4.2.4).

If performance is constant then the change in energy efficiency is proportional to the change in power. For example, based on the data in Figure 1, SNB and IVB have identical performance on Kernel 7. The ratio of average power shows that IVB is 2.0x more efficient than SNB.

In many situations, performance and energy efficiency are treated as competing goals since it is generally possible to improve performance at the expense of energy efficiency and vice versa. Occasionally, an artificial metric that incorporates both performance and energy efficiency, such as the energy delay product [45], is used to determine whether a trade-off between the two is beneficial. Fortunately, this paper does not explicitly encounter this trade-off and therefore does not resort to alternative metrics.

4.2.6 Register Scrambling

The analysis in this paper includes a novel technique, called register scrambling, to artificially alter the instruction-level dependencies of a kernel. As the name suggests, it involves randomly assigning new SSE/AVX register numbers to each instruction. For example, Figure 13 shows two examples of Kernel 1 with the registers scrambled. The instruction mix remains constant, but the performance and power change as the instruction-level parallelism changes. By generating several different "scrambles" of the kernel we are able to generate a regression that relates performance to power. Figure 14 shows the relationship between performance and power of Kernel 1 on HSW. This regression can be used to evaluate improvements in the core architecture by estimating how much a change in performance will affect the energy efficiency.

4.3 Experimental Results

Figure 15 presents the main results of this paper. Averaging across all of the Livermore Loop kernels, there is a 2.9x improvement in energy efficiency from PNY to HSW.

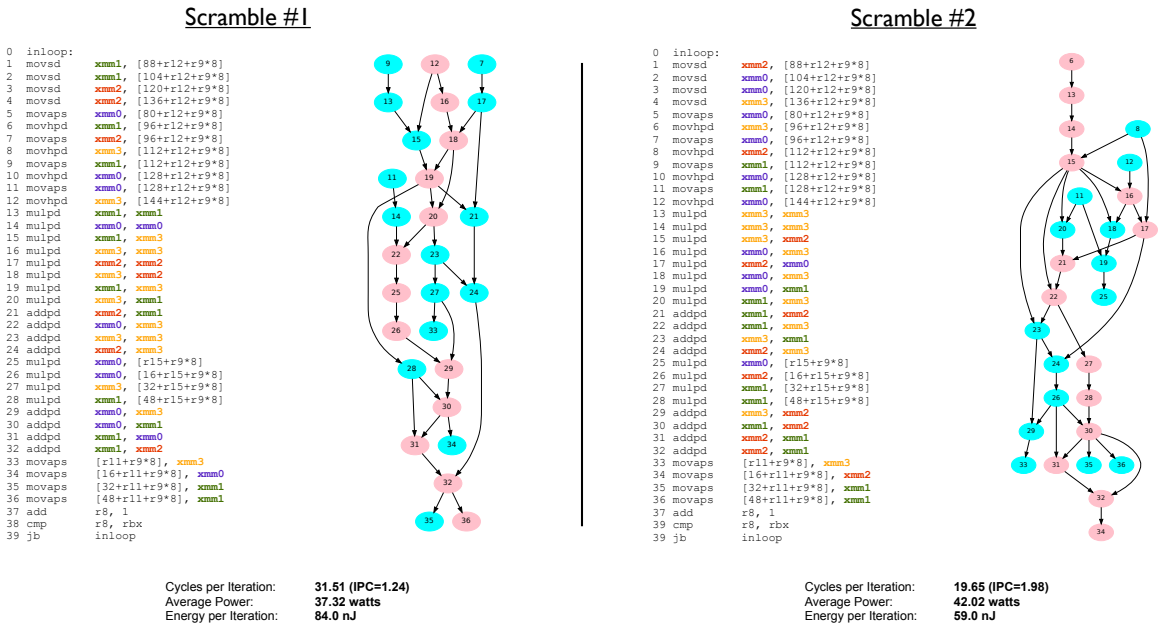

Figure 13: An example of two "scrambled" versions of Kernel 1. This figure also includes sample data from HSW.

Furthermore, this section drills-down on the source of the improvements in energy efficiency by progressively removing the benefit of recent architectural features. This is completed as a five step process:

1. **SIMD Extensions**: we restrict the use of relevant ISA extensions that have been added since PNY: the AVX and AVX2 SIMD extensions.

2. **Frontend Innovations**: we prevent the processor from utilizing new frontend features. The primary innovations since PNY are the addition of a micro-op cache and improvements to the loop caches.

3. **Backend Innovations**: we estimate the impact performance improvements due to backend innovations —such as additional execution units and larger instruction windows— have on power and energy efficiency.

4. **22nm Process Technology Node**: we rollback the advantage of the 22nm process technology to estimate power and energy efficiency on the 32nm process technology node.

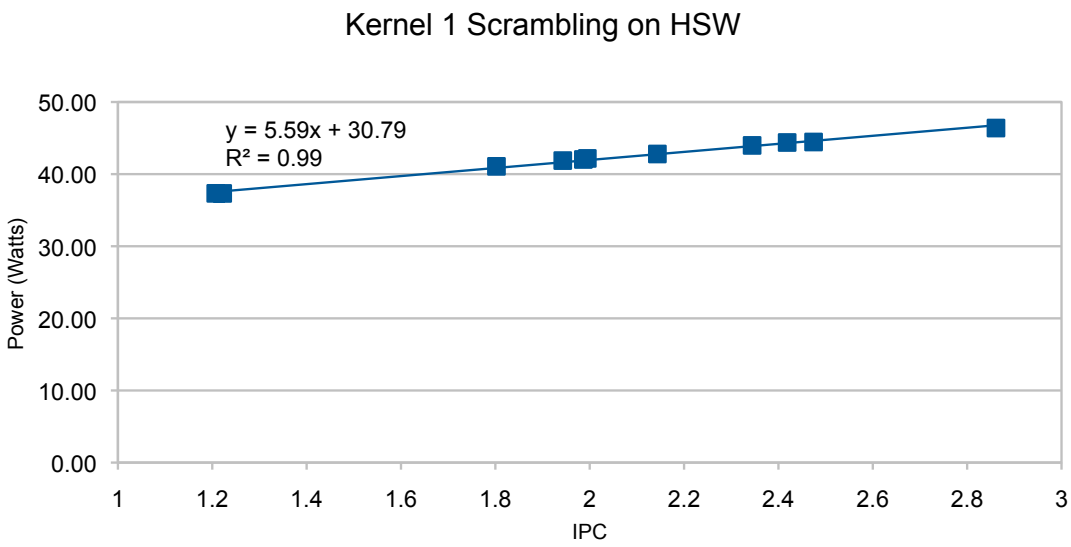

Figure 14: A regression of several scrambles of Kernel 1 on HSW.

5. **32nm Process Technology Node**: we rollback the advantage of the 32nm process technology to estimate power and energy efficiency on the 45nm process technology node.

Figure 15 shows the improvement in energy efficiency after each step. For example, the HSW plot shows that the improvement in energy efficiency drops from 2.9x to 2.1x after removing the use of SIMD extensions (step 1). Similarly, after also removing the impact of frontend innovations (step 2), the improvement drops from 2.1x to 1.8x. After, removing the benefits of the SIMD Extensions, frontend, and backend as well as the process technology, we are left with the "base" energy efficiency. The "base" value represents the efficiency of HSW if it pays the overhead cost of implementing all of its architectural features —such as the micro-op cache and additional execution units— but does not benefit from any of them.

It is important to note that this process is conducted in an additive process – the changes made in previous steps are retained in the following steps. This methodological decision was made out of necessity since each processor needs to be running identical code in order to properly compare the frontends. Similarly, we need to

neutralize changes in the frontend and ISA before comparing the backends.

The details of each step are described in the following subsections.

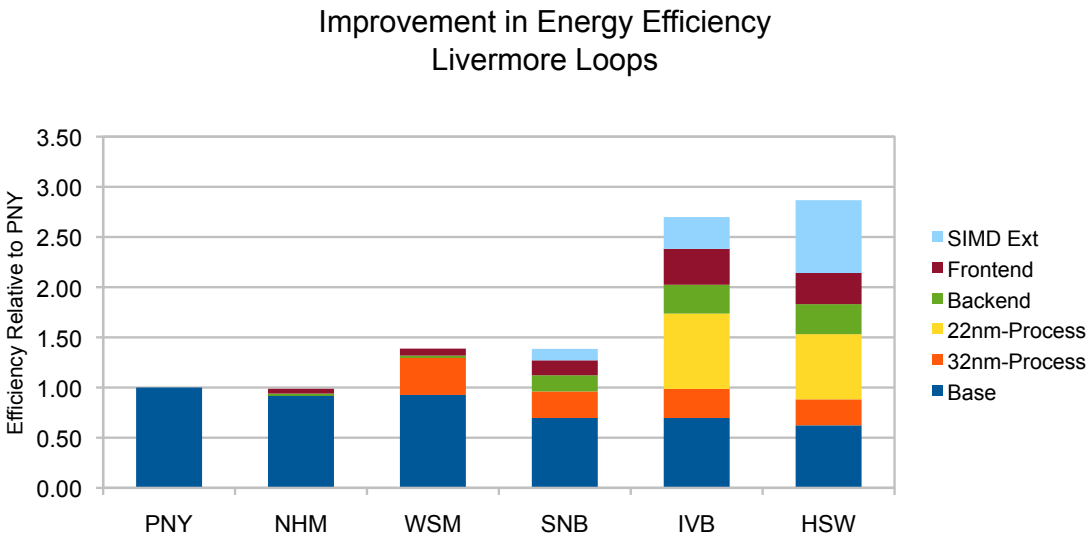

Figure 15: Average improvement in energy efficiency across the Livermore Loops kernels.

4.3.1 SIMD Extensions

The AVX extensions introduce three important features to the ISA that are relevant to the Livermore Loops: (1) four-wide SIMD vector instructions (double the SIMD width of SSE instructions) (2) non-destructive instructions, and (3) 256-bit load and store instructions (double the width of SSE instructions) [36]. In addition, the AVX2 extensions also provide fused multiply-add (FMA) instructions which double the peak floating-point throughput and gather instructions (e.g., vgatherdpd) for vectorizing non-adjacent memory accesses.

To evaluate the impact of the AVX and AVX2 extensions, we create multiple versions of each kernel. The SSE version —the baseline version— is created by constraining the compiler to only generate code that is supported by SSE4.1 machines, then extracting the resulting loop body. The AVX and AVX2 versions are generated by allowing the compiler to take advantage of the additional AVX instructions. We

gauge the effectiveness of the extensions by comparing the performance and energy efficiency of the resulting versions.

Results from Kernel 20 (the peak floating-point throughput kernel) on HSW demonstrate the potential benefits of SIMD extensions. Doubling the SIMD width with AVX instructions doubles performance and only increases total power by 4.3%, which results in a 1.9x improvement in energy efficiency. Similarly, the use of the FMA instructions can double the performance and only increase power by 5.0%, which also nearly doubles energy efficiency. By fully exploiting both the FMA instructions and the wide SIMD width of the AVX instructions, HSW can achieve up to 6.3 GFlops/watt, a 7x improvement over the 0.9 GFlops/watt on PNY.

Unlike the ideal conditions, the AVX and AVX2 extensions have only a moderate impact on the Livermore Loops kernels. Eleven of the nineteen kernels (1, 2, 3, 4, 7, 8, 10, 12, 13, 14, and 18) benefit from the additional SIMD instructions – the other kernels have structural dependencies which make vectorization difficult. Of the kernels that do benefit, energy efficiency improves 4-56% and performance improves 7-83%. In each of these cases the improvement in performance is greater than the improvement in energy efficiency primarily because utilizing SIMD instructions often requires a little extra work to shuffle data into and out of the proper SIMD lanes. Interestingly, in two of the kernels (Kernel 4 and Kernel 9), the use of AVX instructions had a negative impact on both performance and energy efficiency even though the AVX version of the kernel uses fewer instructions. Averaging across all kernels, the AVX extensions delivers a 21% improvement in performance and a 16% improvement in energy efficiency. Utilizing both AVX and AVX2 instruction on HSW provide a 26% improvement in performance and a 21% improvement in energy efficiency over the SSE version.

4.3.2 Frontend Features

The frontend is responsible for fetching and decoding instructions to feed the execution engines in the backend. In an out-of-order architecture, the frontend is integrated with a branch prediction unit so it can fetch and decode instructions down speculative paths in order to keep the backend busy. In addition, modern processors are equipped with loop detectors and instruction caches to reduce the number of instructions that must be fetched and decoded, which improves both performance and energy efficiency. This section examines how the evolution of the frontend has impacted energy efficiency.

4.3.2.1 Loop Caches

The loop cache exploits the temporal locality of instructions to reduce the burden on the frontend [10, 74]. When used, instructions from inside a loop nest are streamed directly from the Instruction Queue (IQ) or Micro-op Queue (MQ), skipping earlier stages of the pipeline. In PNY, the cache is located between the instruction predecode and the instruction decoders. In subsequent generations the cache is located after the decode units, allowing the frontend to power-down the instruction fetch and decode units when streaming from the MQ [105].[4] Ultimately, the loop cache improves performance by eliminating potential frontend bottlenecks and reduces power by eliminating redundant work.

4.3.2.2 Mico-op Cache

In addition to the loop cache, SNB, IVB, and HSW also include a micro-op cache in the frontend, between the loop cache and the L1 instruction cache. It provides many of the benefits of the loop cache but has a higher capacity (1500 micro-ops versus the

[4]Since the loop cache is located before the decoders in PNY, it caches instructions in the Instruction Queue (IQ); whereas later generations cache decoded micro-ops in the Micro-op Queue (MQ).

