ADAPTIVE, EFFICIENT PARALLEL EXECUTION OF PARALLEL PROGRAMS

by

Srinath Sridharan

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2014

Date of final oral examination: 09/23/2014.

Dedicated to my wife, Uthra, and my parents, Bhavani and Sridharan, for their unconditional love and support.

ABSTRACT

Future multicore processors will be heterogeneous, be increasingly less reliable, and operate in dynamically changing operating conditions. Such environments will result in a constantly varying pool of hardware resources which can complicate the programmer's task of efficiently exposing an application's parallelism onto these resources. Coupled with this complexity is the diverse set of performance objectives, such as latency, throughout, energy, power and resources, that users may desire. This dissertation proposes to automate the process of exposing an application's parallelism, in order to alleviate programmers from the burden of this complexity. It develops Varuna, a system that *dynamically*, *continuously*, *rapidly* and *transparently* adapts an application's parallelism to best match the instantaneous capabilities and availability of the hardware resources and the characteristics of the application, while optimizing different performance objectives.

To facilitate rapid parallelism adaptation, this dissertation develops a holistic and resource-agnostic *scalability model* based on Amdahl's law. Varuna first employs the model to rapidly estimates changes in efficiency during an application's parallel execution. It then uses formulae, derived from the model, to instantaneously determine the optimum degree of parallelism (DoP) to employ for different performance objectives and automatically guides the execution to the computed DoP.

In order for Varuna to transparently guide the application's parallel execution to the computed DoP, this dissertation proposes to employ a novel primitive called a *virtual task (vtask)*. Vtasks decouple application-level parallelism from hardware parallelism. They are progress-aware entities and give Varuna the flexibility needed to transparently control an application's parallel execution, without hampering its forward progress.

Varuna is programming model independent. It retains the existing programming abstractions and can be applied to both task-based and multithreaded shared memory parallel applications. Further, it requires no changes to the application or the Operating System, and can tackle arbitrary parallel applications that use standard APIs. This dissertation demonstrates Varuna for three different shared memory parallel programming APIs: Pthreads, Intel Thread Building Blocks (TBB), and Prometheus.

This dissertation evaluates Varuna in three different execution environments, isolated, multi-programmed and asymmetric, using unaltered C/C++ Pthreads, TBB and Prometheus applications from various standard benchmark suites, on three different real hardware platforms with different microarchitectural resource capabilities. Regardless of the execution environment, Varuna always outperformed the state-of-the-art approaches for the performance objectives considered.

ACKNOWLEDGMENTS

First and foremost, I would like to thank my wife, Uthra, for her unconditional love and support. Without her, I would not have come this far. She is my sunshine and my rock. I dedicate this dissertation to her. I also dedicate this dissertation to my mother, Bhavani, and my father, Sridharan, not only for the love and support they have given me, but also for everthing they have done for me in my life so far. My brother, Sriram, and my sister, Srinidhi, have always been there for me, and I will always be there for them.

I am ever indebted to my guru, Prof. Gurindar S. Sohi, who is responsible for etching my research career. He taught me how to approach a research problem, how to write research papers, and how to do presentations. He persisted with me even though I disappointed him several times. His confidence in my abilities has been uplifting and words cannot express my gratitude towards him.

I would like to thank Prof. Michael Swift for his relentless and detailed feedback on all articles I wrote, including this dissertation. He is a walking encyclopedia and I thoroughly enjoyed the conversations I had with him. I also would like to thank Prof. David Wood, who taught me the fundamentals of computer architecture and parallel programming. He is the best teacher I have ever seen. I also would like to thank other members of my thesis committee: Prof. Remzi Arpaci-Dusseau and Prof. Mikko Lipasti.

Most of my technical knowledge and expertise have come from my interactions with my peers in the computer sciences department. Gagan Gupta has been pretty much involved in all my research projects and has taught me how to organize my thinking, how to write papers, and most importantly, how to express my ideas to Guri. Matthew Allen mentored me during the early days of my PhD. He has been a great source of inspiration and taught me the importance of writing readable code. Sankaralingam Panneerselvam is a great friend who could answer every question related to Operating System internals. Hongil Yoon has challenged me with great questions on memory consistency models, coherence protocols, and directory structures. Jayaram Bobba is my go to person whenever I had any questions related to computer architecture. I thank James Wang for the wonderful discussions on transactional memory.

I have made many great friends during my stint at Wisconsin. Sankaralingam Panneerselvam, Ramakrishnan Durairajan, Sandeep Viswanathan, Suryanarayanan Panneerselvam, Venkatanathan Varadharajan, Vijay Chidambaram Velayudham Pillai, Thanumalayan Sankaranarayan Pillai, Sibin K Philip, Madhav Venkateswaran, Ragothaman Avanasi, and Ragunanth Venkatesh are a few. I thank all of them from the bottom of my heart. Without them, my life would have been hell in Madison.

DISCARD THIS PAGE

TABLE OF CONTENTS

Appendix

Page

DISCARD THIS PAGE

LIST OF TABLES

Table Page

DISCARD THIS PAGE

LIST OF FIGURES

Figure Page

Chapter 1

Introduction

For several decades, advances in semiconductor technology have provided computer architects with new opportunities for innovation. Computer architects have responded with waves of innovation that have resulted in a dramatic transformation of the microarchitecture of computer systems. These waves of innovation have resulted in more powerful processors and computing systems. Until recently, improvement in uniprocessor performance has mainly come from improving semiconductor technology and innovative architectural and microarchitectural techniques. Through this combination, uniprocessor performance has improved by several orders of magnitude in the past few decades. As a result, programmers could write applications in a familiar, sequential manner, yet have had remarkable performance improvements in a transparent manner. However, several fundamental limitations, including wire delay, power consumption, and design complexity, have forced hardware manufacturers to move from the decades-old approach of increasingly fast uniprocessors to building multicore processors. Today almost every common multiprocessor chip, even those used in mobile devices, is a multiprocessor with multiple processing cores on a single die communicating through shared caches.

The prevalence of multicore processors has had a significant impact on how software applications are being developed. They no longer automatically benefit from improvements in clock speeds the way did in the uniprocessor era. Multiprocessor computing is more effective only when software expects and take advantage of *parallelism*, i.e., harnessing multiple processing elements to cooperatively solve a single piece of work. Thus, multicore processors have effectively transferred the burden of improving software performance from computer system designers to software programmers.

Justifiably, the rapid transition from sequential software development to parallel software development has imposed additional burdens on software programmers. Specifically, they must:

1. Extract enough parallelism from the software to keep the processing cores busy,
2. Identify *shared data* and ensure accesses to them are synchronized to avoid *data races*, and
3. Manage and optimize the parallelism extracted in the software to maximize its efficiency.

The first aspect, extracting parallelism, is one of the critical aspects of parallel execution. This involves identifying appropriately-sized computations and compose the computations into independent units of work amenable to parallel execution. Since tasking the programmer with this aspect is regarded as excessively burdensome and error prone, there has been a plethora of work in parallel programming models to assist the programmers in extracting and reasoning about parallelism. Some examples include: Pthreads, which is a POSIX C API thread library that has standardized functions to create and managing native threads across various platforms, OpenMP [15], which provides compiler pragmas for expressing loop- and task-based parallelism, Intel's Threading Building Blocks (TBB) [109] which provides a palette of building blocks that run on top of native threads, Cilk Arts' Cilk++ [23, 49] which provides a rich environment to support fork-join parallelism, including a provably efficient work-stealing scheduler [24], support for parallel reductions [48] and parallel performance analysis; and Apple's Grand Central Dispatch [96] which includes elaborate sets of queues and thread pools to automatically handle the parallel execution of blocks of code, and analysis and visualization tools to facilitate an understanding of the parallel execution.

The second aspect of ensuring race free parallel execution is also quite challenging and error prone, especially if the programmer has little help from automatic tools. When a parallel computation identified is deemed independent, the programmer is guaranteeing the underlying hardware that it will not interfere with other computations that have also been identified as independent, in any dynamic run of the software. Since how a software executes dynamically depends upon a number of factors, including the input data set (of which the programmer typically has little knowledge), the programmer must ensure that any potential conflicting situation is handled by inserting appropriate synchronization. This can be quite burdensome for the programmer and there is a delicate balance: too much synchronization and the program tends towards a sequential program; too little synchronization and the program may have race conditions. There has been a large body of work in automatic tools to generate a properly synchronized parallel software once independent computations have been determined, thereby relieving the programmer of a task that has the potential to be cumbersome and error prone. Examples include [13, 34, 45, 60, 79, 91, 97, 98, 100, 111, 125, 126].

Both the above-mentioned aspects are important and we believe that researchers must continue to innovate in these aspects for the continued adoption of multicore processors. This dissertation, however, focuses on the third aspect of parallel software development: *Manage and optimize the parallelism extracted in the software to maximize its efficiency*. In particular, *static* and *dynamic diversity* among multicore processing chips, diversity in performance objectives, and dramatic increase in non-expert programmers and applications are identified as sources of both opportunities and challenges for future parallel application software development. *Attempting to address these opportunities and challenges creates uncertainty in terms of how the extracted parallelism must be dynamically exposed in an application on to the processing cores at a given instant in time in order to avoid underutilization or oversubscription of resources in the system.* Presently, it is the responsibility of programmers to unravel this uncertainty, which can severely hamper their productivity. This dissertation proposes and evaluates a run-time system that addresses this problem in an efficient manner, without the involvement of the application programmer, the OS or any other entity.

1.1 Motivation

1.1.1 Emergence of System Diversity

While the benefit of parallel execution in increasing efficiency of an application is well known, increased parallelism is not always beneficial. While too little parallelism leaves more room for efficient execution, too much parallelism can lead to execution inefficiencies, actually decreasing application performance. These inefficiencies may result from a variety of sources including, for example, increased contention for shared resources, such as memory capacity, bandwidth and locks, caused by increased parallelism, increased processing overhead from implementing increased parallelism, for example, in load balancing or scheduling, and changes in the workload behavior as in the case of web services, such as search and video.

As the computing landscape evolves, at a phenomenal pace, growing system diversity is likely to pose further challenges to efficient parallel execution. *Microarchitectural diversity* is growing since computing devices across the spectrum, from the low end (mobile devices) to the high end (servers), employ rapidly evolving role-specific microarchitectures. For example, today each new multiprocessor has a different number of processing cores, cache/memory configurations, and interconnect and I/O technologies. Further, many computing chips have graphics processing cores, and other accelerators, alongside multiple, homogeneous, general-purpose processing cores. Chips have been announced that have statically heterogeneous general-purpose processor cores (e.g., ARM's Big-Little [68]) and research has proposed multicore processors with a variety of heterogeneity such as static heterogeneity [74, 76, 58, 63, 112, 119], statically-homogeneous but dynamically heterogeneous cores [33, 14], cores that execute different instruction sets [127, 39, 41], and others. Within a system, *dynamic diversity* will arise from a variety of sources, including hardware defects, process variability, dynamic voltage and frequency scaling, dynamic techniques to handle power, etc. Furthermore, the growing popularity of multiprogrammed systems, e.g., mobile devices and cloud services, will increase the diversity of co-located applications.

The above trends share two distinct traits.

- First, application programs will have to run in parallel on a diverse set of hardware whose microarchitectural (cores, sockets, cache capacity, memory bandwidth, I/O bandwidth, etc) resource capabilities are not only unknown statically, but may keep changing dynamically, even as the software executes. Importantly, in such environments, the application must run efficiently, since the value of the application is likely to be determined by its efficiency: application vendors will be able to charge more for applications that are more efficient.

- Second, they create a new layer of complexity for programmers in how the application's extracted parallelism should be efficiently exposed on to the hardware at any given moment. Unraveling this complexity requires an understanding of the application's characteristics, an intimate knowledge of system, knowledge of the system's dynamic operating conditions, and the ability to dynamically and continuously tune the application's parallel execution.

1.1.2 Diversity in Performance Objectives

Coupled with the uncertainty in the capabilities of resources is the diverse set of performance objectives that the application user enforces. In the past, performance used to be the primary objective. Today, objectives typically vary according to the environment the program is running upon. For example, energy efficiency may be the most important in a mobile environment while total resource consumption may be more important in a cloud.

1.1.3 Dramatic Increase in Programmers and Applications

In the past, efficiency was a concern for a select few parallel application programmers, but in the future it will be for many more. Until recently a small set of experts developed parallel application for a very small set of machines. They tuned applications using intimate knowledge of the host, e.g., its microarchitecture. Often, they could assume the host was entirely available to their applications, and expect little or no interference from other applications. Going forward, we expect a multitude of programmers to program commodity parallel systems, e.g., the vast array of cell phones, laptops, desktops, etc. We expect applications to be written without the detailed knowledge of the hardware, software, and dynamic operating conditions. Further, given the possibly large number of diverse target hosts, portability will be highly desired. Therefore, to achieve efficient parallel execution, it will be daunting for common programmers to account for the often complex and non-intuitive factors related to application behavior, operating conditions and system characteristics. They will need solutions that are automatic, yet effective.

1.2 Thesis Statement

The confluence of the above-mentioned factors leads to the following thesis statement:

Optimizing efficiency of a parallel application on future systems will require automatic, dynamic, continuous and programmer-transparent harmonization of its parallelism with the execution environment, due to the emerging challenges and opportunities of static and dynamic diversity among multicore chips, diversity in performance objectives, and dramatic increase in non-expert programmers and applications.

1.3 Shortcomings of Existing Proposals

Prevalent approaches take a limited view of the efficiency issues programmers face. The traditional multithreaded models put the onus of efficient execution largely on programmers. Task-based programming models, e.g., Intel Thread Building Blocks (TBB) [109], ease some of the burden. They automate load-balancing of tasks to prevent resource underutilization. However, they neither address overutilization, nor account for co-scheduled applications.

Current automated approaches to optimize an application's parallel execution fail to take a comprehensive view of the prevailing programming methods and the system diversity. Some proposals require applications be rewritten from scratch using their own APIs that are not widely adopted [106, 107, 101]. Others [113, 116, 37, 38] that work with existing APIs are applicable only to task-based programming models [109, 15, 49]. Most of the approaches [116, 37, 38] can handle only data parallel applications and the ones that propose to tackle arbitrary applications require compiler or programmer support [106, 107, 113]. Importantly, none of these techniques are applicable to arbitrary multithreaded applications. Some prevent oversubscription only some resources, may require compiler support or offline profiling, and may be ineffective in multiprogrammed environments [116, 80, 37, 38]. Further, these approaches use hill-climbing search heuristics to find the right operating point, and hence may fail to react swiftly to changing conditions. Moreover, they optimize only for performance and do not take into account other performance objectives, such as resource consumption, power or energy.

1.4 Thesis Proposal

This dissertation proposes **Varuna**, a system that *dynamically*, *continuously*, *rapidly* and *transparently* adapts an application's dynamically exposed parallelism to best match the dynamic hardware resource capabilities and its workload characteristics, as well as to optimize diverse performance objectives. Specifically, Varuna *automatically* determines how many parallel computations to execute in parallel, how to interplay different computations, when to execute these computations and how to map them on to the underlying processing cores.

The key highlights of Varuna are as follows:

- **No programmer involvement.** Varuna does not perform automatic parallelization of sequential applications. It requires application programmers to identify appropriately-sized computations and the data shared between them, and compose the computations using algorithms and data structures amenable to parallel execution. However, unlike the traditional parallel programming models, Varuna separates the concern of expressing parallelism from the concern of optimizing the parallelism for different execution environments. This relieves the programmers, users or administrators from worrying about the intricacies and vagaries of parallel execution and allows them to concentrate on discovering parallelism and writing functionally correct parallel applications.

- **Programming model agnostic.** Varuna provides a transparent solution to optimize an application's parallel execution. While the optimization strategy employed by existing adaptive techniques are tightly coupled to a particular parallel programming model, Varuna's mechanisms and models are programming model agnostic. Programming models are often useful because of what they forbid, rather than what they allow. Using the wrong programming model for a particular computation can result in subpar performance, as also noted by Pan, et. al [101]. For example, Cilk is considered to be high-performance and general purpose. However, it lacks expressiveness and destroys cache locality for certain linear algebra operations, achieving worse performance than an equivalent pipeline implementation. With Varuna, application programmers are free to choose a suitable programming model to decompose their parallel application and Varuna will automatically optimize the application's dynamically exposed parallelism without any additional inputs from the programmer.

- **Optimize general-purpose parallel applications.** Existing parallelism optimization techniques primarily focus on applications that employ data parallelism. Whereas Varuna's methodology can optimize the parallel execution of general purpose parallel applications with arbitrary dependence patterns.

- **No Operating System (OS) involvement.** Varuna's principles may be incorporated into an operating system, or even the hardware. To broaden its applicability, however, we implemented Varuna as a run-time library agnostic to both the OS and the application.

- **Resource agnostic.** Varuna takes a principled approach to optimizing parallel execution efficiency, as opposed to ad-hoc trial and error methods employed by existing techniques. It employs a novel holistic, resource agnostic scalability model based on Amdahl's law to estimate changes in efficiency during an application's execution. It then uses formulae, derived from the model, to rapidly determine the optimum parallelism to employ and automatically guides the program's execution to match the optimum parallelism.

- **Multiple performance objectives.** Varuna can adapt to multiple performance objectives, rather than just performance. This dissertation demonstrates Varuna's efficacy for two different performance objectives as follows: (i) Improve application throughput, and (ii) Minimize resource consumption, i.e., the CPU consumption-execution time product.

1.5 Dissertation Outline

The rest of the dissertation is organized as follows. Chapter 2 contextualizes the need for dynamic adaptation of parallel execution as enabled by Varuna. Chapter 3 describes Varuna's fundamental working principles and system architecture. Chapter 4 describes how Varuna determines the optimum degree of parallelism for the two different performance objectives considered. Chapter 5 presents Varuna's adaptive execution mechanisms. Chapter 6 then presents the detailed evaluation and results for the three different environments on three disparate platforms, before Chapter 7 concludes.

Chapter 2

Background and Related Work

The key to efficient parallel execution lies in appropriately matching an application's parallelism to characteristics of its execution environment. An application's execution environment is defined by its resource demands and performance objectives, resources provided by the system to execute it, and the utilization of resources by all co-located applications in the system. Current parallel programming practices fail to fully consider the above factors and hence do not fully realize the potential of multicore systems in delivering performance for parallel applications. For example, today, the most common technique for realizing parallel execution is *multithreading*. Multithreading requires developers to *statically* specify the degree of parallelism to be employed for each parallel portion in the application either at development time or launch time. An uninformed developer may select too little or excessive parallelism and thus achieve suboptimal performance. One way to solve this problem is for the application developer to statically produce multiple versions of code with varying degrees of parallelism and dynamically select a version that best fits each execution environment. Unfortunately, the number of scenarios resulting from plethora of platforms and applications is so large that the possibility of incorporating all of them into statically-compiled codes is not feasible.

Recently proposed dynamic task-based programming models [49, 109, 7, 25, 101] free the developers from the burden of statically specifying the degree of parallelism in the application. They probe the platform the application is running on, determine the maximum number of processing cores available, automatically scale the degree of parallelism in the application to fully utilize all the processing cores, and dynamically assign and balance the user-defined tasks across all the processing cores to prevent resource underutilization. However, they cannot prevent resource oversubscription. Resource oversubscription may lead to dynamic serialization of what would otherwise be parallel parts of an application and can lead to negative performance consequences.

2.1 Some Factors Impacting Degree of Parallelism and Performance

In an arbitrary execution environment, contention and the consequent performance degradation can arise from a variety of different sources. Some of them are listed below:

- **Critical Sections.** Parallel applications synchronize accesses to critical sections via locks. Critical sections are regions of code in a parallel application that manipulate shared data. A lock is a programming construct that allows one hardware thread to take control over the critical section, while preventing others from entering it. Therefore, all executions of a lock protected critical section gets dynamically serialized. When a thread acquires a lock, the total time spent in acquiring the lock increases with the number of threads acquiring the same lock at the same instant in time. As the number of threads increase, the fraction of the time spent in the actual parallel parts of the program may reduce compared to the time spent in the critical section. Consequently, it is possible that the time spent in critical sections offsets the performance benefits obtained from adding more threads.

  Arguably, the above problem should be easy to address with known techniques [89]. However, if the application is to be portable, the developer may not always have complete control of the locking primitives and libraries in a particular execution environment. The problem is exacerbated if the locking happens in the Operating System kernel data structures. For example, a user-invoked *mmap* operation results in a global lock acquisition on page table in Linux kernels 2.6.32 or lower [28].

- **Cache coherence delays.** Cache coherence protocols can dynamically serialize parallel execution, when a processor reads/writes data that another processor has already written. The exact details depend on the mechanics of the coherence protocol. However, the high level operations are as follows. When a processor writes data that other have cached, the cache coherence protocol forces the write operation to wait while the protocol finds the cached copies and invalidates them. Similarly, when a processor reads data that another processor has just written, the cache coherence protocol does not return the data until it finds the cache that holds the modified data, annotates that cache to indicate there is a copy of the data, and fetches the data to the reading core. These operations are serial in nature and can cause the processor waiting on data to idle several hundreds of cycles, leading to dynamic serialization and disproportionate performance behavior.

- **Contention to shared microarchitectural resources.** The above two cases describes how data sharing in parallel programs can lead to dynamic serialization. But it is possible that an application's parallel execution can dynamically serialize even if its parallel computations do not share data. For example, computations may compete for shared hardware resources, such as, last level cache, memory bandwidth, I/O bandwidth, interconnection bandwidth, prefetching hardware, etc., and end up spending more of its time waiting for these resources rather than computing. These resources are not architecturally exposed to software (including the OS) and so it is difficult for the programmer to know their exact availability and capabilities. When the application's demands exceeds the capabilities of these resources, contention may result leading to serialization and unexpected performance behavior. Severe contention for a resource can have significant negative consequences, as this can result in increased latencies of other operations, perhaps even resulting in a situation where parallel execution results in a slowdown compared to sequential execution.

- **OS virtualization and scheduling.** In canonical parallelization methodology, the OS virtualizes processing resources using threads to provide programs with an autonomous and clean execution environment. An application can create and operate on its own set of OS threads, and does not need to know about other simultaneously running applications. The OS time-multiplexes the threads from different parallel programs onto the underlying processing resources. While such a methodology simplifies the process of programming, it may result in significant performance degradation when resources are oversubscribed due to the overheads associated with scheduling the execution of multiple simultaneously executing applications. For example, the more threads there are, the more time it takes the OS to cycle through all of them, causing each thread to run at more infrequent and unpredictable times. Computations that communicate often with one another may not be able to run simultaneously, causing serialization. Further, they can cause cache interference, TLB misses and page faults, all of which can significantly serialize parallel execution impacting the program's performance negatively.

- **Reliability and power management techniques.** As size of individual transistors shrink and as more of them are crammed in limited space, they begin to suffer from a variety of problems. First, they become more susceptible to transient, permanent, and intermittent hardware faults, which can impact a computation's reliability [26, 64, 27, 114, 35]. Second, power budgets and heat dissipation limitations can prevent activating all transistors at the same time. A number of techniques to tackle these problems have been proposed, such as dynamically adjusting voltage and/or frequency [30], employing Dual-Modular Redundancy (DMR) [54, 93, 78], or temporarily stopping the use of a processor [32, 56, 123]. Though these solutions are effective in many cases at mitigating transistor issues, they result in loss of resources which can create contention to available resources leading to dynamic serialization.

- **Performance asymmetric multicores.** Performance asymmetry in multicore architectures arises when individual cores have different performance. Building such multicore processors is desirable because many simple cores together provide high parallel performance while a few complex cores ensure high serial performance. However, current parallel systems typically assume computational cores provide equal performance, and such an assumption can create a mismatch between the demands of a computation and the capabilities of the core that it is mapped to. Consider a pipeline style program in which the loop iterations are split into pipe stages and are executed across processing cores while satisfying dependence relationships between them. The throughput of a pipeline depends on how soon the critical stages complete execution. Failing to do so can create subsequent pipe stages to stall, causing serialization.

- **False sharing.** False sharing occurs where threads use different data objects, but those objects happen to be close enough in memory that they fall on the same cache line, and the cache system treats them as a single lump that is effectively protected by a hardware write lock that only one thread can hold at a time. This causes real but invisible resource contention; whichever thread currently has exclusive ownership so that it can physically perform an update to the cache line will silently throttle other threads that are trying to use different, but nearby, data that sits on the same line. It is easy to see why the problem arises when multiple threads are writing to different parts of the same cache line, because only one can hold the exclusive hardware lock at a time. In practice, however, it can be even more common to encounter a reader thread using what it thinks is read-only data still getting throttled by a writer thread updating a different but nearby memory location, because the reading thread has to invalidate its copy of the cache line and wait until after the writer has finished to reload it.

- **Granularity.** Currently, it is the responsibility of the developer to identify computations to execute in parallel. The size of a computation (granularity) greatly affects its parallel performance. When decomposing an application for parallel execution, one approach is to logically partition the problem into as many parallel computations as possible. Within the parallel computations, next determine the necessary communication in terms of shared data and execution order. Since partitioning computations, assigning them to threads, and communicating (sharing) data between computations are all potentially serial operations, failing to mitigate them can negatively impact the application's performance, as the degree of parallelism increases.

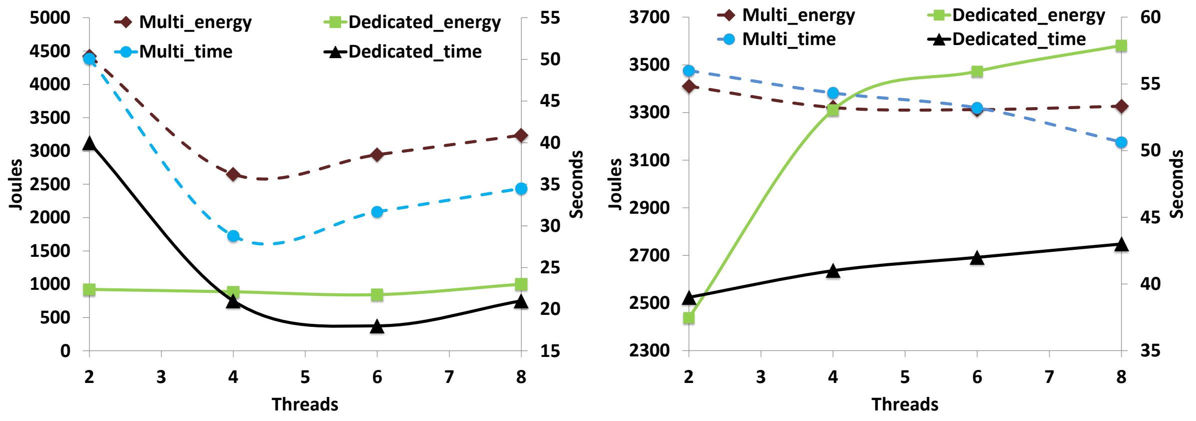
The above-mentioned issue have been known for a while, and the canonical approach to addressing them is to understand the execution of the application in a given environment and manually tune its parallelism such that its performance could be improved. Unfortunately, manual tuning is not a viable approach going forward since it is highly unlikely that developers will have detailed knowledge about the spectrum of parallel microarchitectures on which their applications might be running. Even more likely is knowledge about the characteristics of the execution environment, e.g., which other applications might be running, or how many processing and non-processing resources are likely to be available at any given instant in time, or whether the application has undergone a phase change or not.

Therefore, we argue that it is very important to dynamically control the degree of parallelism that is deployed in an application, even continuously changing it as the resource capabilities (or their usage) in the system changes.

2.2 Quantifying the Impact of Variations in Execution Environment

To put the prior discussion of performance variabilities arising out of variations in the execution environment in to perspective, consider the example shown in Figure 2.1. It plots the energy consumed by, and the execution time of, two of our benchmark applications, *Stream* and *RE*, in two different environments, dedicated, where a benchmark is the only application running, and multiprogrammed (multi), where other applications run simultaneously with the benchmarks, on a 4-core, 8-context Intel Core i7-2600 workstation.

RE, a networking redundancy elimination application [10], has abundant parallelism, and little contention for machine resources. As expected, with more number of parallel threads, both the execution time and the energy required to execute the application are reduced. However, when other extrinsic parallel operations exist, as is the case in the multiprogrammed environment, there is a significant degradation in energy and execution time beyond a thread count of four. This is because beyond four the total number of software threads in the example multiprogrammed environment exceed the total number of hardware thread contexts and threads from RE get descheduled by the OS scheduler, including at times while holding what is otherwise a very low-contention lock. This



(a) RE (b) Stream

Figure 2.1 Execution time and energy trends of two programs, RE and Stream, in dedicated and multiprogrammed (multi) environments on a Core i7-2600 workstation.

is a well know performance degradation scenario in the computer systems and can be viewed as "contention" for the limited number of hardware thread contexts.

In contrast to RE, Stream exhibits an opposite trend (Figure 2.1(b)). Stream, designed to stress the memory bandwidth, has ample parallelism and is indicative of numerical vector kernels whose datasets are larger than the cache capacity [88]. Even a modest attempt at parallel execution in a dedicated environment results in excessive parallelism, creating memory contention, and thus degrades execution time and energy. However, in the multiprogrammed environment, when it is contending with other programs for other resources (e.g., cache capacity or issue slots in the SMT processor), it naturally slows down, thereby slowing down the creation of parallelism and hence the memory contention. The naturally slowed rate of creation of parallelism allows more parallel execution to be beneficial rather than be a detriment.

While it is conceivable for expert programmers to analyse and determine the crossover point when one parallelism configuration becomes better than the other, and dynamically expose the application's parallelism to achieve optimum performance, expecting common programmers to do so can severely hamper their productivity. The objective of this dissertation is to design a parallel system that can automatically tune an application's parallelism without any programmer involvement.

2.3 Related Work

The problem of dynamically exposing an application's parallelism has been widely studied in the community, especially recently, as its importance has increased. This related work section is not intended to be a survey of this foundational research, rather it is intended to present an overview of similar efforts (with a focus on more recent work). To improve clarity, we have divided the related work into three subsections. First, we discuss research in Operating Systems that have enabled applications to create and enforce resource management policies. Second, we discuss architectural and microarchitectural techniques for detecting and managing contention for shared hardware resources as well as satisfying power/performance budgets. Third, we discuss techniques that share a similar motivation to this dissertation: dynamically exposing an application's parallelism on to the execution environment.

2.3.1 Operating System Techniques for Application-Level Resource Management

Determining how to allocate system resources in an environment with many different computation threads is one of the classic problems in operating systems research. With the mainstream adoption of multicores, this problem has received renewed interest in the field with the added complications of asymmetric cores and reducing power consumption as an optimization criteria [75, 36]. Most previous work from the scheduling perspective looked at doing the best job of ordering the threads using various types of performance prediction. That is, these techniques typically work with the threads provided and do not consider the possibility of changing the threads themselves. Nevertheless, the rest of the subsection discusses some of the important proposals in this space.

Traditional operating systems place too much restrictions on applications. Information of resources are hidden behind high level abstractions where applications have little or no access to. This structure aggressively denies applications from expanding or modifying the Operating System. Additionally, Operating Systems are usually large and complex which decreases system reliability and make them very hard to maintain. The design of Exokernel [43], microkernels [5], extensible operating systems [19], introspective systems, such as Infokernel [12], and split-level schedulers, such as like Scheduler Activations [11], target these problems presented by a traditional operating system. Their goal is to give applications efficient control over the management of hardware and software resources. For example, exokernel places abstractions traditionally implemented by operating systems, such as virtual memory and inter-process communication, in the user space where user-level libraries abstract exposed resources. This property gives applications the flexibility to tailor an operating system that best suits their needs. Similarly, Scheduler Activations create a feedback loop between the in-kernel CPU scheduler and the user-space thread scheduler to allow the user-space scheduler to efficiently manage CPU resources during periods of low and high activity in the system. A classic example is a blocking I/O call or a synchronization event. Infokernel is similar to scheduler activations except that it enhances commodity schedulers rather than replacing them. This technique leverages the time and money invested in commodity schedulers, and may help make them more robust by facilitating in-depth user feedback.

