PERFORMANCE ANALYSIS AND MODELING OF PARALLEL APPLICATIONS IN THE CONTEXT OF ARCHITECTURAL ROOFLINES

by

NASHID SHAILA

A THESIS

Presented to the Department of Computer and Information Science
and the Graduate School of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Master of Science

June 2016

THESIS APPROVAL PAGE

Student: Nashid Shaila

Title: Performance Analysis and Modeling of Parallel Applications in the Context of Architectural Rooflines

This thesis has been accepted and approved in partial fulfillment of the requirements for the Master of Science degree in the Department of Computer and Information Science by:

Boyana Norris	Chairperson
Stephen Fickas	Member

and

Scott L. Pratt	Dean of the Graduate School

Original approval signatures are on file with the University of Oregon Graduate School.

Degree awarded June 2016

THESIS ABSTRACT

Nashid Shaila

Master of Science

Department of Computer and Information Science

June 2016

Title: Performance Analysis and Modeling of Parallel Applications in the Context of Architectural Rooflines

Understanding the performance of applications on modern multi- and manycore platforms is a difficult task and involves complex measurement, analysis, and modeling. The Roofline model is used to assess an application's performance on a given architecture. Not much work has been done with the Roofline model using real measurements. Because it can be a very useful tool for understanding application performance on a given architecture, in this thesis we demonstrate the use of architectural roofline data with measured data for analyzing the performance of different benchmarks. We first explain how to use different toolkits to measure the performance of a program. Next, these data are used to generate the roofline plots, based on which we can decide how can we make the application more efficient and remove bottlenecks. Our results show that this can be a powerful tool for analyzing performance of applications over different architectures and different code versions.

CURRICULUM VITAE

NAME OF AUTHOR: Nashid Shaila

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon, Eugene, OR
Bangladesh University of Engineering and Technology, Dhaka, Bangladesh

DEGREES AWARDED:

Master of Science, Computer and Information Science, 2016, University of Oregon
Bachelor of Science, Computer Science and Engineering, 2011, Bangladesh University of Engineering and Technology

AREAS OF SPECIAL INTEREST:

High Performance Computing
Performance Optimization

PROFESSIONAL EXPERIENCE:

Graduate Research Fellow, University of Oregon, Eugene, 2014

Software Engineer, Dohatec New Media, Bangladesh, 2012

Software Developer, Pubali Bank Limited, Bangladesh, 2011-2012

ACKNOWLEDGEMENTS

I'd like to thank my supervisor, Professor Boyana Norris. Her expertise on the subject matter was valuable during the early phase of this thesis. Her assistance in the preparation of this manuscript was valuable. I'll be grateful to her for helping me with useful insights and guiding me to the right direction.

TABLE OF CONTENTS

LIST OF FIGURES

Figure Page

LIST OF TABLES

Table Page

CHAPTER I

INTRODUCTION

High Performance Computing (*HPC*) systems are well known for their complexity, as the architecture makes it very challenging to achieve consistent performance on different platforms. For this reason, theoretical performance of these applications often significantly exceeds the actual performance. There will always be a gap between theoretical performance and measurement, as there is no standard way of evaluating the performance on manycore and multicore platforms. It makes performance optimization more difficult as well. This is especially true for mathematical or numerical functions because their performance can be hard to understand and estimate on cache-based architectures even for sequential runs. Hence, performance models are used to help scientists understand and improve performance. Measurement tools and performance models that rely on hardware counters enable us to achieve better understanding of application performance at the microprocessor level. The goal of a model should be to indicate any bottlenecks present in the application and point out the things that can be altered to achieve better performance on a target architecture.

Tools like *Intel's VTune* and *TAU* can be used to measure the performance counters and bytes of data transferred to and from the memory. The counters measure events such as different cache misses, *TLB misses*, floating-point instructions, total instruction, and time. But these are all data based, meaning they will often generate the reports and for analysis there are no well known visualization tool. As we know, visualization can play a very important part in finding new insights from a simple

text based data. Thus, if a model can be used to visualize all the data gathered from *VTune* and *TAU*, that would make the analysis easier.

One such model is *Roofline Performance Model*. This model is represented with a 2D-graph with arithmetic intensity (*FLOPS/Byte*) on the x-axis and performance (expressed as number of floating-point operations per second) on the y-axis. Thus it defines a performance envelope that helps reveal new information. The memory bandwidth creates an additional upper bound for computations with low intensity as the plot clearly distinguishes memory- and compute-bound computations. It also shows the behavior of intensity across sizes. But the greatest downside is that getting reliable data for this model is incredibly difficult. The papers that have used the model to date have done so through primarily back-of-the-envelope manual calculations rather than measurements Lo et al. [2014]. Here, in this paper, we use real application data, gathered through *TAU* and *VTune* to generate the plots.

Our goal is to analyze the performance of an *HPC* application in the context of the *Roofline* model. We used in the experiment three different benchmarks to generate the data used for this roofline based analysis. The first benchmark is *HPC* Interpolation/Particle Pusher/Maxwell solver. The second one is the *Stream* benchmark. It is a simple synthetic benchmark program that measures sustainable memory bandwidth (in *MB*/s) and the corresponding computation rate for simple vector kernels. The third one is *DGEMM*, which demonstrates the processor's and coprocessor's ability to do a multiplication of two matrices using double precision calculations.

The main challenge is to identify the functions (hotspots) inside the code that could limit the performances of the code (bottlenecks). We evaluate these three benchmarks on a single *HPC* platform for uniformity. The platform we use is *Edison*,

which is a *Cray XC30* supercomputer with *Intel "Ivy Bridge"* processor on the compute nodes.

The contribution of this thesis is twofold. First we devise a strategy to generate the roofline plots with real data. These data are generated using real applications, on real platforms and measured carefully with *TAU* and *VTune*. After a lot of trial and error we devised a technique to get the data reliably using the correct set of hardware counters in a multi-threaded environment.

The second contribution is the performance analysis of numerical kernels such as *Picsar*, and *DGEMM* with the roofline model. For the first time, we generate roofline plots for these benchmarks in a multithreaded environment. The plots prove that execution time and other metrics, such as cache utilization is sensitive to the parallel configuration of the problem. For example, dramatic differences in total time is observed for varying the *MPI* task/*OpenMP* thread combinations. Also, the *Roofline Toolkit* visualization enabled us to quickly evaluate differences between multiple versions of the same code components (or the entire application) in the context of architectural roofline.

In the second chapter, we provide background on the roofline model and arithmetic intensity along with brief description of the toolkits. Chapter III describes the related work. In Chapter IV we introduce the measurement methodology. We explain in detail how the performance data is collected using the hardware counters and profiling tools. Finally, in Chapter V we show a set of experiments on various benchmarks and how it can be used to provide useful information in analyzing and tuning applications. After a discussion of results we conclude.

CHAPTER II

BACKGROUND: DEFINITIONS, TOOLKITS, AND TECHNIQUES

In this chapter, we overview the concepts and tools on which this thesis relies.

Roofline Performance Model

The *Roofline* model is a visually intuitive performance model used to bound the performance of various numerical methods and operations running on multicore, many-core, or accelerator processor architectures Williams et al. [2009]. The model measures the performance of an application by combining locality, bandwidth, and different parallelization paradigms instead of using percent-of-peak estimates. This model can also be used for determining the inherent implementation and performance limitations.

Arithmetic Intensity

The *Roofline's* core parameter is *Arithmetic Intensity*. It is defined by the ratio of total floating-point operations to total bytes transferred (*FLOPS/byte*).

A *BLAS-1* vector-vector increment ($x[i]$ += $y[i]$), where x and y are arrays with N-elements, would have a very low arithmetic intensity of 0.0417 (N *FLOPS*/$24N$ *byte*) and would be independent of the vector size. Conversely, *FFTs* perform $5 * N * logN$ *FLOPS* for a N-point double complex transform. On a write allocate cache architecture, the transform would move at least $48N$ bytes Lab [2008].

Roofline Model

The basic *Roofline* model is used to bound floating-point performance as a function of machine peak performance, machine peak bandwidth, and arithmetic intensity. It can then be visualized by plotting the performance bound ($GFLOP/s$) as a function of *arithmetic intensity*. The resultant curve is the performance envelope under which the kernel or application performance exists.

Runtime vs. Arithmetic Intensity

The *Roofline* expresses the relationship between run time and arithmetic intensity. The example in Figure 1, shows that the runtime is independent of the degree of the polynomial until the machine balance is achieved. After that, runtime increases linearly with the degree of the polynomial Lab [2008].

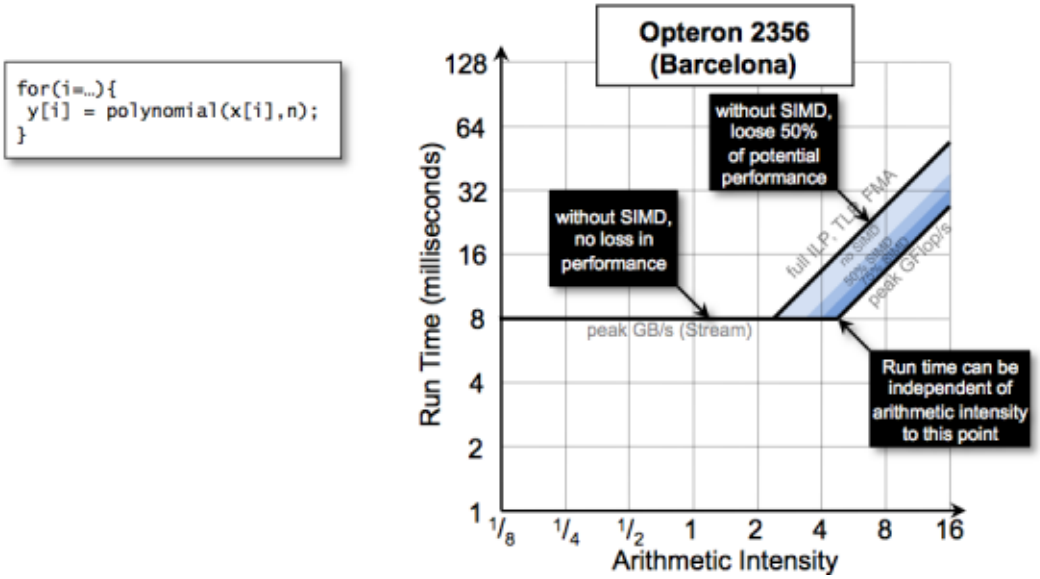

FIGURE 1. Runtime vs arithmetic intensity.