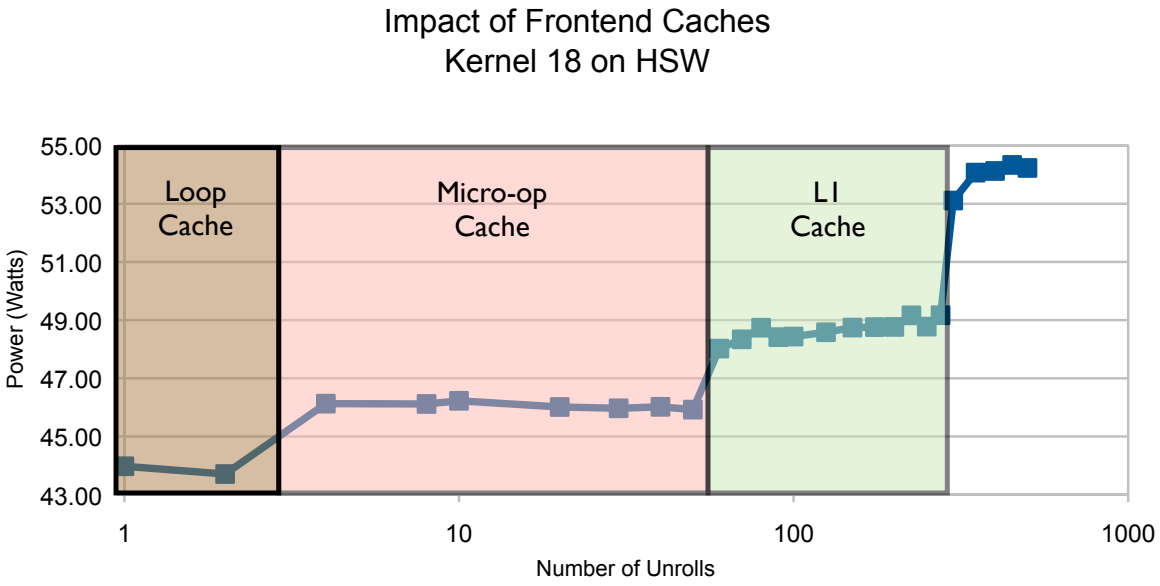

Figure 16: This figure demonstrates the results from manually unrolling Kernel 18 on HSW. Kernel 18 has 19 instructions, not counting the branch instruction. When unrolls=10 the loop nest contains 190 instructions. Performance remains constant until the instruction stream exceeds the capacity of the L1 instruction cache at unrolls =300.

56 micro-ops in the loop cache). Like the loop cache, the micro-op cache can reduce the number of instruction decodes by caching decoded instructions. However, unlike the loop cache, the micro-op cache exists earlier in the pipeline and therefore does not bypass the branch prediction unit. The results in Figure 16 and Figure 17 show that utilizing the loop or micro-op cache can reduce the power by several watts but the power saving advantage of the loop cache over the micro-op cache is relatively low.

We measure the impact of the loop cache and the micro-op cache by comparing the original kernel with a version that has been manually unrolled to the point that the instruction stream exceeds the capacity of the caches. In most cases the kernel only needed to be unrolled two to four times before the loop nest exceeded the capacity of the loop cache; however, some of the kernels are too large to fit in the loop cache even without unrolling. Table 7 lists IQ capacity of the different processors. Performance counters are used to verify the utilization of the loop cache.

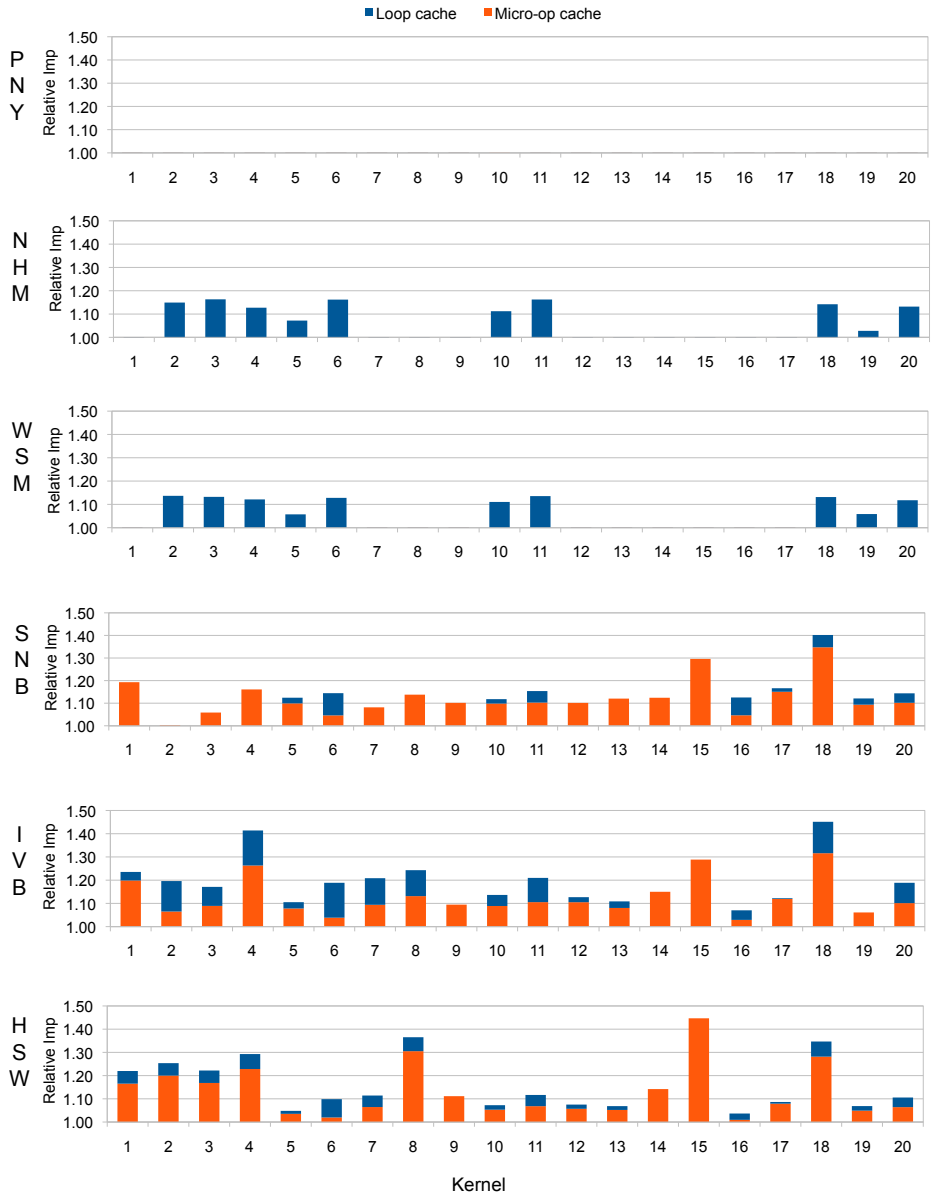

Figure 17: Improvement in energy efficiency from the micro-op and loop caches. Blue denotes the contribution of the loop caches and orange denotes the contribution of the micro-op caches.

Figure 17 shows the results from the loop unrolling. In the case of PNY, only one kernel (Kernel 11) fits into the loop cache because of the size of most kernels exceeds the limited capacity of the cache. The additional capacity in NHM and WSM increases the number of kernels that can utilize the loop cache to six. We also notice that locating the loop cache after the decode stage of the pipeline has increased

the energy savings from 5% in PNY to 15% in NHM and WSM. Similarly, the larger MQ in SNB, IVB, and HSW further increases the number of kernels that utilize the loop cache; however, in this case the improvement in energy efficiency is relatively small because the unrolled kernels only fall back to the micro-op cache, still avoiding the fetch and decode phases of the frontend.

4.3.3 Backend Features

Comparing HSW to PNY with the SSE version of the Livermore Loops, there has been a 1.4x improvement in per-cycle performance which can be attributed to architectural improvements to the backend. While increasing performance is beneficial on its own, it also affects energy efficiency by reducing execution time and increasing power (recall Equation 7). In this section, we analyze the source of improvements in performance and energy efficiency of the backend.

4.3.3.1 Additional Execution Units

Superscalar processors exploit instruction-level parallelism by issuing multiple instructions per cycle. It improves performance and also improves energy efficiency by reducing the execution time. Since PNY, several additional execution units have been added to the core microarchitecture. The most relevant additions are the second load unit added to SNB, which doubles the L1 load bandwidth, and several redundant execution units added to HSW.

4.3.3.2 Additional Out-of-order Scheduling Resources

To take advantage of additional execution units, the out-of-order scheduling resources must be increased to augment the discovery and scheduling of independent instructions. Among these resources, the number of reorder buffer (ROB) entries directly affects the number of parallel instructions in flight. The ROB holds the operands necessary for the instructions' operations and hold the results before they are written

back to the architectural registers. More ROB entries allow more concurrent instructions in the pipeline. The instruction window controls the number of instructions in the execution flow that the processor can analyze for parallel execution. A larger instruction window allows the processors to examine more instructions, which increases the chances of finding independent instructions to be executed concurrently. Table 7 shows the increase in the size of the reorder buffer and the instruction window from PNY to HSW.

4.3.3.3 Backend Experiments and Results

Ideally, the proper approach to quantifying the benefit of the backend improvements is to defeature a processor to make its backend behave like the previous generation's backend. Unfortunately, we do not have the type of controls in modern processors to make this work. Instead, we approximate this approach by manually isolating changes in the backend. For these experiments we use the SSE version of the kernels and unroll them enough so the instruction streams do not fit into the micro-op caches, thereby eliminating affects from the SIMD extensions and frontend. Furthermore, to remove the impact of the manufacturing process, we compare processors from the same process node (i.e., NHM to PNY, SNB to WSM, and HSW to IVB).

Overall, backend features improved performance more than energy efficiency. From PNY to NHM, five of the 19 kernels improved performance, but only three improved energy efficiency. From WSM to SNB, 13 of the 19 kernels improved performance and only three improved energy efficiency. From IVB to HSW, 12 of the 19 kernels improved performance and only two improved energy efficiency. The reason behind these results is that backend improvements require a substantial increase in transistor count which leads to a significant increase in capacitance and power to toggle them.

Using performance counters and architecture simulators [56, 57], we study the reduction in pipeline stalls from one generation to the next. Based on this information, we can discern what feature is responsible for the change in performance. From PNY to NHM, we found that the performance benefits mainly come from increased scheduling resources. From WSM to SNB, we found that the majority of the performance benefits come from the addition of a second load port. Many of the kernels are L1 bandwidth limited on NHM, therefore the additional port alleviates this performance bottleneck. From IVB to HSW, we found that several of the kernels that benefited from the additional load port in SNB now benefit from the additional scheduling resources as the L1 bandwidth is no longer a significant performance bottleneck.

Finally, we use Equation 7 to calculate the effect these performance improvements have on energy efficiency. Since these performance improvements will increase power due to the increase in utilization, we use the Register Scrambling technique (see Section 4.2.6) to map changes in performance to changes in power. For example, Kernel 1 has an IPC of 2.37 for PNY and 2.79 for HSW. Using the regression model for Kernel 1 on HSW (see Figure 14), we can determine that this 1.18x improvement in performance equates to a 1.05x increase in power (from 44.0 watts to 46.4 watts). Overall, this increase in power is countered with a more significant decrease in runtime which translates to a net 1.11x improvement in energy efficiency.

4.3.4 Process/Circuits Innovation

Process technology innovations have been the primary driver of improvements in energy efficiency over the past several decades. These advances enable manufacturers to produce smaller transistors that can operate a lower supply voltages and reduced capacitance, both of which reduce dynamic switching energy. Although, smaller transistors also have a higher leakage ratio, which contribute to higher overall static power dissipation.

We evaluate the impact of process technology improvements by comparing microarchitectures across process technology nodes: NHM with WSM and SNB with IVB. Since the core microarchitecture is largely unchanged, the change in power can be primarily attributed to improvements in circuits and process technology. Figure 18 shows a scatter plot comparing the average power of a kernel on NHM with the average power of the same kernel on WSM. The regression model suggests that moving from the 45nm to the 32nm process technology node has increased static power by 7.55 watts but reduced dynamic power by a factor of 0.57. The increase in the static power is partially due to higher leakage and partially due to a bigger die with two more cores in WSM – we believe the increase in static power would be lower if Intel had produced an equivalent processor with four rather than six cores. Figure 19 shows a similar trend for the progression from 32nm to 22nm process technology nodes. In this case, both the dynamic and static power reduced by a factor of 0.68 and by 10.48 watts respectively. The reduction in static power has been discussed as a benefit of the transition from planar transistors in the 32nm process node to the 3D Tri-Gate transistors which debuted in the 22nm process node [5]. The 40% reduction in dynamic power in both cases is in line with ITRS Roadmap projections [7].

4.4 Discussion

Taking out the impact of process technology changes, micro-architecture changes have increased per-cycle performance 1.9x and energy efficiency by 1.2x. Ignoring the overhead cost of implementing these features, 40% of the energy efficiency can be attributed to SIMD width, 35% to frontend features, and 25% to backend features.

This section section makes several additional observations about the evolution of energy efficiency in these processors.

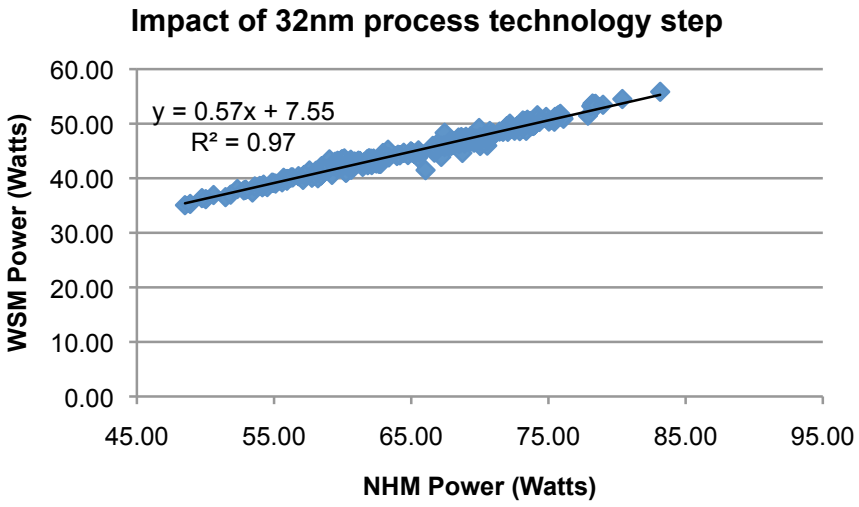

Figure 18: Improvement in energy efficiency attributed to 32nm process technology step.