An alternative approach to direct CPU scheduler feedback is to construct implicit feedback from low-level instrumentation of individual computations. Barham, et. al. [16] and Stewart, et. al. [115] take this approach to detect or predict resource contention. In this approach, a variety of performance information is collected online and complex models are calibrated or constructed offline. This approach is broader than the above-mentioned approaches in that it can detect contention for multiple resources, rather than just CPU resources. However, it requires a learning phase or recalibration for every new application, range of inputs, and hardware configuration.

Statically partitioning CPU resources through virtualization [17, 31] or hierarchical CPU partitioning [121, 55] can be used to ensure a fixed CPU allocation for each individual application. These techniques ensure that there is CPU isolation between competing applications. However, they do not prevent CPU contention between concurrent tasks in the same application; without scheduler feedback, it can be difficult for an application to determine the correct level of parallelism or the severity of CPU slowdown it is suffering.

2.3.2 Hardware Techniques for Contention Management

Multicore systems commonly share a large portion of the memory subsystem between different cores. Main memory and shared caches are two examples of shared resources. Memory requests from different applications executing on different cores can interfere with and delay each other in the shared memory subsystem.

To mitigate this problem, several proposals, such as [65, 66, 71, 94, 95, 99, 42, 61], detect contention in shared memory subsystem and use this information to dynamically adapt the rate at which the different cores inject requests into the shared memory subsystem such that system-level fairness objectives are met. Bitirgen, et. al. [22] propose implementing an artificial neural network that learns each application's performance response to different resource allocations. This technique searches the space of different resource allocations among co-executing applications to find a partitioning in the shared cache and memory controller that improves performance. Herdrich, et. al. [59] reduce the interference caused by a lower-priority application on a higher-priority application by modulating the clock signals in modern multicore processors. Zhang, et. al. [128] propose a technique that uses clock modulation and prefetcher on/off control provided by existing hardware platforms to improve overall system performance in current multi-core systems. Jahre, et. al. [67] dynamically adjust the number of available MSHRs to control the total miss bandwidth available to each thread running on a multicore processor.

All the above mentioned works alleviate the negative impact of a type of contention to a given resource by changing the impact of the contention, while keeping the degree of parallelism the same. The work proposed in this dissertation alleviates contention by dynamically altering the degree of parallelism exposed in the application.

Li and Martinez [83, 82] propose runtime heuristics for parallelism and DVFS control to find power/performance efficient execution points for specific soft performance/power targets. The primary difference between these papers and the work proposed in this dissertation is that the latter can arrive at the optimum operating point without having to extensively search the parallelism configuration space, whereas these papers do.

2.3.3 Application-level Parallelism Management

Several recent papers propose to dynamically vary the degree of parallel execution of an application [37, 38, 81, 80, 116, 106, 69]. This subsection describes these proposals in detail.

- **Feedback Driven Threading (FDT) [116].** In FDT, the authors used simulations to demonstrate that many applications can improve performance and/or power consumption by taking memory bandwidth and synchronization into account. They propose to monitor the execution of the application and use models to predict the appropriate number of threads for the application to employ, given the system state. FDT's approach requires a static compiler to augment parallelizable loops such that a profiling code can later adjust how the threads are divided by dynamically setting a small number of variables in the source code. FDT has three disadvantages as compared to work in this dissertation. First, it is only effective on data-parallel loops, as it does not take inter-iteration dependencies into consideration. Second, it cannot tackle contention to resources other than memory bandwidth and lock synchronization. Third, it cannot perform continuous adaptation, making it ineffective in multiprogrammed environments. It probes into the environment once either at the beginning of the program or at the inception of every phase in the program, identifies the optimum number of threads and fixes that value for the rest of the program or phasal execution. The techniques proposed in this dissertation does not suffer from these limitations. Additionally, FDT requires support from a static compiler. However, the system proposed in this dissertation requires no external support and works on legacy applications, provided it is cognizant of the semantics of the programming model used in the application.

- **Curtis-Maury et al. [37, 38].** These papers propose a very fast and effective online power/performance statistical prediction models to determine optimum parallelism, in which performance, power and combined metrics are predicted with as few as two snapshots of hardware event counters, on a phase by phase basis. As compared to the work presented in this dissertation, these papers have three disadvantages. First, their models employ offline regression analysis and learning that is trained from samples of the power-performance adaptation search space collected from standard workloads. It is not clear how it will work if a new workload is encountered. Second, the models are only effective when their applications execute in isolation and cannot handle multiprogrammed environments. Third, as with FDT, their approach is applicable only to data-parallel programs.

- **Thread Tailor (TT) [80].** TT is a dynamic compilation system that can automatically stitch threads together based on the architecture, dynamic system state, and communication and synchronization relationships between threads. It employs a novel partitioning algorithm and code generation methodology that quickly and effectively selects the appropriate number of threads an application should be using and generates new threads without unnecessary synchronization or communication overheads. Unlike FDT [116] and Curtis-Maury, et al. [37, 38], TT does not require any support from the compiler toolchain and is effective on legacy applications. Further, it can handle applications with arbitrary communication patterns and not just data-parallel applications. However, as compared to this dissertation, TT cannot handle multiprogrammed environments. Further, TT is not model based. It employs a search heuristic which can be inefficient as the number of processors employed to execute the application increases.

- **Degree of Parallelism Executive (DoPE) [106].** DoPE separates the concern of developing parallel applications from that of optimizing them. Using DoPE, the application developer can specify all of the potential parallelism in loop nests just once; the mechanism developer can implement mechanisms for parallelism adaptation; and the administrator can select a suitable mechanism that implements a performance goal of system use. Unlike the previous proposals, DoPE can handle multiple performance objectives, such as throughput, power and energy, nested parallelism and multiple parallelism types, such as do-all, do-any and pipeline-style parallelism. However, it has the following disadvantages. First, similar to the previous proposals it does not perform continuous adaptation. Second, it is not clear whether it can handle multiprogrammed environments. Third, it requires intervention from an *administrator* to statically choose the right set of mechanisms for the given environment. Fourth, it requires application to be rewritten from scratch using its own set of APIs.

- **Parcae [107].** Parcae is a compiler and runtime system that automatically generates flexible parallel applications and that monitors and optimizes the execution of multiple flexible applications running on a shared parallel platform. Of all the adaptive approaches mentioned so far, Parcae is the most general. It can handle arbitrary parallel applications and can handle multiprogrammed environments. However, it has the following disadvantages. First, it optimizes for only one metric, execution time. Second, it employs a finite difference gradient ascent control approach, a variant of a hill climbing search heuristic, to adapt to search for an optimal degree of parallelism, which can be very slow to react when the operating conditions change rapidly and can potentially get stuck in a local optimum.

- **Lithe [101].** Lithe describes how to mix multiple heterogeneous parallel runtimes (e.g., TBB + OpenMP) in a single application to run efficiently on a given set of processing cores. While Lithe can handle the problem of resource oversubscription between parallel libraries, each parallel library can potentially incur degradation due to contention to shared resources in the system. The work in this dissertation is orthogonal to the philosophy of Lithe and can be easily deployed with Lithe to avoid shared resource contention.

- **Autonomous computing.** The system proposed in this dissertation may also be viewed as an example of autonomic computing, in which systems self-monitor and self-regulate to achieve desired objectives, without user-intervention [70]. Its periodic monitoring and control of execution to improve efficiency is analogous to the MAPE cycle of autonomic systems and observer-controller paradigm of organic computing [110]. Invasive computing is another proposal whose one goal is to optimize parallel execution of applications [117]. It requires pervasive changes to all aspects of a system, whereas the proposal in this dissertation works with existing hardware and prevalent programming methods.

- **Auto-tuners.** Auto-tuners have been previously investigated in the area of numerical software and high-performance computing, primarily for single threaded applications [21, 86, 46, 47, 62, 124]. They optimize a set of library kernels by generating many variants of a given kernel and benchmarking each variant by running on the target platform. Recently, auto-tuners to efficiently map the parallelism in an already parallelized parallel application to a multicore processor has gained a lot of attention. Wang, et. al., [122] use machine learning techniques to determine the optimum degree of parallelism to employ on a given hardware platform. Ko, et. al., [72] describe a system that uses a simple heuristic to find the right degree of parallelism for each level in a cluster environment. The ADAPT dynamic optimizer applies loop optimizations at run-time to create new variants of code [120]. Compared to the work in this dissertation, these proposals have two disadvantages. First, they require dynamic compilation support which can destructively interfere with the execution of the application. Second, it is not clear how these approaches can handle multiprogrammed environments.

Chapter 3

Working Principles and System Architecture of Varuna

Chapter 1 introduced the high level motivation of this dissertation and presented the basic idea of the proposed system, Varuna, that transparently optimizes an application's parallel efficiency by dynamically and continuously adapting its parallelism to match the characteristics of the execution environment. Chapter 2 discussed the difficulties associated with traditional parallel programming methodology in achieving predictable parallel execution efficiency in dynamically changing operating conditions and argued that the unpredictability is a direct consequence of statically exposing an application's parallelism. This chapter presents an overview of the working principles and system architecture of Varuna.

3.1 Working Principles

Before we enlist the fundamental working principles of Varuna, it is critical for us to first understand impact of parallel execution on an application's efficiency.

If an application's workload can be perfectly divided into equal sized parallel computations that do not interact, and can be executed on a system with unlimited resource capabilities that do not change, one may expect linear speedups as more processors are employed to execute the application. In practice, however, an application's parallel region is not perfectly parallelizable, computations often interact with each other, and resource capabilities in the system are unknown, limited and can dynamically change. The confluence of these factors typically leads to two types of **side effects** which can impact the application's efficiency unintuitively.

First, they can dynamically increase the latencies of the application's parallel computations, causing **dynamic serialization**. Dynamic serialization can lead to slowdowns, sometimes worse than sequential execution, and can arise from a plethora of sources as described in Chapter 2, including contention to software resources (e.g., locks) and shared hardware resources (e.g., last level cache, memory and disk bandwidth, SMT core, etc.), memory effects (e.g., false sharing and processor affinity), cache coherence delays, cache interference due to multiprogramming, TLB misses and page faults, loss of resources due to power and reliability management techniques, load balancing and scheduling overheads, among others.

Second, they can dynamically decrease the latencies of the application's parallel computations, causing **dynamic acceleration**. Dynamic acceleration can lead to superlinear speedups and can arise primarily due to caching effects resulting from different memory hierarchies on modern processors. For example, as an application uses more processors, the total cache available to it also increases. With more cache, the processors can collectively accommodate more instructions and data and reduce the memory access time, causing a computation to finish faster than expected.

In general, one may expect some combination of the two effects, often depending on the application characteristics and the amount of resources allotted to execute the application. To achieve efficient parallel execution, it is critical for programmers to pick the right degree of parallelism that takes advantage of the dynamic acceleration and mitigates dynamic serialization for a given performance objective. However, statically determining this point is hard because these effects can occur at different times and due to different reasons. For example, they can take hold for the entire application, or only during parts of it when the applications change phases, or when co-located applications occupy or release resources, or when the resources go offline or come online. Further, the combination of these effects can be different for different microarchitectures. Within a microarchitecture, the combination can vary dynamically and differently across runs, impacting the application's efficiency unintuitively, especially in multiprogrammed environments. Therefore, to achieve efficient parallel execution, a dynamic and responsive parallelism optimization strategy is needed. As depicted in Figure 3.1, implementing such a strategy requires:

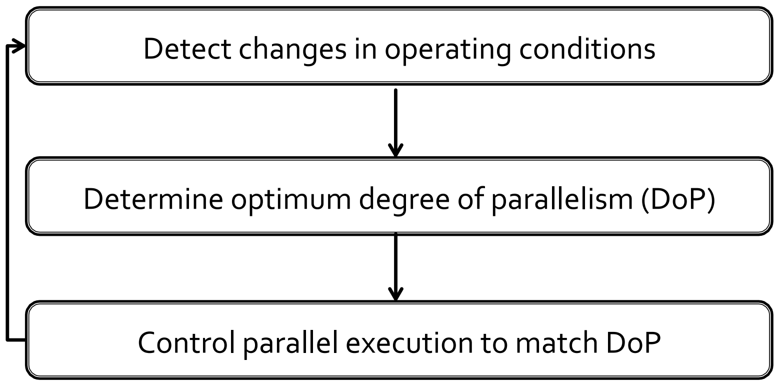

Detect changes in operating conditions
Determine optimum degree of parallelism (DoP)
Control parallel execution to match DoP

Figure 3.1 Working Principles of Varuna.

1. Detecting changes in the system's operating conditions and determining if the change necessitates retuning the application's parallelism,

2. Determining an optimum degree of parallelism (DoP) such that it mitigates dynamic serialization and takes advantage of dynamic acceleration for a given performance objective, and

3. Controlling the application's parallel execution, by suspending, resuming and migrating its parallel computations, to match the determined optimum DoP.

For overall efficiency, the above process needs to be *repeated periodically*, as the application executes, to assess and react to changes. Expecting common programmers to deploy such a complex strategy can severely hamper their productivity.

Canonically, it has been the responsibility of the Operating System to control how many processors to assign to an application at a given instant in time. A natural path, therefore, is to expect the Operating System to be responsible for implementing the above principles. However, there are multiple reasons why this may not be a preferred approach going forward. First, Operating Systems are very slow to change, and it is not clear that novel dynamic parallelism adaptation techniques will be incorporated into them in a timely manner. Second, an application may run on a diverse set of Operating Systems, some of which may or may not have this capability. Third, this requires a way of communicating between an application and an Operating System so that the latter can understand the former's needs, something that is going to be very challenging when there are millions of different ubiquitous applications, and many different Operating Systems, co-existing in the overall computing ecosystem. Further, as more applications move to server clouds, it is not clear how the latter communicates with the former and vice-versa, with several software layers of complexity between them. In short, we do not believe that creators of ubiquitous parallel applications will want an approach that requires intervention by other entities.

Varuna implements the above strategy as a runtime system interposed between the application and the Operating System, with no modifications to either. The rest of the section describes how Varuna realizes each of the capabilities depicted in Figure 3.1.

3.1.1 Detect Changes in Execution Environment

The most basic requirement for Varuna is the ability to detect changes in the execution environment. An application's execution environment can change due to:

- *System events*, such as launches of new applications and termination of old concurrently running applications, context switches, thread migrations, etc.
- *Application events*, such as variations in size or quality of the workload, and
- *Hardware events*, such as Dynamic Voltage and Frequency Scaling (DVFS), component failures, and shutting down cores and parts of cache and memory.

One approach to detect changes in the execution environment is to continuously monitor these events themselves. However, monitoring these events is not practical as several of them are not directly exposed to software, including the Operating System. Exposing them to applications requires changes in both the operating system and the hardware interfaces, and support from programmers—something that is going to be very challenging when there are plethora of different programming models, operating systems, and hardware microarchitectures co-existing in the computing ecosystem.

Several recent proposals [85, 107, 77, 104, 105] have observed that any change in the execution environment ultimately manifests as a change in the processor, memory, I/O or energy usage characteristics of the application and have leveraged this observation to detect changes in the execution environment in a low overhead fashion, without needing to alter the existing hardware or software interfaces. Specifically, these proposals use hardware counters to periodically monitor a gamut of metrics, such as, *Instruction fetch rate*, *last level cache miss rate*, *CPU utilization*, *memory bandwidth utilization*, *instructions retired per second*, *number of page faults*, *processor and system energy consumption*, etc., for changes and reconfigure parallelism when a change in any of these metrics is detected. Most modern processors, whether embedded, desktop or server-class, contain performance and energy monitoring units which are capable of counting and supplying the instantaneous values of the these metrics either directly or indirectly [18, 1]. This dissertation employs a similar approach.

3.1.2 Determine Optimum Degree of Parallelism

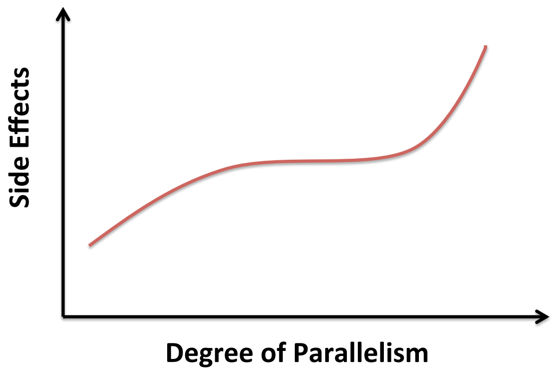
The most important requirement for Varuna to dynamically expose an application's parallelism is the ability to determine the optimum degree of parallelism that an application must employ such that it mitigates dynamic serialization and takes advantage of dynamic acceleration for a given performance objective.

If the parallel region is large enough that an exhaustive search is feasible, an easy way to determine the optimum degree of parallelism is to try all possible parallelism configurations: starting with sequential execution, incrementally increase the application's degree of parallelism, while recording the efficiency of each parallelism point, until the maximum number of hardware contexts is reached and pick the one that yields maximum efficiency. The cost of an exhaustive search is $N * t$, with N being the maximum number of hardware contexts and t being the average time spent in each parallelism configuration. Often times, however, an exhaustive search may not be possible, or its overhead prove prohibitive, because the parallel region may not be large enough to amortize the cost of search. Even if the parallel region is sufficiently large, it is possible that the execution environment may change during the search, rendering the whole search process ineffective. Therefore, it is important to reduce the amount of time spent in searching and converge toward the global optimum much faster.

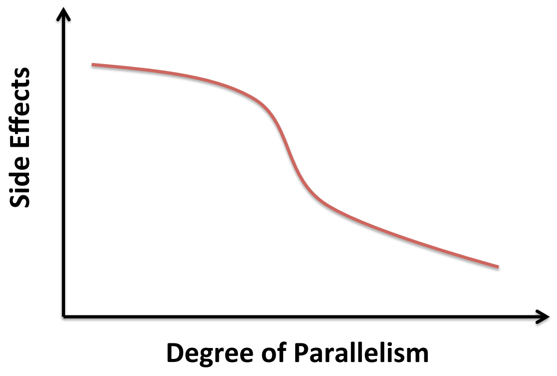
Ideally, one would like to have an approach that can instantaneously supply the optimum degree of parallelism without resorting to any search techniques. However, such an approach requires an oracle view of how dynamic serialization and dynamic acceleration would vary with parallelism. This dissertation argues that by modeling the relationship among dynamic serialization, dynamic acceleration, the application's instantaneous performance and the application's instantaneous degree of parallelism, it is possible to gain valuable insights into how dynamic serialization and dynamic acceleration vary with parallelism, and use these insights to *rapidly* obtain the optimum degree of parallelism without performing an exhaustive search.

The question then is, how do we model such a relationship ? While there could be several ways of modeling it, this dissertation models it as a simple **scalability model** based on Amdahl's law. It first uses the scalability model to rapidly estimate how dynamic serialization and acceleration vary with respect to parallelism. It then uses formulae, derived from the model, to *instantaneously* determine the optimum degree of parallelism to employ for different performance objectives. Specifically, using the model, this dissertation determines the optimum degree of parallelism for two different performance objectives: (i) Maximize application throughput, and (ii) Minimize resource consumption cost, i.e., the CPU consumption-execution time product. These are two popular pricing models employed in cloud-based services. The former, *time-based* pricing, used in Amazon's EC2 and Microsoft's Azure, gives a program a fixed number of cores and charges for how long they are used; thus minimizing execution time is important. In the latter, *consumption-based pricing*, used by VMware, the pricing depends on the average number of cores and the duration of their use. We believe other performance objectives, such as power, Perf/J, and Perf/W, can be similarly targetted. However, we leave them future work. Chapter 4 presents the model as well as the heuristic that leverages the model to determine the optimum degree of parallelism in detail.

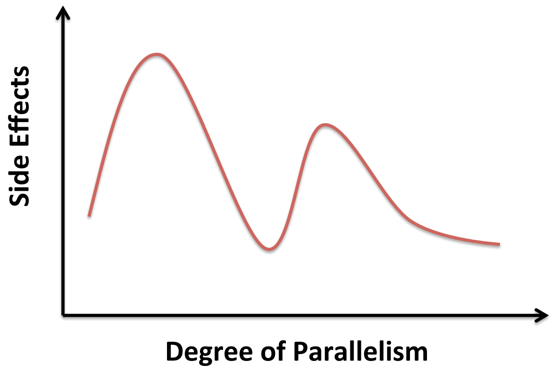
Although the proposed model can rapidly determine the optimum degree of parallelism, it is possible that the model may arrive at a non-optimum solution. This is because the model assumes that both dynamic serialization and dynamic acceleration are **monotonic functions** with respect to parallelism, as shown in Figures 3.2(a) and (b). As we will show in Chapter 6, for a variety of different execution environments and machines, the monotonic assumption is sufficient for the



(a) Monotonically Increasing



(b) Monotonically decreasing



(c) Non-monotonic

Figure 3.2 Assumptions of the scalability model.

model to rapidly and accurately estimate the optimum degree of parallelism. However, there may be scenarios in which these effects could possibly be non-monotonic, as shown in Figure 3.2(c), especially on multi-socket machines, in which case the model may identify a solution that is different from the optimum one. To tackle such scenarios, in addition to the model, this dissertation proposes a **hill climbing based search heuristic**.

Hill climbing can roughly be summarized as an iterative search technique that, starting from an initial feasible solution, progressively improves it by applying a series of local modifications or moves to the solution. At each iteration, the search moves to an improving feasible solution that differs only slightly from the current one. The search terminates when no more improvement is possible. The proposed heuristic employs a similar approach. It systematically varies the degree the parallelism in the application in the search space to locate the one that minimizes dynamic serialization and maximizes dynamic acceleration for the desired performance objective.

One of the fundamental limitations of a hill climbing search heuristic is that it can get “stuck” in a locally optimum solution and fail to reach a globally optimum solution. Clearly, this is an important limitation of the method: unless one is extremely lucky, this local optimum will often be a fairly mediocre solution. To overcome this problem, this dissertation proposes to incorporate *Tabu search* [52, 50] in the above heuristic. Tabu search is a metaheuristic search method employed for mathematical optimization problems to escape the trap of local optimality. The basic idea behind Tabu search is to add a simple notion of “memory” to the hill climbing algorithm that records the recent set of points previously visited in the search and use this memory to pursue the search in the unexplored areas of the search space at periodic time intervals, whenever a local optimum is encountered. Chapter 4 describes this heuristic in detail.

3.1.3 Control Parallel Execution

Another important requirement for Varuna is to dynamically control the execution of parallel computations in the application such that it best matches the instantaneous degree of parallelism determined by the scalability model mentioned in Section 3.1.2. Doing so requires a flexible parallel programming model that possesses the following four capabilities:

- A parallel programming abstraction that enables automatic and dynamic assignment/reassignment of computations in the application without the involvement of the programmer or user or any other entity.

- A mechanism to transparently pause or suspend appropriate computations from being performed when decreasing the parallelism to avoid oversubscription, and migrate or resume computations when increasing the parallelism to avoid underutilization.

- A mechanism to dynamically (re-)partition the remaining parallel work in the application, if any, to avoid load imbalance and/or resource idling,

- A mechanism to honor dependences between computations in the application, if any, to ensure that the application's forward progress is not affected due to deadlocks, livelocks or starvation issues.

A *task-based parallel programming model* possesses the necessary foundations to realize the above four capabilities. Tasks may be viewed as computations that compose a thread in a conventional multithreaded application— the overhead of spawning a task is typically a few times the overhead of an ordinary function call [49] and far less compared to the overhead of spawning a thread. A task programming model separates the concern of expressing parallelism in the application from the concern of managing parallelism, thus relieving programmers of the responsibility of assigning tasks to hardware contexts. Tasks are decoupled from the hardware contexts (threads) and could potentially to extended to have their own notion of context to support seamless suspension, resumption and migration of computations without any Operating System or hardware involvement. The low cost and finer granularity of tasks permits application programmers to create as many tasks as they want and dynamically partition them across available hardware contexts to avoid load imbalance and resource idling. These aspects are also exploited by modern task-based runtime systems [49, 109, 23, 24, 96].

If applications are composed using task programming primitives, we can leverage the efficient dynamic scheduling already provided by the existing task parallel runtime systems to transparently control an application's execution. However, this presents a significant challenge, because these dynamic scheduling algorithms assume tasks are independent, and requires programmers to appropriately enforce dependences between computations, if any. This assumption is not sufficient to permit the necessary control to prevent resource overutilization as well as to ensure the application's forward progress. This is because controlling parallelism requires the ability to arbitrarily block computations from being performed when decreasing parallelism, and resume them when increasing parallelism. However, this can lead to unintended consequences in applications that enforce a specific order between computations.

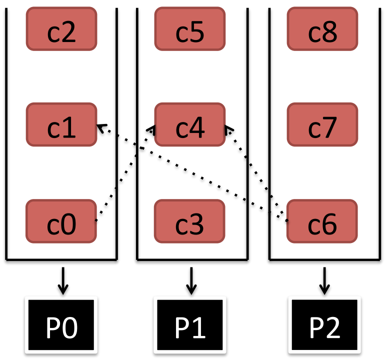


Figure 3.3 Independence-based parallel execution example.

Figure 3.3 shows the the example illustration of the problem. It shows an example dynamic schedule of tasks $c0$ to $c8$ on processing resources $P0$ to $P2$, along with their dependence relationships. For example, $c1$ and $c4$ are dependent on $c6$ (represented by edges). Consider the situation where the application's degree of parallelism is reconfigured from three to two, in which case, one of the executing tasks must be suspended. Wrongly suspending a task, say $c6$, can the hinder the application's forward progress and can even result in a deadlock as the dependent tasks ($c1$ and $c4$) can end up holding their respective processing resources ($P0$ and $P1$), disallowing other tasks to proceed.

There are two ways of tackling this problem in a dependence unaware dynamic scheduling system. First, the developer can insert (partial) barriers before each dependent task in the application code so that the underlying scheduler waits for all the currently scheduled (producing) tasks to complete before scheduling the dependent ones. But doing so can inhibit the utilization of distant parallelism in the application [44]. Second, the developer can alter the lexical ordering of computations in the program by inserting enough other independent tasks between the producing and dependent tasks. This approach has three problems:

1. Two tasks that are statically dependent may not be dynamically dependent [7]. That is, by the time a dependent task is scheduled, the producing task could have already completed. Hence, separating them lexically with static dependence information can have performance implications [44].

2. If there are not enough independent tasks to insert between the producing and dependent tasks, the programmer will have to resort to the first approach of inserting barriers.

3. Dynamic scheduling runtime systems do not reveal any internal data structure details (such as size of the work queues) to programmers and hence it is not straightforward to identify the number of independent tasks that are to be inserted between the producing and dependent tasks.

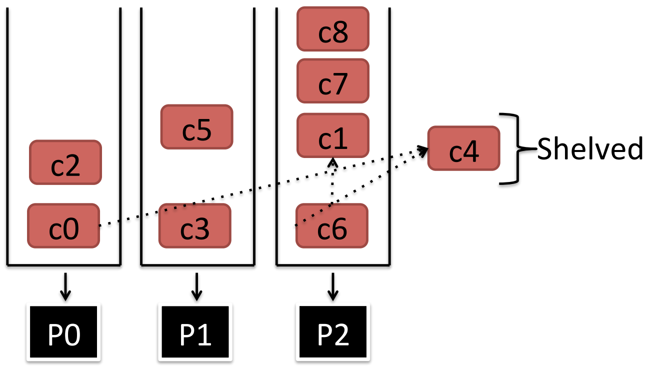


Figure 3.4 Dependence-aware parallel execution example.

Knowledge of dependences between tasks can enable effective scheduling decisions without programmer involvement that can completely avoid the above-mentioned problems. For example, computations $c6$ and $c1$ can be scheduled together on to $P2$ or $P0$. Similarly, computation $c4$ can be shelved until its input producing tasks $c0$ and $c6$ complete execution, and rescheduled for execution once it receives its input data. Figure 3.4 depicts one possible dynamic schedule that is dependence aware. Having such a schedule will enable Varuna to randomly pick a victim task without any complicated dependence tracking algorithms (as there are no cross-dependences), take advantage of the existing dynamic load balancing algorithms (which require that all tasks be independent [49]), and always make forward progress. Chapter 5 discusses these aspects in detail.

Recently, there have been several dependence-aware task-based programming systems proposed, *albeit* for a different purpose, to execute parallel applications. SMPSs [102] and Gupta et al. [57] leverage this philosophy to discover distant parallelism. Prometheus [8] and DPJ [25] execute only independent tasks concurrently to achieve determinacy. One could envision extending one of the above systems for progress aware parallelism adaptation. However, these systems are not widely adopted. Further, they do not embody a wide variety of computational patterns. For example, they do not support divide-and-conquer, branch-and-bound, recursive data and parallel-search idioms.

While it is possible to extend these systems and compose future parallel applications using them, standard dependence unaware parallel programming models, such as Pthreads [2], TBB [109], and Cilk [49], will continue to be important. Many incumbent parallel applications are written using these models, and they will need to run efficiently in evolving dynamic computing environments. Moreover, many developers would prefer to program in a familiar model rather than start afresh with a new one. Thus, a challenge is to regulate the execution of a parallel application written using a standard dependence unaware programming models without changing or breaking the application source code.

To address this challenge, this dissertation proposes a novel primitive called *virtual task* (in short, *vtask*). Vtasks abstract hardware contexts into logical cooperative tasks [6, 4] to which application's parallel computations are transparently mapped. The vtasks, in turn, are dynamically scheduled onto hardware contexts, similar to tasks in a dynamic task-based runtime system. Each vtask maintains the state of the current computation mapped on to it using *contexts*, allowing Varuna to transparently pause, resume, or migrate a computation by saving or restoring its corresponding vtask's context. It also includes the state necessary for Varuna to ensure the computations' forward progress even as their execution is regulated.

This dissertation demonstrates vtasks' versatility using three different parallel programming models: Pthreads [2], Intel Thread Building Blocks (TBB) [109], and Prometheus [9]. The first two are widely adopted industry standard programming models, whereas the third is a research prototype developed at Wisconsin's Multiscalar group. These models capture a wide variety of computational patterns, level of abstraction, and parallel execution models that programmers of today employ. For example, (i) Pthreads exposes *threads* as programming abstraction, whereas

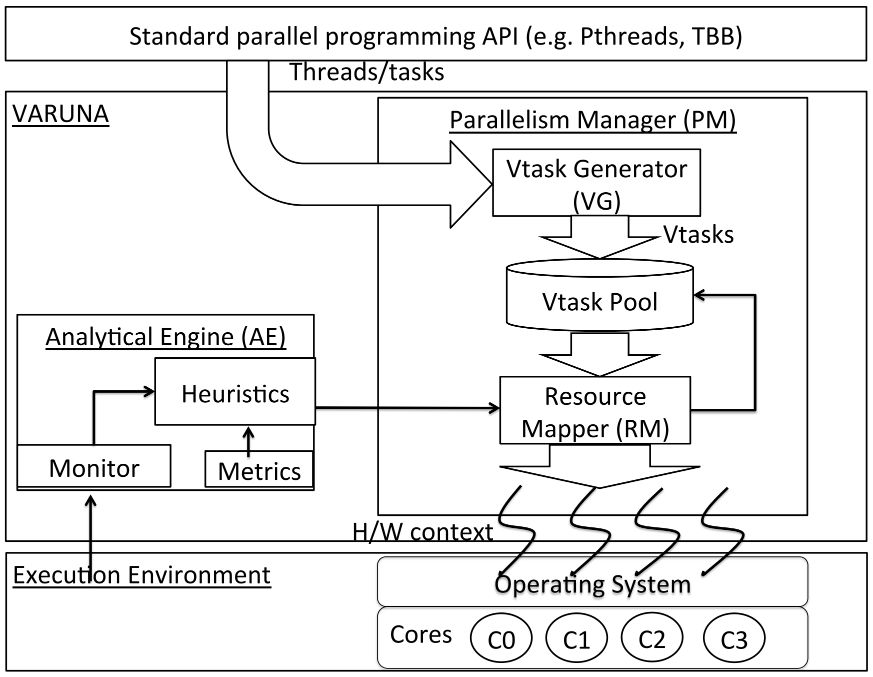


Figure 3.5 Varuna's system architecture.