Effects of Cache Behavior on Arithmetic Intensity

The *Roofline* model requires an estimate of total data movement between *CPU* and main memory. Conflict misses and capacity of the cache increases data movement and as a result arithmetic intensity is reduced. Also, if the number of cache write-allocations is large, it can result in huge data movement. For example, the vector initialization operation $x[i] = 0.0$ allocates one write and write back per cache line. Here, write allocation is huge as every element of the cache line gets overwritten. But, it's very difficult to quantify how much compulsory data movement actually occurred during an execution Lab [2008].

Instruction-Level Parallelism and Performance

Deeply pipelined processor architectures can increase frequency and peak performance as well as they increase the latency of different instructions. To achieve peak performance the programmer has to make sure that independent instructions are issued in sequence. It's called instruction-level parallelism (*ILP*). Absence of this can decreases performance on compute-intensive kernels. On the other hand, it does not effect memory-intensive operations. The example in Figure 2, demonstrates good *ILP* by constructing partial sums and summing them at the end of the loop Lab [2008].

Data-Level Parallelism and Performance

Data-level parallelism such as *vectorization* (or *SIMDization*) is used widely to maximizing performance and energy efficiency Lab [2008]. However, obtaining peak performance depends greatly on the compiler/programmer's ability to insert these instructions in the right places. In cost of high *arithmetic intensity*, absence of

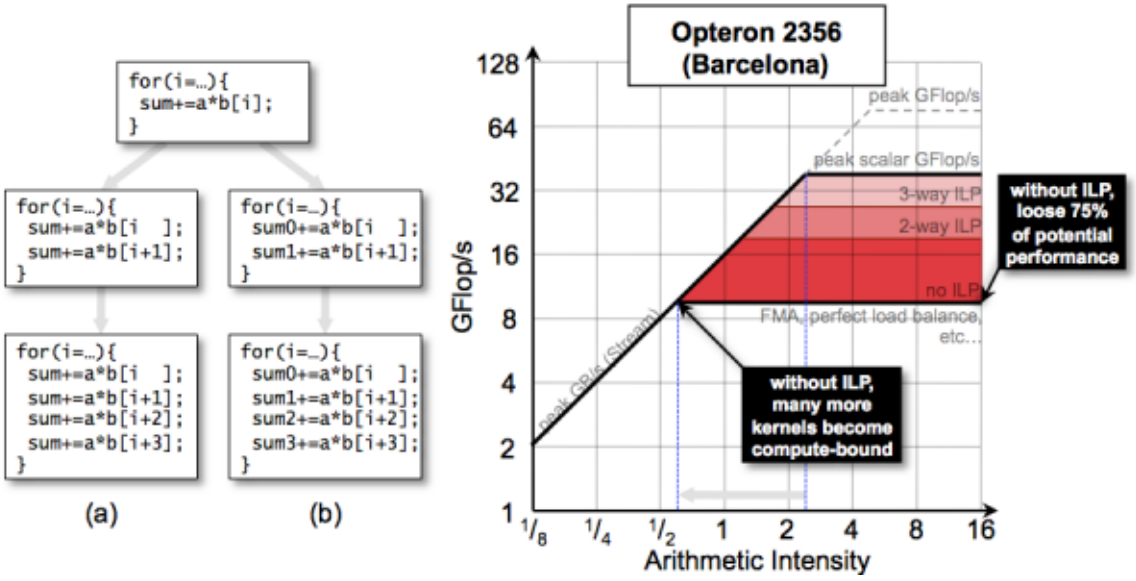

FIGURE 2. Instruction-level parallelism

SIMDization significantly affects performance. For example, if a high arithmetic intensity application such as matrix multiplication is done without *vectorization*, it can take a long time to run even for a moderately sized matrices. In the presence of *vectorization*, computation is fast as data is localized and number of misses becomes significantly lower. However, for low *arithmetic intensities*, the impact is negligible.

Performance Counters

In computing, hardware performance counters (or hardware counters) are a set of special purpose registers built into modern microprocessors to store the counts of hardware activities. Advanced users often rely on those counters to conduct low-level performance analysis or tuning of an application.

Many different sets of performance counters are available for different *CPUs*. They might have the same or different names across these platforms. In the same processor family, different models can differ considerably in the specific performance counters available. In general, these counters measure similar types of things, such

as registering the absolute number of cache misses, number of instructions issued, number of floating-point instructions executed, number of vector instructions, etc.

Nearly every processor commonly used these days is a cache-based machine. Caches offer high-speed access of instructions and data compared to the main memory of the system. They work on the principle of spatial and temporal locality. It is designed in a way that it can take advantage of the applications' tendency to frequently reuse blocks of data, which is denoted by the term temporal locality. Also applications tend to access data items near those already been used, which is denoted by spatial locality. An application's chance of achieving high performance is greater if it follows the above patterns on a cache-based processor. If the application is not performing well, it is the developer's job to find out why the processor is stalling instead of doing meaningful work. This is where performance counters can be useful.

Most of the time, an application is devised explicitly to run on specific computer hardware. The numbers generated by the performance counters are used to measure the exact performance of the application and point out the cases where it might gain additional performance.

Performance Modeling

Performance modeling is a structured approach to understanding the performance of an application. It typically begins during the early phases of application design and continues throughout its lifecycle.

In earlier days, performance of an application was generally ignored until there was a problem. There are several problems with this casual approach:

1. In the design phase, performance problems are frequently encountered.

2. Issues faced in design phase cannot always be fixed by tuning or efficient coding.

3. Fixing design issues later in the development cycle is not always possible. It is usually inefficient and can be very expensive.

The main aim of creating a performance model is to eliminate all those problems. When creating a performance model, performance objectives, such as response time, throughput, and resource utilization (*CPU*, memory, disk *I/O*, and network *I/O*) are set. Different application scenarios are identified as well. Performance scenarios are broken down into steps. Also performance budgets are assigned. This budget defines the resources and constraints across the performance objectives. The most significant benefits of performance modeling are:

1. Performance of the application becomes an important part of the design process.
2. By building and analyzing models, it becomes easier to evaluate the tradeoffs before the actual solution is built.
3. The design decisions that are influenced by performance can be identified. If these decisions are unidentified, it can lead to additional maintenance efforts.
4. Surprises in terms of performance can be avoided when the application is released into production.
5. It helps to see quickly what is important. That translates to where to instrument, what to test for, and how to know whether the application is on or off track for meeting the performance goal.

MPI/OpenMP

MPI

MPI stands for the *M*essage *P*assing *I*nterface. *MPI* addresses primarily the message-passing parallel programming model, in which data is moved from the address space of one process to that of another process through cooperative operations on each process Forum [2015]. This standardized *API* (application program interface) is mainly used in the fields of parallel and distributed computing. It is an specification that defines the syntax of a library routine, not the implementation. *MPI* operations are expressed as functions, subroutines, or methods, according to the appropriate language bindings that for *C* and *Fortran*, are part of the *MPI* standard Forum [2015]. There are several stable and efficient implementations of *MPI*, many of which are open-source. *MPI-1.0* (released in 1994) is the first installment. *MPI-3.1* is the latest stable version. The main advantage of using *MPI* is portability and ease of use. Furthermore it enhances scalability. The main goals of *MPI* include ensuring efficient communication, providing portable implementation, convenient *C* and *Fortran* binding for the interfaces, being language independent, allowing thread safety, and building reliable communication interfaces.

OpenMP

OpenMP stands for *Open M*ulti-*P*rocessing. At its most elemental level, *OpenMP* is a set of compiler directives and callable runtime library routines that extend *Fortran* (and separately, *C* and *C++*) to express shared-memory parallelism Dagum and Menon [1998]. The base language is unspecified and vendors can implement it in any language. It works on most platforms, processor architectures, and operating systems,

including *Solaris*, *AIX*, *HP-UX*, *LINUX*, *OS X*, and *Windows* Dagum and Menon [1998]. It consists of a set of compiler directives, library routines, and environment variables that influences the run-time behavior of an application. *OpenMP* is a flexible standard and can easily be implemented in different platforms. It has four parts: control structure, data environment, synchronization, and runtime library. It is frequently used in conjunction with *MPI* to exploit both shared- and distributed-memory parallelism.

Tuning and Analysis Utilities: TAU

TAU is a performance analysis framework developed at the *Oregon Performance Research Lab*. It consists of a suite of static and dynamic tools that provide graphical user interaction and interoperation to form an integrated analysis environment for parallel *Fortran*, *C++*, *C*, *Java*, and *Python* applications of Oregon Performance Research Lab. The two major features of the *TAU* framework are described below.

TAU Portable Profiling Package

The model that *TAU* uses to profile parallel, multi-threaded programs maintain performance data for each thread, context, and node in use by an application. The profiling instrumentation needed to implement the model captures data for functions, methods, basic blocks, and statement execution at these levels of Oregon Performance Research Lab. In the *TAU* profiling instrumentation, for *C/C++* for example, all the *C++* features are supported. It is available through an *API* at the application level. This *API* also provides selection of profiling groups for organizing and controlling instrumentation.

From the collected profile data, the profile analyzer of *TAU* can generate a range of performance information. For an application, it can show the inclusive and exclusive time spent in each function with resolution of a nanosecond. The breakup time for each installation can be shown for templated entities. Other measures include how many times each function was called, how many profiled functions each function invoked, and what the mean inclusive time per call was. Time information can also be displayed relative to nodes, contexts, and threads. Instead of time, hardware performance data can be shown. Also, user-level profiling is possible of Oregon Performance Research Lab.

TAU's profile visualization tool, *paraprof*, provides graphical displays of all the performance analysis results, in aggregate and per node/context/thread form of Oregon Performance Research Lab. This graphical interface is very useful in detecting the sources of performance bottlenecks in an application. *TAU* can also produce event traces. The *Vampir* trace visualization tool is used to display the traces.