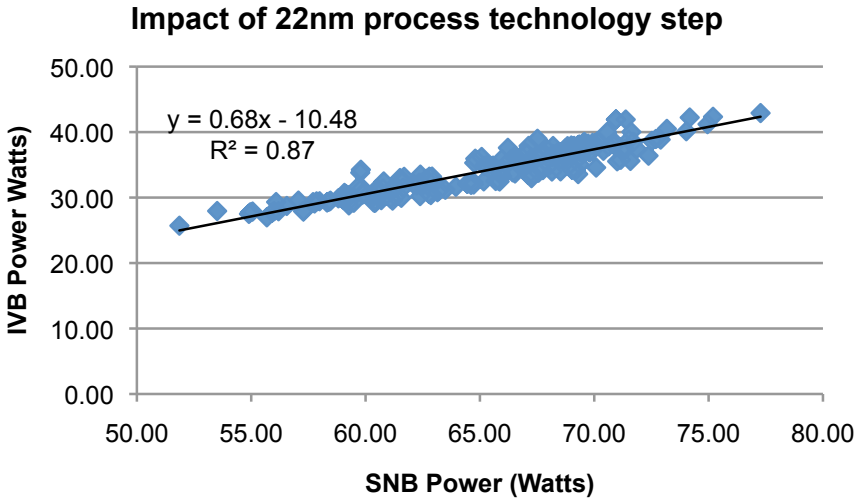

Figure 19: Improvement in energy efficiency attributed to 22nm process technology step.

4.4.1 Energy per instruction (EPI) depends highly on IPC

Efficiency is often gauged by calculating the effective energy per instruction (EPI). EPI makes the most sense as a metric when it remains constant, independent of IPC. However, because real machines expend energy at a certain minimum rate even when idle, actual EPI becomes a function of IPC. Figure 20 demonstrates the effective EPI

of HSW as a function of IPC. The effective cost of an instruction drops from 4.1 nJ at IPC=0.5 to 0.8 nJ when IPC=4. The plot also shows that IPC has a more dramatic impact on EPI than the actual instruction mix.

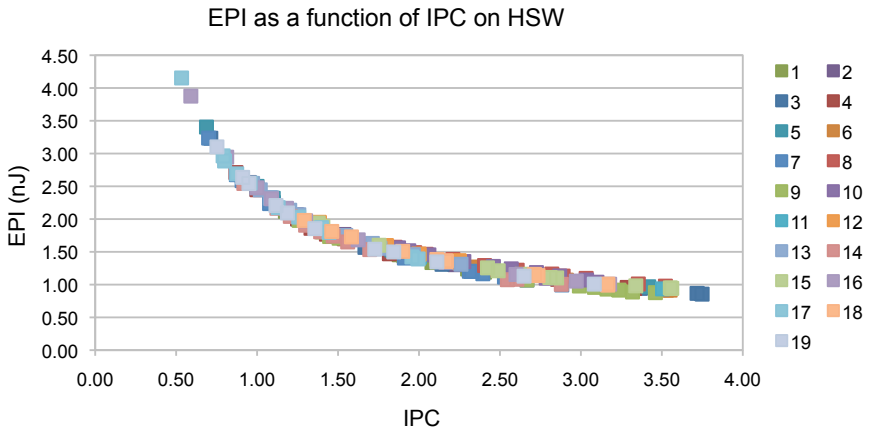

Figure 20: EPI as a function of IPC on HSW. Data samples come from ten "scrambles" of each of the Livermore Loops kernels. Data points are colored according to the kernel.

4.4.2 Fixed costs dominate variable costs

The relationship between IPC and EPI is the result of high static power which is independent of core utilization. We can approximate this overhead by modeling power as a function of IPC (see Figure 21). Table 9 lists a linear regression relating power to IPC on each of the processors. As the table shows, when IPC=2, HSW consumes 11.05 nJ per cycle. Of that cost, 8.31 nJ comes from a fixed overhead cost and 2.74 nJ comes from the variable cost of the instructions. As this data implies, the actual operation cost of an instruction (e.g., the floating-point arithmetic implied by a floating-point instruction) is only a small fraction of the total power of the CPU. This is a typical consequence of general purpose processors [102]. Looking at the trend from PNY to HSW, the variable costs drops with each processor generation, with substantial drops corresponding to the changes in the process technology node. The fixed costs drop during process technology node steps (NHM → WSM, SNB

→ IVB), but increase when subsequent microarchitectures are introduced (PNY → NHM, WSM → SNB, IVB → HSW).

Comparing HSW to PNY, a larger fraction of the total energy is spent on fixed costs. Two reasons contribute to this trend. First, as modern processors integrate more cores together on a single chip, the size of the uncore grows [79]. In our experiment, the uncore is not exercised, but it is not clock gated either. Second, increasing core performance has a cost in area and power. However, the transfer from variable costs to fixed costs is not necessarily bad. For example, the introduction of the micro-op cache increases the fixed cost of the core by adding a cache, but it also reduces the variable cost by reducing the number of instruction decodes and fetches.

Furthermore, comparing HSW to IVB reveals how the new features in HSW affect both the fixed and variable costs. The microarchitectural evolution from IVB to HSW has increased the fixed cost by 22%. Unless HSW is able to exploit these features to improve performance, at IPC=2.0 IVB will be 14% more efficient than HSW. To overcome this deficit, HSW must sustain an IPC of 2.38 to match the energy efficiency of IVB at IPC=2.0 or rely on SIMD extensions to make each instruction more productive.

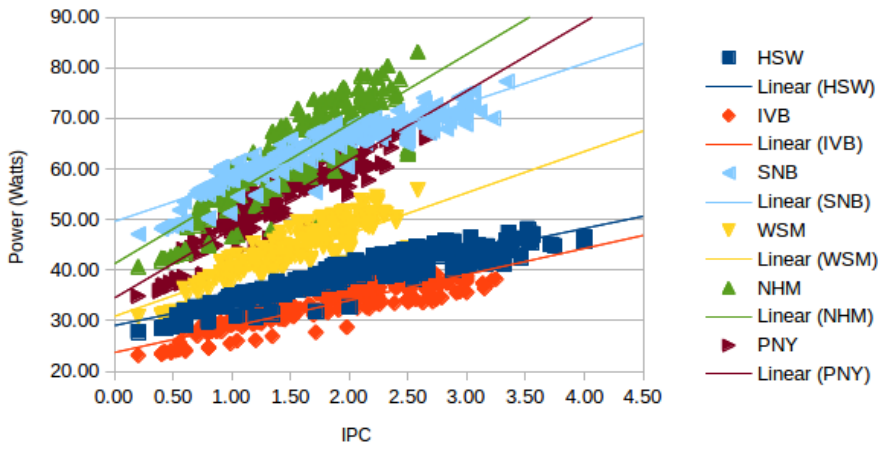

Figure 21: Power as a function of IPC. Data samples come from ten "scrambles" of each of the Livermore Loops kernels.

Table 9: Fixed vs Variable cost analysis of CPU power. The power model is based on a linear regression of the data shown in Figure 21.

	Power Model	Energy Per Cycle (IPC=2)	Fixed Cost Per Cycle (IPC=2)	Variable Cost Per Cycle (IPC=2)	Fixed / Total (IPC=2)
HSW	4.79(IPC) + 29.08	11.05 nJ	8.31 nJ	2.74 nJ	75%
IVB	5.14(IPC) + 23.74	9.72 nJ	6.78 nJ	2.94 nJ	70%
SNB	7.81(IPC) + 49.63	18.64 nJ	14.81 nJ	4.46 nJ	76%
WSM	8.14(IPC) + 30.89	13.97 nJ	9.14 nJ	4.82 nJ	65%
NHM	13.79(IPC) + 41.30	20.39 nJ	12.23 nJ	8.16 nJ	60%
PNY	13.63 (IPC) + 34.54	20.60 nJ	11.51 nJ	9.09 nJ	56%

4.4.3 Performance improvements exceed power increases

According to the "base" value in Figure 15, the increased complexity of the HSW core has increased the overhead cost by nearly 80% when compared to PNY. To avoid a net increase in energy consumption, improvements in performance should be greater than 80%. In the case of the Livermore Loops, the average performance increases 90%.

4.4.4 Frontend features reduce the tax of complex instructions

As mentioned earlier, as the complexity of the core increases, more energy is devoted to the fixed costs. However, the increase in fixed cost does not necessarily reduce energy efficiency. For example, the frontend caches increase the fixed cost, but ultimately improve energy efficiency by dramatically reducing amount of energy spent fetching and decoding instructions.

When compared to the Reduced Instruction Set Computing (RISC), Complex Instruction Set Computing (CISC) is generally considered less efficient. CISC does not have uniform instruction length, which adds significant complexity to the instruction fetch and decode logic. Based on the results in section 4.3.2 we can approximate the power of the fetch and decode units by comparing the power when instructions are streamed from the micro-op cache —which caches decoded micro-ops and therefore

can bypass the fetch and decode stages— with the power when the instruction are streamed from the L1 instruction cache. Figure 16 shows that on HSW, the fetch and decode overhead consumes about five watts of power which is roughly 12.5% of the average 40 watts of power consumed by HSW when running the Livermore Loops kernels. Our results also show that 16 of the 19 kernels fit into the loop cache on HSW and all of the kernels fit in the micro-op cache, which suggest that most of the inefficiencies associated with the CISC decode tax can be eliminated with a well designed frontend.

4.4.5 SIMD extensions increase the productivity of each instruction with minimal impact on power

It is a well known that SIMD computing is energy efficient. Our results show that under the ideal conditions vector instruction can increase performance with only a nominal increase in dynamic energy consumption. However, as SIMD width increases, its applicability diminishes (see Section 4.3.1). Even when there is sufficient data parallelism, the additional work necessary to shuffle data into the proper SIMD lanes can limit the advantage of wide SIMD instructions. For example, algorithms with indirect accesses or random accesses require separate scalar operations to load and pack data into these SIMD lanes, which could take away most of the performance and energy efficiency benefit offered by SIMD. Auxiliary instructions —such as the gather instruction, which loads non-continuous data into SIMD registers— are crucial to attaining the full potential of vectorization. Just like the frontend caches which provide the benefit of reducing complex instructions overhead, the gather instructions help make wider SIMD useful for a broader pool of applications.

4.4.6 High performance computing vs energy efficient computing

Modern semiconductor manufacturing technology enables processors to operate over a wide range of frequency and supply voltages. The processor can operate at a high

voltage to support a high clock frequency or a low voltage to reduce power. In our experiments the voltage and frequency used are near the top end of the supply voltage range, which sacrifices energy efficiency for high single-threaded performance. In this regard, our results may not capture the most energy efficiency way of using the processors; it is possible for these processors to be more efficient operating at a lower frequency and voltage. We therefore caution readers to consider the supply voltage of the processor before comparing the energy efficiency across different classes of processors.

4.5 Related Work

Traditionally, research has focused on the power consumption of individual functional units, yet there has been a growing demand for processor-level analysis of energy efficiency.

At the application level, there have been several studies comparing the energy efficiency of different processors. This has included long-term historical trends [46, 70] as well as comparisons of competing platforms [20, 33, 34]. The demand for energy efficient computing has even motivated the Green500 list, which has become an industry benchmark for comparing the energy efficiency of supercomputers [35]. Of course, the vast differences between these systems (performance levels, ISA, process and manufacturing technology, code quality, etc) makes it difficult to draw any definitive conclusions about the true impact of the underlying microarchitecture.

Bottom-up models and cycle-accurate simulators have been used to evaluate architectural design decisions that affect power [16, 49, 60, 110]. Unfortunately, it is often difficult to validate the relative contribution of the individual components in the underlying model when only the total power of the physical processor can be observed. Furthermore, as the complexity of modern processors swells, it is becoming increasingly difficult to construct a bottom-up model that can accurately capture all

relavent processor features.

Alternatively, several groups have studied power consumption by developing instruction-based power models [11, 75, 89]. This approach focuses on correlating hardware performance counters with observed power measurements. A regression analysis is used to assign energy costs to each architectural event. This can then be used to predict power consumption of an arbitrary application based solely on performance counter values. The primary weakness of this approach is its inability to directly account for the full context within which an instruction is executed. For example, as our paper shows, the energy consumed by an instruction that is streamed from a loop cache can be significantly different from one that instead exercises the instruction fetch and decode units, yet the performance counter events only give a limited view of the internal state of the processor core.

4.6 Future Work and Conclusion

Due to the complexity of modern processors, we had to limit the scope of this project to deliver meaningful insights. We identify several limitations that can be addressed further in future work:

- **Beyond the core**: Our study focuses on the core architecture and thereby tries to minimize the impact of the memory hierarchy, uncore, and system-level components. We acknowledge that these components are an important factor in the overall efficiency of a modern application. Studying the core in isolation provides a foundation for tackling the broader problem.

- **Representative workloads**: The Livermore Loops benchmark suite provides a number of floating-point heavy computations with a variety of instruction-level dependencies. It is well suited for the experiments in our study, but it is not necessarily representative of modern or future workloads. In particular, we want to draw attention to two important characteristics that are absent in

our benchmarks: first, the single-loop structure of our benchmarks eliminates any performance advantage of a sophisticated branch predictor because nearly all branches are taken; second, by design all memory accesses hit the L1 cache, neglecting the uncore and higher levels of the memory hierarchy and thereby eliminate dynamic variation in instruction latency. Studying more representative workloads is a natural next step.

- **Scaling voltage and frequency**: Core voltage and clock frequency can have a tremendous impact on energy efficiency. In our experiments, the processors were configured to run at the default voltage and frequency without dynamic scaling. With a specific application in mind, it would be interesting to study how voltage and frequency scaling impact energy efficiency on a fixed architecture.

- **Comparing competing processors**: The quantitative results of this study are specific to the Intel® Core™ processor, yet the conclusions can be applied in general. For a more general purpose study, this approach can also be extended to compare among different processor families involving more dramatic microarchitectural differences, such as a comparison with the Intel® Atom™ or Xeon Phi™ or even comparing processors with different ISAs.