TBB presents *tasks*, (ii) Pthreads and TBB are independence-based execution models, whereas Prometheus is a dataflow-based execution model [113], and (ii) Pthreads supports arbitrary communication patterns, whereas both TBB and Prometheus are restricted to fork-join parallelism. More details on vtasks are given in Chapter 5.

3.2 System Architecture

As mentioned earlier, the goal of Varuna is to take an unmodified parallel application, running on a stock OS, and dynamically and continuously adapt its parallel execution so as to avoid underutilization or oversubscription of resources while satisfying a specified performance objective. It comprises two components, an **Analytical Engine**, and a **Parallelism Manager**, as shown in Figure 3.5. It assumes an underlying system that provides a pool of worker threads, common in a modern Operating Systems, and a low-overhead means to measure an execution environment's characteristics, e.g., via hardware performance counters provided by modern processors. The two components and their operations are summarized next.

3.2.1 Analytical Engine

The Analytical Engine (AE) continuously monitors changes in the operating conditions using hardware performance monitoring units, models the application's dynamic execution behavior to estimate impact of the side effects, and determines the optimum degree of parallelism. The high-level operations of the AE, as depicted in Figure 3.6, are as follows:

1. Establish the relationship between the application's instantaneous degree of parallelism, instantaneous performance and the side effects using a scalability model.
2. Using this information, determine the optimum degree of parallelism, P_{opt}, for a given performance objective.
3. Passively monitor the execution environment for changes, as the parallelism manager employs P_{opt} parallelism for the application.
4. Go to step 1 if the operating conditions change.
5. If the operating conditions have not changed for a threshold period of time, go to step 6.
6. Determine if the side effects monotonically vary with degree of parallelism.
7. If monotonic, go to step 3.
8. If non-monotonic, invoke the Tabu-based iterative search.

The details of the scalability model, the Tabu-based iterative search, how the AE leverages the model to determine the optimum degree of parallelism as well as to detect if the side effects monotonically vary with parallelism are described in Chapter 4.

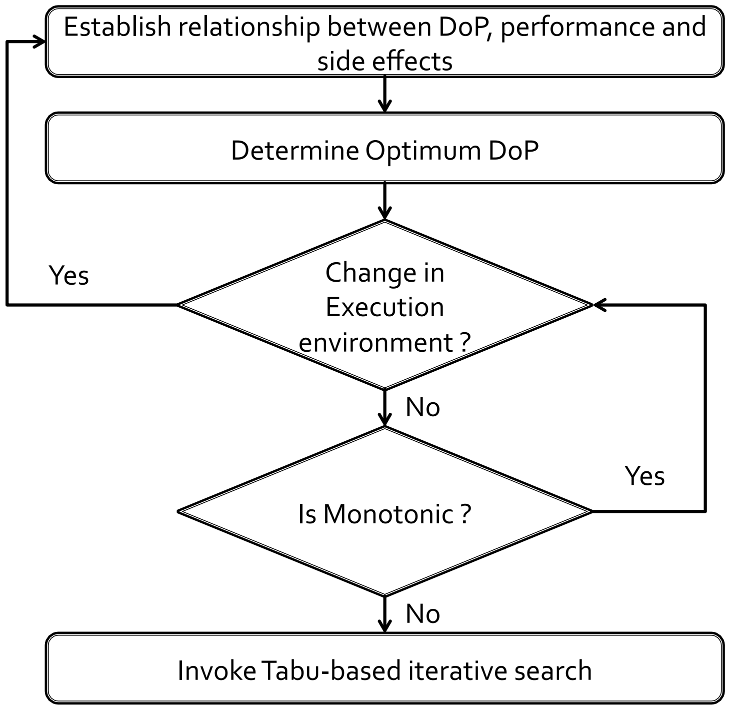

Establish relationship between DoP, performance and side effects
Determine Optimum DoP
Change in Execution environment ?
Yes
No
Is Monotonic ?
Yes
No
Invoke Tabu-based iterative search

Figure 3.6 High level operations of the Analytical Engine.

3.2.2 Parallelism Manager

The role of the Parallelism Manager (PM) is to automatically control the execution of parallel computations in the application in order to match the degree of parallelism determined by the AE. The high level operations of the PM are as follows:

The programmer first decomposes the application into smaller, independent units of work using a supported parallel programming model. As the application begins to execute and starts creating threads or spawning tasks, a Vtask Generator (VG) transparently intercepts these requests, nullifies them and creates vtasks instead. The VG then reassigns the thread's/task's parameters (pointer to actual computation and its arguments) onto vtasks and enqueues them into a priority queue, called vtask pool. A Resource Mapper (RM) then assigns the vtasks from the vtask pool, to a dynamically varying pool of worker threads, whose number is determined by the Analytical Engine. The number of workers executing vtasks directly corresponds to the maximum degree of parallelism exposed by the application. The RM is also responsible for dynamically controlling the execution of vtasks (suspending, resuming and migrating) as well as ensuring the application's forward progress. These details are described in Chapter 5.

Chapter 4

Analytical Engine

The role of the Analytical Engine (AE) is to continuously monitor changes in the operating conditions using hardware performance monitoring units, model the application's dynamic execution behavior to estimate impact of the side effects, and determine the optimum degree of parallelism. Chapter 3 presented the high level operations of the AE. This chapter presents the details of those operations.

The chapter begins with Amdahl's law in Section 4.1. Next, it develops the proposed scalability model based on Amdahl's law in Section 4.2 and demonstrates how the model provides insights into an application's parallel execution behavior. Next, it derives formulae to determine optimum degree of parallelism for two different performance objectives: (i) maximize application throughput, and (ii) minimize resource consumption cost, i.e., the CPU consumption-execution time product, in Section 4.3 and Section 4.4, using the insights in Section 4.2. Then, the chapter presents how the different quantities in the derived formulae can be empirically determined in Section 4.5. Next, in Section 4.6, it describes how the AE monitors the execution environment for changes and recalibrates the optimum degree of parallelism. Then, the chapter describes the limitations of the proposed scalability model in Section 4.7. To tackle this limitation, the chapter presents an alternate heuristic that incorporates *Tabu search* [52], developed using the guidelines due to Gendreau [50] in Section 4.8. The chapter concludes with a summary in Section 4.9.

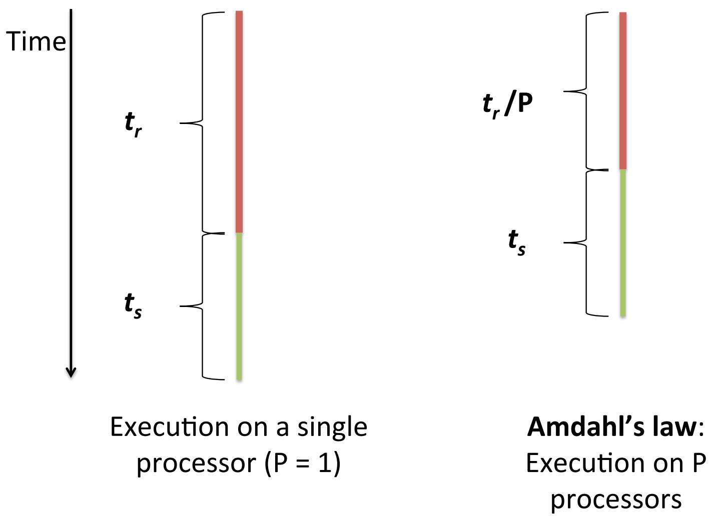

Time
t_r
t_s
t_r/P
t_s
Execution on a single processor (P = 1)
Amdahl's law: Execution on P processors

Figure 4.1 Amdahl's law example.

4.1 Amdahl's Law

As mentioned in Chapter 3, the AE models the relationship among dynamic serialization, dynamic acceleration, an application's instantaneous efficiency and an application's instantaneous degree of parallelism based on Amdahl's law. To better understand the model, it is first critical to understand Amdahl's law.

In 1967, Gene Amdahl argued that there was an inherent limitation to the amount of speedup that could be obtained by increasing the parallelism in the application. His observation of this fact has come to be called Amdahl's Law and has been formalized more mathematically than he actually presented it. If t_r is the time to execute the parallel region of the application on a single processor, and t_s is the time to execute the sequential region of the application on a single processor, then, according to Amdahl's law, the speedup of the application, $S(P)$, employing P degree of parallelism, is given by the following equation:

$$S(P) = \frac{t_r + t_s}{\frac{t_r}{P} + t_s} \quad [4.1]$$

Figure 4.1 illustrates Amdahl's law. Amdahls law applies broadly and has important implications such as:

- If the parallel region time, t_r, is too small, any effort to improve its performance will have little effect on the overall performance of the application.
- As P approaches ∞, the upper bound for the speedup approaches $\frac{t_r+t_s}{t_s}$. In other words, the inverse of the sequential fraction is the most speedup one can ever obtain, so the more the sequential fraction in the application, the less speedup is possible.

4.2 Modeling Side Effects

Amdahl's law assumes that the parallel region is perfectly parallelizable, i.e., speedup in the parallel region is P. It also assumes that the parallel region execution time, t_r, is independent of the degree of parallelism, P, employed by the application.

However, in reality, as mentioned in Chapter 3, the parallel region can incur dynamic serialization and dynamic acceleration, the combination of which can either increase or decrease the parallel region execution time. Further, these effects can vary with respect to P, as discussed earlier in Chapter 3. In order to get an accurate prediction of the realistic scalability of an application, apart from the parallel and sequential execution times, Amdahl's law must reflect these two side effects as well.

To account for dynamic serialization and dynamic acceleration, let t_q(P) be the additional time incurred due to dynamic serialization and t_c(P) be the time saved due to dynamic acceleration, when employing a degree of parallelism of P. Then the new speedup, $S'(P)$, is:

$$S'(P) = \frac{t_r + t_s + t_q(1) - t_c(1)}{\frac{t_r}{P} + t_s + t_q(P) - t_c(P)} \qquad [4.2]$$

Since we are concerned with the net impact of the side effects, we combine the two quantities, $t_q(P)$ and $t_c(P)$ into $t_{qc}(P)$ as follows:

$$t_{qc}(P) = t_q(P) - t_c(P) \qquad [4.3]$$

Substituting this in Equation 4.2, we get:

$$S'(P) = \frac{t_r + t_s + t_{qc}(1)}{\frac{t_r}{P} + t_s + t_{qc}(P)} \qquad [4.4]$$

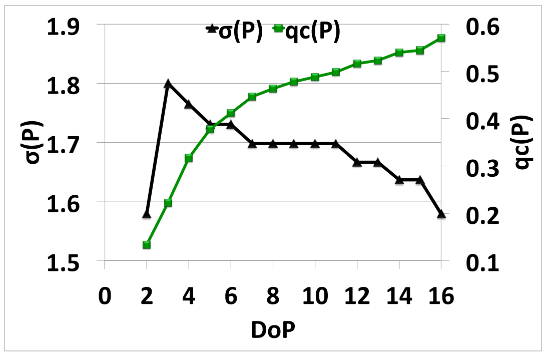
Equation 4.4 captures the impact of side effects on an application's overall performance, which includes both parallel and serial regions. However, we are only concerned with the impact of these effects when the application is executing in parallel, i.e., $P > 1$, since they arise only in the parallel region. Therefore, $t_{qc}(1)$ is zero. Further, the term t_s can be eliminated, since it is applicable only to the serial region (when $P = 1$). Accordingly, for the speedup, $\sigma(P)$, obtained in the parallel region, Equation 4.4 becomes:

$$\sigma(P) = \frac{t_r}{\frac{t_r}{P} + t_{qc}(P)} = \frac{1}{\frac{1}{P} + \frac{t_{qc}(P)}{t_r}}, \; where\ P > 1 \qquad [4.5]$$

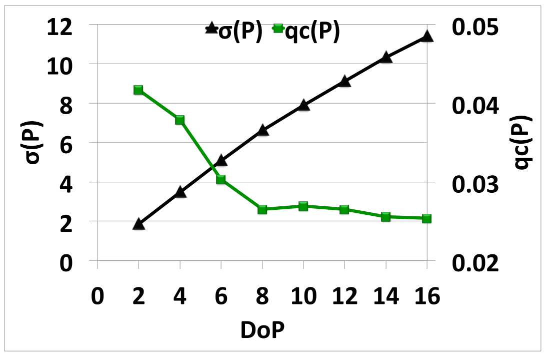
Let $qc(P) = \frac{t_{qc}(P)}{t_r}$. From Equation 4.5 it follows that:

$$qc(P) = \frac{1}{\sigma(P)} - \frac{1}{P} \qquad [4.6]$$

$qc(P)$ provides insights into how the execution of the application's parallel region is influenced by the current operating conditions. A positive $qc(P)$ signifies dynamic serialization, a negative $qc(P)$ signifies dynamic acceleration, and a zero value indicates perfect speedups. An increase in $qc(P)$ with an increase in P indicates that dynamic serialization is dominating, whereas a decrease in $qc(P)$ indicates that dynamic acceleration is dominating. A stable $qc(P)$ indicates that these factors are not influencing the application's scalability. Further, when increasing parallelism from $P1$ to $P2$, if $qc(P2) > qc(P1)$ and $qc(P2) > \frac{1}{P2}$, then $\sigma(P2) < \sigma(P1)$, i.e., the increase in dynamic serialization due to the increase parallelism has resulted in performance degradation.



(a) ReverseIndex



(b) Barneshut

Figure 4.2 Speedup $\sigma(P)$ and $qc(P)$ trends of ReverseIndex and Barneshut on Opteron.

Figures 4.2(a) and 4.2(b) illustrate these aspects. They plot the measured speedup, σ(P) (primary vertical axis), and the computed $qc(P)$ (secondary vertical axis) for two of our applications, ReverseIndex and Barneshut, respectively, with varying degrees of parallelism (P), on one of our experimental platforms, Opteron (details are provided in Chapter 6). ReverseIndex processes files and places significant demands on the disk bandwidth. Even modest attempt at parallel execution results in disk contention, indicated by the higher and increasing values of $qc(P)$. Between $P = 2$ and $P = 3$, the application scales even if $qc(P)$ increases, because $qc(P)$ is less than $\frac{1}{P}$ ($qc(2) = 0.12$ which is less than $\frac{1}{2}$ and $qc(3) = 0.23$ which is less than $\frac{1}{3}$). But, when P exceeds 3, $qc(P)$ not only does it increase with P, but is also greater than $\frac{1}{P}$, resulting in slowdown. For example, at $P = 4$, $qc(4) = 0.32$ is greater that both $qc(3) = 0.23$ and $\frac{1}{4}$.

Barneshut is highly scalable and has few contention concerns. In contrast to ReverseIndex, Barneshut exhibits opposite trends in $qc(P)$. When the parallelism increases, not only does the number of processors change but also the cumulative size of the caches. As the total available cache size increases, more of Barneshut's working set fits in it, reducing the memory access time. This causes $qc(P)$ to reduce, providing additional speedup from $P = 2$ to $P = 8$. Even when $qc(P)$ remains relatively constant, from $P = 8$ to $P = 16$, Barneshut continues to speed up. Hence stable or decreasing $qc(P)$ indicates that additional parallelism is likely to improve performance.

By computing $qc(P)$ from measured σ(P) (speedup of the parallel region) and using these observations, we can determine the optimum degree of parallelism for different performance objectives. The sections to follow show how we determine the optimum degree of parallelism for two different performance objectives: (i) maximize application throughput, and (ii) minimize resource consumption cost. May more performance objectives, such as $\frac{perf}{J}$, $\frac{perf}{W}$, maximize throughput under a power budget, etc., are possible. We leave them for future work.

4.3 Maximize application throughput, $MAX(throughput)$:

We can obtain the optimum degree of parallelism, P_{opt_t} that maximizes throughput of the parallel region by simply differentiating Equation 4.5 wrt to P and equating it to zero as follows:

$$\frac{d\frac{1}{\sigma(P)}}{dP} = -\frac{1}{P^2} + \frac{dqc(P)}{dP} = 0 \qquad [4.7]$$

$$P_{opt\_t} = \sqrt{\frac{1}{\frac{dqc(P)}{dP}}} \qquad [4.8]$$

where $\frac{dqc(P)}{dP}$ is the rate of change of $qc(P)$ or the gradient of the $qc(P)$ curve at a given P. For the purpose of this derivation, we assume that P is a continuous variable, although, in reality, it is discrete.

Note that Equation 4.8 is applicable only when $\frac{dqc(P)}{dP}$ is positive, as in the case of ReverseIndex (Figure 4.2(a)). A negative or zero $\frac{dqc(P)}{dP}$, however, as in the case of Barneshut In Figure 4.2(b), indicates that the application is benefiting from more parallelism. Hence, as many resources as possible, P_{max}, may be allocated to the application. Amending Equation 4.8 with boundary conditions we get:

$$P_{opt\_t} = \begin{cases} \sqrt{\frac{1}{\frac{dqc(P)}{dP}}} & if\ \frac{dqc(P)}{dP} > 0 \\ P_{max} & if\ \frac{dqc(P)}{dP} \leq 0 \end{cases} \qquad [4.9]$$

The AE uses Equation 4.9 as an online metric to determine the instantaneous optimum degree of parallelism for the prevailing operating conditions, without resorting to a time-consuming exhaustive search. This permits Varuna to respond more swiftly to changes. The AE dynamically computes $qc(P)$ and $\frac{dqc(P)}{dP}$ to determine P_{opt_t} and P_{max}, as we shall see in §4.5.

4.4 Minimize resource consumption cost, $MIN(consumption)$:

To find the optimum parallelism, P_{opt_c}, that minimizes the resource consumption cost, we want to minimize the product $P \times \frac{1}{\sigma(P)}$. Similar to the first objective, P_{opt_c} can be obtained by simply differentiating $P \times \frac{1}{\sigma(P)}$ wrt P, and then equating it to zero as follows:

$$\frac{d\frac{P}{\sigma(P)}}{dP} = P \times \frac{dqc(P)}{dP} + qc(P) = 0 \qquad [4.10]$$

$$P_{opt\_c} = -\frac{qc(P)}{\frac{dqc(P)}{dP}} \qquad [4.11]$$

A negative $\frac{qc(P)}{\frac{dqc(P)}{dP}}$ indicates net dynamic acceleration and hence efficient resource consumption (Figure 4.2(b)). In this case P_{opt_c} is computed using the formula $-\frac{qc(P)}{\frac{dqc(P)}{dP}}$. $\frac{qc(P)}{\frac{dqc(P)}{dP}}$ will yield multiple values depending on the value of P. The AE picks the one with minimum value of $\frac{qc(P)}{\frac{dqc(P)}{dP}}$ since it signifies the least contention and hence most efficient consumption of resources. If both $\frac{dqc(P)}{dP}$ and $qc(P)$ are positive, the application is not scaling due to dynamic serialization, and hence the resources are not being consumed efficiently, as shown in Figure 4.2(a). In this case, minimum resources, P_{min}, are allocated to the application. If both $\frac{dqc(P)}{dP}$ and $qc(P)$ are zero or negative, the application is scaling linearly or superlinearly and hence, as many resources as possible, P_{max}, may be allocated to it, as in the case of Barneshut in Figure 4.2(b). Amending Equation 4.11 with boundary conditions we get:

$$P_{opt\_c} = \begin{cases} -\frac{qc(P)}{\frac{dqc(P)}{dP}} & if\ \frac{qc(P)}{\frac{dqc(P)}{dP}} < 0 \\ P_{min} & if\ \frac{dqc(P)}{dP} > 0 \ \&\ qc(P) > 0 \\ P_{max} & if\ \frac{dqc(P)}{dP} \leq 0 \ \&\ qc(P) \leq 0 \end{cases} \qquad [4.12]$$

4.5 Determining $\mathbf{qc(P)}$, $\mathbf{\frac{dqc(P)}{dP}}$, $\mathbf{P_{max}}$ and $\mathbf{P_{min}}$

To apply Equation 4.9 and Equation 4.12 as online metrics, the AE needs to compute $qc(P)$, $\frac{dqc(P)}{dP}$, P_{max} and P_{min} dynamically. Unfortunately, modern processors and systems do not supply this information directly, and hence, we compute them empirically.

4.5.1 Determining $\mathbf{qc(P)}$

To determine $qc(P)$, the AE computes σ(P) empirically using Equation 4.6, and this requires computing a baseline performance measure for the parallel region. To do this, whenever the AE detects a change in the execution environment, it sets the instantaneous parallelism of the application to one, i.e., $P = 1$, for a pre-defined time period in all our experiments, monitors its execution and establishes a baseline performance, $Perf(1)$.

One of the key considerations to determine the instantaneous performance of the application is selecting the appropriate monitoring time interval. A very small interval can increase the system overheads, whereas large intervals can cause delays in determining the optimum degree of parallelism. On the Intel Sandy Bridge microarchitecture and its successors, experiments have shown [103] and the documentation [1] states that performance counters can be accessed as often as every microsecond with our incurring any significant overheads. However, as Chapter 6 will demonstrate, monitoring at this granularity can incur high overheads in the application execution time, especially in multiprogrammed environments. Further, monitoring at this granularity makes it difficult to capture the Operating System context switching overheads, which typically happens, on an average, at $50ms$ granularity. With $100ms$ and bigger time intervals, we observed that the impact of monitoring on the application performance is negligible. Therefore, the AE uses $100ms$ as the monitoring interval for measuring instantaneous performance of the application.

Another consideration to determine the instantaneous performance is to pick the right metric to represent performance. Generally, it depends on the type of the application. For example, for mobile class applications, $Instructions\ per\ Second\ (IPS)$ is a good measure. For server class applications, $Requests\ Per\ Second\ (RPS)$ is a suitable measure. In this dissertation, we use

IPS to represent performance. We make a fair assumption that spin-locks in application code are rare and that users use standard synchronization interfaces to access their critical sections. Spin-locks can occur in the Operating System and we avoid this issue by not counting the Operating System instructions.

Once the baseline performance is measured, AE switches the degree of parallelism to P, allows the application to run for $100ms$ and measures its performance, $Perf(P)$. $\sigma(P)$ can then be obtained by using the following equation:

$$\sigma(P) = \frac{Perf(P)}{Perf(1)} \qquad [4.13]$$

The AE substitutes this value in Equation 4.6 to obtain different values of $qc(P)$.

4.5.2 Determining $\frac{\mathbf{dqc(P)}}{\mathbf{dP}}$

One possible way of determining $\frac{dqc(P)}{dP}$ is to identify the shape of the $qc(P)$ curve. However, this requires sweeping through all the values of $qc(P)$ and may not be a scalable solution, especially when the operating conditions can change frequently, as the process of sweeping can range from several hundreds of milliseconds to seconds and can grow with the increase in number of processors. Hence, we need a much quicker way of computing $\frac{dqc(P)}{dP}$.

As we demonstrate below, we always ensure that the parallel region is no slower than its sequential execution (by controlling the parallelism), i.e., $\sigma(P) \geq 1$. Further, when $\frac{dqc(P)}{dP}$ is used to compute P_{opt_t} (Equation 4.9), it can only be positive, implying that dynamic serialization is increasing with P, i.e., $0 < t_{qc}(P)$. Thus $0 < t_{qc}(P) < t_r$, and $1 \leq \sigma(P) < P$, hence $0 < qc(P) < 1$.

Similarly, when computing P_{opt_c} (Equation 4.12), $\frac{dqc(P)}{dP}$ can only be negative, implying dynamic acceleration. In this case, $-t_r \leq t_{qc} < 0$ (dynamic acceleration can only eliminate t_r in the best case, and no more). Thus $-1 < qc(P) < 0$.

Further, we empirically observed that $\mathbf{qc(P)}$ **is almost always a monotonic function with respect to** $\mathbf{P}$. A function, f, is monotonic if it is either entirely non-increasing or non-decreasing. It is monotonically increasing (also increasing or non-decreasing), if for all x and y such that

$x \leq y$, $f(x) \leq f(y)$, so f preserves the order. Likewise, a function is monotonically decreasing (also decreasing, or non-increasing) if, whenever $x \leq y$, then $f(x) \geq f(y)$, so it reverses the order. Figure 4.2 provides an example illustration of $qc(P)$'s monotonic behavior. In Reverse Index, $qc(P)$ strictly increases with P (from $P = 2$ to $P = 16$), whereas in Barneshut, $qc(P)$ first falls with increase in P (from $P = 2$ to $P = 8$) and then stays almost a constant (from $P = 9$ to $P = 16$), but never rises with P.

Given the above relatively narrow range of *qc(P)* (between 0 and 1 or between -1 and 0) as compared to the range of P (e.g., 2 to 24 in our experiments), and its monotonicity with respect to P, we approximate the $qc(P)$ curve to a straight line of the form:

$$y = mx + c \qquad [4.14]$$

where x and y correspond to P and $qc(P)$, respectively, m is the gradient of the straight line graph and it corresponds to $\frac{dqc(P)}{dP}$, and c is the $y - intercept$ of the straight line graph.

To obtain estimates of $\frac{dqc(P)}{dP}$, the AE uses linear regression, based on the ordinary least squares estimation, on a subset of $qc(P)$ values. Although linear regression may lead to errors, our experiments (Chapter 6) and residual analysis [84] show that it is adequate and leads to better results than the state-of-the-art adaptive methods, which resort to time-consuming iterative search strategies.

Based on our experiments for $1 \leq P \leq 24$ (our experimental platforms have a maximum of 24 hardware contexts), linear regression using data for three parallelism configurations (in addition to $P = 1$ for which $qc(1) = 0$) gave sufficiently accurate estimates of $\frac{dqc(P)}{dP}$[1]. Hence, we restrict our measurement to three points (P_1, P_2 and P_3) in order to make quick decisions. We picked these points to be 2, $N/2$, and N, respectively, where N is the maximum number of hardware contexts available in the machine, to determine the overall scalability behavior of the application. In Chapter 6, we demonstrate that this approach is sufficient to make informed decisions to arrive at the optimum parallelism configuration.

[1]Higher values of P may require more data for accurate estimates.

The AE computes performance at the three different degrees of parallelism, $P_1 = 2$, $P_2 = \frac{N}{2}$ and $P_3 = N$, as described in Section 4.5.1. $\sigma(P_1)$, $\sigma(P_2)$, and $\sigma(P_3)$ can then be obtained by dividing the corresponding performance measures by the baseline performance, using Equation 4.13. The AE verifies that these values are indeed greater than one. Otherwise, it will switch to sequential execution. These values are substituted in Equation 4.6 to get $qc(P_1)$, $qc(P_2)$, and $qc(P_3)$, which are then used to obtain $\frac{dqc(P)}{dP}$ by applying the least square method.

The AE then computes P_{opt}, based on the performance objective considered, by substituting the above values in the corresponding equation.

4.5.3 Determining $\mathbf{P_{max}}$ and $\mathbf{P_{min}}$

The monotonic behavior of $qc(P)$, and our methodology of obtaining $\frac{dqc(P)}{dP}$, makes the determination of P_{max} and P_{min} fairly straightforward. When optimizing for P_{opt_t}, if $\frac{dqc(P)}{dP}$ is negative, it implies that the application scales till the maximum number of hardware contexts, N, and hence, P_{max} is set to N. When optimizing for P_{opt_c}, there are two cases. If $\frac{dqc(P)}{dP} > 0$ & $qc(P) > 0$, it indicates that the application does not consume resources effectively when executing in parallel and that sequential execution is the most optimum. Hence, P_{min} is simply set to 1. If $\frac{dqc(P)}{dP} \leq 0$ & $qc(P) \leq 0$, it implies that the application consumes resources effectively up to N, and hence, P_{max} is set to N.

4.6 Monitoring and Recalibrating $\mathbf{P_{opt}}$

Once P_{opt} is determined for the desired performance objective, the AE conveys this value to the Parallelism Manager (PM). The PM controls the execution of the parallel computations to match the P_{opt} determined by the AE. These details are described in Chapter 5. Once conveyed, the AE then enters into a *passive monitoring mode* where it periodically monitors the processor, memory, I/O and energy usage characteristics of the application for changes. To do so, it maintains a *usage vector* that comprises a variety of metrics, including *Instruction fetch rate*, *last level cache miss rate*, *CPU utilization*, *memory bandwidth utilization*, *instructions retired per second*, *number of page faults*, *processor and system energy consumption*, among others. If a change in any of these metrics is detected, the AE switches the degree of parallelism to 1 and repeats the process described in § 4.5 to recalibrate P_{opt}.

To measure processor energy, the AE, in this dissertation, use the RAPL (Running Average Power Limit) power management interface provided in the modern processors, such as Intel SandyBridge and IvyBridge [1]. To measure overall system energy, the AE uses Wattsup meter [3]. To measure metrics related to processor, memory and I/O usage characteristics, the AE uses the PAPI library APIs [92]. We do not count instructions that may cause potential side effects when determining the efficiency of the application. For example, we omit instructions used for synchronization, and OS mode instructions.

4.7 Limitations

One of the fundamental limitations of the scalability model is that it assumes that $qc(P)$ is a monotonically varying function with respect to P. Although our experiments demonstrate that this assumption is sufficient for the AE to rapidly and accurately determine the optimum parallelism, P_{opt}, to be employed, there are scenarios in which $qc(P)$ could possibly vary non-monotonically with respect to P. In such scenarios, Equations 4.9 and 4.12 may determine a P_{opt} value that may not be optimum to the execution environment. To handle such scenarios, the AE, while in passive monitoring mode, periodically diversifies (at $3s$ granularity in our experiments) by switching the degree of parallelism to a randomly chosen value between 1 and N, say P_{random}, monitors its execution and measures its performance, $Perf(P_{random})$. If $Perf(P_{random})$ is less than or equal to $Perf(P_{opt})$, then the AE switches the execution back to P_{opt} and enters into the passive monitoring mode. Otherwise, the AE concludes that the $qc(P)$ curve could possibly be non-monotonic and invokes a hill climbing based search heuristic, as described next.

4.8 Hill climbing Heuristic

This heuristic is based on the commonly used hill-climbing search algorithm. It locates the optimum degree of parallelism by systematically exploring the direction in the search space that yields higher efficiency. Generally speaking, the heuristic starts at some mid-point number of hardware contexts, P, and computes the application's efficiency. Then, another P between the current configuration and either of the active endpoints is chosen, and the process is repeated. If the optimum for that P is better, the other side is disregarded, and the search continues on that side. Otherwise, the heuristic switches to the other side, disregarding further attempts on the first side. When neither side is better, or we run out of parallelism configurations, the search ends.

Hill climbing algorithms are known to get "stuck" in a local optimum and fail to reach the global optimum. Hence this heuristic, developed using the guidelines due to Gendreau [50], incorporates *Tabu search* [52] to escape local optima. The Tabu search maintains a fixed-size *tabu list* to log previously searched points along with their efficiencies. The heuristic periodically "diversifies" the search from the current optimum by exploring points that are not present in the *tabu list*. Any time a point is searched, it is logged in the *tabu list*. The oldest point in the list is evicted to make room for a new point, if needed.

Section 4.8.1 will first present the basics of tabu search. Subsequently, Section 4.8.2 will discuss the heuristic itself.

4.8.1 Basics of Tabu Search

The basic concept of tabu search, as described by Glover et al. [52], is "a meta-heuristic superimposed on another heuristic". The overall approach is to avoid entrainment in cycles by forbidding or penalizing moves which take the solution, in the next iteration, to points in the solution space previously visited.

The tabu method was partly motivated by the observation that human behavior appears to operate with a random element that leads to inconsistent behavior given similar circumstances. As pointed out by Glover et al. [52], the resulting tendency to deviate from a charted course, might be regretted as a source of error but can also prove to be source of gain. The tabu method operates in this way with the exception that new courses are not chosen randomly. Instead the tabu search proceeds according to the supposition that there is no point in accepting a new (poor) solution unless it is to avoid a path already investigated. This insures new regions of the solution space will be investigated in, with the goal of avoiding local minima and ultimately finding the desired solution.

The tabu search begins by marching to a local minima. To avoid retracing the steps used, the method records recent moves in one or more *tabu lists*. The original intent of the list was not to prevent a previous move from being repeated, but rather to insure it was not reversed.