TAU Code Analysis Package

The *TAU* static analysis tools are based on *PDT* (*P*rogram *D*atabase *T*oolkit) that produces an intermediate language (*IL*) representation. With various parsing tools such as *GNU* gfortran and *EDG* *TAU* supports sophisticated views of program structure, incorporating the latest *C++* language features such as templates, namespaces, and exceptions. Currently, the code analysis systems have been used to analyze *C*, *C++*, and *Fortran* source to automatically generate *TAU* profiling instrumentation of Oregon Performance Research Lab.

PerfExplorer

PerfExplorer is a framework for parallel performance data mining and knowledge discovery. The framework enables the development and integration of data mining operations that will be applied to large-scale parallel performance profiles Oregon Performance Research Lab [2005]. The goal of this framework is to use data mining techniques to analyze parallel performance data.

PerfExplorer supports different data mining approaches, such as clustering, association, regression, correlation, and summarization. Organizing data points into logical groups (or clusters) is defined as cluster analysis. Association is the technique of looking for relations in these clusters. The method of finding dependent/correlated and independent variables in the performance data is called regression. And finally, the process of finding out the similarities and dissimilarities between clusters is called summarization.

In addition, comparative analysis can be done by using *PerfExplorer*. The types of charts available include time steps per second, relative efficiency and speedup of the entire application, relative efficiency and speedup of one event, relative efficiency and speedup for all events, relative efficiency and speedup for all phases and runtime breakdown of the application by event or by phase Oregon Performance Research Lab [2005]. Moreover, events can be grouped together and the percentage of total runtime of the group can be displayed using another chart. All these analyses can be done within different parallel performance profiles and across different phases of execution.

Performance Application Programming Interface: PAPI

The *PAPI* project specifies a standard *A*pplication *P*rogramming *I*nterface (*API*) for accessing hardware performance counters available on most modern microprocessors ICL.

The measurement of these counters provides important insights of the correlation between the structure of the source code and the efficiency of mapping this code to the underlying architecture.. These correlations can be used in various cases, such as hand tuning of the code, compiler optimization, benchmarking, monitoring, debugging, and most importantly performance modeling. Understanding the mapping of the object code with the help of *PAPI* to the underlying architecture reduces the commonly occurring bottlenecks in high performance computing.

Description

PAPI provides two interfaces to the underlying counter hardware: a simple, high level interface for the acquisition of simple measurements and a fully programmable low level interface directed towards users with more sophisticated needs. The low level interface deals with hardware events in groups that are called *EventSets*. It gives a glimpse of how the counters are used in a system. The high level interface, on the other hand, can start, stop, and read specific events, one at a time.

PAPI can be divided into two layers of software. The upper layer contains the *API* and all machine independent functions. The lower layer translates this upper layer to machine dependent functions. These functions access a subtree that may contain assembly functions, operating system, or a kernel extension. Depending on availability, *PAPI* chooses among most efficient of the above three options. The

functionality of the upper level layer depends on the subtree immensely. *TAU* relies on *PAPI* for collecting performance measurements.

VTune

VTune Intel, is an integrated tool that automatically profiles the execution of an application on an Intel platform using performance counters. It generates a report detailing the breakdown of execution cycles. It is widely used for code profiling on *Intel* architectures. Examples include stack sampling, thread profiling, hardware event sampling etc. The report contains measures such as time spent in each subroutine that can be further break down to the instruction level. So, if there are any stalls in the pipeline during the execution, the time taken reflects that. *VTune* also facilitates thread performance analysis.

CHAPTER III

RELATED WORK

The *Roofline* Paper Williams et al. [2009] first introduced the the roofline plots with real data. In this chapter, we put together some papers that are closely related to this thesis and discuss about them.

Roofline Model Toolkit: A Practical Tool for Architectural and Program Analysis

In this paper, the authors describe a prototype architecture characterization engine for the *Roofline Toolkit* that quantifies the bandwidth and compute characteristics of multicore, manycore, and accelerated systems Lo et al. [2014]. The tool was used to benchmark four leading *HPC* systems: *Edison*, *Mira*, *Babbage*, and *Titan*. The paper describes the ability of each architecture to reach its peak bandwidth and the sensitivity to changes in arithmetic intensity or parallelism. They also propose a new benchmark to measure the performance of *CUDA* codes that is superior in performance compared to *Zero Copy* alternatives. Last, they evaluate three other *HPC* benchmarks on *Mira* by plotting their performance on an Roofline model to get a clear view of the application's performance.

Roofline: An Insightful Visual Performance Model for Floating-Point Programs and Multicore Architectures

This paper proposes an easy-to-understand, visual performance model that offers insights to programmers and architects on improving parallel software and hardware for floating point computations Williams et al. [2009]. The model is simple and

visual, and can be used to understand which systems would be a good match to important kernels, or to see how to change kernel code or hardware to achieve better performance. For floating-point kernels that do not fit completely in caches, this paper shows how operational intensity is an important parameter for both the kernels and the multicore computers.

The *Roofline* model offers insights into the difficulty of achieving the peak performance of a computer by making it obvious when a computer is imbalanced. Since nodes have different processing speeds, more jobs should be assigned to nodes with higher processing speed. If this is not present in a system, there will be an imbalance. Finding an optimal load distribution that uses dynamic load balancing is a challenging task. This paper investigates the existence of a synergistic relationship between performance counters and the *Roofline* model. The requirements for automatic creation of a *Roofline* model could guide the designer as to which metrics should be collected when faced with literally hundreds of candidates but a limited hardware budget.

Applying the Roofline Model

The *Roofline* model makes precise notions of memory and compute-bound applications and, can provide an insightful visualization of bottlenecks. To date the model has been used almost exclusively with back-of-the-envelope calculations and not with measured data. In this paper the authors show how to produce roofline plots with measured data on recent generations of *Intel* platforms Ofenbeck et al. [2014]. They also show how to accurately measure the necessary quantities for a given program using performance counters, including threaded and vectorized code, and for warm and cold cache scenarios. Later on, they explain in detail the approach to measure

and validate the results. They generate a set of roofline plots with measured data for some common benchmarks on different platforms.

The goal of the paper is twofold. First, it shows that the roofline model can be used with measurements rather than back-of-the-envelope calculations, but it proved to be very difficult because of the pitfalls arising from lower level system details. The second goal was to show that with measurements, roofline plots can be a valuable tool in performance analysis. This paper focuses on floating-point operations but the original model can easily be instantiated for integer computations. The work described in this thesis aims to address some of these difficulties.

PerfExpert: An Easy-to-Use Performance Diagnosis Tool for HPC Applications

The authors of this paper have developed *PerfExpert*, a tool that combines a simple user interface with a sophisticated analysis engine to detect probable core, socket, and node-level performance bottlenecks in each important procedure and loop of an application Burtscher et al. [2010].

As *HPC* applications take only a small fraction of peak performance to operate, the available performance evaluation tools need a lot of effort to learn this patterns. Performance optimization can be further complicated by the ongoing migration to multi-core and multi-socket compute nodes. As most of the available performance evaluation tools are not very accurate application writers don't use them.

PerfExpert provides an alternate performance assessment that is concise. It also suggests steps to improve performance that can guide application developers. These steps include compiler switches and optimization strategies with code examples. Many experiments have been done using *PerfExpert* and in all cases, it has correctly

identified the critical code sections and provided accurate assessments of their performance.

Online Performance Analysis by Statistical Sampling of Microprocessor Performance Counters

*Hardware p*erformance *c*ounters (*HPC*s) are used to analyze performance and identify the causes behind performance bottlenecks. But these are difficult to use as microprocessors almost always don't provide enough counters to monitor and collect data of these types of events that are important to understand the performance of a specific application.

In this paper Azimi et al. [2005], describe two techniques that help overcome these difficulties, allowing *HPC*s to be used in dynamic realtime optimizers. First, statistical sampling is used to dynamically multiplex *HPC*s and make a larger set of logical *HPC*s available. Using real programs, they show experimentally that it is possible through this sampling to obtain counts of hardware events that are statistically similar (within 15%) to complete non-sampled counts, thus allowing us to provide a much larger set of logical *HPC*s. Second, they observe that stall cycles are a primary source of inefficiencies, and hence they should be major targets for software optimization.

A simple model is built based on these observations. This model associates every stall cycle to a processor to find out what is causing the stall in realtime. This model is generated using the data collected form the *HPC*s multiplexing facility that monitors a large number of hardware components simultaneously. This analysis approach achieves an accurate model in an out-of-order superscalar microprocessor.

The results obtained in the study show that effective analysis of online performance of application and system code running at full speed. The stall analysis shows where performance is being lost on a given processor.

MuMMI: Multiple Metrics Modeling Infrastructure for Exploring Performance and Power Modeling

In this paper, the authors present the *MuMMI* framework, which consists of an Instrumentor, Databases, and Analyzer. Its an infrastructure that facilitates systematic measurement, modeling, and prediction of performance, power consumption and performance-power tradeoffs for parallel systems Wu et al. [2013].

The *MuMMI* system is based on three existing frameworks: *Prophesy* for performance modeling and prediction of parallel applications, *PAPI* for hardware performance counter monitoring, and PowerPack for power measurement and profiling. The *MuMMI* Instrumentor provides low overhead automatic performance and power data collection and storage. The database part extends the database of Prophesy for storing power and energy consumption. Also, hardware performance counter data is stored for different frequency settings. The analyzer part extends the data analysis component of Prophesy to support power consumption and hardware performance counters, and it supports performance and power modeling, performance-power trade-off and optimizations, and Web-based automated modeling system.

The *MuMMI* online system currently supports four modeling techniques: curve fitting, parameterization, kernel coupling, and performance-counters-based performance and power models for scientific applications online. It can also help

performance and power data measurement, storage, modeling and prediction of scientific applications on *XSEDE* website resources.

Prophesy: An Infrastructure for Performance Analysis and Modeling of Parallel and Grid Applications

This paper presents the *Prophesy* system and the use of its coupling parameter (i.e., a metric that attempts to quantify the interaction between kernels that compose an application) to develop application models Taylor et al. [2003].

In grid applications, like other applications, performance is an important issue. Understanding how the system features impact the performance of the applications is essential for faster execution. This knowledge is gathered by analysis and development of performance models. This paper discusses how the modeling techniques can be used in grid application analysis.