- **Optimizing software for energy efficiency**: Beyond the quantitative numbers presented in this study, this work also demonstrates how carefully controlled experiments can be used to isolate individual architectural features for the purposes of empirically measuring energy consumption. In many cases, key architectural features, such as the loop cache, are transparent from a performance perspective and are thus overlooked. We believe this methodology provides the level of precision necessary for exploring the impact software implementation decisions have on energy efficiency. For example, an evaluation of compiler heuristics can lead to compile time optimizations tailored specifically

for reducing energy consumption.

Following the hackneyed business adage, "if you can't measure it, you can't manage it", we firmly believe that long-term improvements in energy efficiency within the field of high performance computing depend on rigorous evaluations of progress. In this spirit, our paper provides an in-depth assessment of energy efficiency on recent generations of the Intel® Core™ processor. Our results indicate that advances in manufacturing and circuits have been the dominate contributor to the improvements in energy efficiency, but architectural innovations have had a positive impact on energy efficiency as well.

CHAPTER V

A THEORETICAL FRAMEWORK FOR ALGORITHM-ARCHITECTURE CO-DESIGN

5.1 Introduction

We seek a formal framework that explicitly relates characteristics of an algorithm, such as its inherent parallelism or memory behavior, with parameters of an architecture, such as the number of cores, structure of the memory hierarchy, or network topology. Our ultimate goal is to say precisely and analytically how high-level changes to the architecture might affect the execution time, scalability, accuracy, and power-efficiency of a computation; and, conversely, identify what classes of computation might best match a given architecture. Our approach marries abstract algorithmic complexity analysis with key physical constraints, such as caps on power and die area, that will be critical in the extreme scale systems of 2018 and beyond [1, 68]. We refer to our approach as one of *algorithm-architecture co-design*.

We say "algorithm-architecture" rather than "hardware-software," so as to evoke a high-level mathematical process that precedes and complements traditional methods based on detailed architecture simulation of concrete benchmark code artifacts and traces [18, 44, 51, 58, 96, 106]. Our approach takes inspiration from prior work on high-level performance analysis and modeling [9, 50–52, 92], as well as the classical theory of circuit models and the area-time trade-offs studied in models based on very large-scale integration (VLSI) [73, 98]. Our analysis is in many ways most

The material in this chapter has appeared in a separate refereed publication [26].

similar to several recent theoretical exascale modeling studies [42, 95], combined with trends analysis [69]. However, our specific methods return to higher-level I/O-centric complexity analysis [13, 14, 19, 23, 41, 111], pushing it further by trying to resolve analytical constants, which is necessary to connect abstract complexity measures with the physical constraints imposed by power and die area. This approach necessarily will *not* yield cycle-accurate performance estimates, and that is not our aim. Rather, our hope is that a principled algorithmic analysis that accounts for major architectural parameters will still yield interesting insights and suggest new directions for improving performance and scalability in the long run.

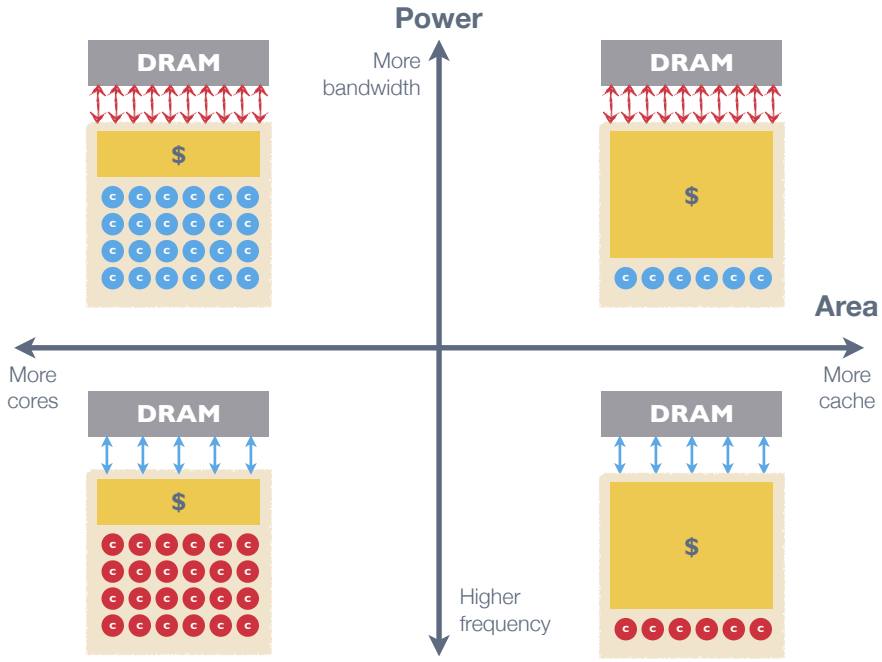

Figure 22: A notional power/transistor allocation problem. In our framework, a fixed die area (allocated between cores and cache) and a fixed power budget (allocated between core frequency and bandwidth), define a space of possible machines.

A formal framework. We pose the formal co-design problem as follows. Let a be an algorithm from a set A of algorithms that all perform the same computation within the same desired level of accuracy. The set A might contain different algorithms,

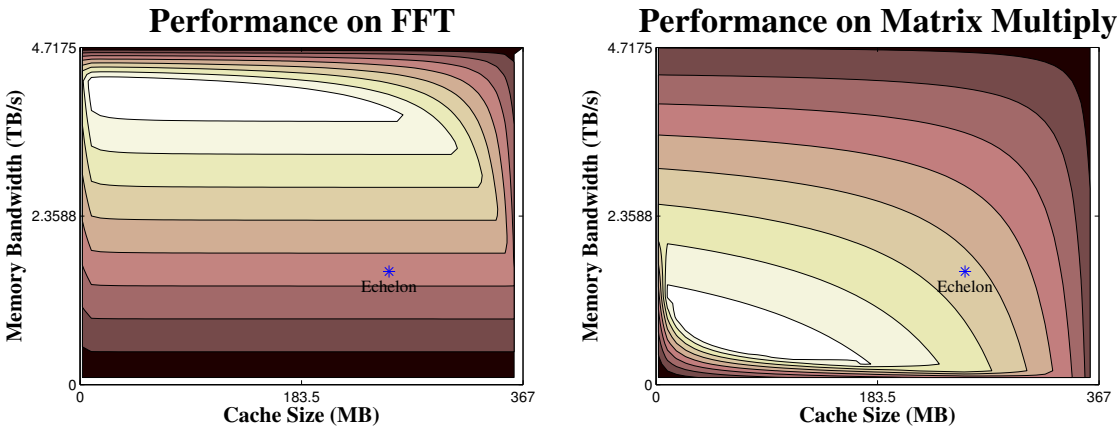

Figure 23: Projected performance for a 3D FFT and matrix multiply at some problem size (lighter is better). Different algorithms may perform differently on these machines. The marker is the approximate "location" within this space of the NVIDIA Echelon GPU-like architecture proposed for the year 2017 [65]. In the 3D FFT example, the optimal configuration is 2.6 times faster than Echelon.

such as "$A = \{$fast Fourier transform, F-cycle multigrid$\}$," for the Poisson model problem [30, 84]. Or, A may be a set of tuning parameters for one algorithm, such as the set of tile sizes for matrix multiply. Next, let μ be a machine architecture from a set M, and suppose that each processor of μ has an area of $\chi(\mu)$. Lastly, let $T(n; a, \mu)$ be the time to execute a on μ for a problem of size n, while using a maximum instantaneous power of $\Phi(\mu)$. Then, our goal is to determine the algorithm a and architecture μ that minimize time subject to constraints on total power and processor die area, e.g.,

$$(a^*, \mu^*) = \underset{(a \in A,\ \mu \in M)}{\operatorname{argmin}} T(n; a, \mu) \tag{8}$$

subject to

$$\Phi(\mu^*) = \Phi_{\max} \tag{9}$$

$$\chi(\mu^*) = \chi_{\max}, \tag{10}$$

where $\Phi_{\max}$ and $\chi_{\max}$ are caps on power and die area, respectively. The central research problem is to determine the form of $T(n; a, \mu)$, $\Phi(\mu)$, and $\chi(\mu)$. The significance and novelty of this analysis framework is that it explicitly binds characteristics

of algorithms and architectures, Equation (8), with physical hardware constraints, Equations (9)–(10).

A demonstration. Suppose we wish to design a manycore processor μ, which we represent by the four-tuple $(\beta_{\text{mem}}, q, f, Z)$: β_{mem} is the processor-memory bandwidth (words per unit time), q is the number of cores per processor, f is the clock frequency of each core (cycles per unit time), and Z is the total size of the aggregate on-chip cache (in words), assuming just a two-level hierarchy (cache and main memory). Further suppose that the $\chi_{\text{max}} = 141$ mm^2 of die area can be divided between on-chip cache (Z) and cores (q). Lastly, suppose the node power budget is constrained to $\Phi_{\text{max}} = 173$ Watts, which can be used to increase cycle-frequency (f) or boost off-die memory bandwidth (β_{mem}). Figure 22 is a cartoon that suggests how these parameters and constraints imply a space of possible designs. Figure 23 shows how, given a specific model of different algorithms on this space of machines, we might then solve the optimization problem of Equations 8–10 to identify optimal systems. Unsurprisingly, a processor tuned for a communication-intensive 3D fast Fourier transform will devote more of a fixed power budget (y-axis) to memory bandwidth, compared to matrix multiply.

However, Figure 23 also suggests an intriguing possibility. Observe that a die area configuration (x-axis) that is good for matrix multiply will also be good for an FFT; to make a system that can perform "optimally" on both workloads, we would need the ability to dynamically *shift* power from the processor to memory bandwidth, by a large factor of roughly 7×. That is, reconfigurability of processor transistors may be relatively less important than extreme power reconfigurability with respect to bandwidth. Whether one can build such a system is a separate question; this demonstration suggests and attempts to quantify the possibility.

The remainder of this paper formalizes this analysis. As a demonstration, we

develop an analytical model of these constraints, $\Phi(\mu)$ and $\chi(\mu)$, as well as performance models, $T(n; a, \mu)$. To suggest the possibilities of the framework, we develop models for a full-system configuration, consisting of a distributed memory machine comprising any number of manycore processors connected by a network, in the case of distributed matrix multiply, distributed 3D FFTs, and distributed stencil algorithms. We do not view any specific models and projections as the main contribution of our work. Rather, we wish to emphasize the basic framework, with the large variety of potential detailed modeling strategies, analyses, and projections as possibilities based on it.

5.2 Background and Related Work

The principal challenge is how to connect power (and implicitly, energy) and area constraints with a complexity analysis. There are numerous approaches. The most widely-cited come from the computer architecture community [20, 33, 48, 50, 80, 117]. In such approaches, the application or algorithm is typically abstracted away through an Amdahl's Law style analysis, which means it can be difficult to relate high-level algorithmic characteristics to architectures precisely. Theorists have also considered a variety of new complexity models that incorporate energy as an explicit cost [71, 85]. However, this body of work is very abstract and focused purely on asymptotics, making these models difficult to operationalize, in our view. Lastly, there is a considerable body of work from the embedded hardware/software community, emphasizing analysis suitable for compiler- and run-time systems [27, 67, 109]. However, this work necessarily focuses on specific concrete code and architecture implementations, and therefore do not explicitly illuminate constraints due to fundamental algorithmic and physical limits. It is these limits that we seek to understand to make forecasts about future algorithm and system behavior.

Regarding area constraints, note that our framework treats die area, $\chi(\mu)$, as a

function of architecture μ only. Thus, to construct this function we need to consider what architectural components we will put on chip (e.g., cores, caches, on-chip networks) and derive cost estimates for the size of each component. Effectively, this choice implies that we are most interested in still building architectures that can execute general-purpose computations; however, the power and area allocations are tuned to specific workloads. Thus, our approach stands in contrast to the classical work on models of computation under very large-scale integration (VLSI) [98, Chap. 12]. That body of work also considers physical area-time trade-offs [73, 93], with connections to energy [4, 114], but does so for specific circuit structures that correspond to specific computations. We imagine a bridging between these two approaches but are for the moment agnostic on the specific analytical form of that bridge.

To develop cost models for both power and area, we are mining the vast literature on models and technology trends for everything from low-level processor device physics, functional units, cores, caches and memory systems, and on-chip and off-chip networks [15, 16, 28, 47, 60, 68, 76, 103]. Since our ultimate goal is to consider potential long-term outcomes, we focus on recent projections of scaling trends [15, 65, 68, 69, 103].

5.3 An Example of Instantiating a Model within the Framework

This section explains how one might go about constructing meaningful cost and constraint models within the framework. In particular, we instantiate specific forms for $T(n; a, \mu)$, $\Phi(\mu)$, and $\chi(\mu)$. These forms are meant to be illustrative, not necessarily definitive. Armed with such a model, Section 5.4 then considers a variety of what-if scenarios at exascale (roughly 1 exaflop/s capable systems) expected in the 2017–2020 timeframe, to illustrate the kinds of insights possible within the framework.

5.3.1 Technological and architectural parameters

To guide parameter selection and model calibration, we start with the basic technology trend assumptions laid out in a recent description of the proposed NVIDIA Echelon processor, scheduled for release in approximately the year 2017 [65].

For our subsequent analysis and projections, we will assume the technology constants that appear in Table 10. These values depend on specific assumptions about technological capabilities in the 2017–2020 time frame, for which we borrow the expectations used to design Echelon. We take those projections as-is; debates about the accuracy of these values are beyond the scope and intention of this paper.