The tabu lists are historical in nature and form the tabu search memory. The role of the memory can change as the algorithm proceeds. At initialization the goal is make a coarse examination of the solution space, known as "diversification", but as candidate locations are identified the search is more focused to produce local optimal solutions in a process of "intensification". In many cases the differences between the various implementations of the tabu method have to do with the size, variability, and adaptability of the tabu list to a particular problem domain.

The tabu search has traditionally been used on combinatorial optimization problems. The technique is straightforwardly applied to continuous functions by choosing a discrete encoding of the problem. Many of the applications in the literature involve integer programming problems, scheduling, routing, traveling salesman and related problems.

4.8.2 Iterative Tabu-based Search Heuristic

The proposed heuristic uses the following schema, as depicted in Figure 4.3, to determine the optimum degree of parallelism: establish a baseline efficiency measure; determine the region of the search space to explore; identify an optimal parallelism configuration by repeatedly searching for a better configuration in the established search direction; control parallel execution to change parallelism configurations, and measuring the efficiency of the new configuration relative to the baseline; trigger a new search if the dynamic execution environment changes or a diversification threshold is reached.

Step 1: Establish Sequential Measure: Before beginning the search, the AE establishes a notion of *Sequential efficiency* (Eff_{seq}) to ensure that the optimum point it finds is indeed profitable.

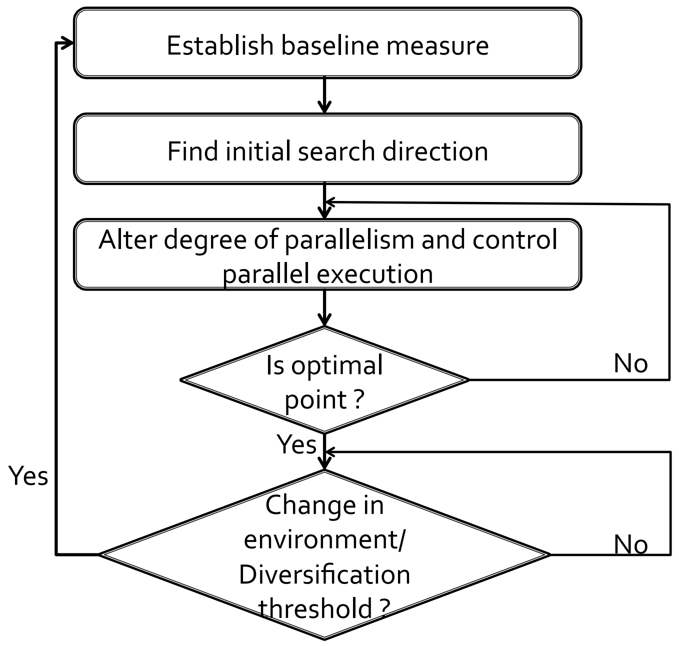

Establish baseline measure
Find initial search direction
Alter degree of parallelism and control parallel execution
Is optimal point ?
No
Yes
Change in environment/ Diversification threshold ?
No
Yes

Figure 4.3 High level operations of the iterative search heuristic.

Hence, whenever the application encounters new conditions, the AE alters the degree of parallelism to one (sequential) and executes the application for a pre-defined duration until it measures the baseline Eff_{seq}[2]. The metric to determine the efficiency of the application depends on the performance objective that the application user enforces. For the *maximize application's throughput* objective, the AE uses $Instructions\ per\ second(IPS)$ as the efficiency metric. For the *minimize the resource consumption cost* objective, it uses $\frac{Degree\ of\ parallelism}{Instructions\ per\ second(IPS)}$ as the efficiency metric.

Step 2: Establish Initial Search Direction: Next, the AE establishes the search direction for the optimization process. It gathers $Effs$ corresponding to three different degrees of parallelism, the mid-point, P_{mid}, of the pre-determined maximum degree of parallelism (often, total hardware contexts in the system), and P_{mid} +/- 1. The application is executed for $100ms$ duration in each configuration. The better of the three points, P_{eff}, sets the search direction. The points are added to the *tabu list* and the execution is switched to P$_{eff}$.

[2]We fixed the pre-defined duration as $100ms$ for the reasons mentioned in Section 4.5

Performance objective	**Efficiency metric**
$Maximize\ throughput:$ $MAX(throughput)$	$Instructions\ per\ second\ (IPS)$
$Minimize\ resource\ consumption\ cost:$ $MIN(consumption)$	$\frac{Degree\ of\ parallelism}{Instructions\ per\ second\ (IPS)}$

Table 4.1 Performance objectives and their corresponding efficiency metrics considered by the Analytical Engine for the iterative search heuristic.

Step 3: Search for Optimum Parallelism: If P_{eff} is the same as P_{mid}, the search has already resulted in a local optimum, else the AE linearly searches in the established direction [3]. It picks new unvisited points (that are not in the *tabu list*) and executes the application in each of these points for $100ms$ duration. Visited points are added to the *tabu list* until a local optimum, P_{lopt}, is found. If Eff_{lopt} is not significantly better than Eff_{seq}, the AE starts over from step 1, otherwise it fixes the current configuration to P_{lopt}, and enters a "sleep" mode.

Step 4: Sleep Mode: In this mode, the AE performs passive monitoring of the system until it (i) detects changes in the execution environment using the strategy described in Section 4.6, or (ii) reaches a diversification threshold. In case of (i), the AE clears the *tabu list* and begins the search once again from step 1. In case of (ii), the AE saves the current Eff_{lopt} and moves to step 5.

Step 5: Diversification: The diversification threshold is a pre-defined interval ($5s$ in all our experiments) after which the AE ensures that the execution is not trapped in a local optimum. It begins the search from the original mid-point in the direction opposite to the original, similar to step 3 (updating the *tabu list* and avoiding already visited points), until a local optimum P_{lopt2} is reached. If Eff_{lopt2} is better than Eff_{lopt}, the configuration is switched to P_{lopt2}, else it is switched back to P_{lopt}. In either case, the AE enters the sleep mode and moves to step 4.

[3]In our experiments on 8- and 24-context machines, the search space is small enough to perform linear search. For larger search spaces, binary search may prove to be better.

4.9 Chapter Summary

This chapter developed a scalability model based on Amdahl's law to calculate the realistic speedup of an application's parallel region and demonstrated how the model provides insights into an application's parallel execution behavior. It then showed how the model could be used to derive formulae that determine optimum degree of parallelism for two different performance objectives: (i) maximize application throughput, and (ii) minimize resource consumption cost. It then presented how the different quantities in the derived formulae can be empirically determined. The chapter then described how the AE monitors the execution environment for changes and recalibrates the optimum degree of parallelism. Finally, the chapter described the limitations of the scalability model and presented an alternate hill climbing heuristic, based on Tabu search, to overcome the limitations.

Chapter 5

Parallelism Manager

The role of the Parallelism Manager (PM) is to automatically control the execution of parallel computations in the applications to match the degree of parallelism determined by the Analytical Engine (AE). Chapter 3 presented the high level operations of the PM. This chapter presents the details of those operations.

Once the AE determines the optimum degree of parallelism, P_{opt}, for a given performance objective, it signals the Parallelism Manager (PM) to control the number of inflight computations to match the P_{opt}. To be able to continuously and transparently alter the number of parallel computations without hampering the application's forward progress, the PM maps units of computation designated for parallel execution, e.g., a task in task-based parallel applications, or a thread in multithreaded applications, to *vtasks*. As mentioned in Chapter 3, vtasks abstract hardware contexts into logical cooperative tasks [6, 4] to which application's parallel computations are transparently mapped. The vtasks, in turn, are dynamically scheduled onto hardware contexts, similar to tasks in a dynamic task-based runtime system. Each vtask maintains the state of the current computation mapped on to it using *contexts*, allowing Varuna to transparently pause, resume, or migrate a computation by saving or restoring its corresponding vtask's context.

Vtasks are closest in spirit to *fibers* [4], an implementation of cooperative tasks in the Windows Operating System, but with three key differences as follows:

1. Unlike fibers, vtasks are not exported as programming abstractions to developers. They abstract away the details of parallelism management from the application without altering the existing parallel programming interfaces, similar to how processor virtualization employs

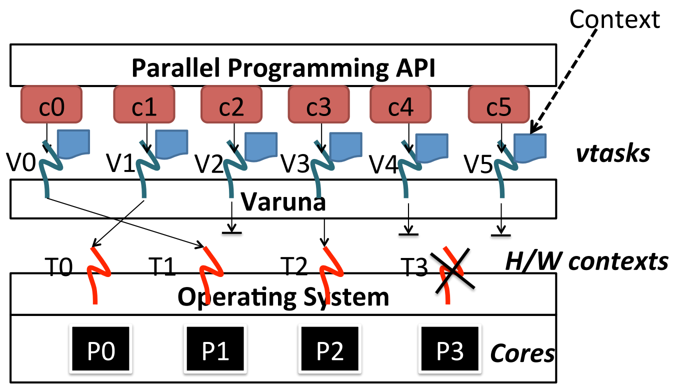


Figure 5.1 Transparent suspension and migration of vtasks.

VCPUs to abstract away the details of on-chip resource management from the system software without changing the existing ISA. Programmers are free to express their parallel computations using any desired parallel programming model. For example, they could express their parallel computation as software threads using Pthreads or as tasks using Intel's TBB. The PM transparently intercepts these constructs, creates vtasks for each of the computation invoked by the programmer and reassigns the computations encapsulated inside these constructs on to vtasks. Such a strategy allows the PM to suspend a computation by blocking its corresponding vtask context, or remap a computation from one hardware context to another by migrating its corresponding vtask context, while abstracting its details and complexity from the application. Figure 5.1 shows transparent migration of computations $c0$ and $c1$ to hardware contexts $T1$ and $T0$, respectively, and the suspension of computations $c2$, $c4$ and $c5$.

2. Vtasks require no additional programming effort to manage their scheduling and context switches, providing Varuna the flexibility to transparently control their execution, whereas fibers must be explicitly created and managed by the application developer.

3. Vasks are *progress-aware* entities. Their contexts include additional state, such as locks and conditional variables, needed to ensure the computations' forward progress even as their

execution is regulated. Using fibers as is for Varuna's objectives can hamper the application's forward progress.

As the application executes, the PM maintains a pool of vtasks. When the AE increases the degree of parallelism, the PM assigns more vtasks from the vtask pool to the hardware contexts. When the AE decreases the degree of parallelism, the PM suspends executing vtasks and returns them to the vtask pool until they can be resumed later. To perform such control, the PM maintains the state of each vtask, tracks its status, and schedules its execution. Since concurrent computations can interact through shared state, care is needed to ensure their forward progress, especially in multithreaded programs.

The rest of the chapter discusses these aspects and is organized as follows. Section 5.1 presents the details of the vtask's context data structure. Section 5.2 describes how vtasks are managed and scheduled. Finally, Section 5.3 describes the mechanisms employed by the PM to enure an application's forward progress.

5.1 Vtask Context

To enable suspension, resumption and migration of a vtask, each vtask contains a *vtask context block (VCB)*. As depicted in Figure 5.2, the VCB contains the following state:

1. a *Program Counter (PC)* that specifies the address of the next instruction in the vtask control flow to be executed,
2. *General Purpose Registers (GPRs)* that contain data generated by the vtask computation,
3. a *Stack Pointer (SP)* that points to the next entry in the vtask's *call stack*,
4. a *Call Stack (CS)* that maintains the activation records of active functions invoked by the computation running on the vtask, and
5. a *Mutex Counter (MC)* that contains the number of mutex variables currently acquired by the vtask computation when executing in user mode.

```
struct context
{
    uint64_t PC;    //Program counter
    uint64_t GPR[MAX_REGISTERS];    //General purpose registers
    uint64_t* SP; //Stack pointer
    call_stack_t* CS // Call stack
    uint64_t MC; //Mutex counter
};
```

Figure 5.2 Vtask context block.

The first four quantities are the standard elements necessary to save an execution's state. The *Mutex Counter (MC)* is needed to ensure forward progress of applications when varying the number of inflight vtasks (§5.3). VCBs serve two purposes. First, they provide the ability to save and restore the state of a running computation at arbitrary points in time. Second, they enable interoperability between vtasks that may need to block due to: (1) mutual exclusion, locks and barriers, and (2) when the number of vtasks exceeds the number of available hardware contexts to execute.

To implement the call stack, the PM employs an approach inspired from lazy threads [53], originally proposed to reduce the overheads of nested parallel function calls. The strategy supplies each vtask with a *stacklet* — a linear stack that stores the activation records of functions invoked by computations mapped on to the vtask. Whenever the application invokes a new parallel computation, the PM creates a new vtask for that computation and associates a stacklet with it. The lifetime of the stacklet is the same as the lifetime of the vtask using it. Any computation mapped on to a vtask is free to use its associated stacklet to perform unrestricted allocation and deallocation of its activation records similar to ordinary subroutine invocations in a sequential program.

There are two advantages for the PM to maintain its own call stack for each vtask in userspace over using the stack already provided by the Operating System for each hardware context. First, it allows unrestricted creation and scheduling of a large number of vtasks independent of the number of hardware contexts. Second, it eliminates kernel mode switching while migrating a vtask from one hardware context to another or replacing a vtask with another on the hardware context.

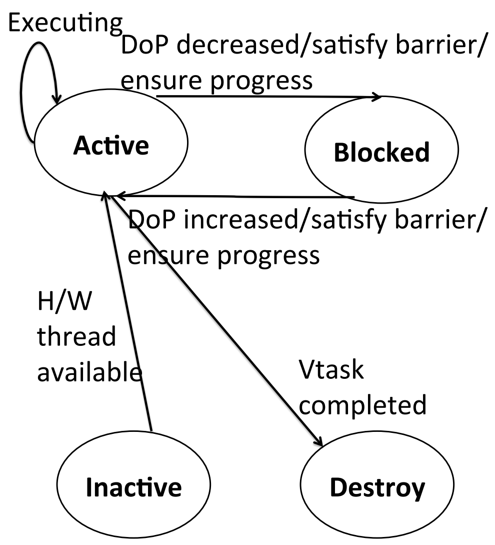

Executing
DoP decreased/satisfy barrier/ ensure progress
Active
Blocked
DoP increased/satisfy barrier/ ensure progress
H/W thread available
Vtask completed
Inactive
Destroy

Figure 5.3 Vtask state transition diagram.

To suspend a vtask, the PM saves all user-level registers and the PC on the vtask's stacklet, so that the stacklet is a self-contained record of the context state. To resume the vtask, the PM restores registers from the stacklet.

There are three alternatives to track contexts: (1) using $setjmp$ and $longjmp$ routines, (2) using the new $glibc$ context routines, or (3) writing custom assembly routines. Obviously the C library routines provide higher portability. However, using them would require two system calls on every vtask context switch, making these routines significantly slower than a pure user-space solution. We chose to employ the third option. That is, we employ own custom user-level context management assembly routines to enable low-overhead vtask context switching.

5.2 Managing and Scheduling Vtasks

Figure 5.3 depicts the state transition diagram the PM uses to manage vtasks, from their creation to destruction. A vtask, when created, is always in one of four states: *inactive*, *active*, *blocked* or *destroyed*. These states reflect whether or not a hardware context exists for a vtask to execute, and transitions between these states occur during its creation, suspension and completion. When a vtask is created by the Vtask Generator (VM) (Figure 3.5), it is in an *inactive* state. A vtask moves to an *active* state when the Resource Mapper (RM) assigns it for execution, either when the degree of parallelism is increased or when a current hardware context becomes free. To decrease the degree of parallelism, the RM pre-empts excess vtasks when they reach a *safe point*, transitions them to a *blocked* state, and moves them back to the vtask pool. We describe what *safe points* are in Section 5.3. Safe point pre-emption is necessary to ensure a vtask's forward progress. When decreasing degree of parallelism, the RM moves vtasks to a *blocked* state in the order in which they arrive at the safe point. A vtask is also moved to a *blocked* state when it arrives at a barrier and the barrier condition is not satisfied. When assigning vtasks for execution, the RM prioritizes vtasks that have waited the longest, to ensure fairness. Finally, if a vtask finishes its assigned quota of work, it transitions to a *destroy* state before its state is destroyed.

To efficiently schedule vtasks, the RM employs a Cilk-style work stealing scheduler [49]. At the start of the application execution, the RM creates a pool of worker threads, one per hardware context allocated to it by the Operating System. A double-ended work queue (*deque*) is then assigned to each worker in the system. A worker schedules vtasks for execution by queuing them in its work deque. Each worker seeks vtasks from its own deque, failing which it steals from someone else's deque.

To avoid memory explosion when creating vtasks, the RM uses lazy vtask creation, an idea borrowed from Mohr, et. al., [90]. There are two possible ways to handle the creation of new vtasks in a work-stealing scheduler: *eager* and *lazy*. With eager vtask creation, a master worker spawns a vtask by pushing it into its work deque and then continues executing the rest of the application code, i.e., the continuation of the vtask. When the master worker finishes spawning all the vtasks, it begins drawing vtasks from its own deque and executing them. In the meantime, other slave workers may steal vtasks from the master worker's deque and execute them. Eager vtask creation is straightforward to implement, but it can result in an memory explosion when an application spawns a large number of vtasks [90]. Creating vtasks lazily can avoid this problem.

Using lazy vtask creation, when a worker creates a vtask, rather than pushing the actual vtask into its deque, it pushes the continuation of the vtask. The worker then executes the vtask itself. Once the vtask completes, it attempts to pop the continuation from the deque to resume execution.

While a worker is executing a vtask, it is possible that another worker may steal the vtasks continuation. The stealing worker will execute the continuation until it encounters another creation call, at which point it will save a new continuation into its deque and commence execution of the new vtask. This process continues until all the vtasks in the application are created and destroyed. As it is evident, applying this approach to creating vtasks limits the number of active vtasks to the number of active workers, and would require only one entry in each deque to hold the continuation.

5.3 Ensuring Forward Progress

Controlling parallelism requires the ability to arbitrarily block vtasks from being performed when decreasing the degree of parallelism, and resume, migrate or introduce new vtasks when increasing the degree of parallelism. For example, in Figure 2.1(a), in the multiprogrammed environment, increasing the parallelism from four to six degrades RE's execution efficiency. In this case, it is ideal to execute no more than four workers. Doing so requires preventing additional vtasks from being assigned to more workers. However, vtasks are cooperative tasks and are not pre-emptively scheduled, unlike Operating System threads. In general, tasks in task-based applications are usually independent, and can run to completion without communicating with each other. This, however, may not be the case with threads in multithreaded applications. Hence, arbitrarily pausing/resuming a vtask's execution can potentially affect the forward progress of other vtasks, especially in applications with arbitrary dependence patterns.

A vtask's progress can be affected when it is waiting on:

- a mutex lock held by a *blocked* vtask, or
- a signal from a *blocked* vtask (e.g., a consumer computation waiting to receive data from a producer computation, in a producer-consumer style application).

5.3.1 Handling Blocked Mutexes

Uhlig, et al., [118] have proposed one technique to avoid Operating System lock-holder pre-emption, in a overcommitted virtualized environment where a large number of virtual CPUs (VC-PUs) must be scheduled on a restricted number of available cores. They make the observation that a VCPU executing in user mode is not holding a kernel lock, and can thus be safely preempted. These pre-emptible locations are referred to as *safe points*.

To avoid forward progress issues due to blocked mutexes in vtasks, the RM employs a similar approach. It pauses a vtask only when it reaches a *safe point* in its execution. In the context of vtasks, *a safe point is defined as a control point in the vtask execution flow at which the vtask is currently executing in user mode and does not hold any user mode mutex locks*.

Ensuring that a vtask is executing in user mode is necessary to avoid suspending the vtask after acquiring a kernel-level (spin)lock. Utilizing the fact that the Operating System will release all kernel locks before returning to user-level, the RM can monitor all switches between user-level and kernel-level. However, such monitoring is hard as it requires modifying the Operating System calls or requires prediction mechanisms based on monitoring privileged instructions [118]. Instead, the RM takes a much simpler approach. It suspends a vtask only if it reaches one of the following synchronization points in the control flow: before or after a mutex lock or unlock, respectively, after reaching a barrier, and before or after a conditional wait or signal, respectively. These points are guaranteed to be free of kernel-level locks. If none of the above points are reached, the RM simply waits until the vtask completes its execution.

Ensuring that a vtask is not suspended when holding any user-level mutex locks is necessary to avoid unsafe pre-emptions when in user mode. For example, consider the example shown in Figure 5.4. Vtasks V1 and V2 are contending for the same user-level lock, L1, and V1 acquires its first. The RM cannot allow V2 to proceed when the lock is already acquired by V1, and so it pauses and saves the context of V2, and moves it to a *blocked* state. Later, V1 acquires a second user-level lock, L2 (before releasing L1), at which point the RM decides to suspend V1 due to oversubscription in the system. However, doing so can hamper the forward progress of V2 which is waiting on L1 to be released by V1.

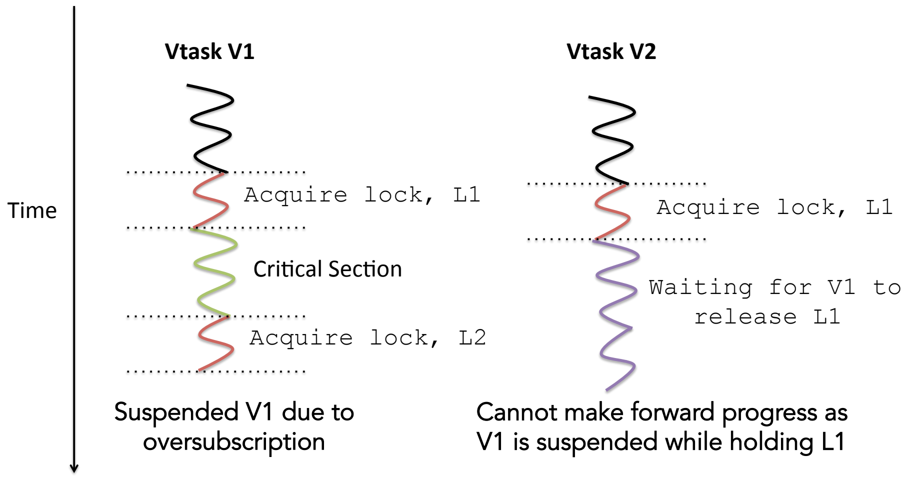

Time
Vtask V1
Acquire lock, L1
Critical Section
Acquire lock, L2
Suspended V1 due to oversubscription
Vtask V2
Acquire lock, L1
Waiting for V1 to release L1
Cannot make forward progress as V1 is suspended while holding L1

Figure 5.4 Deadlock example.

To address this problem, each vtask maintains a count of the number of mutexes it has currently acquired in a *Mutex Counter (MC)*. It increments the counter when acquiring a user-level lock and decrements it when releasing the same. The RM leverages this information and ensures that a vtask is never suspended until its counter becomes zero. For example, in Figure 5.4, the RM will not suspend V1 until it releases both the locks, L1 and L2.

Many multithreaded applications have few safe points since these applications seldom synchronize. Hence they provide few opportunities to control the parallel execution of vtasks. In such cases, more safe points can be created by spawning more vtasks than the actual number of hardware contexts. Note that task-based applications already create many more tasks than actual hardware threads, naturally achieving this. We observe that many multithreaded applications are generally written to take the number of threads as an argument, which is used to divide the work into as many independent portions at run-time. Consider the code in Figure 5.5, which lists a simplified version of the barneshut benchmark from the Lonestar suite [73]. This benchmark gets the number of threads as an argument from the programmer ($line$ 17) and uses it to the divide up its input array, $body$, into independent chunks ($lines$ $6 - 12$) and spawns each chunk for parallel execution. Hence, spawning more threads, similar to tasks, is as simple as altering the command line argument to the application.

```
static void* Process(void* arg)
{
    int start, end;
    const int slice = (long)arg;
    start = slice * nbodies / threads;
    end = (slice + 1) * nbodies / threads;
    for (int i = start; i < end; i++)
    {
        body[i]->ComputeForce(groot, gdiameter);
    }
    return NULL;
}
threads = getNumThreads ();
//main loop
for (int i = 1; i < threads; i++)
{
    // launch the parallel worker threads
    pthread_create(&worker[i], NULL, Process, (void*)i);
}
Process((void *)0);  // run on main thread
for (int i = 1; i < threads; i++) {
    // wait for the parallel worker threads to finish
    pthread_join(worker[i], NULL);
}
```

Figure 5.5 Barneshut example.

As we demonstrate in Chapter 6, spawning a high number of vtasks in Varuna does not have the same overheads as spawning a high number of threads, due to the following reasons:

- Vtasks are userspace objects and hence the cost of switching between them is extremely low,
- the RM employs lazy vtask creation [90], which avoids memory explosion as described in Section 5.2.

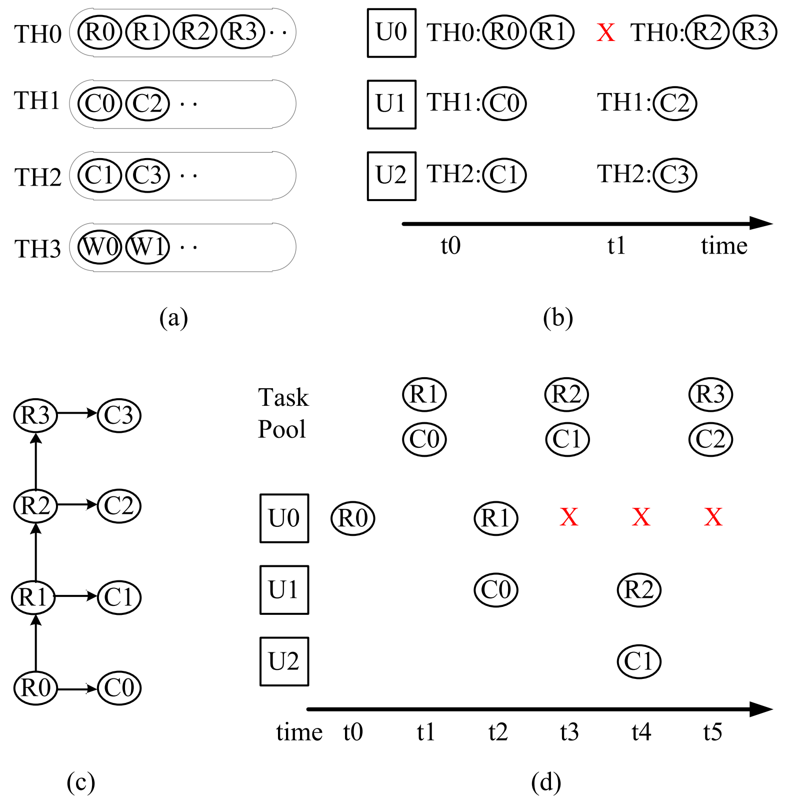


Figure 5.6 Comparison of independence-based and dependence-aware execution of Pbzip2

5.3.2 Handling Blocked Signals

Signaling is another way computations in multithreaded applications communicate with each other (task-based programming models usually do not support signaling). If a computation's execution is dependent on another, there are two generic approaches to mitigate starvation among computations which are not executing concurrently:

1. Avoid pre-empting a computation that is responsible for producing the signal/data to other computations, and

2. Pro-actively pre-empt an executing computation in favor of executing a more productive computation.

To employ the first approach, the PM requires knowledge of dependences between computations. However, predominant of the parallel programming models that exist today, are *independence-based*. That is, these models do not automatically detect data objects accessed by different computation and enforce dependences between them. Rather, they require programmers to honor dependences between computations by using appropriate synchronization primitives. While the dependence ordering between computations could be arbitrary in certain cases, such as accessing lock protected critical sections, in certain other instances the enforced order could be specific, e.g., to execute the producer before a consumer in a produce-consumer style application. In such cases, controlling the execution, without being aware of the dependence relationships, e.g., blocking the producer, can hinder forward progress, or worse, deadlock the execution.

Consider the popular Pthreads implementation of Pbzip2 [51]. It reads data blocks serially from an input file, one at a time, compresses the blocks in parallel, and writes the results to an output file. The computations are organized into threads, as shown in Figure 5.6(a). Block-read operations are grouped into one thread, e.g., $R0$ to $R3$ in thread $TH0$. Compress operations are grouped across multiple *compress* threads (to achieve concurrency), e.g., $C0$ to $C3$ in threads $TH1$ and $TH2$. Block-write operations are grouped into one thread, e.g., $W0$ and $W1$ in thread $TH3$. The threads are expected to be co-scheduled.

When the application executes, say three processors, $U0$ to $U2$, are allotted to it. Threads $TH0$, $TH1$ and $TH2$ execute in epoch $t0$, (Figure 5.6(b))[1]. $R0$ and $R1$ in $TH0$ read blocks which are compressed by $C0$ and $C1$, respectively. Now, say, at the start of epoch $t1$, $U0$ is taken away from the application to control the parallelism, and $TH0$ is blocked. $TH1$ and $TH2$ complete $C0$ and $C1$, and advance to execute $C2$ and $C3$. $C2$ and $C3$, in turn, wait for $R2$ and $R3$ in $TH0$ to read the blocks from the input file. However, if no more processors are available, $TH0$, which can make progress, remains blocked, stalling the program's progress, or even deadlocking it, while available resources are occupied by computations that cannot make progress, defeating the very purpose of controlling the execution. Although the example is from a Pthreads application, a task-based implementation with a similar structure would be equally vulnerable.

[1]For brevity, we do not show $TH3$ operations.

One possible solution to address this problem is to rewrite the application such that it does not enforce a specific order between computations. For example, we rewrote Pbzip2, using TBB, without resorting to producer-consumer style parallelism. The application is divided into three non-overlapping phases: read, compress and write. The sequential read phase is executed first, followed by a full barrier. Then the parallel compress phase is executed which is again followed by a full barrier. Finally, the sequential write phase is executed. While such an approach may work, it can potentially impact performance.

Another possible solution to address this problem is for the PM to expose a dependence-aware programming model, such as SMPSs [102], Gupta et al. [57], Prometheus [8] and DPJ [25]. These programming models require the programmers to statically identify data objects that a parallel computation will potentially access, using which they dynamically construct a dataflow graph between computations to uncovers large amounts of parallelism, *akin* to dataflow processor architectures. The PM could leverage this dependence construction facility to completely avoid forward progress issues. For example, the VG, in the PM, after mapping all computations exposed by the application on to vtasks, may admit only independent vtasks into the vtask pool; while suspending dependent vtasks, when identified, until their dependences are resolved. Consider the same Pbzip2 example in Figure 5.6. Figure 5.6(c) shows its computations formulated as vtasks, $R1$, $R2$, $C0 - C3$, and the dependences between them[2]. Figure 5.6(d) shows the dependence-aware execution in finer time steps. At time $t0$, $U0$, $U1$ and $U2$ are alloted to the application. $R0$ executes. Since no other independent computations exist, the vtask pool is empty. When $R0$ completes in $t1$, dependences of $R1$ and $C0$ have resolved, and hence they are added to the vtask pool. The RM assigns them to $U0$ and $U1$ in $t2$. In $t2$ the vtask pool is once again empty since $R2$ and $C1$ are suspended. $R2$ and $C1$ are added to the vtask pool once $R1$ completes in $t3$. In $t3$, say $U0$ is taken away. The RM assigns $R2$ and $C1$ to $U1$ and $U2$ in $t4$. Note that unlike in Figure 5.6(b), $C2$ does not get scheduled unless $R2$ completes. Thus, the PM ensures dependent tasks do not occupy hardware contexts. The application can make forward progress as long as it is allotted at least one worker.

[2]Dependences not germane to the discussion are not shown.

However, as mentioned in Chapter 3, dependence-aware programming models have two fundamental problems. First, they are not widely adopted. And, second, they do not support a wide variety of computational patterns. Therefore, to avoid starvation issues due to blocked signals, the PM takes the second approach of pro-actively pre-empting an executing computation in favor of executing a more productive computation. The approach is as follows. We observe that threads in producer-consumer style multithreaded applications typically communicate via conditional variables. Therefore, everytime a vtask invokes a call that corresponds to a conditional variable in the application, if there are more vtasks in the vtask pool waiting to be scheduled for execution, the RM suspends the current vtask, enqueues it at the tail of the vtask pool and schedules the oldest vtask in the vtask pool in the former's place. In this way, each communicating vtask gets a slice of the resource to execute, avoiding starvation and potential deadlock situations.