Design and Implementation of Prophesy Automatic Instrumentation and Data Entry System

In this paper, the authors present the *P*rophesy *A*utomatic *I*nstrumentation and *D*ata *E*ntry (*PAIDE*) system and describe its design framework and implementation supporting *C*, *Fortran77* and *Fortran90* programs on diverse systems Taylor and Stevens [2001].

The benchmarks used in this study are *NAS C*onjugate *G*radient (*CG*) and *I*nteger *S*ort (*IS*). They are used to analyze the *PAIDE's* instrumentation overheads for two granularities of instrumentation when problem size and number of processors increase. The experimental results on the *SGI Origin2000* show that the instrumentation overhead for *CG* benchmarks is less than 3.4%, while that for parallel

IS benchmarks is less than 1%. The result also shows that the *PAIDE* system does not affect the performance data at all. It nicely summarizes performance information of repeated events as sequential events. The *PAIDE* system is appropriate for parallel and distributed applications, which take a long time to execute.

Performance Analysis Using the MIPS R10000 Performance Counters

In this paper, the authors describe support in the *MIPS R10000* for non-intrusively monitoring a variety of processor events – support that is particularly useful for characterizing the dynamic behavior of multi-level memory hierarchies, hardware-based cache coherence, and speculative execution Zagha et al. [1996].

Performance tuning of supercomputer application needs proper analysis of the interaction of that application and the underlying architecture. This paper first explains the performance data collection approach. An integrated set of hardware mechanisms, operating system abstractions, and performance tools are used for that. Next, some scientific applications are used as examples to illustrate how the counters and profiling information can help the developers to analyze and tune the application.

Modeling Performance of Parallel Programs

The authors of this paper discuss performance modeling, an approach to understanding the performance of parallel systems. They present a survey of current approaches to modeling (both analytical modeling based on system parameters, and structural modeling based on the structure of the program) and propose a combination of these two approaches as a promising direction for new work Meira [1995].

If the actual performance of parallel programs is compared to the highest performance given by the underlying hardware, it is often disappointing.

Understanding the source of this difference is necessary for improving the performance of the application. The combination proposed in this paper is evaluated and later improvements are proposed that combines measurement and modeling.

Performance Measurement and Modeling of Component Applications in a High Performance Computing Environment

Parallel component environments place various constraints on performance measurement. Some applications require reusable performance interfaces for component interface monitoring. Moreover, observing component operation without access to the source code must be possible. This paper describes a non-intrusive, coarse-grained performance measurement framework that allows the user to gather performance data through the use of proxies that conform to these constraints Trebon [2005].

Performance models for individual components and also for the entire application can be generated from this data. They also describe a framework for validation that is used to validate the behavior and measurement methodologies of the framework by using known performance models of simple component based applications. Finally, modeling and measurement of a real application are studied and presented in the paper.

CHAPTER IV

METHODOLOGY

In this section, we discuss the approach taken for measuring the code-specific quantities and constructing the roofline plots. First, we describe the environment we used for experiments. Next, we explain the general way of using the performance counters to obtain the measurements and derived metrics. Finally, we illustrate how the experiments are performed, data is exported into the database, and plots are generated.

Environment and Specs: Edison

The *N*ational *E*nergy *R*esearch *S*cientific *C*omputing *C*enter (*NERSC*) is the primary scientific computing facility that we are using for our experiments. It operates under the Office of Science in the U.S. Department of Energy. *NERSC* is a division of the Lawrence Berkeley National Laboratory, located in Berkeley, CA. It is also one of the three divisions in the Berkeley Lab Computing Sciences area.

We used the machine *Edison* that is named after scientist Thomas Alva Edison. It is a *Cray XC30* machine, with a peak performance of 2.57 *Peta-FLOPS/sec.* It has $133,824$ compute cores, 357 terabytes of memory, and 7.56 petabytes of disk.

Edison has a total of 24 cores on each compute node. It has two sockets on each compute node. All the processors have *Intel H*yper-*T*hreading (*HT*) enabled, which means user can run with 48 logical cores per node. At runtime the user can decide to run with 24 cores per node (which is the default setting) or 48 logical cores per node.

Edison uses *Cray Aries* interconnects for inter-node communication. *Aries* provides a higher bandwidth, lower latency interconnect than *Gemini*, and should

exhibit reduced network congestion. *Edison's Aries* network is connected through a *Dragonfly* topology.

Edison has 12 login nodes that are external to the main compute portion of the system. Since the login nodes are external, users can login, access file systems and submit jobs when the main compute portion of the system is down for maintenance. The login nodes on *Edison* have 512 GB of memory.

Measurement

Edison has two sockets per node and each socket is populated with a 12-core Intel *"Ivy Bridge"* processor. There are 24 cores per node and 2 threads per core. So we experimented with up to 48 threads in various *MPI* process/threads combinations. We made sure to include 32 (e.g., 4 *MPI* tasks with 8 threads each) because that's what the architecture rooflines were generated with. The architectural roofline is the best performing *MPI* process/*OpenMP* thread configuration for a specific kernel. In general, the number of *MPI* processes and threads can be mixed and matched depending one the architecture. We did not use hyperthreading as the floating point measurements couldn't be done otherwise.

Counters for Measuring Floating-Point Operations

Performance counters exist on most modern microprocessors. These count hardware performance events such as cache misses, floating-point operations, etc. These counters are only incremented when either arithmetic or comparison instructions are issued. We used the *P*erformance *D*ata *S*tandard and *API* (*PAPI*) for measuring the counters. It provides a uniform interface to access these

performance counters. Table 1 lists the counters used for measuring floating-point operations.

TABLE 1. Counters for measuring floating-point operations and instructions on *Ivy Bridge*.

Event Mask Mnemonic	Events
PAPI_FP_OPS	Floating point operations
PAPI_FP_INS	Floating point instructions

Measuring Runtime

For measuring the performance of the application we focused only on the total runtime. To measure it we used the *TIME* variable of *UNIX gettimeofday()* function. The regular *Intel* timestamp counter is not used in our experiments. Because the regular timestamp is a system-wide counter, it measures any effects of the operating system. Moreover, in the parallel scenario, the reference cycles will report how many cycles are spent per *CPU*, but reconstructing their sum back into wall-clock time is not trivial due to partially overlapping regions. Hence, we used the *TIME* variable.

Measuring Memory Traffic

Various memory uses and leaks are measured with performance counters, derived metrics, and *VTune* during the experiments.

Counters for Memory Measurement

TAU can evaluate memory utilization options that examine how much heap memory is currently used and how well caches are being utilized . We used the memory measurement counters to collect these measurements. Memory instructions and shuffles are also recorded by the counters while the specific application executes

on the processor. Getting this measurement right is the most challenging task, as the set of counters differ between different microarchitectures. The counters we used for the *Ivy Bridge* microarchitecture are listed in Table 2.

TABLE 2. Counters for measuring memory operations on *Ivy Bridge*.

Event Mask Mnemonic	Events
PAPI_L3_TCM	Level 3 cache misses
PAPI_L2_TCM	Level 2 cache misses
PAPI_L3_TCA	Level 3 total cache accesses
PAPI_L2_TCA	Level 2 total cache accesses

Caveats

L3 cache misses are not a good way of estimating intensity as the caches are all write-allocate. Moreover, other events may cause data transfers like the prefetcher, page table loading, and streaming memory operations. An alternative approach is to measure the raw traffic on the memory controller. We used *VTune* measurements for getting the number of bytes transferred to and from memory. Figure 3 shows the list of counters we have measured on *Edison*.

Steps of the Experiments

Here we list all the necessary steps to generate and store results in the database.

Sampling with TAU

For this thesis, we used *TAU* *E*vent-*B*ased *S*ampling (*EBS*) to integrate sample information into *TAU* profile measurements at runtime. To enable sampling we used *"-ebs"* flag with the *tau_exec* command. This takes an un-instrumented binary

TABLE 3. List of the counters measured during the experiments on *Ivy Bridge*.

Event Mask Mnemonic	Events
PAPI_TLB_IM	Instruction translation lookaside buffer misses
PAPI_DP_OPS	Floating point operations; optimized to count scaled double precision vector operations
PAPI_TOT_INS	Instructions completed
PAPI_TOT_CYC	Total cycles
PAPI_SR_INS	Store instructions
PAPI_BR_INS	Branch instructions
PAPI_LD_INS	Load instructions
PAPI_REF_CYC	Reference clock cycles

as input and generates a flat profile with the sample data. While sampling the application is periodically interrupted and the running state of the program is examined. The samples are aggregated and a histogram of where the program spends its time is built. We used sampling because our codes have many lightweight functions and if we introduce instrumentation, too much overhead will be added. We sampled the codes by running it with the *tau_exec* command. The *tau_exec* command then tracks the memory events and creates separate profiles with the data.

Sampling MPI Applications

The *tau_exec* command allows us to sample an *MPI* application at runtime. Because the instrumentation is not done at runtime, the linking is all done dynamically. To use the *MPI* option, we simply place the *tau_exec* command before the application's executable when executing *mpirun*, and add the *mpi* flag:

$\$ > mpirun\ -n\ 4\ tau\_exec\ -T\ mpi,papi\ -ebs\ ./a.out$

TAU Configuration

We configure *TAU* to use *PAPI* for accessing the hardware performance counters. To do so, we downloaded and installed *PAPI* on *Edison*. Then, we build *TAU* using the $-papi = <dir>$ configuration option specifying the location of *PAPI*. *TAU* can be configured to record more than one hardware performance counter, along with time for each timer and routine. For enabling this feature, we added *"–MULTIPLECOUNTERS"* option while building *TAU*.