Our target system is a supercomputer. We parameterize the high-level architecture μ by the following: the number of cores per processor (q), cycle-frequency of each core (f), the aggregate on-chip cache capacity (Z), memory bandwidth (β_{mem}), on-chip network bandwidth (β_{noc}), off-chip network link bandwidth (β_{net}), and total number of nodes (p). By "node" we really mean a single unit of distributed memory processing consisting of a (manycore) processor, local memory, persistent storage (disk). This definition constitutes a simplification of how a real "node" might look in a future system; however, we do model the power required by such a node (see ϵ_{node} in Table 10). We use the term "core" to represent the basic unit of processing in the system. We endow a core with general-purpose processing capabilities, e.g., ALU, address generation, branch unit, register file; however, because the sample algorithms we analyze are floating-point intensive, we characterize the core essentially by its peak rate of floating-point instructions completed per cycle. The caches are core-private but the value Z is aggregate over all cores on a chip. We connect cores via a $\sqrt{q} \times \sqrt{q}$ 2D Mesh with an on-chip link bandwidth of β_{noc}. The on-chip cache is evenly distributed to all cores so that each cores has a private cache of size $\frac{Z}{q}$. We connect nodes by a $p^{\frac{1}{3}} \times p^{\frac{1}{3}} \times p^{\frac{1}{3}}$ 3D torus with a link bandwidth of β_{net}. As a reference point, the values of these parameters as proposed for the Echelon design appear in

column 2 of Table 11. (Columns 3 and 4 will be discussed in Section 5.4.)

Our specific algorithmic analyses will also assume an abundance of parallelism. Though this assumption seems strong, it is also a necessary condition for any application that can hope to scale to very large numbers of nodes relative to today's standards.

Table 10: Technology Constants: Projected values for 2018.

Parameters		2018 (10 nm) Value
System Power Cap:	Φ_{max}	20 MW
Chip Die Area:	χ_{max}	141.7 mm^2
Cache Density:	σ_{cache}	.386 mm^2/MB
Core Density:	σ_{core}	.0105 mm^2/MB
Memory BW Power:	λ_{mem}	36 mW/$\frac{\text{GB}}{\text{s}}$
Network BW Power:	λ_{net}	36 mW/$\frac{\text{GB}}{\text{s}}$
On-Chip Network BW Energy:	λ_{noc}	.75 pJ/mm Byte
Dynamic Power Coefficient:	c_{dynamic}	0.00129704
Short-Circuit Coefficient:	c_{short}	0.0032426
Leakage Coefficient:	c_{leakage}	0.002026625
Node Overhead:	ϵ_{node}	2 W

5.3.2 A model of physical constraints

We use the following models of power and area, based on the parameters shown in Tables 10 and 11.

The total system power comprises the power of the cores (P_{comp}), memory (P_{mem}), on-chip interconnect (P_{noc}), and the system network (P_{net}). There is also a nominal per node power cost (ϵ_{node}) that represents the inherent overhead cost of maintaining a node, e.g., power supply, chipset, disk.

$$\Phi(\mu) \;=\; p\,(P_{\text{comp}} + P_{\text{mem}} + P_{\text{noc}} + \epsilon_{\text{node}}) + P_{\text{net}} \tag{11}$$

Power consumption of CMOS circuits are frequently modeled with a three term equation of the form, $P = ACV^2f + \tau AVIf + VI_{\text{leakage}}$, which accounts for the dynamic

Table 11: Hardware Characteristics (μ).

Parameters		Echelon Value	Ideal Matrix Multiply Value	Ideal FFT Value	Ideal Stencil Value
Cores:	q	4,096	11,491	11,295	9,494
Frequency:	f	2.0 GHz	.275 GHz	3.0 GHz	.280
Aggregate On-chip Cache:	Z	256 MB	54.5 MB	59.8 MB	109 MB
Memory Bandwidth:	β_{mem}	1.6 TB/s	.034 TB/s	29.5 TB/s	.787 TB/s
On-chip Network Bandwidth:	β_{noc}	4.0 GB/s	.11 GB/s	38.2 GB/s	.19 GB/s
Network Bandwidth:	β_{net}	67 GB/s	11.4 GB/s	18,100 GB/s	12.5 GB/s
Number of nodes:	p	102,500	1,280,000	3,400	480,000
Peak:	$2qfp$	1.7 EF/s	8.1 EF/s	230 PF/s	2.5 EF/s

power consumption, short-circuit current, and leakage current [91]. The key variables from our perspective are voltage V and frequency f. Since the maximum operating frequency f is roughly linearly related to the voltage V, we can simplify the expression for the power consumption of each core to be a function that is cubic in f:

$$P_{\text{comp}} = q\left(c_{\text{dynamic}}f^3 + c_{\text{short}}f^2 + c_{\text{leakage}}f\right). \quad (12)$$

We can obtain suitable coefficients by fitting this equation to the different voltage, frequency, and energy settings provided in the Echelon paper [65].

Bandwidth power (Joules/sec or Watts) is bandwidth (Bytes / sec) times energy cost per byte (Joules/Byte):

$$P_{\text{mem}} = \beta_{\text{mem}} \cdot \lambda_{\text{mem}} \quad (13)$$

$$P_{\text{net}} = \beta_{\text{net}} \cdot \text{Links}(p) \cdot \lambda_{\text{net}} \quad (14)$$

The die area dedicated to each core is $\frac{Z}{q}\sigma_{\text{cache}} + \sigma_{\text{core}}$. Assuming each core is square in shape, the distance between the center of two neighboring cores is $\sqrt{\frac{Z}{q}\sigma_{\text{cache}} + \sigma_{\text{core}}}$,

which we will use to approximate the length of the on-chip interconnect links. Assuming a 2D mesh topology, which is the most natural considering the current planar manufacturing process of modern processors—there are a total of $4q-4\sqrt{q}$ links. The power of the on-chip network is therefore

$$P_{\text{noc}} = (4q - 4\sqrt{q}) \left(\sqrt{\frac{Z}{q}\sigma_{\text{cache}} + \sigma_{\text{core}}} \right) \beta_{\text{noc}} \lambda_{\text{noc}}. \tag{15}$$

The limited processor die area constrains the number of cores and cache that can be placed on a single node. A larger cache capacity means there is less space for cores and vice-versa. Thus, given a total die area of Ω_{die}, we constrain total core and cache capacity by requiring that

$$\Omega_{\text{die}} \;\; = \;\; (q \cdot \sigma_{\text{core}}) + (Z \cdot \sigma_{\text{cache}}) \, . \tag{16}$$

5.3.3 Algorithmic cost models

Given the basic architectural model and parameters, the next step is to analyze an algorithm or class of algorithms, so that we can compute $T(n; a, \mu)$. We specifically consider the total execution time to be the maximum of four component times:

$$T \;\; = \;\; \max \{T_{\text{comp}}, T_{\text{net}}, T_{\text{mem}}, T_{\text{noc}}\} \tag{17}$$

where T_{comp} is the time performing compute (flops), T_{net} is the time spent in network communication, T_{mem} is the time spent performing processor-memory communication, and T_{noc} is the time spent in on-chip network communication. Below, we give sample analyses for 2.5D matrix multiply, a three-dimensional FFT, and a stencil computation.

5.3.3.1 Example: Distributed 2.5D matrix multiply

The 2.5D matrix multiply algorithm of Solominik and Demmel [107] is a particularly interesting case for our framework. In particular, it contains a tuning parameter that

can be used to reduce communication at the cost of increasing memory capacity, a trade-off that we subsequently analyze.

The 2.5D matrix multiplication algorithm decomposes a $n \times n$ matrix multiply, distributed across p nodes, into a sequence of $(p/C)^{3/2}$ matrix multiplies, each of size $(n\sqrt{C/p}) \times (n\sqrt{C/p})$. The value C is the tuning parameter that, when increased, decreases communication at the cost of increased (replicated) storage.

We take computation time to be that of the conventional (non-Strassen) algorithm:

$$T_{\text{comp}} = \frac{2n^3}{pqf}. \tag{18}$$

Network communication costs are based on the asymptotically optimal bandwidth costs of the 2.5D algorithm [107]:

$$T_{\text{net}} = \frac{2n^2}{\sqrt{Cp}\beta_{\text{net}}}. \tag{19}$$

Each node will compute $(p^{1/2}/C^{3/2})$ local matrix multiplies of size $\left(n\sqrt{\frac{C}{p}}\right) \times \left(n\sqrt{\frac{C}{p}}\right)$ during the computation. From this fact, we can calculate the time spent locally transferring data between the processor and memory, given the aggregate cache of size Z. Assuming an asymptotically I/O-optimal blocked algorithm with block size $b = \sqrt{\frac{Z}{3}}$, we obtain

$$T_{\text{mem}} = \left(\frac{\sqrt{p}}{C^{\frac{3}{2}}}\right)\left(\frac{n\sqrt{C}}{\sqrt{p}}\right)^3\left(\frac{2\sqrt{3}}{\sqrt{Z}\beta_{\text{mem}}}\right) \tag{20}$$

$$= \frac{2\sqrt{3}n^3}{p\sqrt{Z}\beta_{\text{mem}}}. \tag{21}$$

Since we assume private caches and a 2D mesh network, we can treat the local matrix multiply as a distributed computation across the cores. We estimate the on-chip network communication time assuming the communication-optimal Cannon algorithm [17],

$$T_{\text{noc}} = \left(\frac{\sqrt{p}}{C^{\frac{3}{2}}}\right)\left(\frac{2n^2C}{p\sqrt{q}\beta_{\text{noc}}}\right) = \frac{2n^2}{\sqrt{Cpq}\beta_{\text{noc}}}, \tag{22}$$

where the constants reflect the additional assumption of the matrix operands being distributed in "skew" order across the private caches.

5.3.3.2 Example: Distributed 3D FFTs

Our second example is the 3D FFT. We assume a problem of size $N = n^3$ using the pencil decomposition [78]. The algorithm consists of three computation phases separated by two communication phases. Each computation phase computes n^2 1D FFTs of size n in parallel. Each communication phase involves $\sqrt{P}$ independent P-node personalized all-to-all exchanges.

Each 1D FFT of size n is computed locally on a node using $\Theta(n \log n)$ floating point operations. We assume the classic Cooley-Tukey radix-2 algorithm, which requires approximately $5n \log_2 n$ flops; thus,

$$T_{\text{comp}} = 3 \times \frac{5n^3 \log_2 n}{fpq}. \tag{23}$$

During each of the two communication phases, approximately n^3 data points are transferred across the network. Assuming a 3D torus network with a bisection bandwidth of $\mathcal{O}\left(p^{\frac{2}{3}}\beta_{\text{net}}\right)$, the communication cost is approximately

$$T_{\text{net}} = 2 \times \frac{n^3}{p^{\frac{2}{3}}\beta_{\text{net}}}. \tag{24}$$

During each of the three computation phases, each node must compute $\frac{n^2}{p}$ 1D FFTs. The number of processor-memory transactions necessary to compute each local 1D FFT depends on the total cache capacity Z of the node. If the entire 1D FFT can fit within the on-chip caches ($n \leq Z$), then memory transfers are limited to just $\mathcal{O}(n)$ compulsory cache misses. Otherwise, the computation will incur an additional $\Theta(n \log_Z n)$ capacity misses [53]. We have previously estimated a lower-bound on this constant to be 2.5 [25], using hardware counters measurements of

last-level cache misses incurred by the highly-tuned FFTW [40]. Thus,

$$T_{\text{mem}} = 3 \times \frac{n^2}{p} \times \frac{2.5n \max(\log_Z n, 1)}{\beta_{\text{mem}}} \quad (25)$$

$$= \frac{7.5n^3 \max(\log_Z n, 1)}{p\beta_{\text{mem}}}, \quad (26)$$

where the max function ensures that the transfers include at least the compulsory misses.

If an entire 1D FFT can fit in the private cache of a single core ($n < \frac{Z}{q}$), then no on-chip communication is necessary beyond the compulsory cache misses to DRAM. Otherwise, the 1D FFT must be distributed across $\hat{q} = \frac{nq}{Z}$ cores, which requires $\mathcal{O}\left(\frac{n}{\sqrt{\hat{q}}\beta_{\text{noc}}}\right)$ time assuming a 2D mesh topology. In total, each group must compute at least $\frac{n^3}{pZ}$ of these distributed FFTs. Additionally, we only consider large problems sizes ($q \ll n$) in this paper, so that load balance should not be a significant factor. Thus,

$$T_{\text{noc}} = 3 \times \left(\frac{n^3}{pZ}\right)\left(\frac{n\sqrt{Z}}{\beta_{\text{noc}}\sqrt{q}\sqrt{n}}\right) \quad (27)$$

$$= \frac{3n^3\sqrt{n}}{p\sqrt{q}\sqrt{Z}\beta_{\text{noc}}}. \quad (28)$$

5.3.3.3 Example: Distributed 3D Stencil

Our final example is a 3D stencil. For simplicity of demonstration, we will only consider a 3D cross-shaped stencil of width w and total of $6w + 1$ points, and we will ignore the possibility of algorithms that tile in time. We assume a problem of size $N = n^3$.

The most direct method consists of $12w+1$ floating point operations per element; thus,

$$T_{\text{comp}} = \frac{n^3(12w + 1)}{fpq}. \quad (29)$$

Assuming each node owns a $\frac{n}{\sqrt[3]{p}} \times \frac{n}{\sqrt[3]{p}} \times \frac{n}{\sqrt[3]{p}}$ block of the dataset, each node will need a $\frac{n}{\sqrt[3]{p}} \times \frac{n}{\sqrt[3]{p}} \times w$ sub-block from each of the six adjacent nodes on the 3D torus

network. Therefore, the communication cost is approximately

$$T_{\text{net}} = \frac{6wn^2}{p^{\frac{2}{3}}\beta_{\text{net}}}. \tag{30}$$

Without reuse, the number of processor-memory transactions necessary to compute each element is $6w+2$. A cache of size Z can be used to reduce the total number of reads by a factor of $\mathcal{O}\left(Z^{\frac{1}{3}}\right)$. Thus,

$$T_{\text{mem}} = \left(\frac{n^3}{p}\right)\left(\frac{6w}{Z^{\frac{1}{3}}}+2\right)\left(\frac{1}{\beta_{\text{mem}}}\right), \tag{31}$$

where $\frac{n^3}{p}$ is the number of elements processed on each node.