Currently, the PM can automatically ensure forward progress when applications use standard synchronization APIs exported by programming models such as Pthreads, TBB and Prometheus. If the application uses spin loops or home-grown synchronization primitives, the PM requires programmers to identify the call sites to the runtime. Automatically determining these primitives is a subject of future work.

5.4 Chapter Summary

This chapter presented the design, implementation and management of Vtasks, a primitive that the Parallelism Manager (PM) uses to regulate the execution of parallel computations in the application without any programmer involvement. It first presented the details of the vtask's context data structure. It then described how the PM schedules and manages vtasks. The chapter then identified different scenarios in which an application's forward progress can be affected and proposed solutions to tackle each scenario.

Chapter 6

Evaluation

This chapter presents an evaluation of the implementation of Varuna. The first part of this chapter will describe the machines, benchmarks, baselines and configurations used in our experiments. The second part of the chapter will present experimental results for the implementation of Varuna.

6.1 Methodology

To evaluate Varuna's efficacy we applied it to threaded and task-based applications, optimizing them for throughput, and resource consumption cost. We tested under three execution environments, isolated, multiprogrammed and asymmetric, on three stock multiprocessor machines with different microarchitectures. We report the total execution time, the energy consumed and the resource consumption cost for each application, along with the harmonic mean (HM) for the entire benchmark set, when optimizing for different objectives.

In the experiments, we sought to assess the following:

1. Varuna's overheads,
2. Benefits of applying vtasks to unmodified threaded and task applications,
3. Further benefits of applying adaptive optimization to them,
4. Effectiveness in highly dynamic operating conditions,
5. Agility in responding to changes, and
6. Effectiveness in asymmetric execution platforms.

6.1.1 Machines, Benchmarks and Baselines

	Opteron-8350	**Xeon E5-2420**	**Core i7-2600**
# Sockets	4	2	1
# Hardware Contexts	16	24	8
SMT	no	yes	yes
Clock Speed	2 GHz	1.9 GHz	3.4 GHz
Total Cache	16 MB	15 MB	8 MB
Memory	16GB	32 GB	16 GB
Linux Kernel	3.4.4	2.6.32	2.6.32
Memory Controllers/Channels	4/8	2/6	1/2

Table 6.1 Machine configurations used in experimentation

Benchmark	**Description**	**Characteristics**	**Input**
1	2	3	4
Barneshut [73]	N-Body simulation	Barriers	100000 bodies, 25 steps
Blackscholes [20]	Financial analysis	Data-parallel	10000000 options
Bzip2 [51]	Compression	Pipeline-parallel	1.3 GB file
Canneal [20]	Simulated annealing	Atomics & barriers	2,500,000 netlist elements
Dedup [20]	Compression	Pipeline-parallel	672 MB
Fluidanimate [20]	Interactive animation	Locks & barriers	500,000 particles, 500 frames
Hash Join[40]	In-memory DB join	Cache & memory intensive	28 MB tables
Histogram [108]	Image analysis	OS locks	1.4 GB bitmap file
RE [10]	Packet deduplication	Locks	9000000 packets
ReverseIndex [108]	HTML Analysis	Disk-intensive	1.3 GB directory
Stream[88]	Memory Streaming	Memory bandwidth intensive	915MB of data
Swaptions [20]	Financial analysis	Data-parallel	128 swaptions, 1000000 simulations
WordCount [108]	Text processing	Parallel-reduction	100 MB directory
X264 [20]	H.264 video encoding	Pipeline-parallel	1920X1080 pixels, 512 frames

Table 6.2 Multithreaded applications used in experimentation

Table 6.1 provides the details of the three machines used in the evaluation. To demonstrate Varuna's generality, we present results of select applications from different suites that exhibit different characteristics. Table 6.2 shows the list of multithreaded applications (column 1) we used along their characteristics (column 3). We used large input sizes for each application obtained from their respective suites (column 4). The baseline multithreaded versions use the fast NPTL Pthreads library (provided with the Linux kernels). To test Varuna with task-based applications, we applied Varuna to five TBB and Prometheus applications: Barneshut, Bzip2, Histogram, RE, and ReverseIndex.

Since TBB application do not always convey dependence information, the Vtask Generator in the Parallelism Manager simply passes all dynamically discovered tasks into the vtask pool. For reasons described in Section 5.3, we ensured that the TBB applications did not explicitly enforce a specific order between their tasks.

Prometheus exploits sequential program order and data objects accessed by tasks to automatically serialize dependent tasks while parallelizing the execution of independent tasks. It handles WAW dependences between tasks, but relies on the programmer to quiesce the parallel execution to prevent RAW and WAR hazards; the application may resume parallel execution once the hazards are avoided. This limits the degree of parallelism that the runtime can expose.

We enhanced the Prometheus execution model to also automatically handle RAW and WAR dependences, much like out-of-order superscalar processors. As the application executes, tasks stalled due to any dependence are skipped over in search of other dependence-free tasks, resources permitting. The Vtask Generator uses the dependence information to only expose independent tasks to the vtask pool.

We also compared Varuna to two recent adaptive proposals: Feedback Driven Threading (FDT) [116] and Parcae [107]. FDT and Parcae are adaptive approaches applicable only to dependence-unaware task-based applications. FDT can adapt to contention for locks and memory bandwidth. Parcae is more general, but optimizes for only one metric, throughput. It uses a hill climbing search method to adapt to dynamic changes. We faithfully implemented FDT mechanisms and a Parcae-like search heuristic in the TBB runtime. Note that neither Parcae nor FDT can be applied to threaded applications. Hence we compare them with Varuna only for task-based applications.

Config.	**Description**
1	2
PT_CG	Pthreads applications compiled with `-lpthread` with the default number of threads
PT_FG	Pthreads applications compiled with `-lpthread` executing with a higher number of threads (spawned by changing the command line argument)
TBB	TBB applications compiled with `-ltbb` with default number of tasks and threads
PM	Prometheus applications compiled with `-lprometheus` with default number of tasks and threads
Parcae [107]	State-of-the-art adaptive scheme implemented in TBB and Prometheus
FDT [116]	State-of-the-art adaptive scheme implemented in TBB and Prometheus
V_base	Pthreads applications compiled with `-lvaruna` with adaptation capability disabled (number of threads fixed to default value and never varied during runtime)
V_PT_T	Pthreads applications compiled with `-lvaruna` optimized for MAX(throughput)
V_PT_C	Pthreads applications compiled with `-lvaruna` optimized for MIN(consumption)
V_TBB_T	TBB applications compiled with `-lvaruna` optimized for MAX(throughput)
V_TBB_C	TBB applications compiled with `-lvaruna` optimized for MIN(consumption)
V_PM_T	Prometheus applications compiled with `-lvaruna` optimized for MAX(throughput)
V_PM_C	Prometheus applications compiled with `-lvaruna` optimized for MIN(consumption)

Table 6.3 Different configurations used in experiments.

6.1.2 Compilation Options

To operate with Varuna, the Pthreads, TBB and Prometheus baseline applications were simply *re-linked* with a `-lvaruna` flag, instead of `-lpthread`, `-ltbb`, and `-lprometheus`, respectively. We compiled all the applications (for Pthreads, TBB, Prometheus and Varuna) with $GCC\ 4.4.3$ using $-O3$ optimization and the architecture flag, $-march = native$. Varuna automatically detects the number of hardware contexts in a system and uses it as the default number of worker threads.

Benchmark	**PT_CG**	**PT_FG**	**TBB**	**PM**	**V_PT_***	**V_TBB_***	**V_PM_***
1	2	3	4	5	6	7	8
Barneshut [73]	16	100,000	100,000	100,000	100,000	100,000	100,000
Blackscholes [20]	16	10,000	-	-	10,000,000	-	-
Bzip2 [51]	18	1566	1566	1566	16	1566	1566
Canneal [20]	16	96	-	-	96	-	-
Dedup [20]	50	50	-	-	16	-	-
Fluidanimate [20]	16	384	-	-	384	-	-
Hash Join[40]	16	1024	-	-	1024	-	-
Histogram [108]	16	1024	1024	1024	1024	1024	1024
RE [10]	16	100,000	100,000	100,000	100,000	100,000	100,000
ReverseIndex [108]	16	78371	78371	78371	16	78371	78371
Stream[88]	16	1024	-	-	1024	-	-
Swaptions [20]	16	384	-	-	384	-	-
WordCount [108]	16	256	-	-	256	-	-
X264 [20]	18	512	-	-	16	-	-

Table 6.4 Thread, task and vtask count used for different benchmarks on the Opteron. * = PT/TBB/PM

6.1.3 Configurations

The results to follow in the next section show data for the configurations listed in Table 6.3. All data for a given experiment are normalized to PT_CG, which serves as the base case for comparison, and hence is not shown in the figures. Tables 6.4, 6.5, 6.6 list the thread, task and vtask count for PT_CG, PT_FG, TBB, PM and Varuna configurations, for the three platforms, Opteron, Xeon, and core i7, respectively. The higher thread and vtask values are chosen based on the input size and are spawned by giving a different parameter to the command line; the source code is left untouched. The task count for the TBB and Prometheus versions are same as PT_FG. For V_PT_T and V_PT_C versions of Dedup, Bzip2, ReverseIndex and X264, we did not spawn a higher number of vtasks as they are already written with enough periodic safe points and dynamic load balancing capabilities. To measure the instantaneous IPS needed to compute *qc(P)* in the scalability model,

Benchmark	**PT_CG**	**PT_FG**	**TBB**	**PM**	**V_PT_***	**V_TBB_***	**V_PM_***
1	2	3	4	5	6	7	8
Barneshut [73]	24	100,000	100,000	100,000	100,000	100,000	100,000
Blackscholes [20]	24	10,000	-	-	10,000,000	-	-
Bzip2 [51]	26	1566	1566	1566	24	1566	1566
Canneal [20]	24	96	-	-	96	-	-
Dedup [20]	74	74	-	-	24	-	-
Fluidanimate [20]	24	384	-	-	384	-	-
Hash Join[40]	24	1024	-	-	1024	-	-
Histogram [108]	24	1024	1024	1024	1024	1024	1024
RE [10]	24	100,000	100,000	100,000	100,000	100,000	100,000
ReverseIndex [108]	24	78371	78371	78371	24	78371	78371
Stream[88]	24	1024	-	-	1024	-	-
Swaptions [20]	24	384	-	-	384	-	-
WordCount [108]	24	256	-	-	256	-	-
X264 [20]	26	512	-	-	24	-	-

Table 6.5 Thread, task and vtask count used for different benchmarks on the Xeon. * = PT/TBB/PM

we used the PAPI library APIs [92]. Energy was measured using a Wattsup meter to which the experimental machines were connected.

6.1.4 Results Exposition

The exposition of the results is grouped along the lines of the execution environments, isolated, multiprogrammed and asymmetric. The first tests Varuna's basic capabilities (Section 6.2). The second stress tests Varuna in a range of highly dynamic, multiprogrammed operating conditions (Section 6.3). The third tests Varuna's efficacy when executed on cores with different capabilities (Section 6.4).

Benchmark	PT_CG	PT_FG	TBB	PM	V_PT_*	V_TBB_*	V_PM_*
1	2	3	4	5	6	7	8
Barneshut [73]	8	100,000	100,000	100,000	100,000	100,000	100,000
Blackscholes [20]	8	10,000	-	-	10,000,000	-	-
Bzip2 [51]	10	1566	1566	1566	8	1566	1566
Canneal [20]	8	96	-	-	96	-	-
Dedup [20]	26	26	-	-	8	-	-
Fluidanimate [20]	8	384	-	-	384	-	-
Hash Join[40]	8	1024	-	-	1024	-	-
Histogram [108]	8	1024	1024	1024	1024	1024	1024
RE [10]	8	100,000	100,000	100,000	100,000	100,000	100,000
ReverseIndex [108]	8	78371	78371	78371	8	78371	78371
Stream[88]	8	1024	-	-	1024	-	-
Swaptions [20]	8	384	-	-	384	-	-
WordCount [108]	8	256	-	-	256	-	-
X264 [20]	10	512	-	-	8	-	-

Table 6.6 Thread, task and vtask count used for different benchmarks on the Core i7. * = PT/TBB/PM

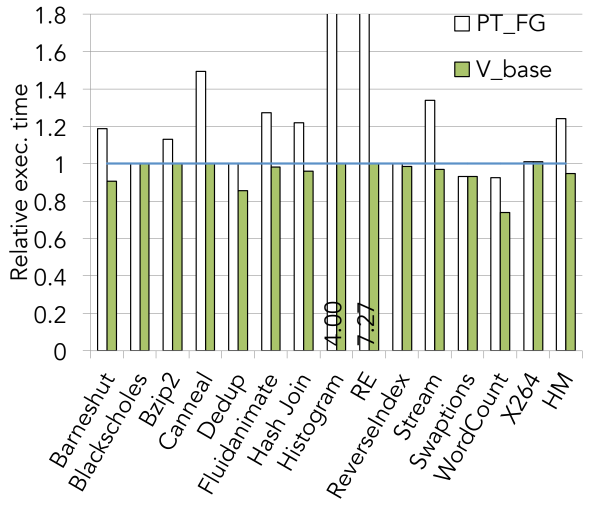
6.2 Isolated Execution Environment

An isolated environment is one in which each application is the only benchmark application running on our experimental platforms.

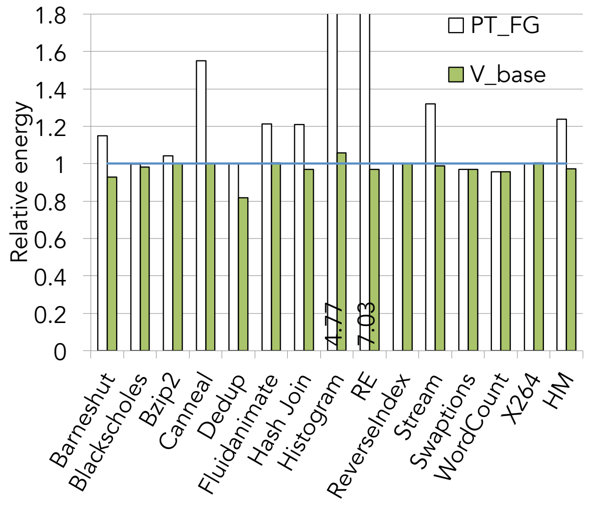
6.2.1 Overheads of Varuna

Result 1. *Varuna's vtask capability incurs negligible overheads (V_base).*

Figures 6.1, 6.2, and 6.3 show the results of multithreaded applications on the Xeon, the Opteron and the Core i7, respectively. V_base incurs no noticeable overheads as compared to PT_CG despite creating a large number of vtasks for several applications (Tables 6.4, 6.5, 6.6, column 6). This is because vtasks are userspace objects and have negligible creation and preemption overheads. For some of the applications (Barneshut, Dedup, Swaptions and Wordcount), V_base

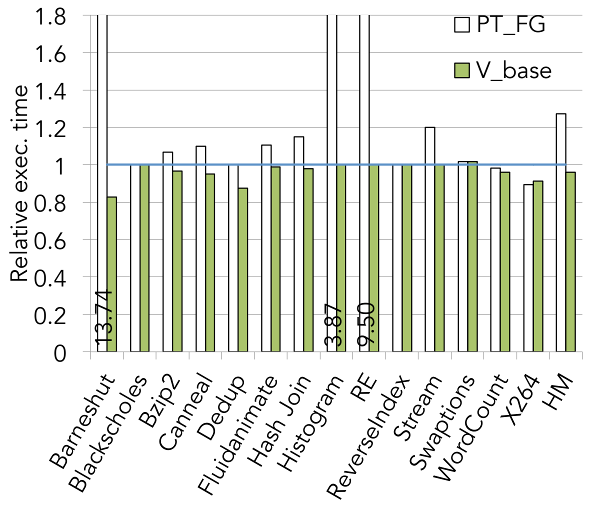


(a) Execution Time

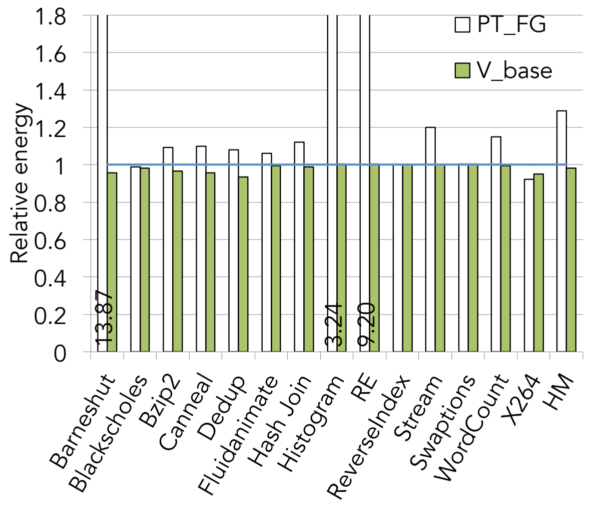


(b) Energy

Figure 6.1 Execution time and energy comparison of PT_CG, PT_FG, and V_base on the Xeon.



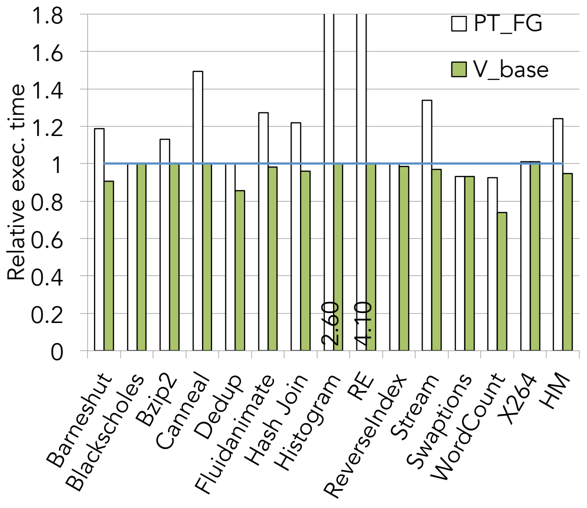
(a) Execution Time



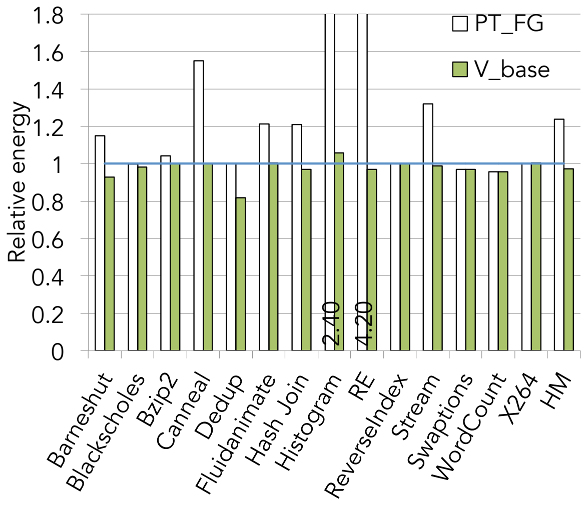
(b) Energy

Figure 6.2 Execution time and energy comparison of PT_CG, PT_FG, and V_base on the Opteron.

actually improved performance. This is because these applications exhibit irregular memory access patterns and V_base is able to improve their efficiency by applying: (i) lazy vtask creation (which avoids memory explosion), and (ii) fine-grained dynamic load balancing via randomized



(a) Execution Time



(b) Energy

Figure 6.3 Execution time and energy comparison of PT_CG, PT_FG, and V_base on the Core i7.

work-stealing. V_base reduces the execution time (Figure 6.1(a)) and energy consumption (Figure 6.1(b)) as compared to PT_CG on average (HM) by 6% and 3%, respectively, on the Xeon. On the Opteron, it reduces execution time (Figure 6.2(a)) and energy (Figure 6.2(b)) on average (HM) by 5% and 2%, respectively. On the Core i7, it reduces execution time (Figure 6.3(a)) and energy (Figure 6.3(b)) on average (HM) by 6% and 3%, respectively.

Result 2. *PT_FG does not benefit programs as does V_base.*

Although PT_FG creates fine-grained work like V_base, it degrades performance in most cases as compared to PT_CG as shown in Figures 6.1, 6.2, and 6.3. This is due to the high overheads involved in creating a large number of OS threads. Since each thread seeks OS resources, large number of threads can create contention, e.g., in Barneshut due to frequent barrier synchronization, in Histogram due to page table lock contention [28] and in RE due to memory exhaustion.

6.2.2 Optimizing $\mathrm{MAX}(\mathbf{throughput})$

Result 3. *Varuna further improves time and energy efficiency of multithreaded applications that exhibit contention to shared resources (V_PT_T) .*

Result 4. *Varuna is platform and resource-agnostic and can handle contention to hardware/software resources.*

Result 5. *Varuna does not degrade efficiency of non-contending applications.*

Our benchmarks exhibit five different contention cases as follows:

6.2.2.1 Contention in external software components

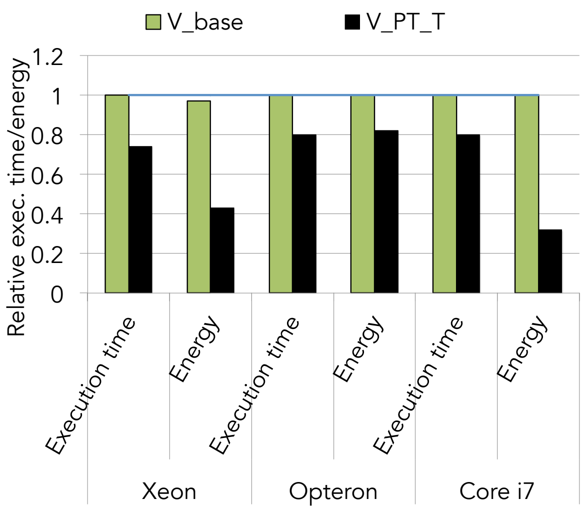


Figure 6.4 Execution time and energy comparison of Histogram on the Xeon, the Opteron, and the Core i7.

Histogram is a data parallel benchmark that carries out computations on a 1.4GB bitmap file. The benchmark invokes *mmap* which allocates memory in a region in the kernel space, marks the new region as allocated but defers updating the page table entries. Pages are loaded from the disk only when the application accesses them. When a new page is accessed, a page fault occurs and the kernel locks the entire region with a read lock. If multiple threads try to access different pages in parallel, concurrent page faults can occur, potentially creating contention for the lock [29]. The benchmark scales poorly and incurs degradation in both performance and energy for even a modest increase in the number of threads. Tables 6.7, 6.8, and 6.9 (column 2) show the optimum DoP that Varuna reaches for each of our multithreaded benchmarks on the Xeon and the

Opteron, respectively, for the MAX(throughput) objective. As it can be seen, Histogram does not scale beyond one thread on any of the experimental machine.

Histogram is an example of a very likely future scenario: a programmer can work hard to avoid contention, but has little or no control over external software components (e.g., libraries or the OS). By dynamically adapting the number of hardware threads, V_PT_T is able to reduce its execution time and energy by 26% and 57%, respectively, on the Xeon, 20% and 18%, respectively, on the Opteron, and 20% and 68%, respectively, on the Core i7, as compared to PT_CG (Figure 6.4). The energy savings are greater on the Xeon, and the Core i7 since with V_PT_T, some processors are idle and are put into deep sleep states, a feature that is not available in the Opteron.

	MAX(throughput)			MIN(consumption)		
	DoP	Sec.	Joules	DoP	Sec.	Joules
Barneshut	24 (24)	14.5 (14)	1579 (1560)	20 (20)	15 (14.6)	1704 (1676)
Bzip2	24 (24)	23 (21)	3094 (2854)	19 (19)	34 (34)	4123 (4100)
Canneal	24 (24)	89 (89)	8393 (8400)	5 (5)	100 (100)	9260 (9201)
Dedup	24 (24)	12 (12)	1274 (1267)	16 (16)	16 (16)	1300 (1297)
Fluidanimate	24 (24)	61 (61)	8622 (8602)	16 (16)	84 (84)	12003 (12003)
Histogram	1 (1)	11.2 (11)	740 (734)	1(1)	11.2 (11)	740 (734)
RE	12 (12)	29 (29)	7500 (7454)	1 (1)	50 (50)	13074 (12987)
ReverseIndex	16 (16)	68 (67)	5748 (5700)	1 (1)	117 (117)	9340 (9341)
Swaptions	24 (24)	80 (80)	11409 (11405)	20 (20)	89 (89)	11409 (11356)
WordCount	24 (24)	5 (5)	201 (200)	20 (20)	6.5 (6.5)	303 (301)
X264	24 (24)	90 (90)	11700 (11653)	20 (20)	98 (98)	13564 (13564)
Blackscholes	24 (24)	36 (36)	3592 (3567)	20 (20)	41 (41)	3865 (3865)
Stream	7 (7)	17 (17)	2434 (2400)	1 (1)	23 (23)	2987 (2986)
Hash Join	11 (11)	24 (24)	3454 (3400)	1 (1)	54 (53)	7694 (7611)

Table 6.7 Optimum degree of parallelism (DoP) chosen by Varuna, and resulting execution time (Sec.) and energy consumption (Joules) on the Xeon. The numbers shown in parantheses are the best possible DoP, execution time and energy consumption that could be obtained for each benchmark (determined by performing a full static thread sweep).

6.2.2.2 Contention due to memory bandwidth

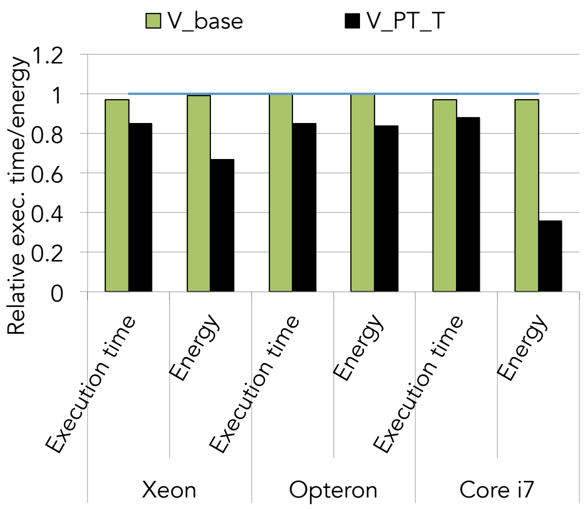


Figure 6.5 Execution time and energy comparison of Stream on the Xeon, the Opteron, and the Core i7.

Stream mainly moves data from one region of the memory to another. Again, there is ample application-level parallelism, however, there is a significant demand for the memory (bus) bandwidth. Core i7 has a single memory controller with two memory channels, Xeon has two memory controllers (one in each socket) with three channels each, and Opteron has four memory controller (one in each socket) with two channels each. Any parallelism beyond the total number of channels available on these platforms will result in contention leading to performance and energy degradation.

As it can be seen in Tables 6.7, 6.8 and 6.9 (column 2), the best parallelism point for Stream is different on different machines. On the Xeon, it scales up to 7 threads, on the Opteron it scales up to 8 threads, while on the Core i7 it does not scale beyond 2 threads.

Varuna is able to control the parallelism, alleviating the contention, thus preventing increase in memory latencies. This results in reducing the processor idle time, and thus more time- and energy-efficient execution. As shown in Figure 6.5, Varuna reduces the execution time and energy consumption by 12% and 64% on the Core i7, respectively, 15% and 23% on the Xeon, respectively, and 15% and 16% on the Opteron, respectively, as compared to PT_CG.

	MAX(throughput)			MIN(consumption)		
	DoP	Sec.	Joules	DoP	Sec.	Joules
Barneshut	16 (16)	20.2 (19.3)	8480 (8345)	8 (8)	29 (27.8)	10200 (9997)
Bzip2	16 (16)	28 (27.6)	13723 (13712)	11 (10)	33 (31.6)	13233 (12655)
Canneal	16 (16)	115 (114)	47442 (47434)	10 (10)	123 (122)	50331 (49976)
Dedup	16 (16)	14 (14)	5560 (5545)	8 (8)	16 (15.5)	5945 (5844)
Fluidanimate	16 (16)	88 (88)	44303 (44298)	10 (10)	115 (115)	55664 (55534)
Histogram	1 (1)	25 (25)	9156 (9156)	1 (1)	25 (25)	9156 (9156)
RE	8 (8)	37 (36)	65432 (65334)	1 (1)	55 (55)	97654 (97650)
ReverseIndex	3 (3)	53 (53)	20500 (20475)	1 (1)	100 (100)	38765 (37884)
Swaptions	16 (16)	125 (124)	66555 (66455)	11 (11)	168 (167)	87338 (86534)
WordCount	16 (16)	13 (13)	4303 (4300)	9 (9)	15 (15)	4500 (4467)
X264	16 (16)	95 (94)	49649 (49008)	9 (9)	104 (104)	53218 (53125)
Blackscholes	16 (16)	88 (88)	36337 (36465)	9 (9)	131 (131)	55968 (55874)
Stream	8 (8)	27 (27)	10453 (10356)	1 (1)	41 (40)	16882 (15987)
Hash Join	13 (13)	39 (39)	17655 (17434)	1 (1)	68 (66)	29765 (28994)

Table 6.8 Optimum degree of parallelism (DoP) chosen by Varuna, and resulting execution time (Sec.) and energy consumption (Joules) on the Opteron. The numbers shown in parantheses are the best possible DoP, execution time and energy consumption that could be obtained for each benchmark (determined by performing a full static thread sweep).

6.2.2.3 Contention due to disk bandwidth

Reverse Index is a benchmark whose parallel execution makes excessive demands for another resource, disk bandwidth. Each parallel computation in Reverse Index does the following: (i) open an HTML file, (ii) parse the file contents to identify links to other pages, (iii) extract the links and update a local data structure based upon some computation, and (iv) close the HTML file. In addition to significant disk activity, the benchmark performs CPU activity. Though there is

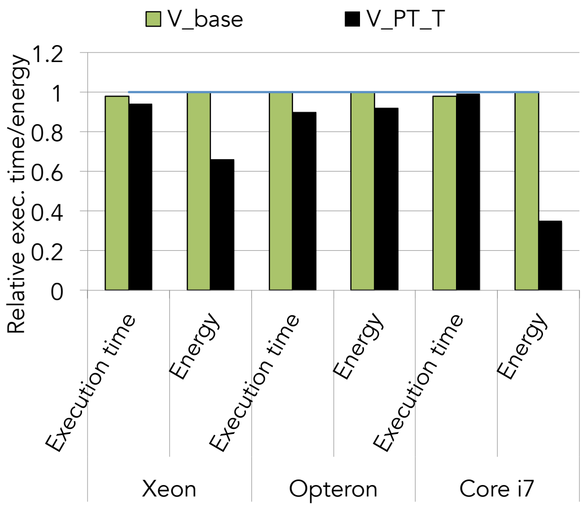


Figure 6.6 Execution time and energy comparison of ReverseIndex on the Xeon, the Opteron, and the Core i7.

abundant parallelism—each file can be processed independently—parallel accesses to the disk can be a source of contention.

As in the case of Stream, the best parallelism point for ReverseIndex is different on different machines. On the Xeon, it scales up to 16 threads (Table 6.7, column 2), on the Opteron it does not scale beyond 3 (Table 6.8, column 2), and on Core i7 it scales up to 4 threads (Table 6.9, column 2). As shown in Figure 6.6, V_PT_T reduces ReverseIndex's execution time and energy by 6% and 34%, respectively, on the Xeon, and 10% and 8%, respectively, on the Opteron, and 1% and 65%, respectively, on the Core i7, as compared to PT_CG. Reductions in execution time are modest because the performance of this benchmark does not degrade significantly for higher thread counts wrt the best DoP.