PAPI Events

Before measuring the events, we have to find out which *PAPI* events *Edison* actually supports. The command *papi_avail* is used for getting the whole list of hardware counters supported on *Edison*, as shown in Figure 3.

```
nshaila@edison12:~> papi_avail
Available PAPI preset and user defined events plus hardware information.
--------------------------------------------------------------------------------
PAPI Version             : 5.4.1.3
Vendor string and code   : GenuineIntel (1)
Model string and code    : Intel(R) Xeon(R) CPU E5-4603 0 @ 2.00GHz (45)
CPU Revision             : 7.000000
CPUID Info               : Family: 6  Model: 45  Stepping: 7
CPU Max Megahertz        : 2000
CPU Min Megahertz        : 2000
Hdw Threads per core     : 2
Cores per Socket         : 4
Sockets                  : 4
NUMA Nodes               : 4
CPUs per Node            : 8
Total CPUs               : 32
Running in a VM          : no
Number Hardware Counters : 11
Max Multiplex Counters   : 192
--------------------------------------------------------------------------------

================================================================================
  PAPI Preset Events
================================================================================
    Name        Code    Avail Deriv Description (Note)
PAPI_L1_DCM  0x80000000  Yes   No   Level 1 data cache misses
PAPI_L1_ICM  0x80000001  Yes   No   Level 1 instruction cache misses
PAPI_L2_DCM  0x80000002  Yes   Yes  Level 2 data cache misses
PAPI_L2_ICM  0x80000003  Yes   No   Level 2 instruction cache misses
PAPI_L3_DCM  0x80000004  No    No   Level 3 data cache misses
PAPI_L3_ICM  0x80000005  No    No   Level 3 instruction cache misses
PAPI_L1_TCM  0x80000006  Yes   Yes  Level 1 cache misses
PAPI_L2_TCM  0x80000007  Yes   No   Level 2 cache misses
PAPI_L3_TCM  0x80000008  Yes   No   Level 3 cache misses
PAPI_CA_SNP  0x80000009  No    No   Requests for a snoop
```

FIGURE 3. Partial output of *papi_avail* (on *Edison*).

Next, to find out the compatibility between each metric that *PAPI* is going to to profile, we used the *papi_event_chooser* command that produces the matching events that can be measured together as a list.

For example, in the following example, the event chooser tells us that there is an incompatibility in the choice of these four metrics: *PAPI_LD_INS*, *PAPI_SR_INS*, *PAPI_STL_ICY*, *PAPI_VEC_SP*. So they cannot be profiled together as shown in Figure 4. After we remove *PAPI_VEC_SP* and event chooser verifies that *PAPI_LD_INS* and *PAPI_SR_INS, and PAPI_STL_ICY* are compatible metrics and generates a list of compatible events that can be measured with this combination. This scenario is depicted by Figure 5.

```
nshaila@edison12:~> papi_event_chooser PRESET PAPI_LD_INS PAPI_SR_INS PAPI_STL_ICY PAPI_VEC_SP
Event Chooser: Available events which can be added with given events.
-------------------------------------------------------------------------------
PAPI Version             : 5.4.1.3
Vendor string and code   : GenuineIntel (1)
Model string and code    : Intel(R) Xeon(R) CPU E5-4603 0 @ 2.00GHz (45)
CPU Revision             : 7.000000
CPUID Info               : Family: 6  Model: 45  Stepping: 7
CPU Max Megahertz        : 2000
CPU Min Megahertz        : 2000
Hdw Threads per core     : 2
Cores per Socket         : 4
Sockets                  : 4
NUMA Nodes               : 4
CPUs per Node            : 8
Total CPUs               : 32
Running in a VM          : no
Number Hardware Counters : 11
Max Multiplex Counters   : 192
-------------------------------------------------------------------------------

Event PAPI_VEC_SP can't be counted with others -1
```

FIGURE 4. Output of *papi_event_chooser* (when metrics are not compatible).

Environment Variables Setup

After the *PAPI* events are chosen, we have to set the environment variable such as *TAU_METRICS*, *PATH*, *TAU_DIR*, and *TAU_MAKEFILE*. To set these, we have to export the variable values and list the *PAPI* metrics we would like to use in a

```
nshaila@edison12:~> papi_event_chooser PRESET PAPI_LD_INS PAPI_SR_INS PAPI_STL_ICY
Event Chooser: Available events which can be added with given events.
---------------------------------------------------------------------------
PAPI Version             : 5.4.1.3
Vendor string and code   : GenuineIntel (1)
Model string and code    : Intel(R) Xeon(R) CPU E5-4603 0 @ 2.00GHz (45)
CPU Revision             : 7.000000
CPUID Info               : Family: 6  Model: 45  Stepping: 7
CPU Max Megahertz        : 2000
CPU Min Megahertz        : 2000
Hdw Threads per core     : 2
Cores per Socket         : 4
Sockets                  : 4
NUMA Nodes               : 4
CPUs per Node            : 8
Total CPUs               : 32
Running in a VM          : no
Number Hardware Counters : 11
Max Multiplex Counters   : 192
---------------------------------------------------------------------------

    Name        Code    Deriv Description (Note)
PAPI_L1_DCM  0x80000000  No   Level 1 data cache misses
PAPI_L1_ICM  0x80000001  No   Level 1 instruction cache misses
PAPI_L2_ICM  0x80000003  No   Level 2 instruction cache misses
PAPI_L2_TCM  0x80000007  No   Level 2 cache misses
PAPI_L3_TCM  0x80000008  No   Level 3 cache misses
PAPI_TLB_IM  0x80000015  No   Instruction translation lookaside buffer misses
```

FIGURE 5. Output of *papi_event_chooser* (when metrics are compatible).

comma-separated list (along with *TIME* counter). Also, for some *MPI* applications, we need to set the *OMP_NUM_THREADS* environment variable to the number of threads to be used.

Generating Event Traces/Profiles

Profiling shows the distribution of execution time across routines. It also shows the code locations associated with specific measurements. Tracing the execution of a parallel program shows when and where an event occurred, in terms of the process that executed it and the location in the source code. For getting these profiles we first compile the source code of the application with appropriate flags and compiler optimization (described above). Next, we get the profiles by running the following command.

$\$ > srun\ -n\ \#procs\ tau\_exec\ -T\ ompt, mpi, papi\ -ompt\ -ebs\ ./picsar$

This generates the $profile.*.*$ files inside the *MULTI_TIME* and *MULTI_PAPI_** directories. The * in the profile file denotes the thread numbers and * in the *MULTI_PAPI_** denotes the name of the *PAPI* metric. All the *MULTI_PAPI_** directories are generated inside the main *TAU* directory.

We wrote a batch script for setting all the environment variables and submitting the batch jobs to *Edison's* scheduler using *sbatch* command. This is more efficient than setting variables and executing applications by hand.

Getting Text Summary of the Profiles

For quick representation of the summary of *TAU* performance, we use the *pprof* command. It reads and prints a summary of the *TAU* data on terminal window for the profiles present inside the current directory. For performance data with multiple metrics (*MULTI_PAPI_** and *MULTI_TIME* profiles), we have to move into one of the directories to get the information about that metric. This is shown in Figure 6.

```
nshaila@edison12:~/picsar_data_edison/mpi/new_ppcell10/n4_th8/MULTI__TIME> pprof
Reading Profile files in profile.*

NODE 0;CONTEXT 0;THREAD 0:
---------------------------------------------------------------------------------------
%Time    Exclusive    Inclusive       #Call      #Subrs  Inclusive Name
              msec   total msec                          usec/call
---------------------------------------------------------------------------------------
100.0        4,035       27,960           1       32843   27960863 .TAU application
 42.0            5       11,741          32          64     366932 OpenMP_PARALLEL_REGION: L_push_particles__23__par_loop0_2_0
 42.0        0.912       11,736          32          32     366765 OpenMP_IMPLICIT_TASK: L_push_particles__23__par_loop0_2_0
 42.0       11,735       11,735          32           0     366737 OpenMP_LOOP: L_push_particles__23__par_loop0_2_0
 40.6            0       11,349         227           0      49997 OpenMP_LOOP: L_push_particles__23__par_loop0_2_0 => [CONTEXT] OpenMP_LOOP: L_push_particles__23__par_loop0_2_0
 40.6            0       11,349         227           0      49997 [CONTEXT] OpenMP_LOOP: L_push_particles__23__par_loop0_2_0
 26.3            4        7,342          32          64     229442 OpenMP_PARALLEL_REGION: L_depose_currents_on_grid_jxjyjz__27__par_region0_2_0
 26.2            1        7,332          32         128     229134 OpenMP_IMPLICIT_TASK: L_depose_currents_on_grid_jxjyjz__27__par_region0_2_0
 26.2        7,325        7,325          64           0     114461 OpenMP_LOOP: L_depose_currents_on_grid_jxjyjz__27__par_region0_2_0
 25.8            0        7,200         143           0      50350 OpenMP_LOOP: L_depose_currents_on_grid_jxjyjz__27__par_region0_2_0 => [CONTEXT] OpenMP_LOOP: L_depose_currents_on_
grid_jxjyjz__27__par_region0_2_0
 25.8            0        7,200         143           0      50350 [CONTEXT] OpenMP_LOOP: L_depose_currents_on_grid_jxjyjz__27__par_region0_2_0
 25.6        7,150        7,150         142           0      50352 OpenMP_LOOP: L_depose_currents_on_grid_jxjyjz__27__par_region0_2_0 => [CONTEXT] OpenMP_LOOP: L_depose_currents_on_
grid_jxjyjz__27__par_region0_2_0 => [SAMPLE] depose_jxjyjz_scalar_1_1_1
 25.6        7,150        7,150         142           0      50352 [SAMPLE] depose_jxjyjz_scalar_1_1_1
 14.8            0        4,149          83           0      50000 .TAU application => [CONTEXT] .TAU application
 14.8            0        4,149          83           0      50000 [CONTEXT] .TAU application
 14.3        3,999        3,999          80           0      50000 OpenMP_LOOP: L_push_particles__23__par_loop0_2_0 => [CONTEXT] OpenMP_LOOP: L_push_particles__23__par_loop0_2_0 =>
[SAMPLE] getb3d_energy_conserving_1_1_1
 14.3        3,999        3,999          80           0      50000 [SAMPLE] getb3d_energy_conserving_1_1_1
 12.9        3,599        3,599          72           0      50000 OpenMP_LOOP: L_push_particles__23__par_loop0_2_0 => [CONTEXT] OpenMP_LOOP: L_push_particles__23__par_loop0_2_0 =>
[SAMPLE] gete3d_energy_conserving_1_1_1
 12.9        3,599        3,599          72           0      50000 [SAMPLE] gete3d_energy_conserving_1_1_1
  6.6            0        1,850          36           0      51410 OpenMP_LOOP: L_diagnostics_mp_calc_diags__31__par_region0_2_0 => [CONTEXT] OpenMP_LOOP: L_diagnostics_mp_calc_diag
s__31__par_region0_2_0
```

FIGURE 6. Partial output of *pprof* (metric is *TIME*).