We can approximate the amount of on-chip communication by comparing the cache misses incurred by a core with a private cache of size $\frac{Z}{q}$ with the number of cache misses incurred by a processor with an aggregate cache of size Z. Thus,

$$T_{\text{noc}} = \frac{\left(\frac{n^3}{p}\right)\left[\left(\frac{6w}{\left(\frac{Z}{q}\right)^{\frac{1}{3}}}+2\right)-\left(\frac{6w}{Z^{\frac{1}{3}}}+2\right)\right]}{q\beta_{\text{noc}}} \tag{32}$$

$$= \frac{6wn^3\left(q^{\frac{1}{3}}-1\right)}{Z^{\frac{1}{3}}qp\beta_{\text{noc}}}. \tag{33}$$

5.4 Analysis

Given the models and architectural parameters of Section 5.3, we can now carry out a series of what-if analyses to gain some insight into possible futures and high-level architectural designs, and even compute solutions to Equation 8.

5.4.1 Ideal architectures

We solved the optimization problem of Equation 8 for the 3D FFT, matrix multiply, and stencil algorithms. For this first analysis, we fixed the matrix multiply algorithm to be the Cannon (2D) algorithm, rather than the 2.5D algorithm considered in the next section, and fix the stencil width ($w = 10$). The ideal configuration for each appears in columns 3-5 of Table 11. Think of these configurations as being the ones

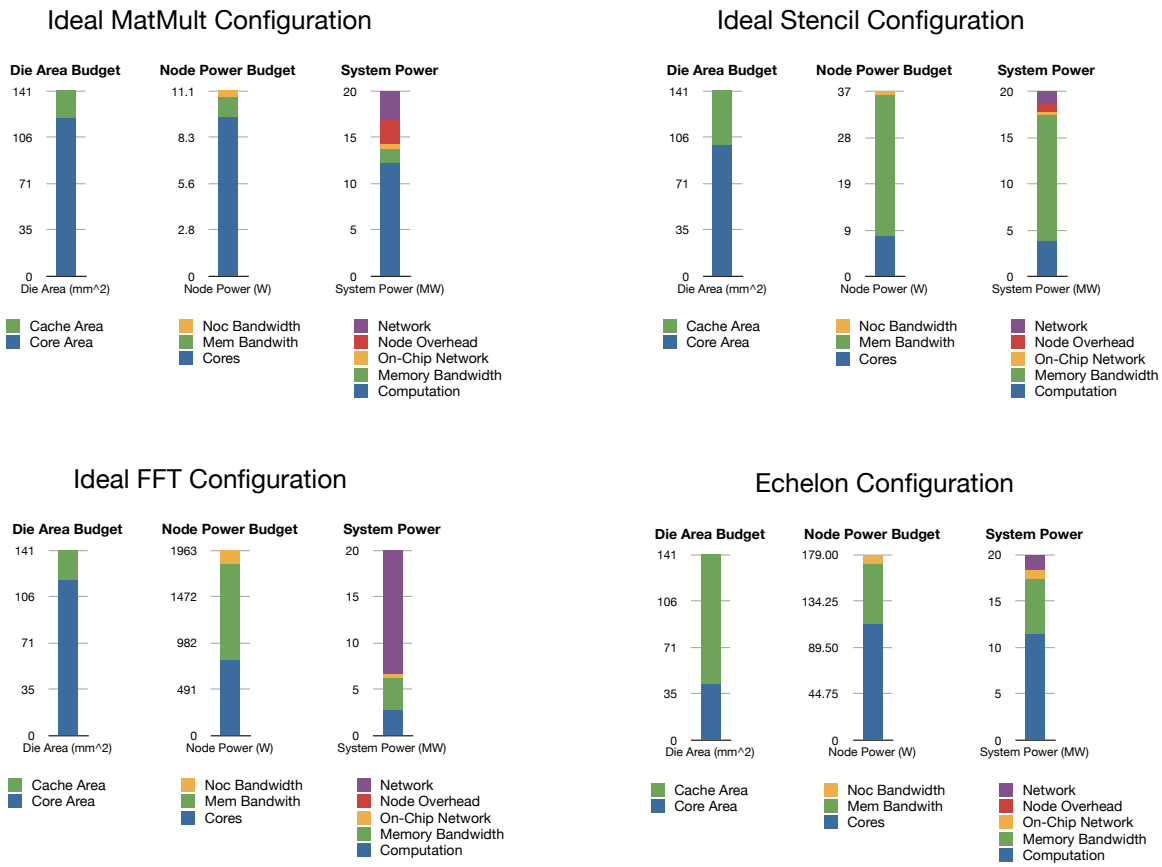

Figure 24: Hardware configurations for the hypothetical machines. The subplots break down the power and die area resource allocations.

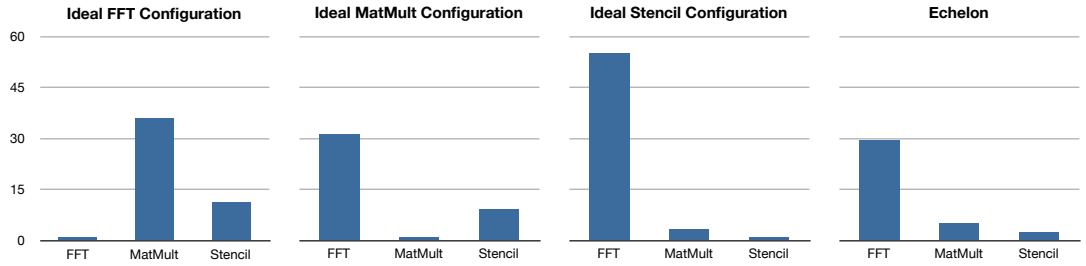

Figure 25: Relative execution times for the hypothetical machines. The subplots show execution time relative to the ieal FFT, Stencil, and MatMult configurations.

optimally tuned for the corresponding algorithm, though recall that the model is for a general-purpose system. Figure 24 shows how resources are allocated in each of these tuned configurations, as well as in the proposed Echelon configuration. Figure 25 shows execution times for each of the hypothetical machines on the 3D FFT, stencil, and matrix multiply workloads. We can make a number of observations about these

results.

Even under optimistic assumptions and an optimal machine configuration, the ideal FFT machine has a peak of only 230 petaflop/s (PF/s) with 20 MW of power, which is just 1/36 of the 8 exaflop/s (EF/s) possible on the ideal matrix multiply system. This means that relative to a matrix multiplication, the FFT computation requires 36× more energy per floating-point operation. However, tuning for a 3D FFT means we will necessarily divert power resources (and, therefore, energy efficiency) elsewhere in the system.

The Echelon design calls for 256 MB of on-chip cache which is over four times more than the ideal FFT and ideal MatMult. This is interesting because relative to NVIDIA's current GPU, the core count increased by a factor of 16 (the scaling factor from a 40nm to 10nm process technology) but the cache capacity by 64 (4x more than the scaling factor).

It is interesting to further consider these configurations in light of our motivating demonstration of Section 5.1. There, we observed that a single-processor system with extreme reconfigurability of *power*—rather than die area—*might* lead to designs capable of performing both a compute-intensive matrix multiply and a communication-intensive 3D FFT. The three ideal configurations, depicted in Figure 24 and enumerated in Table 11, are similarly suggestive. From Figure 24, observe that the processor configurations for the three ideal systems are not too dissimilar. Rather, the dramatic differences come from shifting power allocations to processors (Ideal MatMult), memory bandwidth (Ideal Stencil), or network (Ideal FFT). The node counts also different significantly (Table 11). Thus, an intriguing question is whether there is any way to engineer a single system having the same processors but a mechanism to perform dramatic power reconfigurations (drawing down or shutting off nodes as needed, and diverting power to bandwidth).

5.4.2 Architecture trade-offs: lightweight vs. heavyweight designs

Recent discussions surrounding the direction of high-end systems often characterize design strategies as either "lightweight" or "heavyweight." The key distinction between these two strategies is node density. Lightweight designs, exemplified by the Blue Gene-style processors, consist of many lower power processors. Alternatively, heavyweight designs, exemplified by Jaguar-class machines, consists of fewer but more powerful processors, each operating at high clock frequencies.

Interestingly, these characterizations apply to the ideal machine configurations in Figure 24. The ideal matrix multiply configuration, "Ideal MatMult," reflects a lightweight strategy, whereas the ideal 3D FFT configuration, "Ideal FFT," resembles a heavyweight strategy. The difference is extreme: Ideal FFT has only 3,400 nodes, which is 376× fewer nodes than Ideal MatMult. Essentially, an FFT is communication-bound and is therefore highly sensitive to the decreased energy-efficiency of a large network — on a 3D torus, the energy-efficiency will decrease at a rate of $\mathcal{O}\left((p^{2/3})/p\right)$ as p increases. Unlike the FFT, matrix multiply benefits from large node counts because it can exploit the increased core count to make the computation more efficient with a lower clock frequency. Figure 26 shows the stark contrast in performance between the two computations as a function of system density.

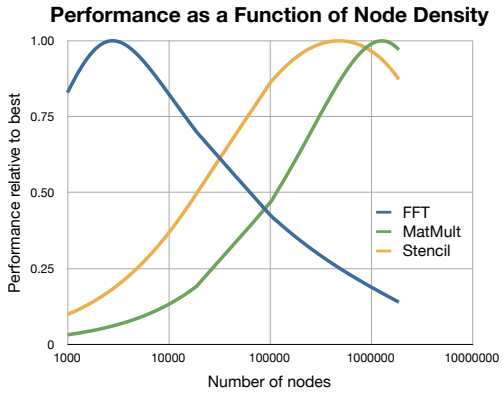

Figure 26: Node Density Plots.

5.4.3 Algorithm trade-offs: computation v. communication

A convolution is an algorithmic pattern that can capture the characteristics of many scientific computing problems. We may interpret a convolution as the application of a "filter" to a "signal." Computationally, a convolution may be implemented as a stencil computation, when the filter is compact, or alternatively by an FFT. A small value of the stencil width w in our model (Section 5.3.3.3) corresponds to our measure of compactness. In a stencil-based approach, the computational complexity of convolution will be $\mathcal{O}(wn)$. When the filter is not compact, an FFT-based method may be more suitable as it reduces computational complexity to $\mathcal{O}(n \log n)$. However, the FFT-based methods have a higher communication cost. For moderate sized stencils this results in a computation versus communication trade-off: the FFT-based method requires fewer floating point operations but more data movement than the stencil method [22, 32].

In the case of a large 3D convolution ($n = 2^{17}$), the FFT-based method will in our model become more efficient when $w \geq 22$. Figure 27 compares the execute time of the stencil and FFT-based methods over various stencil sizes. The results show that an ideal stencil machine is actually much faster than the ideal FFT machine until the stencil size is significantly larger ($w \approx 600$) than expected. The reason for the discrepancy is the relative cost of a floating-point operation versus data movement. As expected trading communication for computation is beneficial until the trade-offs become extreme.

5.4.4 Algorithm trade-offs: space vs communication

In Section 5.4.1, we found the ideal architecture for Cannon's matrix multiply algorithm (the "2D" approach). While this algorithm is asymptotically optimal when all of the available system memory is utilized, the class of "2.5D" algorithms can further reduce network communication by a factor of $\sqrt{C}$ by making an additional

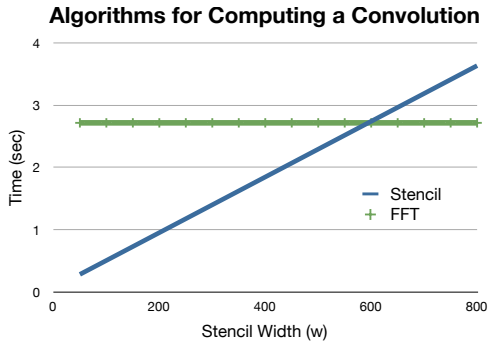

Figure 27: Plot of the time to compute a convolution using different stencil sizes. The figure compares the Stencil method with an FFT based method in the 3D case.

C copies of the data. Based on historical trends, we estimate that in 2018 increasing the system memory capacity could require an extra watt of power for every eight GB of additional DRAM [94, 116]. Thus, the power consumed by the requisite memory capacity, $3Cn^2$ nanowatts, increases as C increases. This introduces a trade-off between memory utilization and network communication. An interesting question is to what extent replication can be used to improve performance without increasing the total power and energy costs.

To find the optimal balance, we solve Equation 8 for the optimal algorithm, a^*, and the corresponding architecture, μ^*, considering the set of all 2.5D implementations (i.e., values of the replication factor, C). Figure 28 shows the performance and resource allocation of these algorithms. The results show that indeed, replication can slightly improve upon Cannon's algorithm under these conditions. However, the optimal balance is not at one of the extremes, but rather somewhere in-between.

5.4.5 Increasing the power budget

The U.S. Department of Energy, one of the primary customers of top-tier supercomputers, has instituted a strict 20 MW power cap for future supercomputers. This power constraint is one of the dominant challenges to reaching exascale.

We can consider how much performance improves if the power cap is relaxed a little by changing the power constraint, $\Phi_{\max}$, in Equation 8. Figure 29 compares

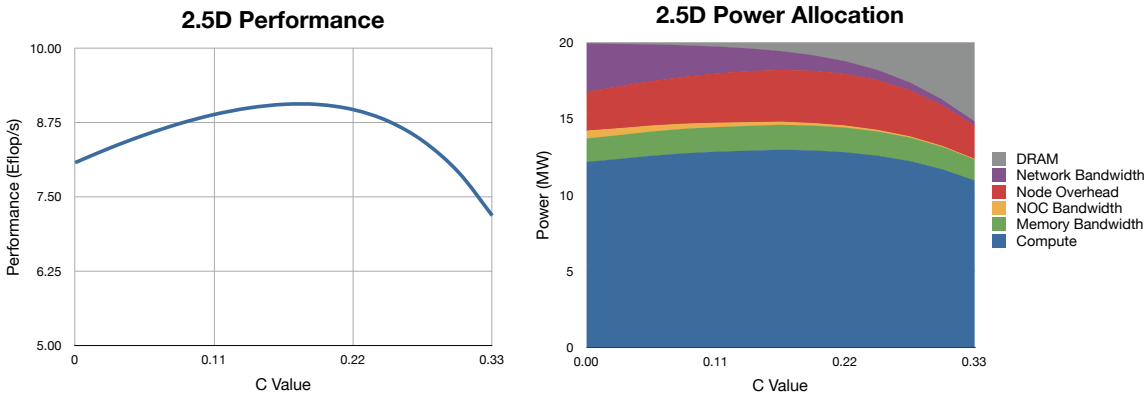

Figure 28: Performance of 2.5D Matrix Multiply variants. The 2.5D algorithm is parameterized by a value $C = p^{\alpha}$ where $0 \leq \alpha \leq \frac{1}{3}$. 'DRAM' represents the power consumed by the requisite memory capacity.

the performance of an 'ideal' machine, designed for a 20 MW power budget, with a similar machine that is designed for a larger power budget. As the figure shows, communication-heavy applications like the FFT will improve at a slower rate than applications that are less dependent on communication. This is because the most efficient way to scale the machine is to increase the number of nodes, which in turn increases the communication overheads.