6.2.2.4 Contention due to cache capacity

In Hash Join there is potential for contention in a shared cache. The working set size of each thread is over 2MB. Core i7 has a shared L3 cache capacity of 8 MB, Xeon has L3 cache capacity of 15MB, and Opteron has L3 capacity of 16MB, and hence contention is likely if the working set size exceeds the respective capacities.

	MAX(throughput)			MIN(consumption)		
	DoP	Sec.	Joules	DoP	Sec.	Joules
Barneshut	8 (8)	26 (26)	1163 (1162)	8 (8)	26 (26)	1163 (1162)
Bzip2	8 (8)	36 (36)	1608 (1608)	8 (8)	36 (36)	1608 (1608)
Canneal	8 (8)	164(164)	15527 (15527)	4(4)	197 (195)	17432 (17389)
Dedup	8 (8)	22 (22)	2356 (2355)	8 (8)	22 (22)	2356 (2355)
Fluidanimate	8 (8)	112 (112)	15950 (15945)	8 (8)	112 (112)	15950 (15950)
Histogram	1 (1)	12 (12)	810 (810)	1(1)	12(12)	810 (810)
RE	5 (5)	45 (44)	11637 (11630)	1 (1)	68 (68)	16054 (16045)
ReverseIndex	4 (4)	60 (60)	4933 (4931)	1 (1)	106 (106)	9100 (9100)
Swaptions	8 (8)	148 (148)	14567 (14567)	8 (8)	148 (148)	14567 (14567)
WordCount	8 (8)	7 (7)	289 (289)	8 (8)	7 (7)	289 (289)
X264	8 (8)	126 (126)	14567 (14537)	8 (8)	126 (126)	14537 (14537)
Blackscholes	8 (8)	48 (48)	4987 (4987)	8 (8)	48 (48)	4987 (4987)
Stream	2 (2)	32 (32)	3789 (3677)	1 (1)	36 (36)	3899 (3898)
Hash Join	4 (4)	10 (10)	1632 (1627)	1 (1)	27 (27)	3544 (3542)

Table 6.9 Optimum degree of parallelism (DoP) chosen by Varuna, and resulting execution time (Sec.) and energy consumption (Joules) on the Core i7. The numbers shown in parantheses are the best possible DoP, execution time and energy consumption that could be obtained for each benchmark (determined by performing a full static thread sweep).

Varuna controls the parallelism to ensure that the working set does not exceed the available cache capacity, thereby reducing both energy consumption and execution time. Varuna reduces the execution time and energy consumption by 25% and 42% on the core i7, 25% and 46% on the Xeon, and 19% and 18% on the Opteron, respectively, as compared to PT_CG (Figure 6.7).

6.2.2.5 Contention due to user-level locks

RE has abundant packet-level parallelism but uses a lock protected shared hash table that each concurrent packet must access. Only one packet (thread) can update the hash table at a given time. As the number of concurrent packets increases, updates to the hash table increase, which increases the contention to the critical section.

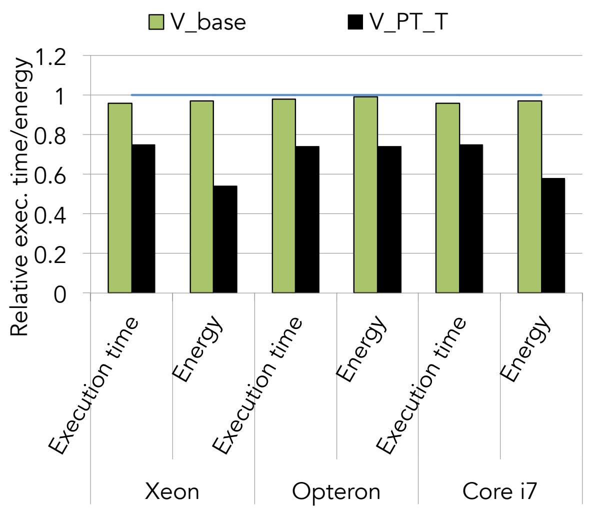


Figure 6.7 Execution time and energy comparison of Hash Join on the Xeon, the Opteron, and the Core i7.

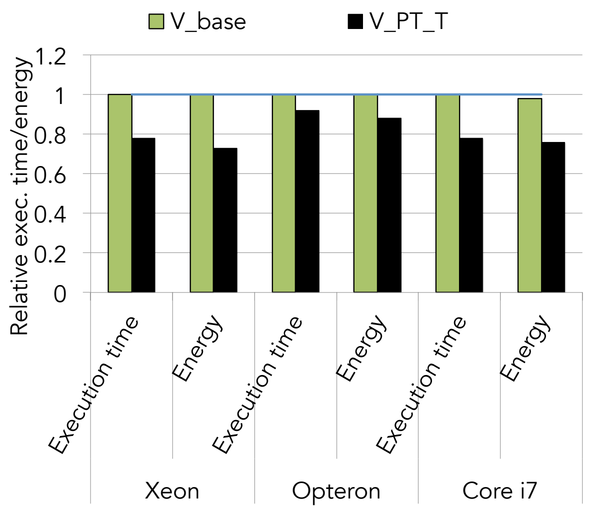
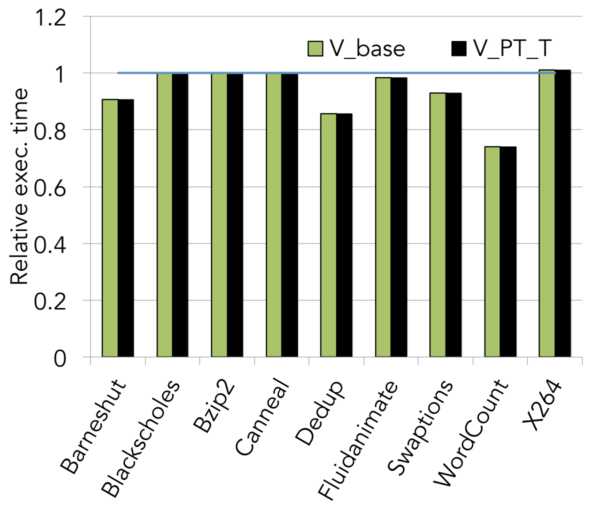


Figure 6.8 Execution time and energy comparison of RE on the Xeon, the Opteron, and the Core i7.

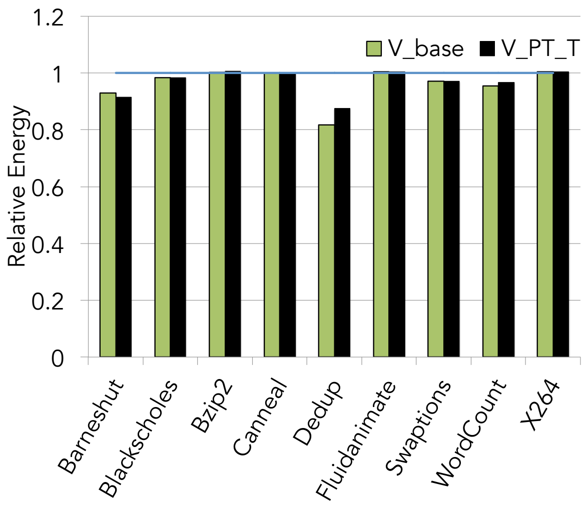
The application incurs degradation both in execution time and energy beyond a thread count of 8 on the Opteron (Table 6.8, column 2), 12 on the Xeon (Table 6.7, column 2), and 5 on Core i7 (Table 6.9, column 2). V_PT_T reduces the execution time and energy consumption by 22% and

27%, respectively, on the Xeon, by 8% and 12%, respectively, on the Opteron, and by 22% and 24%, respectively, on the Core i7, as compared to PT_CG (Figure 6.8).

6.2.2.6 No Contention Scenarios



(a) Execution Time



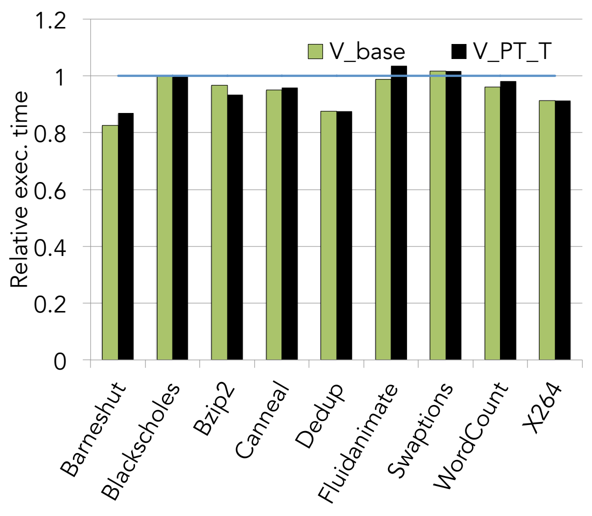
(b) Energy

Figure 6.9 Execution time and energy comparison of non-contending benchmarks on the Xeon.

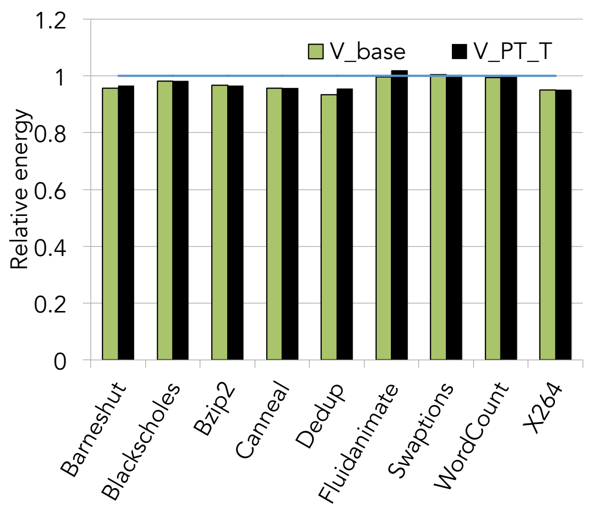
We consider nine benchmarks, Barneshut, Blackscholes, Bzip2, Canneal, Dedup, Fluidanimate, Swaptions, Wordcount, and X264, to illustrate no contention scenarios. As shown in Tables 6.7, 6.8 and 6.9 (column 2), all these benchmarks scaled up to the maximum number of hardware contexts. This is because these benchmarks have abundant parallelism and few contention concerns in common circumstances. Varuna neither incurs any degradation nor provides any savings across all the experimental platforms for these benchmarks, as compared to V_base (Figures 6.9, 6.10, 6.11).

6.2.2.7 Summary

V_PT_T reduces the execution time and energy consumption as compared to PT_CG on average (HM) by 12% and 22%, respectively, across all the applications on the Xeon. On the Opteron, it

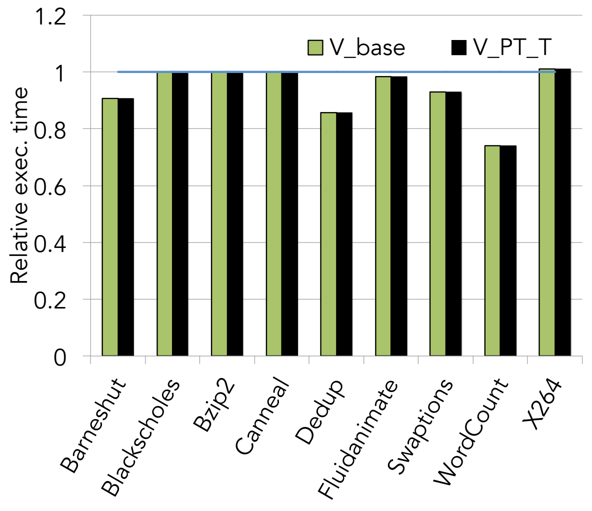


(a) Execution Time

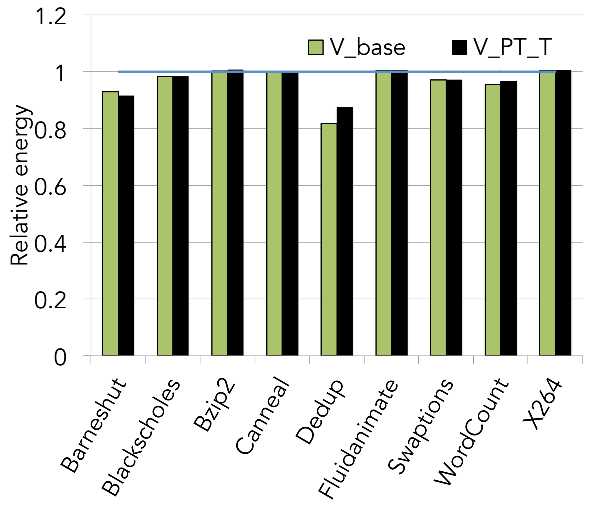


(b) Energy

Figure 6.10 Execution time and energy comparison of non-contending benchmarks on the Opteron.



(a) Execution Time



(b) Energy

Figure 6.11 Execution time and energy comparison of non-contending benchmarks on the Core i7.

reduces execution time and energy on average (HM) by 10% and 8%, respectively. On the Core i7, it reduces execution time and energy on average (HM) by 12% and 23%, respectively.

6.2.3 Handling Task-based Applications

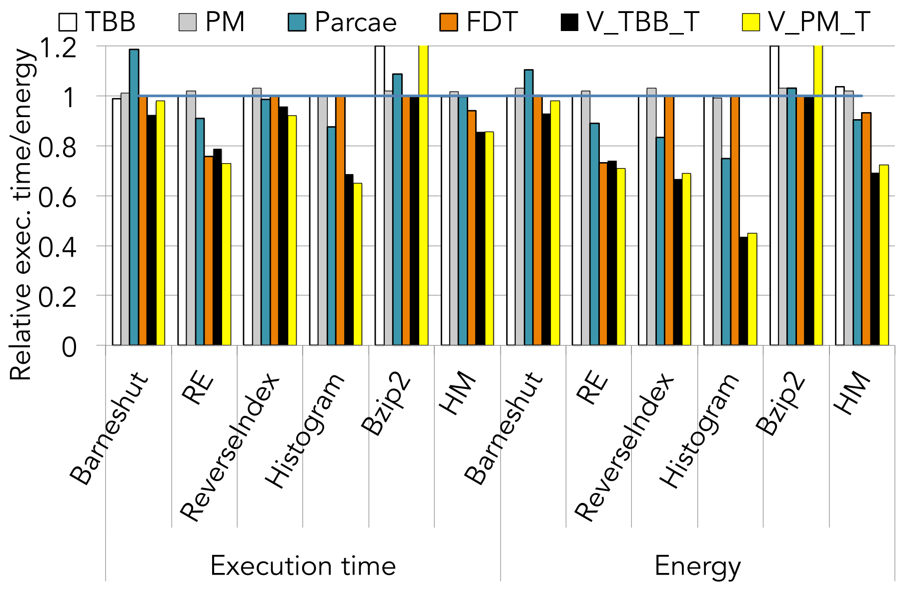


Figure 6.12 Execution time and energy comparison of task-based Varuna, TBB, Prometheus, Parcae and FDT on the Xeon.

Result 6. *Varuna is as effective for task-based applications as it is for threaded applications.*

Result 7. *Varuna outperforms state-of-the-art approaches that are applicable only for task-based applications.*

Figures 6.12, 6.13, and 6.14 show Varuna's (V_TBB_T and V_PM_T) time and energy efficiency for the $MAX(throughput)$ objective when applied to the unmodified TBB and Prometheus applications, on the Xeon, the Opteron, and the Core i7, respectively. It also compares the results with Parcae and FDT, two of the recent adaptive approaches. FDT employs a resource-specific mechanism to detect and avert contention. It can detect contention to locks (RE), however, it cannot detect contention to either the disk bandwidth (ReverseIndex) or the page table (Histogram). While Parcae improves performance and energy consumption of all contending applications (RE, ReverseIndex, Histogram), its slow hill climbing search-based approach degrades time and energy efficiency of non-contending applications (Barneshut and Bzip2) over the PT_CG baseline. Varuna, on the other hand, due to its holistic and quick adaptation, improves over PT_CG on an average

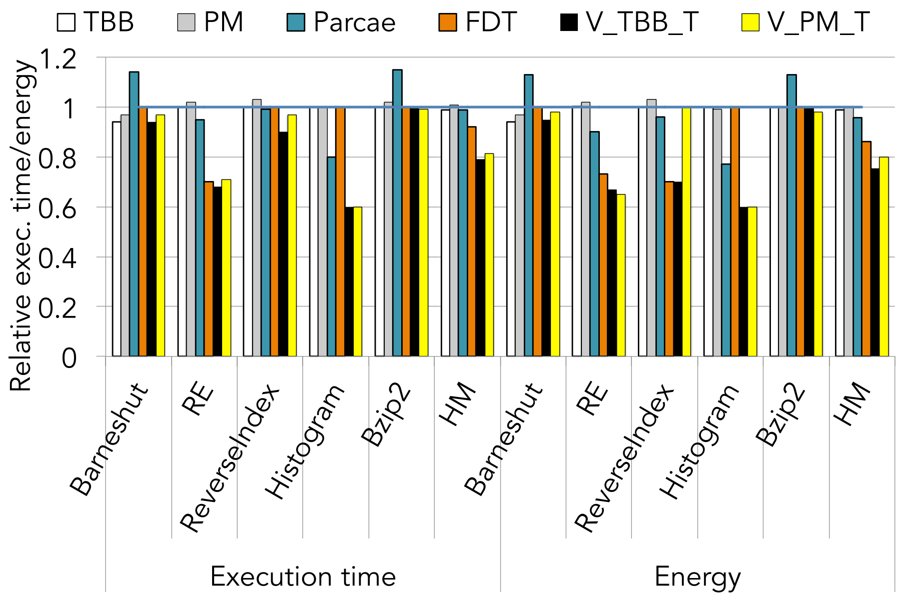


Figure 6.13 Execution time and energy comparison of task-based Varuna, TBB, Prometheus, Parcae and FDT on the Opteron.

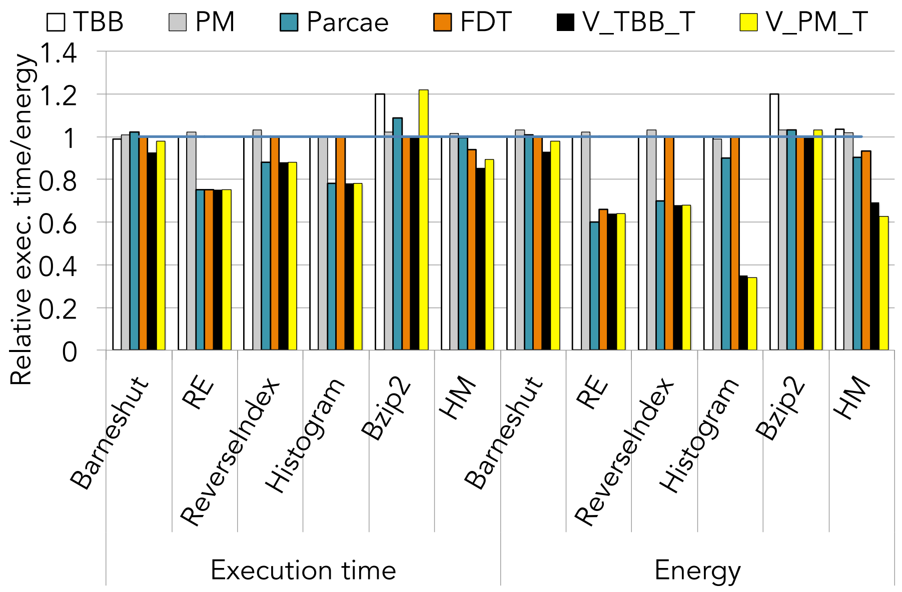


Figure 6.14 Execution time and energy comparison of task-based Varuna, TBB, Prometheus, Parcae and FDT on the Core i7.

(HM) by 15%, and outperforms FDT by 8% and Parcae by 14%, on the Xeon, for V_TBB_T. The average energy savings are even higher, 31% over PT_CG, 23% over FDT and 21% over Parcae, on the Xeon. On the Opteron, V_TBB_T reduces execution time and energy consumption by 21% and 25% over PT_CG, respectively, by 13% and 11% over FDT, respectively, and by 19% and 20% over Parcae, respectively. On the Core i7, it reduces time and energy by 15% and 31% over PT_CG, respectively, by 8% and 24% over FDT, respectively, and by 14% and 21% over Parcae, respectively.

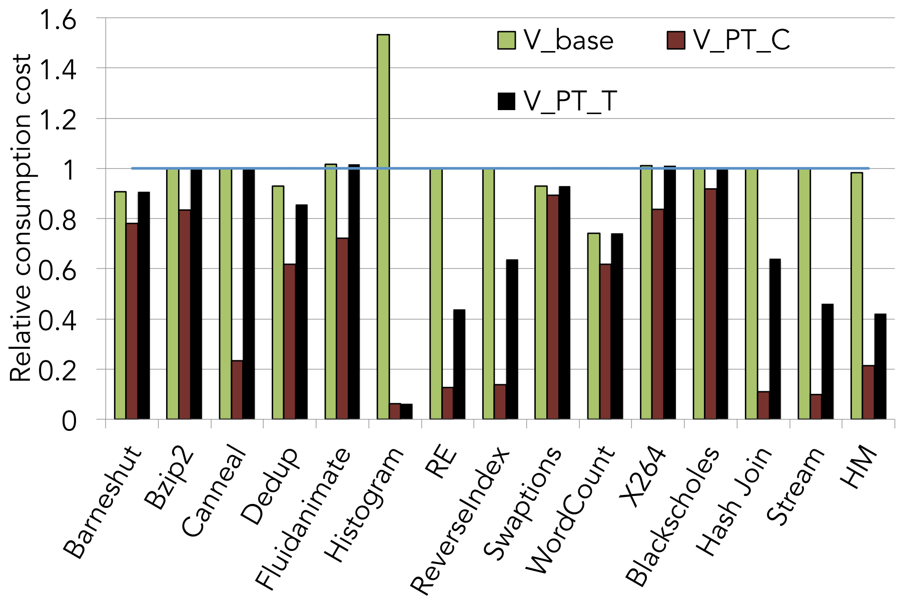
For V_PM_T, the average savings are similar to V_TBB_T. On the Xeon, it reduces the execution time and energy consumption by 15% and 28% over PT_CG, respectively, by 8% and 21% over FDT, respectively, and by 14% and 18% over Parcae, respectively. On the Opteron, it reduces time and energy by 19% and 20% over PT_CG, respectively, by 12% and 6% over FDT, respectively, and by 17% and 15% over Parcae, respectively. On the Core i7, it reduces time and energy by 11% and 38% over PT_CG, respectively, by 5% and 31% over FDT, respectively, and by 11% and 28% over Parcae, respectively.

6.2.4 Optimizing $\mathrm{MIN}(\mathrm{consumption})$

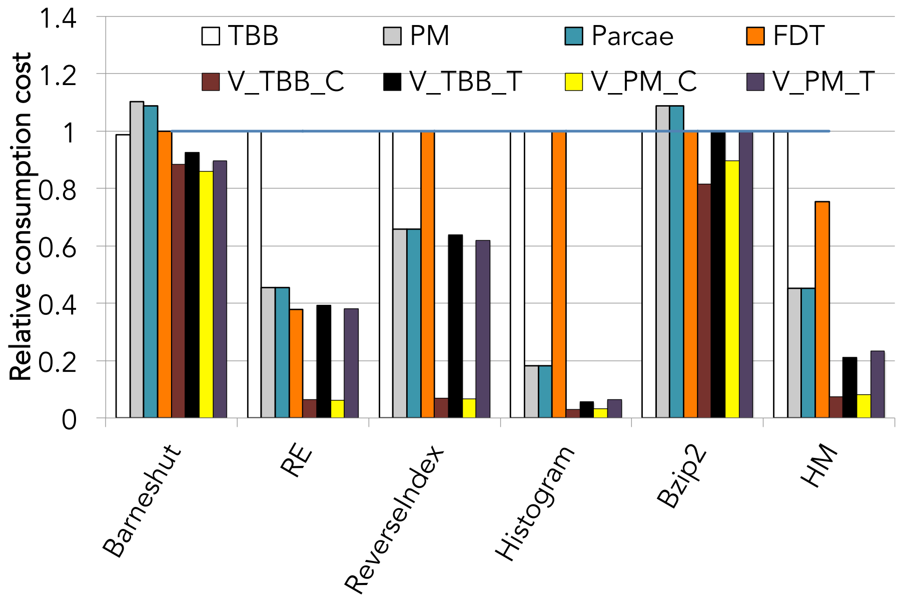
Result 8. *Varuna can better optimize for the resource consumption metric than the state-of-the-art approaches.*

Recall that resource consumption is the product of the average number of hardware threads used by the application and its total execution time. Figure 6.15 shows the resource consumption cost of both threaded and task-based application on the Xeon. We do not show the cost for other machines, as the trends are similar. On an average (HM), V_PT_C reduces the consumption cost by 79% for multithreaded applications over PT_CG and outperforms V_PT_T by 19% (Figure 6.15(a)), on the Xeon.

For task-based applications, V_TBB_C (V_PM_C), on an average, reduces the consumption cost by 93% (92%), respectively, over PT_CG, and outperforms V_TBB_T (V_PM_T) by 14% (15%), respectively (Figure 6.15(b)), on the Xeon.



(a) Multithreaded applications



(b) Task-based applications

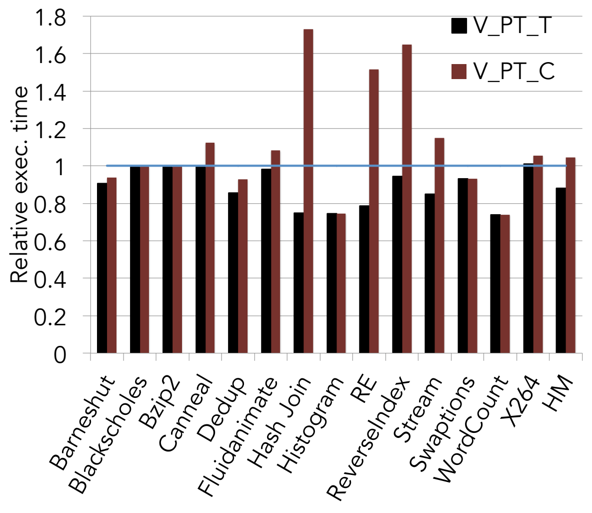
Figure 6.15 Resource consumption cost on the Xeon.

FDT and Parcae are unable to optimize for this objective. Parcae, similar to V_TBB_T (V_PM_T), applies its $MAX(throughput)$ adaptation, which incidentally also improves resource consumption cost, but only to some extent. V_TBB_C (V_PM_C) outperforms Parcae by 38% (37%) and FDT by 68% (69%), on the Xeon (Figure 6.15(b)).

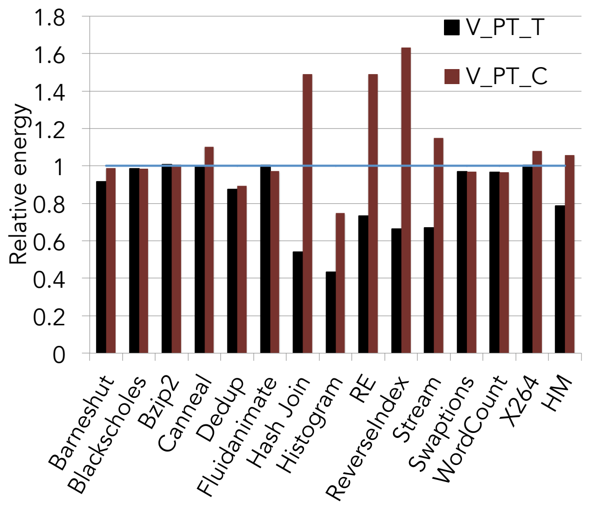
Figures 6.16 and 6.17 show the time and energy efficiency when Varuna (V_PT_C, V_TBB_C and V_PM_C) optimizes for MIN(consumption) metric, on the Xeon. When compared to V_PT_T (V_TBB_T/V_PM_T), V_PT_C (V_TBB_C/V_PM_C) degrades the execution time and energy consumption of several applications. This is because the MIN(consumption) metrics essentially permits use of a resource only if it is effectively utilized. For example, it picked an optimum degree of parallelism of 10 for Barneshut on the Opteron because the application speeds up linearly up to 10 threads, beyond which the gains are only sub-linear (Figure 4.2(b)). Even a trivial increase in parallelism of Histogram, RE and ReverseIndex increases the contention to resources. When applied for this metric, Varuna throttles back their DoP to 1 (Table 6.8).

For multithreaded applications, on an average, V_PT_C degrades the execution time and energy consumption by 12% and 22%, respectively, as compared to V_PT_T, on the Xeon. However, the average degradation is significantly lower than PT_CG. For example, on the Xeon, the degradation is only 0.04% in execution time and 0.05% in energy.

For task-based applications, V_TBB_C (V_PM_C) incurs 9% (10%) degradation in execution time over PT_CG, on the Xeon. However, it saves the average energy consumption by 12% (7%).



(a) Execution Time



(b) Energy

Figure 6.16 Execution time and energy comparison of V_PT_T and V_PT_C on the Xeon.

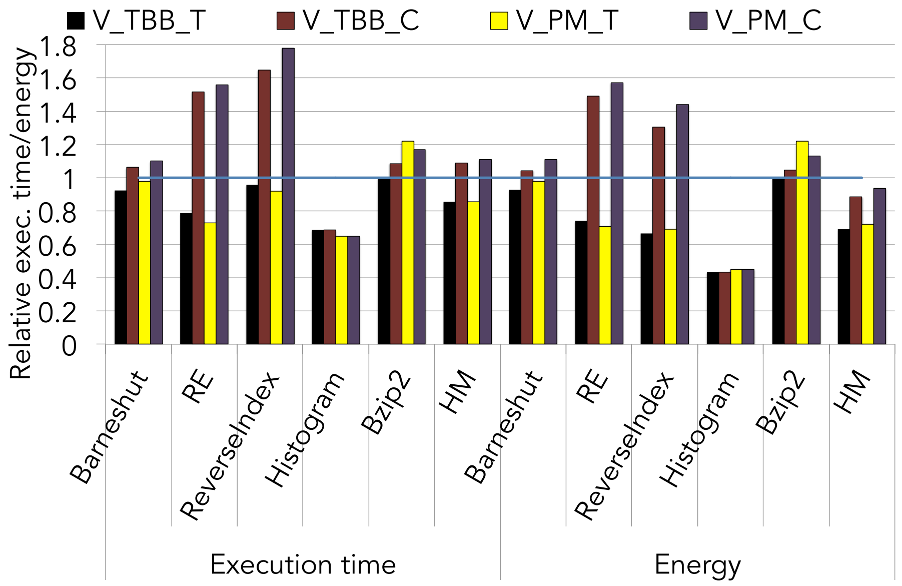


Figure 6.17 Execution time and energy comparison of V_TBB_T, V_TBB_C, V_PM_T, and V_PM_C on the Xeon.

6.2.5 Parallelism Determination Accuracy

Result 9. *Varuna always finds the best DoP regardless of the metric and platform under consideration.*

When adaptive optimization is applied, Varuna determines the DoP as per the $MAX(throughput)$ and $MIN(consumption)$ objectives. Tables 6.7, 6.8 and 6.9 show the optimum DoP that Varuna reaches for each of our multithreaded benchmarks on the Xeon and the Opteron, respectively, for the two objectives, MAX(throughput) and MIN(consumption), respectively, along with their execution time and energy consumed. The numbers shown in parantheses are the best possible DoP, execution time, and energy consumption that could be achieved for each of our benchmarks on the given experimental platform. We obtained these numbers by running the PT_CG versions of the benchmarks with varying degrees of parallelism, and statically selecting the configuration that yielded the best execution time and energy efficiency. As it can be seen, Varuna always arrives at the best DoP for all the benchmarks across all the machines. Further, it incurs less than 1% overhead in execution time and energy efficiency, as compared to the best PT_CG configuration.