Uploading Profiles on TAUdb

TAUdb (*TAU Data*base), previously known as *PerfDMF* (*P*erformance *D*ata *M*anagement *F*ramework), is a an *API/Toolkit* and a *DBMS* to manage and analyze performance data. The *API* is available in *Java* and *C*. We upload the profiles with a specific *MPI* rank, thread number of an application in the *TAUdb* by using the *taudb_loadtrial* command. The command takes several parameters to determine the application and experiment name.

$ *taudb_loadtrial* $-a$ <*appName*> $-x$ <*expName*> $-n$ <*name*>

We can use the multiple metrics profiles for the upload or we can pack it in **.ppk* format for the upload. For viewing the data uploaded in the database, the *paraprof* command is used.

Generating VTune Traces

For the *VTune* measurement, it is necessary to instrument the code with the *__itt_resume()* and *__SSC_MARK(0x111)* flags for starting the *VTune/SDE* profiling. For stopping the profiling, *__itt_pause()* and *__SSC_MARK(0x222)* flags are used. It will then get the data for that portion of the code and display the result in a text file. It is very important to use the *"-littnotify"* flag for compilation of the code, as it makes sure that *VTune* actually collects the data. This memory data is then parsed into a *CSV* (*C*omma *S*eparated *V*alue) file by another script and imported to the database.

```
__itt_resume();     // start Vtune
__SSC_MARK(0x111);  // start SDE instruction tracing
for (c = 0; c < m; c++) {
  for (d = 0; d < q; d++) {
    for (k = 0; k < p; k++) {
      sum = sum + first[c][k]*second[k][d];
    }

    multiply[c][d] = sum;
    sum = 0;
  }
}
__itt_pause();      // stop Vtune
__SSC_MARK(0x222);  // stop SDE tracing
```

FIGURE 7. *VTune* flags inside a program.

Generating Derived Metrics

For generating the derived metrics we write a script that takes the **.ppk* file of a trial and generates different derived metrics. This script uses the *perfexplorer* command to execute and outputs the data into a text file. Another script takes this file as input and generates *CSV* files. These derived metrics are then uploaded into the database using a *CSV* file loader.

During the experiments some of the above mentioned steps were executed manually. In future, all the steps will be partially or fully automated. The time limitation restricted our ability to do the automation at this time. However, we plan on starting the automation as soon as possible.

CHAPTER V

RESULTS AND ANALYSIS

In this section, we discuss the results of experiments with three codes. This discussion includes description of the benchmarks, problems faced for getting the correct results, analyzing performance, and studying the results within the roofline framework.

Benchmarks Used in the Experiments

We ran all our experiments on the *Intel*-based platform *Edison* (described in the Methodology section) of *NERSC*, running the *LINUX* operating system. For our experiments, we considered the following numerical kernels:

1. *Stream* is a simple synthetic benchmark program that measures sustainable memory bandwidth (in MB/s) and the corresponding computation rate for simple vector kernels. We used the latest *Stream OpenMP* version for the experiments. The source code has a hardcoded parameter, N, that is the length of the three arrays (A, B, and C) used in the tests. All of them are of type *DOUBLE PRECISION*. Hence, for 8 byte doubles $3 * 8 * N$ bytes are required. N should be big enough so that the tests touch a large portion of the memory.

2. *DGEMM*, the *Intel MKL* routine for multiplying matrices. It is the most widely used matrix multiplication routine of Intel. It calculates the product of *DOUBLE PRECISION* matrices. This benchmark is designed to measure the sustained, floating-point computational rate of a single node. However, we used

the *OpenMP* version of it to run it with different process/thread combination. It also uses the Intel *M*ath *K*ernel *L*ibrary (*MKL*) routines.

3. *Picsar* is a *P*article-*I*n-*C*ell (*PIC*) code where the main operations include particle pushing and field deposition and gathering. It uses *Hybrid MPI/OpenMP* scheme. The *HPC* kernel does interpolation, and simulates a particle pusher as well as a *Maxwell* solver. All of these are implemented in *Fortran90/Python* and parallelized with *MPI* and *OpenMP*. Parallelization on distributed memory architectures is very efficient in the implementation.

The Picsar benchmark is compiled using *ftn* (*Intel Fortran* compiler version 15.0.120141023). On the other hand, the *DGEMM* and *Stream* benchmarks are compiled with *cc* (*Intel C* compiler verison 15.0.120141023) compiler. All of the benchmarks uses flags *"-g -dynamic -openmp"* with different optimization flags. For *VTune* and *SDE* calculations, the *"-littnotify"* library flag is used during compilation for all the kernels.

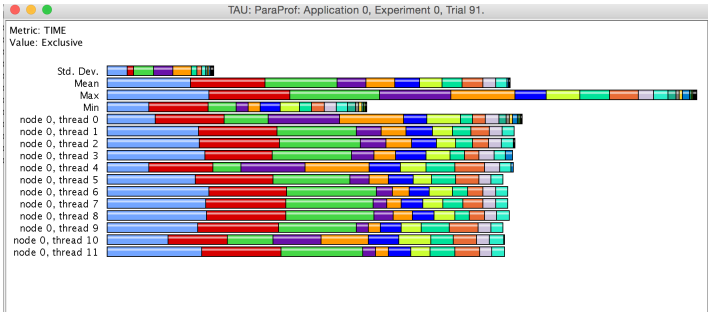

FIGURE 8. Load balance among different *DGEMM* threads on *Edison*.

Analysis of the Results

On a multicore environment, computation efficiency depends largely on load balancing, parallelization, code optimization, and proper utilization of memory

hierarchy. Hence, the data we collected on *Edison* included *CPU*, cache, memory measurements for each thread.

The next few sections will highlight the finding of the thesis along with pointing out the limitations of analyzing some of the cases properly.

A Study of Load Balancing in the Multithreaded Versions

First, we will look at how the multithreaded versions of different kernels were balanced among threads. Figure 8 shows the wall-clock time metric for different threads of node 0 for a *DGEMM* run. The runtime for each of the thread is a little unbalanced. The *pthread* overhead and *OpenMP* overhead are the two main overheads found on each thread.

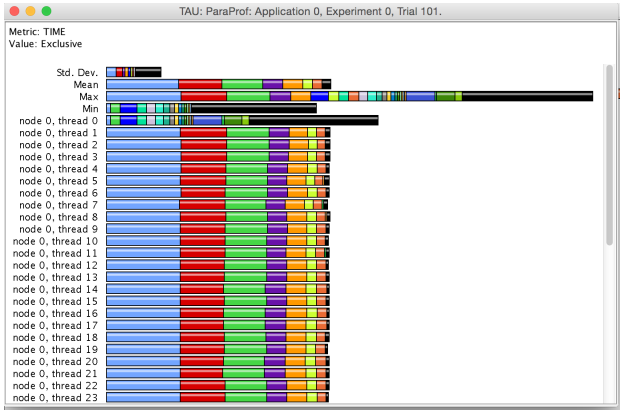

FIGURE 9. Perfectly balanced load among different *Picsar* threads.

Figure 9 shows the time metric calculated on different threads on a node for *Picsar*. From the figure, it is evident that the threads are almost perfectly balanced. All the threads have the same amount of *pthread* and *OpenMP* overheads, and the same computation time. Hence, additional load balancing for *Picsar* is not necessary on this architecture.

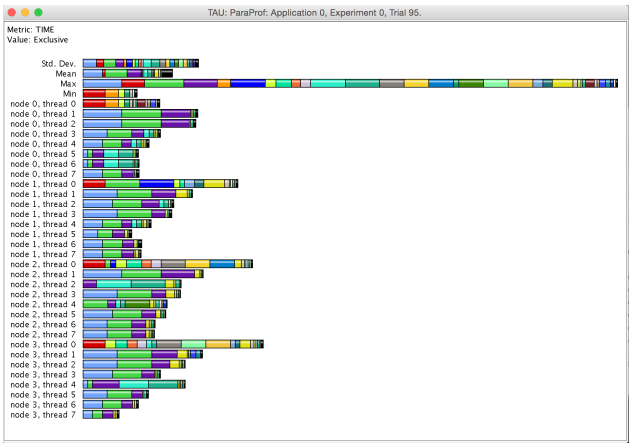

FIGURE 10. Load imbalance among different *Stream* threads.

Figure 10 shows the wall-clock time metric for *Stream* benchmark. Looking at figure, its can be easily seen that the threads are not at all balanced. The *pthread*, and *OpenMP* overheads, computation time is not at all same for the threads on different nodes. Moreover, if we look closely, it becomes clear that, the first thread of each node is not doing any numeric computation, rather it is acting like a master thread and distributing workload among the other threads. In addition, worker threads are performing different amounts of computation. So, load balancing measure can be taken to remove this imbalance and make the threads more balanced.

A Study of Total Execution Time

In this section, we will discuss the sensitivity of the execution time to the parallel configuration of the problem. For example, Figure 11 shows sizable differences in total time for different *MPI* task/*OpenMP* thread combinations (x-axis) for *Picsar*. As the number of threads go higher than 8, a significant jump in the runtime of the program because additional threads incur more *OpenMP* and *pthread* overheads.

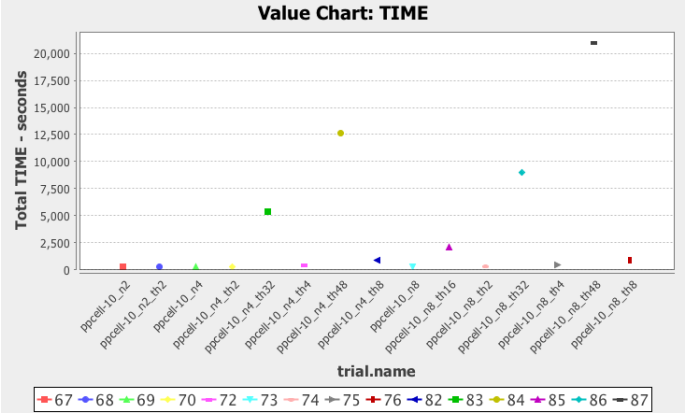

FIGURE 11. Total execution time of *Picsar*.