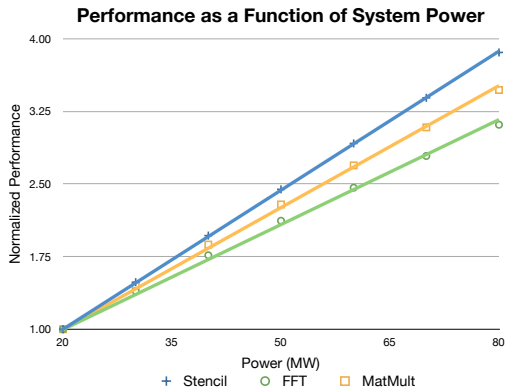

Figure 29: Plot of performance as a function of the power budget. Performance values for the algorithms are scaled to their performance at a 20 MW power budget.

CHAPTER VI

CONCLUSION

Over the past several decades, the prosperity of Moore's Law has ushered in a golden age in the field of High Performance Computing. The exponential increase in transistor budgets has brought about monstrously powerful processors which are faster and more sophisticated than previous generations. This has enabled supercomputers with immense computational horsepower to tackle large-scale science problems that were previously intractable. The explosion of processor technology has also made computing ubiquitous, touching nearly every aspect of daily life.

Unfortunately, as the pace of hardware performance improvements —provided by the increasing supply of transistors driven by Moore's Law— slows, the demand for performance improvements has not waned. This insatiable demand has focused attention on the efficiency of software as source of performance improvements.

Towards that end, this dissertation has taken a detailed look at the efficiency of modern HPC software and studied the bottlenecks that limit performance. We have shown that core HPC software applications are inefficient. Our analysis of key metrics of performance efficiency —such as instruction throughput (IPC), memory bandwidth utilization, and FLOPS— on the HPC proxy apps achieve only a small fraction of the theoretical throughput of the hardware.

In most cases, the complexity of the hardware and software applications hide the underlying performance bottlenecks, which reduce the efficiency of applications. Without a clear understanding of limiting factors, optimizing software to improve efficiency can be prohibitively difficult.

We formalized and address this problem with a technique called Pressure Point

Analysis (PPA). This new approach for diagnosing performance limitations has been shown to systematically identify pressure points in software. While this research does not solve the diagnostic problem, we believe that further refinements of this approach can bring about an entirely automated process for diagnosing and potentially optimizing software.

Taking this analysis of computation performance one step further, we introduced a formal mathematical framework for reasoning about algorithmic and architectural tradeoffs necessary for designing the next generation of supercomputers. Our approach to algorithm-architecture co-design clearly demonstrates the impact architectural design decisions have on the efficiency of algorithms which run on the resulting hardware. This framework also gives us an anlytical tool to align the development of algorithms with the architectural trends destined for future supercomputers.

Bibliography

[1] *The potential impact of high-end capability computing on four illustrative fields of science and engineering.* Washington, DC, USA: The National Academies Press, 2008.

[2] "Intel architecture core analyzer," 2012.

[3] Adhianto, L., Banerjee, S., Fagan, M., Krentel, M., Marin, G., Mellor-Crummey, J., and Tallent, N. R., "HPCToolkit: Tools for performance analysis of optimized parallel programs," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 685–701, 2010.

[4] Aggarwal, A., Chandra, A., and Raghavan, P., "Energy consumption in VLSI circuits," in *Proceedings of the twentieth annual ACM symposium on Theory of computing - STOC '88*, (New York, New York, USA), pp. 205–216, ACM Press, 1988.

[5] Ahmed, K. and Schuegraf, K., "Transistor wars," *Spectrum, IEEE*, vol. 48, no. 11, pp. 50–66, 2011.

[6] Aragón, J. L., González, J., and González, A., "Power-aware control speculation through selective throttling," in *High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on*, pp. 103–112, IEEE, 2003.

[7] Association, S. and others, "International technology roadmap for semiconductors," *Semiconductor Industry Association, Tech. Rep*, 2010.

[8] Bailey, D. H., Chame, J., Chen, C., Dongarra, J., Hall, M., Hollingsworth, J. K., Hovland, P., Moore, S., Seymour, K., Shin,

J., and OTHERS, "PERI auto-tuning," in *Journal of Physics: Conference Series*, vol. 125, p. 012089, IOP Publishing, 2008.

[9] BARKER, K. J., HOISIE, A., and KERBYSON, D. J., "An early performance analysis of POWER7-IH HPC systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '11*, (New York, New York, USA), p. 1, ACM Press, 2011.

[10] BELLAS, N., HAJJ, I., POLYCHRONOPOULOS, C., and STAMOULIS, G., "Energy and performance improvements in microprocessor design using a loop cache," in *Computer Design, 1999.(ICCD'99) International Conference on*, pp. 378–383, IEEE, 1999.

[11] BERTRAN, R., GONZÀLEZ, M., MARTORELL, X., NAVARRO, N., and AYGUADÉ, E., "Counter-based power modeling methods: Top-down vs. bottom-up," *The Computer Journal*, 2012.

[12] BEYLS, K. and D'HOLLANDER, E. H., "Refactoring for data locality," *Computer*, vol. 42, no. 2, pp. 62–71, 2009.

[13] BLELLOCH, G. E., "Programming parallel algorithms," *Communications of the ACM*, vol. 39, pp. 85–97, March 1996.

[14] BLELLOCH, G. E., GIBBONS, P. B., and SIMHADRI, H. V., "Low depth cache-oblivious algorithms," in *Proc. ACM Symp. Parallel Algorithms and Architectures (SPAA)*, (Santorini, Greece), June 2010.

[15] BORKAR, S. and CHIEN, A. A., "The future of microprocessors," *Communications of the ACM*, vol. 54, p. 67, May 2011.

[16] BROOKS, D., TIWARI, V., and MARTONOSI, M., "Wattch: A framework for architectural-level power analysis and optimizations," in

Proc. Int'l. Symp. Computer Architecture (ISCA), (Vancouver, British Columbia, Canada), pp. 83–94, June 2000.

[17] Cannon, L. E., *A cellular computer to implement the Kalman filter algorithm*. PhD thesis, Montana State University, 1969.

[18] Casas, M., Badia, R. M., and Labarta, J., "Prediction of behavior of MPI applications," in *2008 IEEE International Conference on Cluster Computing*, pp. 242–251, IEEE, Sept. 2008.

[19] Chowdhury, R. A., Silvestri, F., Blakeley, B., and Ramachandran, V., "Oblivious algorithms for multicores and network of processors," in *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pp. 1–12, IEEE, 2010.

[20] Chung, E. S., Milder, P. A., Hoe, J. C., and Mai, K., "Single-chip heterogeneous computing: Does the future include custom logic, FPGAs, and GPGPUs?," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, (Atlanta, GA, USA), pp. 225–236, 2010.

[21] Cicotti, P., Mniszewski, S. M., and Carrington, L., "An evaluation of threaded models for a classical MD proxy application," in *Hardware-Software Co-Design for High Performance Computing (Co-HPC), 2014*, pp. 41–48, IEEE, 2014.

[22] Clapp, R., Fu, H., and Lindtjorn, O., "Selecting the right hardware for reverse time migration," *The leading edge*, vol. 29, no. 1, pp. 48–58, 2010.

[23] Culler, D., Karp, R., Patterson, D., Sahay, A., Schauser, K. E., Santos, E., Subramonian, R., and von Eicken, T., "LogP: Towards a realistic model of parallel computation," *ACM SIGPLAN Notices*, vol. 28, pp. 1–12, July 1993.

[24] Czechowski, K., Lee, V. W., Grochowski, E., Ronen, R., Singhal, R., Vuduc, R., and Dubey, P., "Improving the energy efficiency of big cores," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 493–504, 2014.

[25] Czechowski, K., McClanahan, C., Battaglino, C., Iyer, K., Yeung, P.-K., and Vuduc, R., "On the communication complexity of 3D FFTs and its implications for exascale," in *Proc. ACM Int'l. Conf. Supercomputing (ICS)*, (San Servolo Island, Venice, Italy), June 2012. (*to appear*).

[26] Czechowski, K. and Vuduc, R., "A theoretical framework for algorithm-architecture co-design," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pp. 791–802, IEEE, 2013.

[27] D'Alberto, P., Püschel, M., and Franchetti, F., "Performance/energy optimization of DSP transforms on the XScale processor," in *Proc. High Performance Embedded Architectures and Compilers (HiPEAC)*, vol. LNCS 4367, (Ghent, Belgium), pp. 201–214, January 2007.

[28] Dally, W. J., Balfour, J., Black-Shaffer, D., Chen, J., Harting, R. C., Parikh, V., Park, J., and Sheffield, D., "Efficient Embedded Computing," *Computer*, vol. 41, no. 7, pp. 27–32, 2008.

[29] de Melo, A. C., "The new linux perf tools," in *Slides from Linux Kongress*, 2010.

[30] Demmel, J. W., *Applied Numerical Linear Algebra*. SIAM, 1997.

[31] Eklov, D., Nikoleris, N., Black-Schaffer, D., and Hagersten, E., "Bandwidth bandit: Quantitative characterization of memory contention," in *Code Generation and Optimization (CGO), 2013 IEEE/ACM International Symposium on*, pp. 1–10, IEEE, 2013.

[32] Ernst, M., "Serializing parallel programs by removing redundant computation," Master's thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 1992.

[33] Esmaeilzadeh, H., Blem, E., St. Amant, R., Sankaralingam, K., and Burger, D., "Dark Silicon and the End of Multicore Scaling," in *Proceedings of the 28th International Symposiumn Computer Architecture (ISCA)*, (San Jose, CA, USA), 2011.

[34] Esmaeilzadeh, H., Cao, T., Yang, X., Blackburn, S., and McKinley, K., "What is happening to power, performance, and software?," *Micro, IEEE*, vol. 32, no. 3, pp. 110–121, 2012.

[35] Feng, W., Feng, X., and Ce, R., "Green supercomputing comes of age," *IT professional*, vol. 10, no. 1, pp. 17–23, 2008.

[36] Firasta, N., Buxton, M., Jinbo, P., Nasri, K., and Kuo, S., "Intel AVX: New frontiers in performance improvements and energy efficiency," *Intel white paper*, 2008.

[37] Fog, A., "The microarchitecture of Intel, AMD and VIA CPUs/An optimization guide for assembly programmers and compiler makers," 2014.

[38] Fog, A., McCalpin, J. D., and Murky, T., "Test results for Intel's Sandy Bridge processor," 2013.

[39] Fracis-Landau, M. D., Pemberton, N. T., and Schwermer, S., "Too many cooks in the kitchen: Gang scheduling for predictable performance," 2014.

[40] Frigo, M. and Johnson, S. G., "The design and implementation of FFTW3," *Proc. IEEE: Special issue on "Program Generation, Optimization, and Platform Adaptation"*, vol. 93, no. 2, pp. 216–231, 2005.

[41] Frigo, M., Leiserson, C. E., Prokop, H., and Ramachandran, S., "Cache-oblivious algorithms," in *Proc. Symp. Foundations of Computer Science (FOCS)*, (New York, NY, USA), pp. 285–297, October 1999.

[42] Gahvari, H., Baker, A. H., Schulz, M., Yang, U. M., Jordan, K. E., and Gropp, W., "Modeling the performance of an algebraic multigrid cycle on HPC platforms," in *Proceedings of the international conference on Supercomputing - ICS '11*, (New York, New York, USA), p. 172, ACM Press, 2011.

[43] Germann, T. C., McPherson, A. L., Belak, J. F., and Richards, D. F., "Exascale co-design center for materials in extreme environments," 2013.

[44] Gonzalez, J., Gimenez, J., Casas, M., Moreto, M., Ramirez, A., Labarta, J., and Valero, M., "Simulating whole supercomputer applications," *IEEE Micro*, vol. 31, pp. 32–45, May 2011.

[45] Gonzalez, R. and Horowitz, M., "Energy dissipation in general purpose microprocessors," *Solid-State Circuits, IEEE Journal of*, vol. 31, no. 9, pp. 1277–1284, 1996.

[46] Grochowski, E., Ronen, R., Shen, J., and Wang, H., "Best of both latency and throughput," in *Computer Design: VLSI in Computers and Processors, 2004. ICCD 2004. Proceedings. IEEE International Conference on*, pp. 236–243, IEEE, 2004.

[47] Hardavellas, N., Ferdman, M., Ailamaki, A., and Falsafi, B., "Power scaling: the ultimate obstacle to 1K-core chips," *Northwestern University, Technical Report NWU-EECS-10-05*, pp. 1–23, 2010.

[48] Hardavellas, N., Ferdman, M., Falsafi, B., and Ailamaki, A., "Toward dark silicon in servers," *IEEE Micro*, vol. 31, pp. 6–15, July 2011.

[49] Herczeg, Z., Kiss, Á., Schmidt, D., Wehn, N., and Gyimóthy, T., "Xeemu: An improved XScale power simulator," *Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation*, pp. 300–309, 2007.

[50] Hill, M. D. and Marty, M. R., "Amdahl's Law in the multicore era," *IEEE Computer*, vol. 41, pp. 33–38, July 2008.

[51] Hoefler, T., Schneider, T., and Lumsdaine, A., "LogGOPSim: Simulating large-scale applications in the LoGOPS model," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing - HPDC '10*, (New York, New York, USA), p. 597, ACM Press, 2010.