6.2.6 Selecting Appropriate Monitoring Time Interval

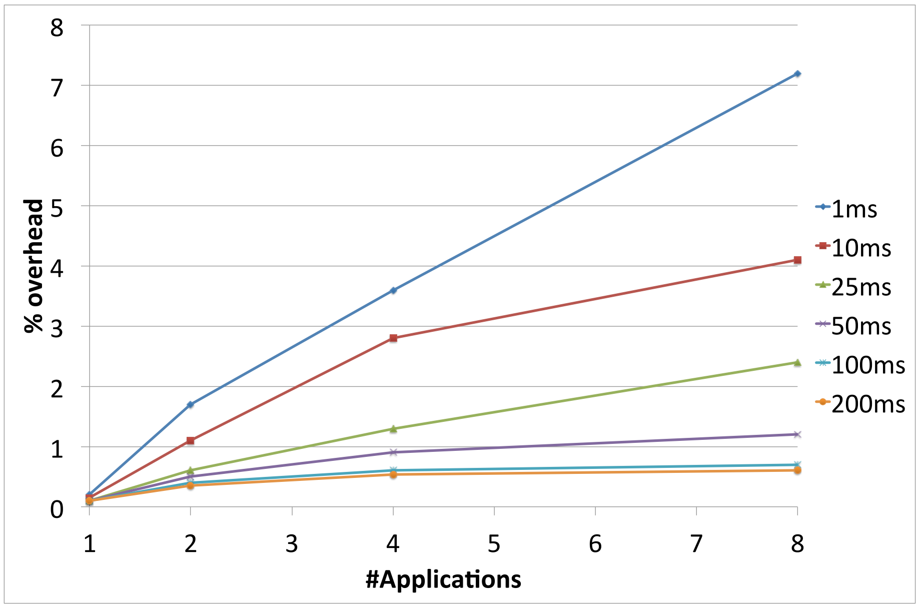


Figure 6.18 Time interval versus overheads on the Xeon.

As mentioned in Chapter 4, Varuna collects processor, memory, I/O and energy usage characteristics of the target applications to respond to changes in the execution environment. However, using an appropriate time-interval for collecting usage characteristics data is very important as it has significant impact on the overhead of Varuna. Though a very small time-interval permits the collection of very fine-grain details of the resource usage data, it increases the total system overhead. Therefore selecting appropriate time-interval is very important. For this, as shown in Figure 6.18, we evaluated Varuna with different time-intervals for monitoring several multithreaded applications simultaneously running on the Xeon. As Figure 6.18 shows, when Varuna is used with intervals less than $100ms$, the total system overhead is considerably high. This is because the number of registers offered by the hardware to monitor these events is limited [1, 87] and must be temporally shared across multiple applications. For small time intervals, as the number of applications increases, the rate at which these registers must be saved and restored to accommodate multiple applications also increases, and consequently leads to high system overheads. With

$100ms$ and greater time-intervals, the overheads of Varuna are negligible, less than 1%. Therefore, Varuna uses $100ms$ as the monitoring interval for detecting changes in the execution environment.

6.3 Multiprogrammed Environment

To evaluate Varuna in multiprogrammed environments, we consider three scenarios: (i) the first introduces a high degree of variability in resource capabilities, (ii) the second creates a highly multithreaded, oversubscribed environment with high context switch rates, and (iii) the third creates an environment with benchmarks with different resource demands. Experiments show that:

Result 10. *Varuna continuously assesses and adapts parallelism to dynamically changing conditions.*

Result 11. *Varuna responds much faster to changing conditions than the state-of-the-art approaches and consequently performs better for both the objectives.*

Result 12. *Varuna improves performance in the presence of both adaptive and non-adaptive mix of co-scheduled applications.*

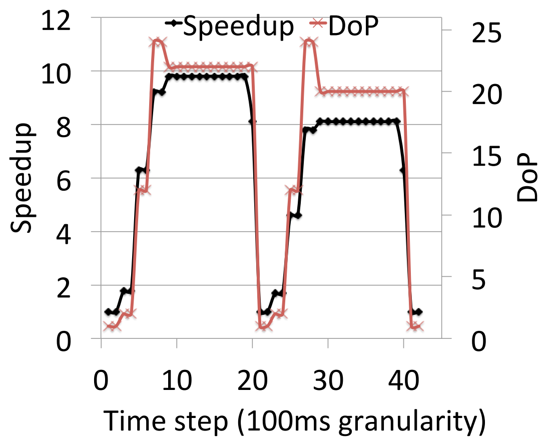
PT_FG results for these experiments are not shown since they were poor (as was also the case in Section 6.2). Prometheus results are also not shown for these experiments as their results are similar to TBB.

6.3.1 Adapting to Variabilities in Resource Capabilities

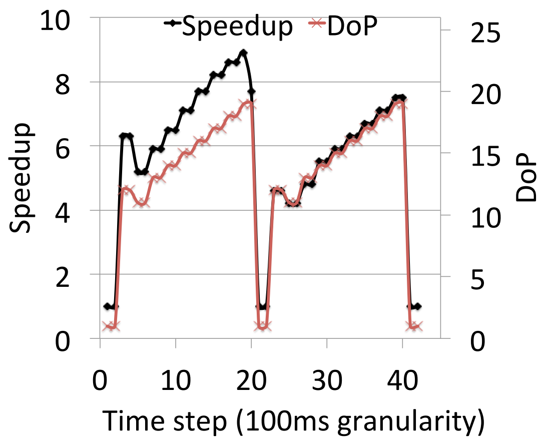
In this scenario, we co-scheduled the benchmark applications with variable instances of a highly cache- and memory-intensive program from the SPEC2006 suite, *mcf*, on the Xeon. Specifically, we launched one instance of *mcf* with our application and then added up to seven more mcf instances, one at a time, at $2s$ granularity. We then reduced the instances, by killing them one at a time, also at $2s$ granularity, until the count reached one, and repeated the above process until our application completes.

Figure 6.19(a) shows Varuna adapting Barneshut's DoP to optimize for the $MAX(throughput)$ objective in response to the demands placed by the varying number of mcf instances. The X-axis shows time incremented in $100ms$. There are two vertical axes: the primary shows instantaneous speedup and the secondary shows instantaneous DoP. From *t=8* to *t=20*, Barneshut executes with *DoP=22*. At this point, there is only one co-scheduled instance of mcf. A change in speedup at *t=20* indicates that the resource capability has changed due to the launch of a new mcf instance. At *t=21*, Varuna reacts to the change by breaking out of the passive monitoring loop and restarts the search to assess the new optimum DoP. It computes the speedups at DoP=2, DoP=12, and DoP=24 (Xeon has 24 contexts) to compute $\frac{dqc(P)}{dp}$ and the new DoP. At *t=28*, Varuna determines and establishes the new DoP to 20, and enters the passive monitoring mode until it detects another change, e.g., at *t=40*. Thus, Varuna continuously alters the parallelism to best suit the dynamic variations in the execution environment.

Note that in Figure 6.19(a), there is no single best operating point for Barneshut unlike in the isolated environment. FDT cannot handle this scenario as it assumes static operating conditions and does not have the ability to continuously adapt. It identifies the optimum DoP, typically once at the beginning of the application or at the inception of every user-defined phase, and fixes that value for the rest of the program/phase.



(a) Varuna



(b) Iterative search

Figure 6.19 Comparison of search heuristics

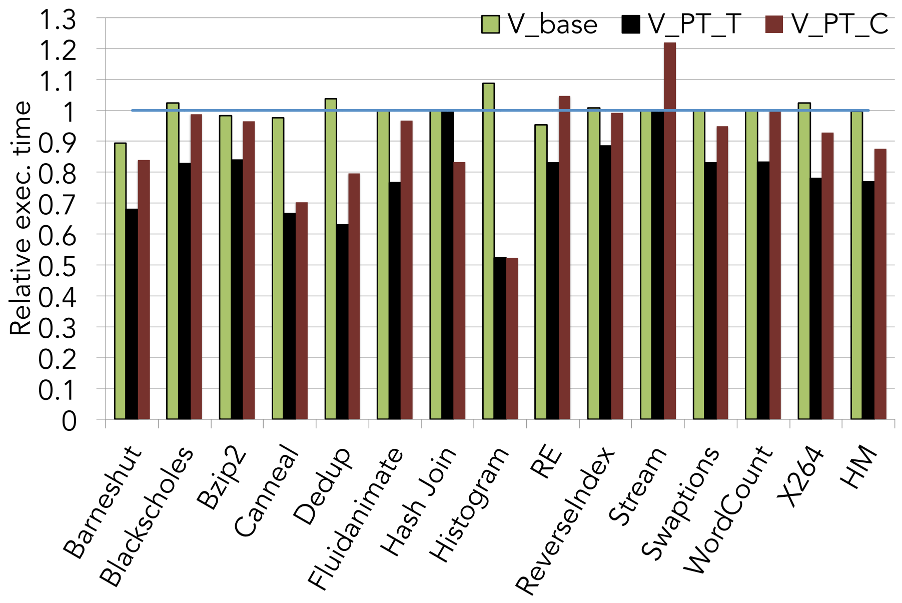
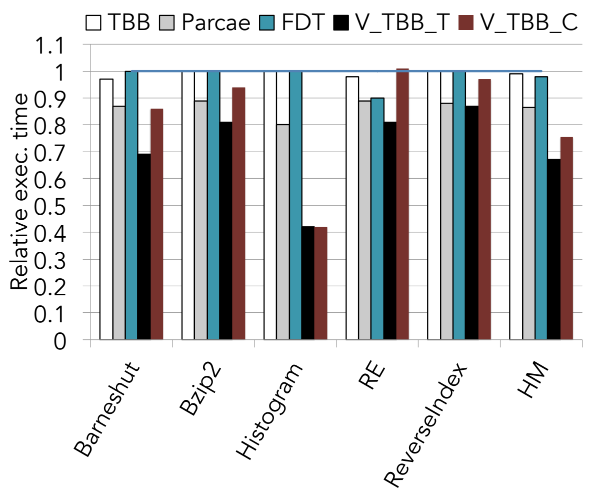
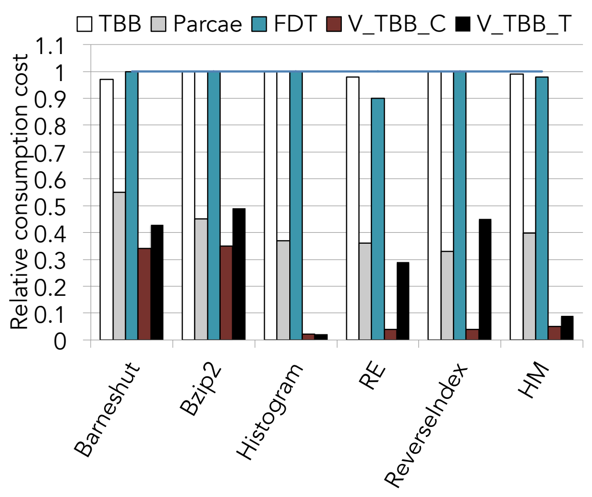


Figure 6.20 Execution time of threaded applications when scheduled with dynamically varying instances of *mcf* on the Xeon, relative to PT_CG running in the same environment.



(a) Execution time



(b) Consumption cost

Figure 6.21 Execution time and resource consumption cost of task-based applications when scheduled with dynamically varying instances of *mcf* on the Xeon.

	Execution time (s)	
	Isolated	Multiprogrammed
Barneshut	14.5	75.4
Bzip2	23	56.2
Canneal	89	165
Dedup	12	55.1
Fluidanimate	61	94.5
Histogram	11	23
RE	29	65
ReverseIndex	68	130.1
Swaptions	80	135
WordCount	5	10
X264	90	127
Blackscholes	36	87
Stream	17	35
Hash Join	24	51

Table 6.10 Execution time comparison of PT_CG in isolated and multiprogrammed environment on the Xeon.

Figure 6.19(b) shows the adaptation using a Parcae-like search for the same scenario. It begins to determine the optimum DoP at *t=5*, like Varuna, but since it searches for the optimum by iteratively trying different DoPs, it is unable to find the optimum immediately. In this case, at *t=20* it is still searching, when the operating conditions change (due to the new mcf instance), causing it to restart the search to adapt to the new conditions. As the figure shows, an iterative search strategy can take longer to adapt to the conditions, and if the conditions change rapidly, they may be far less effective than Varuna.

Table 6.10 presents the execution times achieved by the PT_CG applications in this environment against the isolated environment, on the Xeon. As it can be seen, all the PT_CG applications in this environment incur significant degradation (average of 1.93x), as compared to the isolated environment. This is because of the additional contention caused by the co-scheduled *mcf* instances not only to the shared resources, but also to processing cores and private caches.

Figures 6.20 and 6.21(a) present the execution times achieved for threaded and task-based applications, respectively, by Varuna relative to PT_CG for this multiprogrammed scenario (Table 6.10). Energy savings are not presented since their trends looked similar to the corresponding execution times. V_PT_T (V_TBB_T) reduces the execution time of threaded (task) applications on an average by 26% (33%) as compared to PT_CG. As compared to PT_CG in the isolated environment, V_PT_T limits the average degradation to 1.47x (Table 6.10). V_TBB_T outperforms FDT and Parcae by 30% and 20%, respectively. Unlike in the isolated environment, V_PT_C (V_TBB_C) reduces the average execution time by 14% (25%) over PT_CG and is only 12% (12%) slower than V_PT_T (V_TBB_T). This is because, in this environment, an application receives fewer resources, due to sharing of resources with other applications, and hence the optimum DoPs computed by these metrics are not far apart.

Stream, in this environment, shows particularly interesting results as compared to the results shown in Figure 6.5 for the same platform, Xeon. Unlike in the isolated environment, in which it scales poorly due to memory bandwidth contention, Stream scaled up to the maximum number of hardware contexts in this environment, similar to the one shown in Figure 2.1(b). This is because the contention created by *mcf* instances to cores and caches slowed down Stream's demand for resources, thus avoiding the memory contention in the first place.

Figures 6.22 and 6.21(b) present the resource consumption cost achieved by Varuna for threaded and task-based applications, respectively, relative to PT_CG. V_PT_C (V_TBB_C) reduces the average (HM) consumption cost for threaded (task) applications by 90% (95%) over PT_CG. However, it outperforms V_PT_T (V_TBB_T) only by 6% (5%) since their optimum DoPs are similar. As in the isolated environment, Parcae's $MAX(throughput)$ adaptation incidentally improved the resource consumption cost by 60% over PT_CG. However, Varuna (V_TBB_C) outperforms FDT and Parcae by 92% and 35%, respectively.

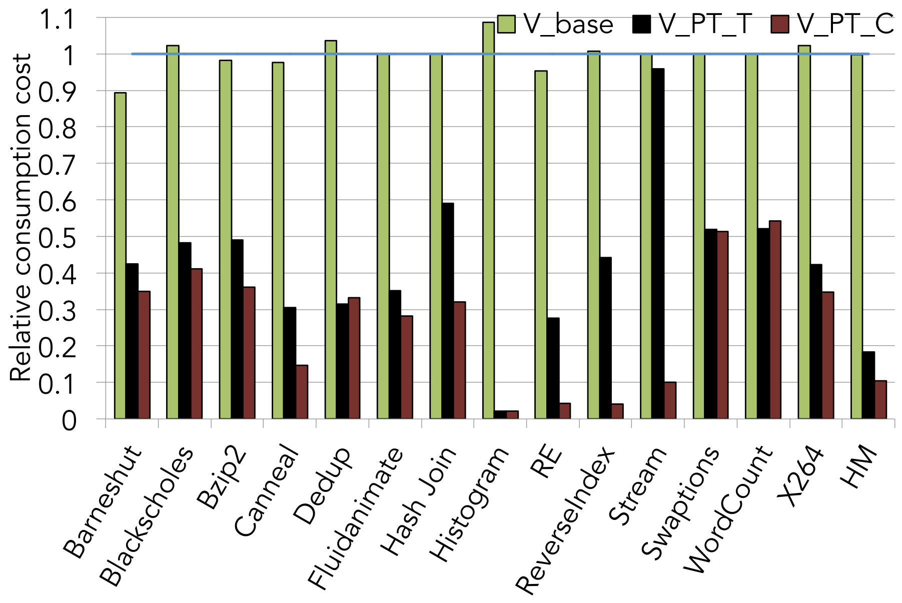


Figure 6.22 Resource consumption cost of threaded applications when scheduled with dynamically varying instances of *mcf* on the Xeon.

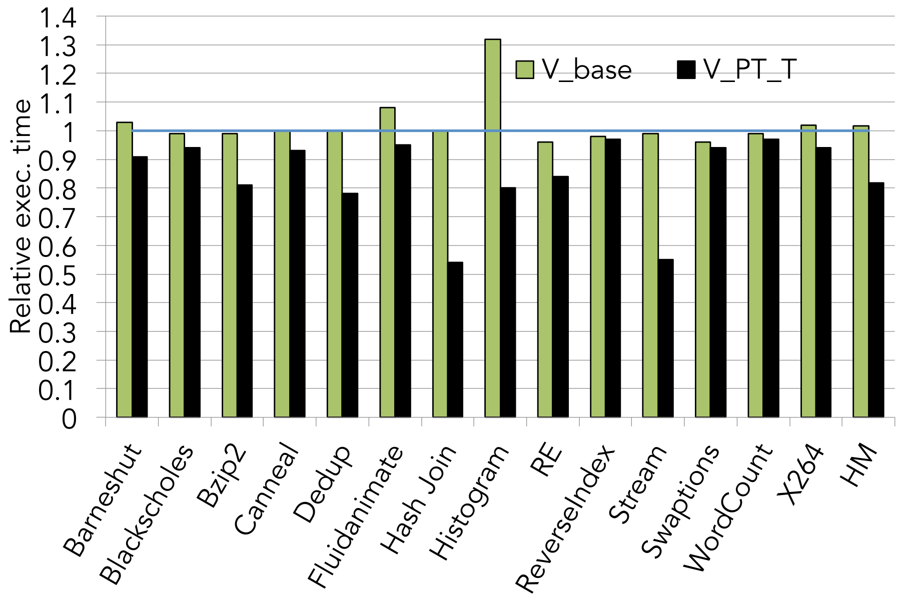


Figure 6.23 Execution time when 8 instances of the same thread application are scheduled together on the Opteron.

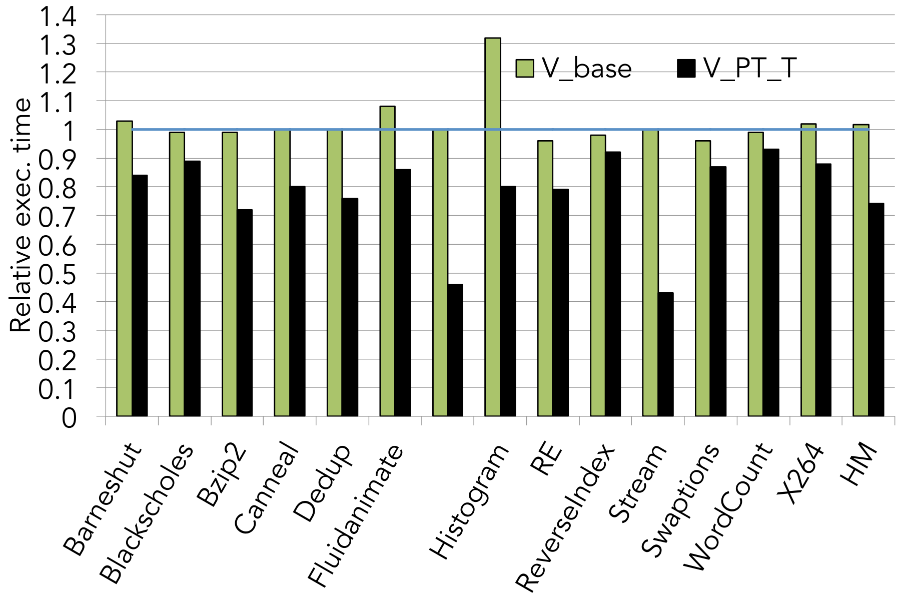


Figure 6.24 Execution time when 12 instances of the same thread application are scheduled together on the Opteron.

6.3.2 Adapting to Contention Due to Excessive Threads

In this scenario, we successively launched 8 instances of our benchmarks, with the same input, on the Opteron. Each instance creates 16 threads (maximum number of cores on that machine), A total of 128 threads execute simultaneously in the platform, effectively oversubscribing the system. An oversubscribed system increases the context switch rate of threads (due to reduction in the allotted time quanta per thread) which can lead to erratic program behavior. For example, it may destroy the cache locality of a given thread if the thread cannot be scheduled on the same core on which it last ran, potentially degrading its performance. It may also increase the contention to lock variables and create starvation in producer-consumer style applications (Bzip2, Dedup and X264).

Figure 6.23 presents the execution time achieved by Varuna for the threaded applications, when optimized for $MAX(throughput)$ objective, relative to PT_CG. Unlike PT_CG, which tries to allocate resources to all 128 threads at the same time, V_PT_T reduces the number of threads employed for each instance individually, thereby avoiding unnecessary context switches and hence its negative impact. It reduces the execution time on an average by 11% over PT_CG.

Unlike the previous scenarios, V_base degrades the execution time for some of the applications as compared to PT_CG. This is due to the interference caused by excessive context switching to its runtime data structures.

Figure 6.24 presents the execution time results of V_base and V_PT_T when the simultaneous instances are increased from 8 to 12. The savings in execution time increases on an average (HM) by 7% when compared to execution of V_PT_T with 8 instances.

6.3.3 Benchmarks with Different Resource Demands

#	Benchmarks		PT_CG		V_PT_T		DoP	
	B1	B2	T(B1)	T(B2)	T(B1)	T(B2)	B1	B2
1	Reverse Index	Barneshut	102	63	78	24	9	15
2	Hash Join	Barneshut	97	68	38	21	11	13
3	Hash Join	Hash Join	101	103	25	28	13	11
4	Hash Join	Stream	46	23	37	23	10	14
5	Barneshut	Bzip2	25	36	23	38	12	12

Table 6.11 Execution time comparison of Pthreads and Varuna (optimized for $MAX(throughput)$ objective) on the Xeon when benchmarks with different resource constraints are co-scheduled. T(B*): B*'s execution time in seconds; DoP: stable parallelism points for B1 and B2.

In the next scenario, we co-schedule selected pairs of our benchmarks on the Xeon. We compare the execution time of Varuna with PT_CG. Table 6.11 presents the results. The first main column shows the pairs of co-scheduled benchmarks. The next two main columns, one for each runtime, show the execution time of each benchmark. The last main column gives the stable optimum degree of parallelism (DoP) that Varuna reaches for each benchmark.

Pthreads create as many threads as hardware contexts (24) to execute the benchmarks. As Table 6.7 shows, 24 contexts are already too many for most of them. Depending on their type, simultaneously executing Pthreads benchmarks can make things worse. Note that Varuna ensures that only as many threads as the hardware contexts are created (Table 6.11, DoP column). We describe the other results using three types of benchmark pairs.

6.3.3.1 Contention-prone and contention-free

In this case, we co-schedule a benchmark prone to contention in the isolated environment, e.g., Reverse Index and Hash Join, with one that is not, e.g., Barneshut. Row #1 and row #2 from Table 6.11 are examples of this case.

As shown in Table 6.7, Reverse Index and Hash Join do not scale beyond 16 and 11 threads, respectively, when executed in isolation. When their Pthreads versions are co-scheduled with Barneshut, the contention created by Barneshut to cores and private caches further degrade their performance by a factor of 1.5x and 4x, respectively, as compared to the isolated environment (Table 6.7).

By using only the required number of contexts and returning the rest to the OS, Varuna versions of Reverse Index and Hash Join reduce contention not only to shared resources (disk and shared cache), but also to cores and private caches, reducing their execution times by 23% and 60% as compared to its Pthreads counterpart. When compared to their isolated versions (Table 6.7), Varuna limits the degradation in execution times of Reverse Index and Hash Join by 1.14x and 1.5x, respectively, as compared to 1.5x and 4x degradation incurred by PT_CG.

Varuna is also able to reduce the execution time of Barneshut by a factor of 3 when compared to PT_CG. Reducing the number of contexts to execute both Reverse Index and Hash Join allows Barneshut to take advantage of the free resources available to execute its computations in an uninterrupted fashion, thereby improving its performance.

6.3.3.2 Contention-prone and contention-prone

In the second case, both co-located benchmarks are prone to contention in the isolated environment, e.g., Hash Join and Stream (row #3 and row #4 in Table 6.11).

When two instances of Hash Join (Pthreads versions) are co-scheduled (Table 6.11, row #3), they create more contention to the shared L3 cache, degrading the performance (by a factor of 4 compared to the isolated environment (Table 6.7). Varuna alleviates this problem by dynamically decreasing the parallelism in each instance close to the best DoP (11). The execution time is almost comparable to the one in the isolated environment (Table 6.7).

Duty Cycle Encoding	Performance Reduction
0000B	0%
1001B	12.5%
1010B	25.0%
1011B	37.5%
1100B	50.0%
1101B	63.5%
1110B	75.0%
1111B	87.5%

Table 6.12 Clock modulation levels on Intel machines.

The result for Hash Join co-scheduled with Stream (row #4, Table 6.11) reiterates the inference made in Section 6.3.1. The interference caused by Hash Join slows down Stream's demand for memory bandwidth, thereby reducing the negative impact on performance for both its Pthreads and Varuna versions. However, the Pthreads versions of Hash Join slowed down by a factor of 2 due to Stream's interference as compared to isolated environment (Table 6.7). Varuna improves the performance of Hash Join by 20% against Pthreads by reducing its DoP to 10 (Table 6.11, row #4, column DoP-B1).

6.3.3.3 Contention-free and contention-free

In the final case, both benchmarks were scalable, e.g., Barneshut and Bzip2 (row #5, Table 6.11). Varuna has only marginal impact since neither suffers from contention. It simply divides the resources evenly amongst the benchmarks. For both the versions, the execution times are worse than the isolated times since both benchmarks can benefit from higher number of resources which they are not provided in the multiprogrammed environment.

6.4 Asymmetric Execution Environment

An asymmetric hardware in one in which the individual cores exhibit different power/performance characteristics. Since asymmetric processing hardware is not yet available, we evaluate

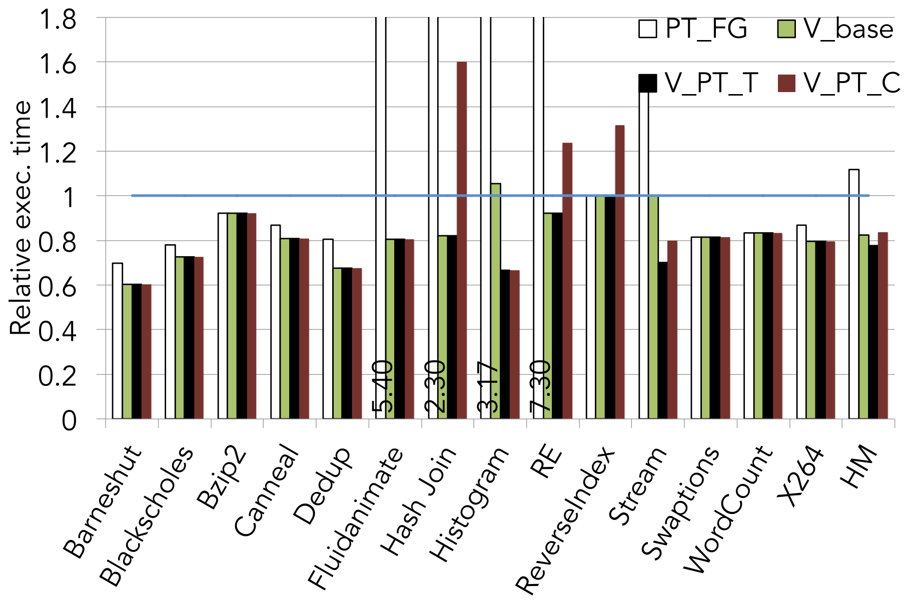
Core	Performance
0	100%
1	100%
2	50%
3	50%

Table 6.13 Clock modulation levels used for core i7 with SMT switched off.

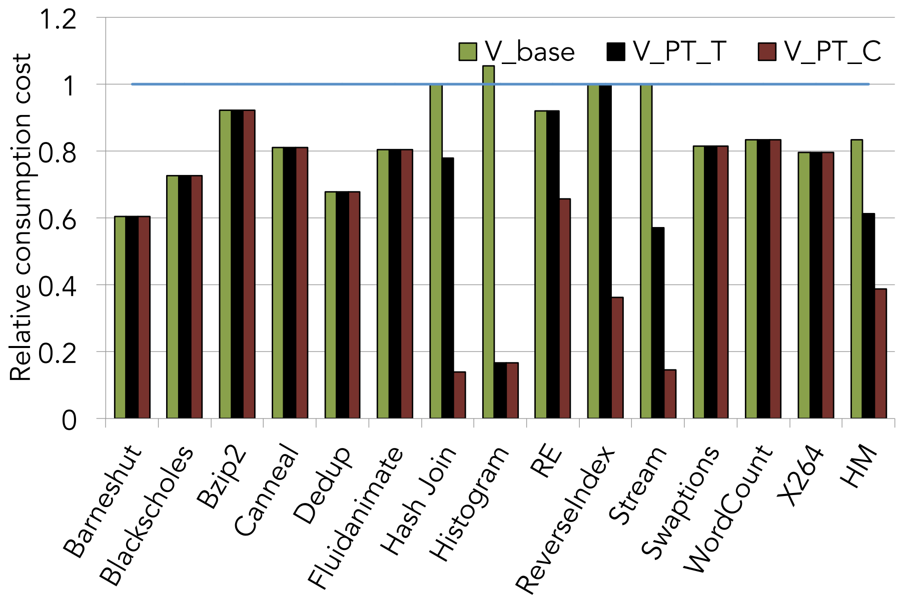
Varuna through emulation in this environment. We could not use DVFS to emulate asymmetric performance, as on our experimental platforms it applies to an entire socket rather than a single core. Instead, we use Intel's clock-modulation feature [1]. This mechanism is used for thermal throttling and controls the processor duty cycle by stopping the clock for short periods (less than $3\mu s$) at regular intervals. There are eight levels available through the `IA32_CLOCK_MODULATION` model specific register (MSR), as shown in Table 6.12. These levels reduce performance from 100% down to 12.5% of full performance in steps of 12.5%. Unlike real asymmetric processors, the performance impact of clock modulation is independent of the code execution. Thus, an application sees a performance drop of 50% if the duty cycle is cut in half. We performed all our experiments in this environment on the Core i7 with SMT option switched off. Table 6.13 lists the performance levels that we used for each core on the Core i7.

We chose this environment to demonstrate the following: (1) how statically partitioning work in the application assuming that all cores are of equal performance can result in unintended negative impacts on the application performance, and (2) how Varuna is able to adapt to the underlying hardware asymmetry and improve performance and resource consumption cost without exposing this asymmetry to the application.

Since the emulated platform consists of only 4 physical cores there were not many opportunities to control the parallelism except for the non-scalable applications, Histogram, ReverseIndex, RE, Stream and Hash Join. Figure 6.25 presents the execution time and resource consumption cost results for Pthreads and Varuna configurations. We do not present energy numbers as the savings



(a) Execution Time



(b) Resource consumption cost

Figure 6.25 Execution time and resource consumption cost of Varuna and Pthreads on the emulated asymmetric environment.

are similar to their execution time counterparts. As it can be seen, Varuna's versions on an average (HM) outperform the pthread versions in this environment.

PT_FG does better than PT_CG for all the non-contending benchmarks (Barneshut, Blackscholes, Bzip2, Canneal, Dedup, Swaptions, Wordcount, and X264). This is because the Operating System is able to achieve better load balancing with a higher number of threads. However, for the benchmarks that contend for resources (Fluidanimate, Hash Join, Histogram, RE and Stream), PT_FG incurs performance degradation. This is because the benefits of fine grained Operating System scheduling is offset by the overheads caused by the contention in the system.