Figure 12 shows the same effect on total time for different *MPI* task/ *OpenMP* thread combinations (y-axis), now for *DGEMM*. Here, the runtime grows significantly as the *MPI*/thread combination goes higher. The correlation is subtle compared to the *Picsar*. However, with enough data points it can be demonstrated.

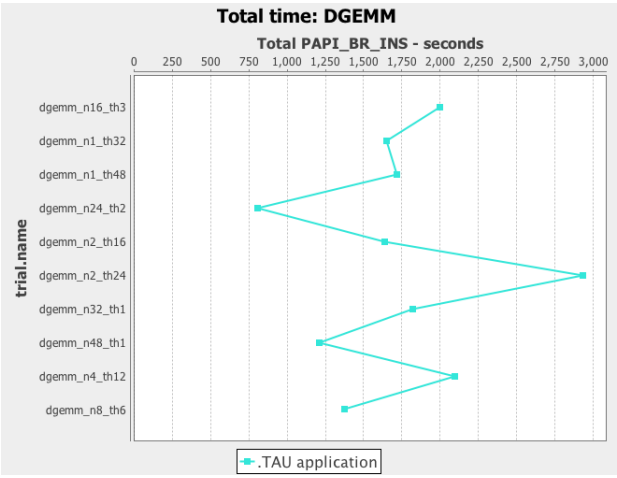

FIGURE 12. Total execution time of *DGEMM*.

Finally, for *Stream*, Figure 13 also shows the characteristics that total runtime goes higher with the different *MPI* task/ *OpenMP* thread combinations (y-axis). The runtime of the program grows proportionally with the number of threads per *MPI*

task. This occurs because *Stream* is limited by the available memory bandwidth, so adding more threads beyond a certain number decreases performance due to contention.

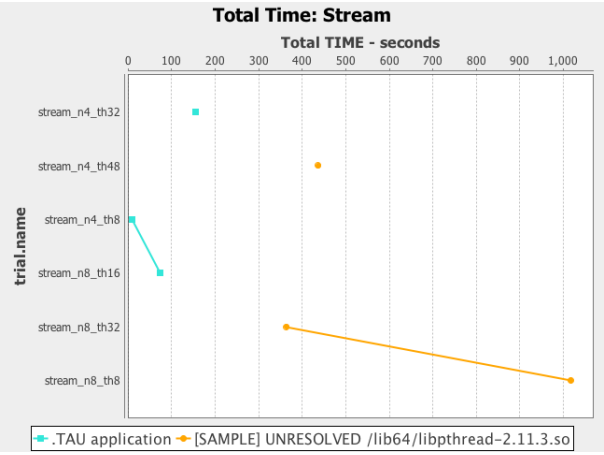

FIGURE 13. Total execution time of *Stream*.

A Study of L3 misses

We observed during the experiment that cache utilization is sensitive to the parallel configuration of the problem. Figure 14 shows the mean number of *L3* cache misses over varying numbers of threads for the same problem size for the dominant loop (particle push of Picsar).

From Figure 15, it is clear that cache performance is sensitive to the *MPI*/thread combination of the problem. It shows the number of *L3* cache misses over varying numbers of *MPI* processes and threads for the same problem size for the whole application (*Stream*).

From Figure 16, it is clear that cache miss is smaller when the *MPI* process is small and vice versa. This figure shows the number of *L3* cache misses over varying numbers of *MPI* processes and threads for the same problem size for the whole application

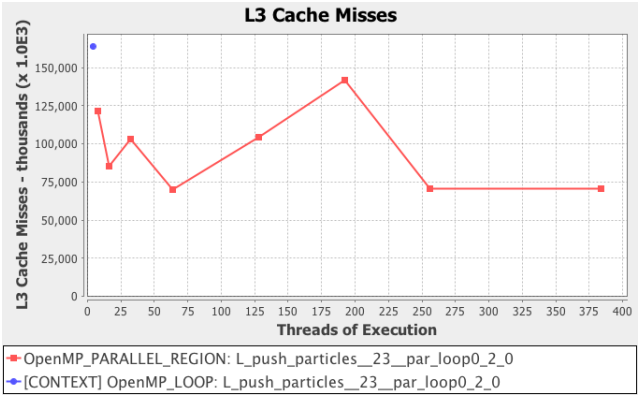

FIGURE 14. Cache misses vs threads of execution in the dominant loop *particle_push* (*Picsar*).

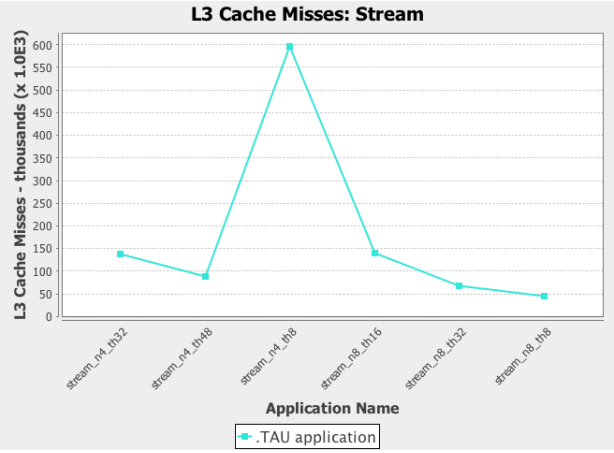

FIGURE 15. Cache misses vs *MPI*/thread combination in the application (*Stream*).

(*DGEMM*). As the process count goes higher, the number of misses grows as well. Hence, using fewer processes is good if the goal is to minimize the number of cache misses.

Application Performance in the Context of the Roofline Model

In addition to the hardware counters, another very effective method of evaluating how well a code is performing is by measuring its operational intensity. In this section

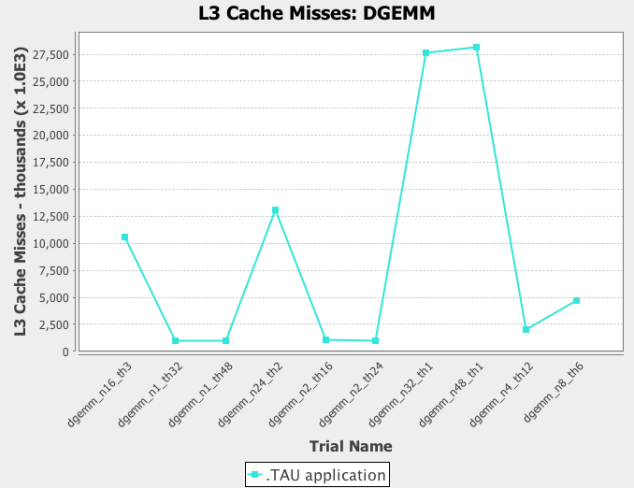

FIGURE 16. Cache misses vs *MPI*/thread combination in the application (*DGEMM*).

```
Search stanza is "EMIT_GLOBAL_DYNAMIC_STATS"
elements_fp_single_1 = 0
elements_fp_single_4 = 0
elements_fp_single_8 = 0
elements_fp_single_16 = 0
elements_fp_double_1 = 64000000000
elements_fp_double_2 = 0
elements_fp_double_4 = 0
elements_fp_double_8 = 0
--->Total single-precision FLOPs = 0
--->Total double-precision FLOPs = 64000000000
--->Total FLOPs = 64000000000
mem-read-1 = 702480
mem-read-2 = 16088
mem-read-4 = 320049296856
mem-read-8 = 96002435992
mem-read-16 = 64024
mem-read-32 = 0
mem-read-64 = 0
mem-write-1 = 526352
mem-write-2 = 0
mem-write-4 = 32032944304
mem-write-8 = 32001827040
mem-write-16 = 0
mem-write-32 = 0
mem-write-64 = 0
--->Total Bytes read = 2048218434400
--->Total Bytes written = 384146919888
--->Total Bytes = 2432365354288
```

FIGURE 17. *SDE* data for a single run of *DGEMM*.

we will describe how we used the RooflineToolkit to visualize the data we gathered for the kernels and how it can indicate performance of an application on a specific platform.

We modified and used the *RooflineToolkit* visualization front end (developed at the University of Oregon) to enable easy computation and visualization of the widely accepted arithmetic intensity metric. The x and y coordinates of the plots are generated using the *SDE* and *PAPI* measurements.

Figure 17 shows the *SDE* data for a single run of *DDGEMM*. From this text data we generate a *CSV* file and use that for computing the *operational intensity* and *GFLOPS/sec*. For example, if we want to compute the *operational intensity*, we take the total *Bytes* form the *CSV* file and divide it by total *FLOPS*. It will result in the x coordinate.

$$x = \frac{totalFLOPS}{totalBytes}$$

For calculating the *GFLOPS/sec*, we need to convert the total *FLOPS* to *GIGA FLOPS* (by dividing it by 10^9). Total runtime is fetched form *TAU* measurements, which is the inclusive runtime of the application. By dividing the *GFLOPS* and the inclusive runtime, we get the y coordinate.

$$y = \frac{totalFLOPS/10^9}{totalruntime}$$

This is how we calculate the (x, y) coordinates for each trial of an application and insert them in the *Roofline* visualization tool to generate the plots.

Figure 18 gives a general idea of the tool for the dominant loop in the *Picsar* kernel. Here, the performance of three versions of the *Picsar* kernel is shown along with *Edison's* roofline plot. The highest attainable performance on this machine is denoted by the roofline envelope. Compared to the highest performance of the system, *Picsar* performs moderately well. This plot takes into consideration the fact

that each node on Edison can take up to 48 threads. So, here is the plot for three different combination of *MPI*/thread combinations. Among the them, 48/1 attains highest performance (*GFLOP/sec*). 16/3 comes second and 8/6 comes third. From this, we can conclude that the *Roofline* visualization makes it easier to choose among different versions of the computation.