[52] Hoisie, A., Johnson, G., Kerbyson, D. J., Lang, M., and Pakin, S., "A performance comparison through benchmarking and modeling of three leading supercomputers: Blue Gene/L, Red Storm, and Purple," in *Proc. ACM/IEEE Conf. Supercomputing (SC)*, no. 74, (Tampa, FL, USA), November 2006.

[53] Hong, J.-W. and Kung, H., "I/O complexity: The red-blue pebble game," in *Proc. ACM Symp. Theory of Computing (STOC)*, (Milwaukee, WI, USA), pp. 326–333, May 1981.

[54] Intel, "Intel Composer XE 2013," 2013.

[55] Intel, R., "Intel 64 and IA-32 architectures optimization reference manual," *Intel Corporation, May*, 2015.

[56] Intel Corp., *Intel® Architecture Code Analyzer*, June 2012.

[57] Intel Corp., *Intel® 64 and IA-32 Architectures Software Developer's Manuals*, September 2013.

[58] Jagode, H., Knupfer, A., Dongarra, J., Jurenz, M., Mueller, M. S., and Nagel, W. E., "Trace-based performance analysis for the petascale simulation code FLASH," *International Journal of High Performance Computing Applications*, Dec. 2010.

[59] Jin, H., Frumkin, M., and Yan, J., "The OpenMP implementation of NAS parallel benchmarks and its performance," tech. rep., Technical Report NAS-99-011, NASA Ames Research Center, 1999.

[60] Kahng, A. B., Li, B., Peh, L.-S., and Samadi, K., "Orion 2.0: A fast and accurate noc power and area model for early-stage design space exploration," in *Proceedings of the Design, Automation, and Test in Europe (DATE) Conference & Exhibition*, vol. 29, pp. 1–6, IEEE, 2009.

[61] Karlin, I., Bhatele, A., Chamberlain, B. L., Cohen, J., Devito, Z., Gokhale, M., Haque, R., Hornung, R., Keasler, J., Laney, D., and others, "LULESH programming model and performance ports overview," *Lawrence Livermore National Laboratory (LLNL), Livermore, CA, Tech. Rep*, 2012.

[62] Karlin, I., Bhatele, A., Keasler, J., Chamberlain, B. L., Cohen, J., DeVito, Z., Haque, R., Laney, D., Luke, E., Wang, F., and others, "Exploring traditional and emerging parallel programming models using a proxy application," in *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pp. 919–932, IEEE, 2013.

[63] Karlin, I., McGraw, J., Gallarado, E., Keasler, J., Leon, E., and Still, B., "Memory and parallelism tuning exploration using the LULESH proxy application," *2012 SC Companion: High Performance Computing, Networking Storage and Analysis (SCC 2012)*, 2012.

[64] Kashnikov, Y., de Oliveira Castro, P., Oseret, E., and Jalby, W., "Evaluating architecture and compiler design through static loop analysis," in *High Performance Computing and Simulation (HPCS), 2013 International Conference on*, pp. 535–544, IEEE, 2013.

[65] Keckler, S. W., Dally, W. J., Khailany, B., Garland, M., and Glasco, D., "GPUs and the future of parallel computing," *IEEE Micro*, vol. 31, pp. 7–17, Sept. 2011.

[66] Kerbyson, D. J., Barker, K. J., Gallo, D. S., Chen, D., Brunheroto, J. R., Ryu, K. D., Chiu, G. L., and Hoisie, A., "Tracking the performance evolution of Blue Gene systems," in *Supercomputing*, pp. 317–329, Springer, 2013.

[67] Kirschenmann, W. and Plagne, L., "Optimizing computing and energy performances on GPU clusters: Experimentation on a PDE solver," in *Proceedings of the COST Action IC0804 on Large Scale Distributed Systems* (Pierson, M. and Hlavacs, H., eds.), pp. 1–5, IRIT, 2010.

[68] Kogge, P. and others, "Exascale Computing Study: Technology challenges in acheiving exascale systems," September 2008.

[69] Kogge, P. and Dysart, T., "Using the TOP500 to trace and project technology and architecture trends," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, p. 28, ACM, 2011.

[70] Koomey, J. G., Berard, S., Sanchez, M., and Wong, H., "Implications of historical trends in the electrical efficiency of computing," *Annals of the History of Computing, IEEE*, vol. 33, no. 3, pp. 46–54, 2011.

[71] Korthikanti, V. A. R. and Agha, G., "Analysis of parallel algorithms for energy conservation in scalable multicore architectures," in *Proc. Int'. Conf. Parallel Processing (ICPP)*, (Vienna, Austria), September 2009.

[72] Kufrin, R., "Perfsuite: An accessible, open source performance analysis environment for linux," in *6th International Conference on Linux Clusters: The HPC Revolution*, vol. 151, p. 05, Citeseer, 2005.

[73] Kung, H., "Let's design algorithms for VLSI systems," in *Proceedings of the Caltech Conference on VLSI: Architecture, Design, and Fabrication*, pp. 65–90, 1979.

[74] Lee, L. H., Moyer, B., and Arends, J., "Instruction fetch energy reduction using loop caches for embedded applications with small tight loops," in *Low Power Electronics and Design, 1999. Proceedings. 1999 International Symposium on*, pp. 267–269, IEEE, 1999.

[75] Lewis, A., Tzeng, N., and Ghosh, S., "Runtime energy consumption estimation for server workloads based on chaotic time-series approximation," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 3, p. 15, 2012.

[76] Li, S., Ahn, J. H., Strong, R. D., Brockman, J. B., Tullsen, D. M., and Jouppi, N. P., "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture - Micro-42* (Albonesi, D. H., Martonosi, M., August, D. I., and Mart'inez, J. F., eds.), no. c in MICRO 42, (New York, New York, USA), p. 469, HP Laboratories, ACM Press, 2009.

[77] Lim, K., Meisner, D., Saidi, A. G., Ranganathan, P., and Wenisch,

T. F., "Thin servers with smart pipes: designing SoC accelerators for memcached," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, pp. 36–47, ACM, 2013.

[78] Loan, C. V., *Computational frameworks for the Fast Fourier Transform*. SIAM, 1992.

[79] Loh, G. H. and Xie, Y., "3D stacked microprocessor: Are we there yet?," *IEEE Micro*, vol. 30, pp. 60–64, May 2010.

[80] Loh, G., "The cost of uncore in throughput-oriented many-core processors," in *Workshop on Architectures and Languages for Throughput Applications*, no. June, pp. 1–9, Citeseer, 2008.

[81] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K., "Pin: building customized program analysis tools with dynamic instrumentation," in *ACM Sigplan Notices*, vol. 40, pp. 190–200, ACM, 2005.

[82] Maleki, S., Gao, Y., Garzaran, M. J., Wong, T., Padua, D., and others, "An evaluation of vectorizing compilers," in *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pp. 372–382, IEEE, 2011.

[83] Malladi, K. T., Lee, B. C., Nothaft, F. A., Kozyrakis, C., Periyathambi, K., and Horowitz, M., "Towards energy-proportional datacenter memory with mobile DRAM," *ACM SIGARCH Computer Architecture News*, vol. 40, no. 3, pp. 37–48, 2012.

[84] Mandel, J. and Parter, S. V., "On the multigrid F-cycle," *Applied Mathematics and Computation*, vol. 37, pp. 19–36, May 1990.

[85] Martin, A. J., "Towards an energy complexity of computation," *Information Processing Letters*, vol. 77, pp. 181–187, February 2001.

[86] Marusarz, J., "Understanding how general exploration works in Intel VTune Amplifier XE," 2015.

[87] Mathis, M. M. and Kerbyson, D. J., "Performance modeling of unstructured mesh particle transport computations," in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, p. 245, IEEE, 2004.

[88] Maudlin, P., Bingert, J., House, J., and Chen, S., "On the modeling of the taylor cylinder impact test for orthotropic textured materials: experiments and simulations," *International Journal of Plasticity*, vol. 15, no. 2, pp. 139–166, 1999.

[89] McCullough, J., Agarwal, Y., Chandrashekar, J., Kuppuswamy, S., Snoeren, A., and Gupta, R., "Evaluating the effectiveness of model-based power characterization," in *USENIX Annual Technical Conf*, 2011.

[90] McMahon, F. H., "The livermore fortran kernels: A computer test of the numerical performance range," tech. rep., Lawrence Livermore National Lab., CA (USA), 1986.

[91] Mudge, T., "Power: A first-class architectural design constraint," *Computer*, vol. 34, no. 4, pp. 52–58, 2001.

[92] Numrich, R. W. and Heroux, M. a., "Self-similarity of parallel machines," *Parallel Computing*, vol. 37, pp. 69–84, Feb. 2011.

[93] Omtzigt, E. T. L., *Domain flow and streaming architectures: A paradigm for efficient parallel computation*. Phd dissertation, Yale University, 1993.

[94] Patterson, D., "Technology trends: The datacenter is the computer, the cellphone/laptop is the computer," October 2007. `www.hpts.ws/papers/2007/TechTrendsHPTSPatterson2007.ppt`.

[95] Pennycook, S. J., Hammond, S. D., Jarvis, S. A., and Mudalige, G. R., "Performance analysis of a hybrid MPI/CUDA implementation of the NAS-LU benchmark," in *Proceedings of the International Workshop on Performance Modeling, Benchmarking and Simulation*, (New Orleans, LA, USA), Nov. 2010.

[96] Rodrigues, A. F. and others, "The structural simulation toolkit," *ACM SIGMETRICS Performance Evaluation Review*, vol. 38, p. 37, Mar. 2011.

[97] Rouet-Leduc, B., Barros, K., Cieren, E., Elango, V., Junghans, C., Lookman, T., Mohd-Yusof, J., Pavel, R. S., Rivera, A. Y., Roehm, D., and others, "Spatial adaptive sampling in multiscale simulation," *Computer Physics Communications*, vol. 185, no. 7, pp. 1857–1864, 2014.

[98] Savage, J. E., *Models of Computation: Exploring the power of computing*. CC-3.0, BY-NC-ND, electronic ed., 2008.

[99] Schkufza, E., Sharma, R., and Aiken, A., "Stochastic superoptimization," in *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, pp. 305–316, ACM, 2013.

[100] Schulte, E., Dorn, J., Harding, S., Forrest, S., and Weimer, W., "Post-compiler software optimization for reducing energy," in *Proceedings of the eighteenth international conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS*, vol. 14.

[101] Schulz, M., Barry Rountree, B., Guix, M. C., and Bronevetsky, Greg Laguna, I., "Gremlin: Emulating exascale conditions on today's platforms," 2013.

[102] Shacham, O., Azizi, O., Wachs, M., Qadeer, W., Asgar, Z., Kelley, K., Stevenson, J. P., Richardson, S., Horowitz, M., Lee, B., and others, "Rethinking digital design: Why design must change," *Micro, IEEE*, vol. 30, no. 6, pp. 9–24, 2010.

[103] Shalf, J., Dosanjh, S., and Morrison, J., "Exascale computing technology challenges," *High Performance Computing for Computational Science–VECPAR 2010*, pp. 1–25, 2011.

[104] Shende, S. S. and Malony, A. D., "The TAU parallel performance system," *International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, 2006.

[105] Singhal, R., "Inside Intel next generation Nehalem microarchitecture," in *Hot Chips*, vol. 20, 2008.

[106] Snavely, A., Wolter, N., and Carrington, L., "Modeling application performance by convolving machine signatures with application profiles," in *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, pp. 149–156, IEEE.

[107] Solomonik, E. and Demmel, J., "Communication-optimal parallel 2.5D matrix multiplication and LU factorization algorithms," tech. rep., University of California, Berkeley, CA, USA, 2011.

[108] Sopeju, O. A., Burtscher, M., Rane, A., and Browne, J., "AutoSCOPE: Automatic suggestions for code optimizations using PerfExpert," *Evaluation*, 2011.

[109] Takizawa, H., Sato, K., and Kobayashi, H., "SPRAT: Runtime processor selection for energy-aware computing," in *Proc. IEEE Int'l. Conf. Cluster Computing (CLUSTER)*, (Tsukuba, Japan), pp. 386–393, October 2008.

[110] Thoziyoor, S., Muralimanohar, N., Ahn, J., and Jouppi, N., "Cacti 5.1," *HP Laboratories, April*, vol. 2, 2008.

[111] Toledo, S., "Locality of reference in LU decomposition with partial pivoting," *SIAM J. Matrix Anal. Appl.*, vol. 18, pp. 1065–1081, October 1997.

[112] Tramm, J. R., Siegel, A. R., Forget, B., and Josey, C., "Performance analysis of a reduced data movement algorithm for neutron cross section data in Monte Carlo simulations," in *Solving Software Challenges for Exascale*, pp. 39–56, Springer, 2014.

[113] Tramm, J. R., Siegel, A. R., Islam, T., and Schulz, M., "Xsbench-the development and verification of a performance abstraction for Monte Carlo reactor analysis," *The Role of Reactor Physics toward a Sustainable Future (PHYSOR)*, 2014.

[114] Tyagi, A., "Energy-Time Trade-offs in VLSI Computations," in *Foundations of Software Technology and Theoretical Computer Science*, vol. LNCS 405, pp. 301–311, 1989.

[115] Vetter, J. S., Lee, S., Li, D., Marin, G., McCurdy, C., Meredith, J., Roth, P. C., and Spafford, K., "Quantifying architectural requirements of contemporary extreme-scale scientific applications," in *High Performance Computing Systems. Performance Modeling, Benchmarking and Simulation*, pp. 3–24, Springer, 2014.

[116] Vogelsang, T., "Understanding the energy consumption of dynamic random

access memories," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 363–374, IEEE Computer Society, 2010.

[117] Woo, D. H. and Lee, H.-H. S., "Extending Amdahl's Law for energy-efficient computing in the many-core era," *IEEE Computer*, vol. 41, pp. 24–31, December 2008.

[118] Yasin, A., "A top-down method for performance analysis and counters architecture," in *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*, pp. 35–44, IEEE, 2014.