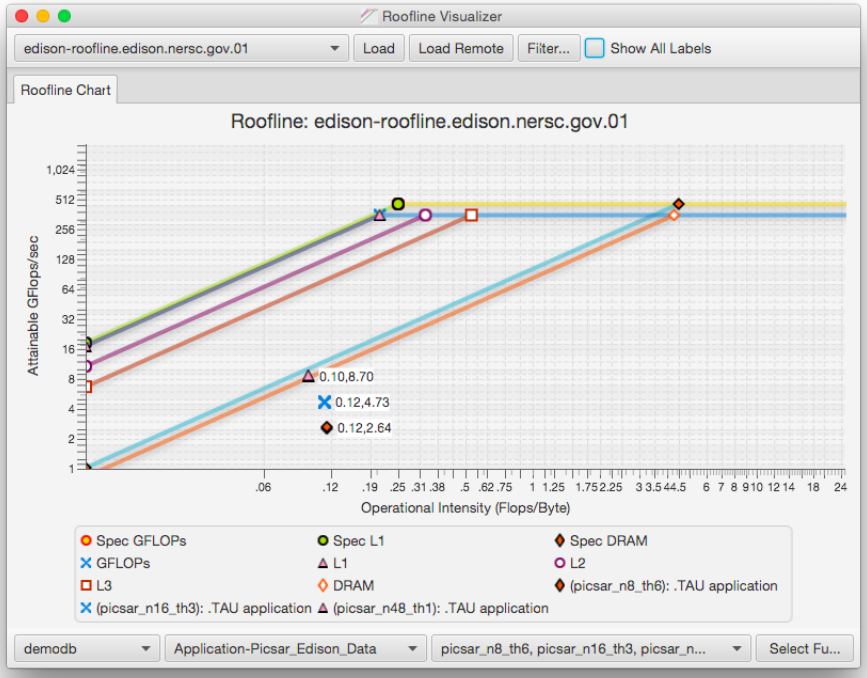

FIGURE 18. Arithmetic intensity of three versions of *Picsar* on *Edison*.

Figure 19 shows the same experiment but on *DGEMM* kernel. The figure shows plot for three different combination of *MPI*/threads. It is evident from this plot, that the *DGEMM* performs very well on *Edison*. 48/1 *MPI*/thread attains highest performance, which is almost equal to the highest performance possible. Here too, 16/3 comes second and 8/6 comes third. So, from this example, its clear that as the number of *MPI* tasks increases and threads decreases, performance increases.

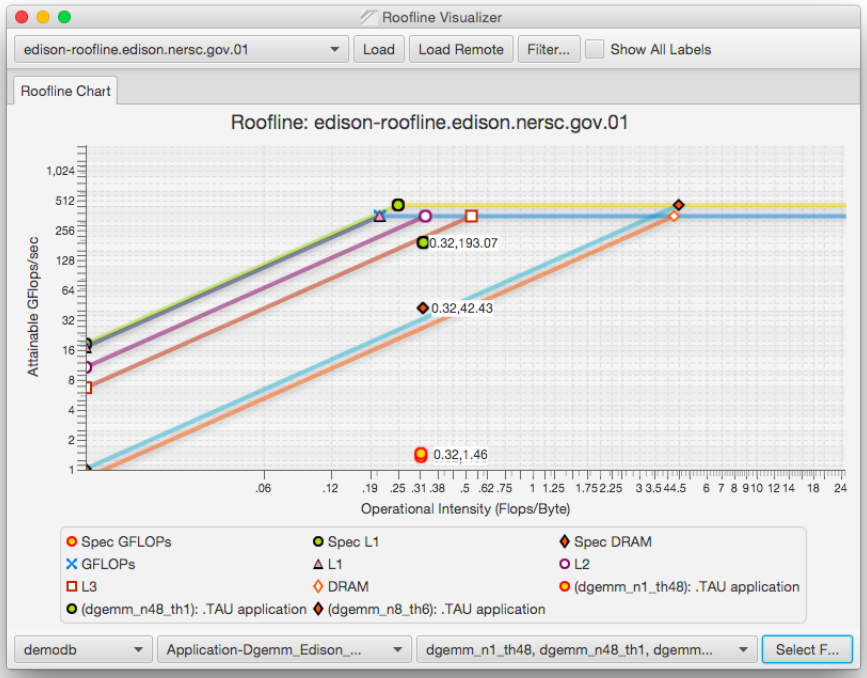

FIGURE 19. Arithmetic intensity of three versions of *DGEMM* kernel on *Edison*.

One important thing to note from the plots is that the operational intensity remains unchanged (mostly) across the trials. This is intuitive as all the trials are doing the same numeric calculation. Only their performance changes with the change in *MPI*/thread combination.

The biggest advantage of using the Roofline visualization graph is to validate the result and get a clear idea about how a specific benchmark would perform on a specific architecture. If the points are all over the place in this plot, most likely the kernel data is not captured correctly, which can be the case when using hardware counters. Also, just by looking at the plots, we can say what will be the best *MPI*/thread combination for a benchmark on a specific machine.

At the beginning, our goal was to enable quick evaluation of differences between multiple versions of the same code components (or the entire application) in the context of the architectural roofline, which shows both the upper bound and can also point to specific deficiencies (e.g., insufficient vectorization). The space being sampled include varying numbers of *MPI* tasks, varying *OpenMP* threads, different compiler optimization options, and as we create different code versions, they will also form one of the dimensions that must be tested to determine the best level and type of parallelism for a particular problem size (number of nodes, *MPI* and *OpenMP* configuration) with the objective of minimizing execution time. The outcome of the experiments proves that our approach is helpful in evaluating the performance of applications on different architecture.

CHAPTER VI

CONCLUSION

Visualizing application performance within the Roofline model can give a useful indication of how an application will run on a specific architecture with a specific *MPI* process/thread counts. This can be very helpful in predicting the performance of the same application on a different machine of similar specs. The plots also give an idea of which *MPI* process/thread counts works best for a given architecture. Moreover, calculating the cache misses in complex benchmarks, such as *Picsar* is almost impossible to do by hand because of the complexity of runtime cache utilization on multi- and manycore architectures. Hence, plotting the non-measured data in these cases can be misleading.

There were two main goals of this thesis. The first goal was to generate the roofline plots with measured data. However, it was very difficult to get the data as there were numerous obstacles. The compatibility issue of the toolkits with the compilers and different modules of *Edison*, default features, compiler optimizations flags, etc., were some of the major hindrances that we have overcome. Moreover, data generated by *PAPI* was not at all reliable. For example, cache misses measured by *PAPI* counters were way off the actual values of bytes transferred from and to main memory as measured by *SDE*. All of the above mentioned obstructions made the seemingly straight forward data generation immensely difficult and often time impossible. After a lot of trial and error we chose a correct set of hardware counters, modules, optimization flags to be used in a multi-threaded environment. The second goal was to analyze the performance of numerical kernels within the roofline plots to determine the efficiency of these kernels on different architecture.

We focused on floating-point computations mostly as we considered $FLOPS/Bytes$ as the computational intensity measure.

The experiments we did resulted in various performance insights. The roofline plots enabled us to get a quick comparison of performance between multiple versions of the same code components on a specific architecture. Also by analyzing the hardware counter values, we can estimate how an application characterized by a particular computational intensity will run given a specific set of *MPI* process/thread combination. We generated a number of plots to demonstrate how to use all these tools efficiently to analyze an application's performance.

In our experiments some of the measurement and analysis work was done manually. In future, we hope to automate all the steps fully or partially.

REFERENCES CITED

Yu Jung Lo, Samuel Williams, Brian Van Straalen, Terry J Ligocki, Matthew J Cordery, Nicholas J Wright, Mary W Hall, and Leonid Oliker. Roofline model toolkit: A practical tool for architectural and program analysis. In *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*, pages 129–148. Springer, 2014.

Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

Berkeley Lab. Performance and algorithm research, 2008. URL `http://crd.lbl.gov/departments/computer-science/PAR/research/roofline`.

Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard, Version 3.1*. High Performance Computing Center Stuttgart (HLRS), USA, 2015.

Leonardo Dagum and Ramesh Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, January 1998. ISSN 1070-9924. doi: 10.1109/99.660313. URL `http://dx.doi.org/10.1109/99.660313`.

University of Oregon Performance Research Lab. Tau homepage. URL `https://www.cs.uoregon.edu/research/tau/home.php`.

University of Oregon Performance Research Lab. Perfexplorer user's manual, 2005. URL `https://www.cs.uoregon.edu/research/tau/docs/perfexplorer`.

ICL. Papi website. URL `http://icl.cs.utk.edu/papi`.

Intel. Vtune website. URL `https://software.intel.com/en-us/intel-vtune-amplifier-xe`.

Georg Ofenbeck, Ruedi Steinmann, Victoria Caparros, Daniele G Spampinato, and Markus Puschel. Applying the roofline model. In *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*, pages 76–85. IEEE, 2014.

Martin Burtscher, Byoung-Do Kim, Jeff Diamond, John McCalpin, Lars Koesterke, and James Browne. Perfexpert: An easy-to-use performance diagnosis tool for hpc applications. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE Computer Society, 2010.

Reza Azimi, Michael Stumm, and Robert W Wisniewski. Online performance analysis by statistical sampling of microprocessor performance counters. In *Proceedings of the 19th annual international conference on Supercomputing*, pages 101–110. ACM, 2005.

Xingfu Wu, Hung-Ching Chang, Shirley Moore, Valerie Taylor, Chun-Yi Su, Dan Terpstra, Charles Lively, Kirk Cameron, and Chee Wai Lee. Mummi: multiple metrics modeling infrastructure for exploring performance and power modeling. In *Proceedings of the Conference on Extreme Science and Engineering Discovery Environment: Gateway to Discovery*, page 36. ACM, 2013.

website. Xsede website. URL `https://www.xsede.orgi`.

Valerie Taylor, Xingfu Wu, and Rick Stevens. Prophesy: an infrastructure for performance analysis and modeling of parallel and grid applications. *ACM SIGMETRICS Performance Evaluation Review*, 30(4):13–18, 2003.

Xingfu Wu Valerie Taylor and Rick Stevens. Design and implementation of prophesy automatic instrumentation and data entry system. In *Proc. 13th IASTED Intl Conf. Parallel and Distributed Computing and Systems (PDCS01)*. Citeseer, 2001.

Marco Zagha, Brond Larson, Steve Turner, and Marty Itzkowitz. Performance analysis using the mips r10000 performance counters. In *Supercomputing, 1996. Proceedings of the 1996 ACM/IEEE Conference on*, pages 16–16. IEEE, 1996.

Jr Wagner Meira. Modeling performance of parallel programs. *TR859. Computer Science Department, University of Rochester*, 1995.

Nicholas Dale Trebon. *Performance measurement and modeling of component applications in a high performance computing environment*. PhD thesis, University of Oregon, 2005.