All Varuna versions are able to do better than both PT_CG and PT_FG for all the applications, on an average (HM). This is because of fine-grained vtask creation and low-overhead dynamic load balancing via work stealing. The execution of V_PT_T is similar to V_base except for Histogram and Stream. Histogram and Stream do not scale beyond thread count of 1 and 2, respectively and V_PT_T controls their parallelism and further improves performance of this application. As in the isolated environment (Section 6.2), V_PT_C resorted to sequential execution for Histogram, ReverseIndex, RE, Stream and Hash Join. Consequently, it increased the execution time of RE, ReverseIndex and Hash Join. However, it makes up for the lost performance by reducing the total resource consumption cost on an average by 32%. This reduction is 23% more than V_PT_T.

Chapter 7

Conclusions and Future Work

The industry is rapidly deploying multicore processors in systems ranging from mobile devices to exascale computers. Parallel programming for these systems and their dynamically changing operating environments pose significant challenges to everyday programmers in the effort to improve productivity and to achieve efficient parallel execution of their applications. In this thesis, we presented a solution that achieves efficient execution of parallel applications, without the involvement of the programmer, the OS or any other entity.

7.1 Thesis Summary and Contributions

This dissertation developed and evaluated a run-time system, Varuna, that automates the process of managing and optimizing an application's parallelism, in order to alleviate programmers from the burden of this complexity. Varuna *dynamically*, *continuously*, *rapidly* and *transparently* adapts an application's parallelism to best match the instantaneous capabilities and availability of the hardware resources and the characteristics of the application, while optimizing different performance objectives.

The most important requirement for Varuna to dynamically adapt an application's parallelism is the ability to determine the optimum degree of parallelism that an application must employ at a given instant in time for a given performance objective. To rapidly determine the optimum degree of parallelism, this dissertation developed a holistic and resource-agnostic *scalability model* based on Amdahl's law. Varuna first employs the model to rapidly estimates changes in efficiency during an application's parallel execution. It then uses formulae, derived from the model, to instantaneously determine the optimum degree of parallelism (DoP) to employ for two different performance objectives: (i) Maximize application throughput, and (ii) Minimize resource consumption cost.

The model makes certain assumptions about the application's parallel execution behavior in dynamic operating conditions. However, there are scenarios in which those assumptions could be violated. To tackle such scenarios, in addition to the model, this dissertation proposed using a *hill climbing based search heuristic*. The heuristic locates the optimum degree of parallelism by iteratively exploring the direction in the search space that yields higher efficiency. Hill climbing algorithms are known to get "stuck" in a local optimum and fail to reach the global optimum. Hence this heuristic, developed using the guidelines due to Gendreau [50], incorporates *Tabu search* [52] to escape local optima.

Another equally important requirement for Varuna is to transparently control the execution of parallel computations in the application such that it best matches the instantaneous degree of parallelism determined by the scalability model or the heuristic mentioned above, without hampering the application's forward progress. To realize this capability, this dissertation employed a primitive called a *virtual task (vtask)*. Vtasks abstract hardware contexts into logical cooperative tasks [6, 4] to which application's parallel computations are transparently mapped. The vtasks, in turn, are dynamically scheduled onto hardware contexts, similar to tasks in a dynamic task-based runtime system. Each vtask maintains the state of the current computation mapped on to it using *contexts*, allowing Varuna to transparently pause, resume, or migrate a computation by saving or restoring its corresponding vtask's context. It also includes the state necessary for Varuna to ensure the computations' forward progress even as their execution is regulated.

Vtasks retain the existing parallel programming abstractions and can be applied to both task-based and multithreaded shared memory parallel applications. Further, they require no changes to the application or the Operating System, and can tackle arbitrary parallel applications that use standard APIs. This dissertation demonstrated the versatility of vtasks for three different shared memory parallel programming APIs: Pthreads, Intel Thread Building Blocks (TBB), and Prometheus.

Finally, this dissertation evaluated Varuna in three different execution environments, isolated, multiprogrammed and asymmetric, using unaltered C/C++ Pthreads, TBB and Prometheus applications from various standard benchmark suites, on three different real hardware platforms with different microarchitectural resource capabilities. For the $MAX(throughput)$ objective, Varuna reduced the execution time on an average by 15% in the isolated environment and 33% in the multiprogrammed environment. The concomitant energy savings are 31% and 32%, respectively. For the $MIN(consumption)$ metric, Varuna saved resource consumption cost by 84% and 90% in isolated and multiprogrammed environments, respectively.

7.2 Limitations and Implications

In our experience, Varuna has achieved its original goal of improving performance portability of parallel programs amongst multiple disparate platforms where the degree of parallelism may change dynamically without any programmer involvement. However, it is not without limitations.

The efficiency of an application's parallel execution depends on a variety of parameters including, utilization of hardware and software resources, the application's data layout, data and work distribution algorithms used, parallel algorithm implemented, scheduling parameters, among others. Varuna provides the basic capabilities to dynamically manage and optimize an application's parallel execution. In this dissertation, we primarily focused on the first parameter of effectively utilizing resources in the system such that they are neither underutilized nor oversubscribed. One could envision leveraging Varuna's capabilities to optimize for other parameters to further improve an application's efficiency.

The current implementation of Varuna is purely in user space. This was a deliberate design decision to enable portability amongst disparate host operating systems. Nonetheless, if Varuna were to be widely adopted, then it would be reasonable to assume that essentially many applications on a running system would employ it. Thus, multiple applications running Varuna is an important problem going forward. So far, Varuna can make only local decisions specific to an application. This can sometimes lead to oscillation problems, when multiple Varuna applications are trying to adapt their parallelism at the same. To avoid this problem, a co-ordinated approach is needed.

The most obvious place to employ such an approach will be in the Operating System. We expect the approach to work similarly to Scheduler Activations [11], where the duties are split between the Operating System and the runtime system. Here is a sketch of one possible implementation. The Operating System uses the scalability model to determine the optimum degree of parallelism to employ for each Varuna-compliant concurrent application in the system, and exposes it to the applications using a Scheduler Activation like interface. The runtime scheduler (attached to each application) appropriately adapts the execution of computations in the application to match the parallelism determined by the Operating System. Having support from the Operating System also enables I/O calls to interoperate seamlessly with Varuna. In contrast, a pure user-level implementation limits applications to using the asynchronous subset of the Operating System's I/O interface or employing special wrappers around blocking I/O calls.

One of the ways that Varuna creates a large number of *safe points* to quickly and efficiently control the execution of parallel computations in the application is by creating a large number of fine-grained vtasks. Currently, this process is performed manually and we carefully chose the size of each vtask such that they are sufficiently large enough to yield performance improvements on the target platform, while at the same time multiple of them could finish executing within the monitoring time-interval. Ideally, one would like to automate this approach to ease the burden on programmers. However, this is a non-trivial problem and requires further research.

Currently, Varuna monitors changes in the execution environment at $100ms$ granularity due to the overheads associated with fine-grained monitoring in multiprogrammed environments. As a consequence, it may not be possible for Varuna to quickly respond to changes happening at finer timescales. While support from hardware could potentially address this issue, it requires appropriately creating new interfaces between the hardware and the software to facilitate seamless information flow.

Single-ISA asymmetric performance multicore processors are shown to deliver higher performance per watt and area for applications with diverse architectural requirements, and so it is likely that future multicore processors will combine fast cores characterized by complex pipelines, high clock frequency, high area requirements and power consumption and many slow cores characterized by simple pipelines, low clock frequency, low area requirements and power consumption [14]. Currently, Varuna assumes that the underlying processing cores are homogeneous, and hence can fail to leverage the underlying asymmetry in accelerating performance- and energy-critical computation bottlenecks. Some of the bottlenecks include critical sections, Amdahl's serial portions and critical stages in a software pipeline. Varuna already possesses the necessary capability to transparently migrate computations using vtasks. One could envision extending this capability to accelerate critical computations in the application to further improve the efficiency of an application on asymmetric hardware.

7.3 Future Directions

During the development of Varuna and the preparation of this dissertation, we have identified several directions for future research. One clear immediate direction is the application of Varuna to a broader range of applications and programming models to demonstrate its applicability as well as to identify any shortcomings of its capabilities and mechanisms. Another clear immediate direction is the application of Varuna in large scale environments. Adapting to varying resources dynamically is especially important for large scientific clusters, where the number of nodes available for computation varies dramatically over time. Varuna can be extended to manage execution in such environments. More generally, on large-scale datacenters, Varuna can be scaled up to maximize utilization and minimize power consumption of clusters of multicore processors.

Cloud computing is emerging as a promising field offering a variety of computing services to end users. These services are offered at different prices using various pricing schemes and techniques. End users will favor the service provider offering the best Quality-of-Service (QoS) with the lowest price. Therefore, applying an effective pricing model will attract more customers and achieve higher revenues for service providers. Although the scalability model proposed in this dissertation is primarily used as a performance model to optimize an application's execution, one could envision using it as a pricing model in cloud environments to charge applications based on their resource consumption characteristics.

Speculation is a key enabler of parallelism in general-purpose programs. Previous studies have shown that excessive speculation can lead to wasted computation as more transactions are aborted, resulting in performance and energy loss. Speculation can be throttled dynamically at run-time, depending on observed misspeculation rates. Varuna can be extended such that it can throttle the degree of parallelism to reduce misspeculation. This may improve energy efficiency since the amount of computation that is discarded is reduced.

LIST OF REFERENCES

[1] Intel64 and IA-32 Architectures Software Developer's Manual Combined Volumes 3A and 3B: System Programming Guide, Parts 1 and 2. http://www.intel.com/Assets/PDF/manual/325384.pdf.

[2] Posix threads programming. In *https://computing.llnl.gov/tutorials/pthreads/*, Livermore, California, USA. IEEE.

[3] WattsUp. http://wattsupmeters.com/.

[4] Windows fiber. In *http://msdn.microsoft.com/en-us/library/windows/desktop/ms682661(v=vs.85).aspx*.

[5] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for unix development. pages 93–112, 1986.

[6] Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, and John R. Douceur. Cooperative task management without manual stack management. In *Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*, ATEC '02, pages 289–302, Berkeley, CA, USA, 2002. USENIX Association.

[7] Matthew D. Allen, Srinath Sridharan, and Gurindar S. Sohi. Serialization sets: A dynamic dependence-based parallel execution model. In *Proceedings of the 14th symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 85–96, 2009.

[8] Matthew D. Allen, Srinath Sridharan, and Gurindar S. Sohi. Serialization sets: a dynamic dependence-based parallel execution model. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 85–96, New York, NY, USA, 2009. ACM.

[9] Matthew D. Allen, Srinath Sridharan, and Gurindar S. Sohi. Serialization sets: a dynamic dependence-based parallel execution model. *SIGPLAN Not.*, 44:85–96, February 2009.

[10] Ashok Anand, Chitra Muthukrishnan, Aditya Akella, and Ramachandran Ramjee. Redundancy in network traffic: findings and implications. In *Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*, SIGMETRICS '09, pages 37–48, New York, NY, USA, 2009. ACM.

[11] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: effective kernel support for the user-level management of parallelism. In *Proceedings of the thirteenth ACM symposium on Operating systems principles*, SOSP '91, pages 95–109, New York, NY, USA, 1991. ACM.

[12] Andrea C. Arpaci-dusseau, Remzi H. Arpaci-dusseau, Nathan C. Burnett, Timothy E. Denehy, Thomas J. Engle, Haryadi S. Gunawi, James A. Nugent, and Florentina I. Popovici. Transforming policies into mechanisms with infokernel. In *In Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 90–105. ACM Press, 2003.

[13] David F. Bacon and Seth Copen Goldstein. Hardware-assisted replay of multiprocessor programs. In *Proceedings of the Workshop on Parallel and Distributed Debugging (PADD)*, pages 194–206, 1991.

[14] Saisanthosh Balakrishnan, Ravi Rajwar, Mike Upton, and Konrad Lai. The impact of performance asymmetry in emerging multicore architectures. In *Proceedings of the 32nd annual international symposium on Computer Architecture*, ISCA '05, pages 506–517, Washington, DC, USA, 2005. IEEE Computer Society.

[15] Gabriele Jost Barbara Chapman and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*. The MIT Press, 2007.

[16] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using magpie for request extraction and workload modelling. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 18–18, Berkeley, CA, USA, 2004. USENIX Association.

[17] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177, October 2003.

[18] Frank Bellosa, Andreas Weissel, Martin Waitz, and Simon Kellner. Event-driven energy accounting for dynamic thermal management. In *Proceedings of the Workshop on Compilers and Operating Systems for Low Power (COLP'03)*, New Orleans, LA, September 27 2003.

[19] Brian N. Bershad, Craig Chambers, Susan Eggers, Chris Maeda, Dylan McNamee, Przemys law Pardyak, Stefan Savage, and Emin Gun Sirer. Spin&mdash;an extensible microkernel for application-specific operating system services. *SIGOPS Oper. Syst. Rev.*, 29(1):74–77, January 1995.

[20] Christian Bienia et al. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel Achitectures and Compilation Techniques (PACT)*, October 2008.

[21] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and Jim Demmel. Optimizing matrix multiply using phipac: A portable, high-performance, ansi c coding methodology. In *Proceedings of the 11th International Conference on Supercomputing*, ICS '97, pages 340–347, New York, NY, USA, 1997. ACM.

[22] Ramazan Bitirgen, Engin Ipek, and Jose F. Martinez. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, pages 318–329, Washington, DC, USA, 2008. IEEE Computer Society.

[23] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. In *Proceedings of the 5th symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 207–216, 1995.

[24] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, 1999.

[25] Robert L. Bocchino Jr. and et al. A type and effect system for deterministic parallel java. In *OOPSLA: Object Oriented Programming, Systems, Languages and Applications*, USA, October 2009. ACM.

[26] S. Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *Micro, IEEE*, 25(6):10–16, 2005.

[27] Shekhar Borkar, Tanay Karnik, Siva Narendra, James Tschanz, Ali Keshavarzi, and Vivek De. Parameter variations and impact on circuits and microarchitecture. In *DAC '03: Proceedings of the Annual Conference on Design Automation*, pages 338–342, 2003.

[28] Silas Boyd-Wickizer, Austin Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An analysis of linux scalability to many cores. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*, Vancouver, Canada, October 2010.

[29] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An analysis of linux scalability to many cores. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.

[30] David Brooks and Margaret Martonosi. Dynamic thermal management for high-performance microprocessors. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, HPCA '01, pages 171–, Washington, DC, USA, 2001. IEEE Computer Society.

[31] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *SIGOPS Oper. Syst. Rev.*, 31(5):143–156, October 1997.

[32] K. Chakraborty, P.M. Wells, G.S. Sohi, T. Benson, A. Akella, D. Maltz, Y. Hou, D.P. Moynihan, M.D. Hill, D. Hower, et al. A case for an over-provisioned multicore system: Energy efficient processing of multithreaded programs. *Technology*, 90(65nm):45nm, 2007.

[33] Koushik Chakraborty. *Over-provisioned Multicore Systems*. PhD thesis, University of Wisconsin-Madison, 2008.

[34] Guang-Ien Cheng, Mingdong Feng, Charles E. Leiserson, Keith H. Randall, and Andrew Stark. Detecting data races in Cilk programs that use locks. In *Proceedings of the 10th Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 298–309, 1998.

[35] C. Constantinescu. Trends and challenges in vlsi circuit reliability. *Micro, IEEE*, 23(4):14 – 19, july-aug. 2003.

[36] Julita Corbalán, Xavier Martorell, and Jesús Labarta. Performance-driven processor allocation. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, pages 5–5, Berkeley, CA, USA, 2000. USENIX Association.

[37] Matthew Curtis-Maury, James Dzierwa, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos. Online power-performance adaptation of multithreaded programs using hardware event-based prediction. In *Proceedings of the 20th annual international conference on Supercomputing*, ICS '06, pages 157–166, New York, NY, USA, 2006. ACM.

[38] Matthew Curtis-Maury, Ankur Shah, Filip Blagojevic, Dimitrios S. Nikolopoulos, Bronis R. de Supinski, and Martin Schulz. Prediction models for multi-dimensional power-performance optimization on many cores. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pages 250–259, New York, NY, USA, 2008. ACM.

[39] Matthew DeVuyst, Ashish Venkat, and Dean M. Tullsen. Execution migration in a heterogeneous-isa chip multiprocessor. In *ASPLOS*, pages 261–272, 2012.

[40] David J DeWitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker, and David A. Wood. Implementation techniques for main memory database systems. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, SIGMOD '84, pages 1–8, New York, NY, USA, 1984. ACM.

[41] Santanu Dutta, Rune Jensen, and Alf Rieckmann. Viper: A multiprocessor soc for advanced set-top box and digital tv systems. *IEEE Des. Test*, 18(5):21–31, September 2001.

[42] Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N. Patt. Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS '10, pages 335–346, New York, NY, USA, 2010. ACM.

[43] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, SOSP '95, pages 251–266, New York, NY, USA, 1995. ACM.

[44] Yoav Etsion, Felipe Cabarcas, Alejandro Rico, Alex Ramirez, Rosa M. Badia, Eduard Ayguade, Jesus Labarta, and Mateo Valero. Task superscalar: An out-of-order task pipeline. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 89–100, Washington, DC, USA, 2010. IEEE Computer Society.

[45] Mingdong Feng and Charles E. Leiserson. Efficient detection of determinacy races in cilk programs. In *Proceedings of the 9th Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 1–11, 1997.

[46] M. Frigo and S.G. Johnson. Fftw: an adaptive software architecture for the fft. In *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, volume 3, pages 1381–1384 vol.3, May 1998.

[47] M. Frigo and S.G. Johnson. The design and implementation of fftw3. *Proceedings of the IEEE*, 93(2):216–231, Feb 2005.

[48] Matteo Frigo, Pablo Halpern, Charles E. Leiserson, and Stephen Lewin-Berlin. Reducers and other Cilk++ hyperobjects. In *Proceedings of the 21st Symposium on Parallelism in Algorithms and Architectures*, pages 79–90, 2009.

[49] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the 1998 conference on Programming Language Design and Implementation (PLDI)*, pages 212–223, 1998.

[50] M. Gendreau. An Introduction to Tabu Search. In F. Glover and G. Kochenberger, editors, *Handbook of Metaheuristics*, chapter 2, pages 37–54. Kluwer Academic Publishers, 2003.

[51] J. Gilchrist. Parallel data compression with bzip2. In *IASTED International Conference on Parallel and Distributed Computing and Systems*, pages 559–564, 2004.

[52] Fred Glover and Manuel Laguna. *Tabu Search*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[53] S.C. Goldstein, K.E. Schauser, and D.E. Culler. Lazy threads: Implementing a fast parallel call. *Journal of Parallel and Distributed Computing*, 37(1):5–20, 1996.

[54] Mohamed Gomaa, Chad Scarbrough, T. N. Vijaykumar, and Irith Pomeranz. Transient-fault recovery for chip multiprocessors. *SIGARCH Comput. Archit. News*, 31(2):98–109, May 2003.

[55] Pawan Goyal, Xingang Guo, and Harrick M. Vin. Readings in multimedia computing and networking. chapter A Hierarchical CPU Scheduler for Multimedia Operating Systems, pages 491–505. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[56] S. Gunther, F. Binns, D.M. Carmean, and J.C. Hall. Managing the impact of increasing microprocessor power consumption. *Intel Technology Journal*, 1, 2001.

[57] Gagan Gupta and Gurindar S. Sohi. Dataflow execution of sequential imperative programs on multicore architectures. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44 '11, pages 59–70, New York, NY, USA, 2011. ACM.

[58] Mark Hempstead, Gu-Yeon Wei, and David Brooks. Navigo: An early-stage model to study power-contrained architectures and specialization. In *Proceedings of Workshop on Modeling, Benchmarking, and Simulations (MoBS)*, 2009.

[59] Andrew Herdrich, Ramesh Illikkal, Ravi Iyer, Don Newell, Vineet Chadha, and Jaideep Moses. Rate-based qos techniques for cache/memory in cmp platforms. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pages 479–488, New York, NY, USA, 2009. ACM.

[60] Derek R. Hower and Mark D. Hill. Rerun: exploiting episodes for lightweight memory race recording. In *Proceedings of the 35th International Symposium on Computer Architecture (ISCA)*, pages 265–276, 2008.

[61] Ramesh Illikkal, Vineet Chadha, Andrew Herdrich, Ravi Iyer, and Donald Newell. Pirate: Qos and performance management in cmp architectures. *SIGMETRICS Perform. Eval. Rev.*, 37:3–10, March 2010.

[62] Eun-Jin Im, Katherine Yelick, and Richard Vuduc. Sparsity: Optimization framework for sparse matrix kernels. *Int. J. High Perform. Comput. Appl.*, 18(1):135–158, February 2004.

[63] Engin Ipek, Meyrem Kirman, Nevin Kirman, and Jose F. Martinez. Core fusion: accommodating software diversity in chip multiprocessors. In *ISCA '07*, pages 186–197.

[64] Semiconductor Industry Association (SIA), Design, International Roadmap for Semiconductors, 2011 edition. http://public.itrs.net.

[65] Ravi Iyer. Cqos: a framework for enabling qos in shared caches of cmp platforms. In *Proceedings of the 18th annual international conference on Supercomputing*, ICS '04, pages 257–266, New York, NY, USA, 2004. ACM.

[66] Ravi Iyer, Li Zhao, Fei Guo, Ramesh Illikkal, Srihari Makineni, Don Newell, Yan Solihin, Lisa Hsu, and Steve Reinhardt. Qos policies and architecture for cache/memory in cmp platforms. In *Proceedings of the 2007 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '07, pages 25–36, New York, NY, USA, 2007. ACM.

[67] Magnus Jahre and Lasse Natvig. A light-weight fairness mechanism for chip multiprocessor memory systems. In *Proceedings of the 6th ACM Conference on Computing Frontiers*, CF '09, pages 1–10, New York, NY, USA, 2009. ACM.

[68] Brian Jeff. Big.little system architecture from arm: saving power through heterogeneous multiprocessing and task context migration. In Patrick Groeneveld, Donatella Sciuto, and Soha Hassoun, editors, *DAC*, pages 1143–1146. ACM, 2012.

[69] Changhee Jung, Daeseob Lim, Jaejin Lee, and SangYong Han. Adaptive execution techniques for smt multiprocessor architectures. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '05, pages 236–246, New York, NY, USA, 2005. ACM.

[70] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, January 2003.

[71] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Parallel Architecture and Compilation Techniques, 2004. PACT 2004. Proceedings. 13th International Conference on*, pages 111 – 122, 2004.

[72] W. Ko, M. Yankelevsky, D.S. Nikolopoulos, and C.D. Polychronopoulos. Effective cross-platform, multilevel parallelism via dynamic adaptive execution. In *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM*, April 2002.

[73] Milind Kulkarni, Martin Burtscher, Keshav Pingali, and Calin Cascaval. Lonestar: A suite of parallel irregular programs. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 65–76, April 2009.

[74] Rakesh Kumar, Keith I. Farkas, Norman P. Jouppi, Parthasarathy Ranganathan, and Dean M. Tullsen. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In *Proceedings of the 36th Annual International Symposium on Microarchitecture*, pages 81–92, June 2003.

[75] Rakesh Kumar, Keith I. Farkas, Norman P. Jouppi, Parthasarathy Ranganathan, and Dean M. Tullsen. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, 2003.

[76] Rakesh Kumar, Dean M. Tullsen, Parthasarathy Ranganathan, Norman P. Jouppi, and Keith I. Farkas. Single-isa heterogeneous multi-core architectures for multithreaded workload performance. In *Proceedings of the 31st annual international symposium on Computer architecture*, ISCA '04, pages 64–, Washington, DC, USA, 2004. IEEE Computer Society.

[77] Kishore Kumar Pusukuri, Rajiv Gupta, and Laxmi N. Bhuyan. Adapt: A framework for coscheduling multithreaded programs. *ACM Trans. Archit. Code Optim.*, 9(4):45:1–45:24, January 2013.

[78] Christopher LaFrieda, Engin Ipek, Jose F. Martinez, and Rajit Manohar. Utilizing dynamically coupled cores to form a resilient chip multiprocessor. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '07, pages 317–326, Washington, DC, USA, 2007. IEEE Computer Society.

[79] T. J. Leblanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computing*, 36:471–482, 1987.

[80] Janghaeng Lee, Haicheng Wu, Madhumitha Ravichandran, and Nathan Clark. Thread tailor: dynamically weaving threads together for efficient, adaptive parallel applications. *SIGARCH Comput. Archit. News*, 38:270–279, June 2010.

[81] Dong Li, B.R. de Supinski, M. Schulz, K. Cameron, and D.S. Nikolopoulos. Hybrid mpi/openmp power-aware computing. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1 –12, april 2010.

[82] J. Li and J.F. Martinez. Power-performance implications of thread-level parallelism on chip multiprocessors. In *Performance Analysis of Systems and Software, 2005. ISPASS 2005. IEEE International Symposium on*, pages 124 –134, march 2005.

[83] J. Li and J.F. Martinez. Dynamic power-performance adaptation of parallel computation on chip multiprocessors. In *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*, pages 77 – 87, feb. 2006.

[84] Robert S Lockhart. *Introduction to Statistics and Data Analysis: For the Behavioral Sciences*. Macmillan, 1998.

[85] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-up: increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44 '11, pages 248–259, New York, NY, USA, 2011. ACM.

[86] Henry Massalin. Superoptimizer: A look at the smallest program. In *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems*, ASPLOS II, pages 122–126, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

[87] John M. May. Mpx: Software for multiplexing hardware performance counters in multithreaded programs. In *Proceedings of the 15th International Parallel &Amp; Distributed Processing Symposium*, IPDPS '01, pages 22–, Washington, DC, USA, 2001. IEEE Computer Society.

[88] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.

[89] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9:21–65, February 1991.

[90] E. Mohr, D. A. Kranz, and R. H. Halstead, Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Trans. Parallel Distrib. Syst.*, 2(3):264–280, July 1991.

[91] Pablo Montesinos, Luis Ceze, and Josep Torrellas. DeLorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In *Proceedings of the 35th International Symposium on Computer Architecture (ISCA)*, pages 289–300, 2008.

[92] P.J. Mucci, S. Browne, C. Deane, and G. Ho. Papi: A portable interface to hardware performance counters. In *Proc. Dept. of Defense HPCMP Users Group Conference*, pages 7–10, 1999.

[93] Shubhendu S. Mukherjee, Michael Kontz, and Steven K. Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. *SIGARCH Comput. Archit. News*, 30(2):99–110, May 2002.

[94] O. Mutlu and T. Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*, pages 146 –160, 2007.

[95] Onur Mutlu and Thomas Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 63–74, Washington, DC, USA, 2008. IEEE Computer Society.

[96] Vandad Nahavandipoor. *Concurrent Programming in Mac OS X and iOS: Unleash Multicore Performance with Grand Central Dispatch.* " O'Reilly Media, Inc.", 2011.

[97] Satish Narayanasamy, Cristiano Pereira, and Brad Calder. Recording shared memory dependencies using strata. In *Proceedings of the 12th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 229–240, 2006.

[98] Satish Narayanasamy, Gilles Pokam, and Brad Calder. BugNet: Continuously recording program execution for deterministic replay debugging. In *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA)*, pages 284–295, 2005.

[99] Kyle J. Nesbit, James Laudon, and James E. Smith. Virtual private caches. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, pages 57–68, New York, NY, USA, 2007. ACM.

[100] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: efficient deterministic multithreading in software. In *Proceedings of the 14th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 97–108, 2009.

[101] Heidi Pan, Benjamin Hindman, and Krste Asanović. Composing parallel software efficiently with lithe. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 376–387, New York, NY, USA, 2010. ACM.

[102] J.M. Perez, R.M. Badia, and J. Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *Cluster Computing, 2008 IEEE International Conference on*, pages 142 –151, 29 2008-oct. 1 2008.

[103] Allan Porterfield, Rob Fowler, Sridutt Bhalachandra, and Wei Wang. Openmp and mpi application energy measurement variation. In *Proceedings of the 1st International Workshop on Energy Efficient Supercomputing*, E2SC '13, pages 7:1–7:8, New York, NY, USA, 2013. ACM.

[104] Kishore Kumar Pusukuri, Rajiv Gupta, and Laxmi N. Bhuyan. Thread reinforcer: Dynamically determining number of threads via os level monitoring. In *Proceedings of the 2011 IEEE International Symposium on Workload Characterization*, IISWC '11, pages 116–125, Washington, DC, USA, 2011. IEEE Computer Society.

[105] Kishore Kumar Pusukuri, Rajiv Gupta, and Laxmi N. Bhuyan. Thread tranquilizer: Dynamically reducing performance variation. *ACM Trans. Archit. Code Optim.*, 8(4):46:1–46:21, January 2012.

[106] Arun Raman, Hanjun Kim, Taewook Oh, Jae W. Lee, and David I. August. Parallelism orchestration using DoPE: the degree of parallelism executive. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 26–37, New York, NY, USA, 2011. ACM.

[107] Arun Raman, Ayal Zaks, Jae W. Lee, and David I. August. Parcae: a system for flexible parallel execution. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, pages 133–144, New York, NY, USA, 2012. ACM.

[108] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor systems. In *Proceedings of the 13th international symposium on High Performance Computer Architecture (HPCA)*, pages 13–24, 2007.

[109] James Reinders. *Intel Threading Building Blocks*. O'Reilly Media, Inc., 2007.

[110] Urban Richter, Moez Mnif, Jürgen Branke, Christian Müller-Schloer, and Hartmut Schmeck. Towards a generic observer/controller architecture for organic computing. *GI Jahrestagung (1)*, 93:112–119, 2006.

[111] Micheil Ronsse and Koen De Bosschere. RecPlay: a fully integrated practical record/replay system. *ACM Transactions on Computer Systems*, 17(2):133–152, 1999.

[112] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: A many-core x86 architecture for visual computing. *ACM Transactions on Graphics*, 27(3):1–15, 2008.

[113] Srinath Sridharan, Gagan Gupta, and Gurindar S. Sohi. Holistic run-time parallelism management for time and energy efficiency. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 337–348, New York, NY, USA, 2013. ACM.

[114] Jayanth Srinivasan, Sarita V. Adve, Pradip Bose, and Jude A. Rivers. The Impact of Technology Scaling on Lifetime Reliability. In *Proceedings of International Conference on Dependable Systems and Networks*, June 2004.

[115] Christopher Stewart, Terence Kelly, and Alex Zhang. Exploiting nonstationarity for performance prediction. *SIGOPS Oper. Syst. Rev.*, 41(3):31–44, March 2007.

[116] M. Aater Suleman, Moinuddin K. Qureshi, and Yale N. Patt. Feedback-driven threading: power-efficient and high-performance execution of multithreaded workloads on cmps. In *In Proc. 13th ACM Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 277–286, 2008.

[117] Jrgen Teich, Jrg Henkel, Andreas Herkersdorf, Doris Schmitt-Landsiedel, Wolfgang Schroder-Preikschat, and Gregor Snelting. Invasive computing: An overview. In *Multiprocessor System-on-Chip*, pages 241–268. 2011.

[118] Volkmar Uhlig, Joshua LeVasseur, Espen Skoglund, and Uwe Dannowski. Towards scalable multiprocessor virtual machines. In *Proceedings of the 3rd conference on Virtual Machine Research And Technology Symposium - Volume 3*, VM'04, pages 4–4, Berkeley, CA, USA, 2004. USENIX Association.

[119] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. Conservation cores: reducing the energy of mature computations. In *ASPLOS '10*, pages 205–218.

[120] M.J. Voss and R. Eigenmann. Adapt: Automated de-coupled adaptive program transformation. In *Parallel Processing, 2000. Proceedings. 2000 International Conference on*, pages 163–170, 2000.

[121] Carl A. Waldspurger and William E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation*, OSDI '94, Berkeley, CA, USA, 1994. USENIX Association.

[122] Zheng Wang and Michael F.P. O'Boyle. Mapping parallelism to multi-cores: A machine learning based approach. *SIGPLAN Not.*, 44(4):75–84, February 2009.

[123] Philip M. Wells, Koushik Chakraborty, and Gurindar S. Sohi. Adapting to intermittent faults in multicore systems. *SIGOPS Oper. Syst. Rev.*, 42(2):255–264, March 2008.

[124] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, SC '98, pages 1–27, Washington, DC, USA, 1998. IEEE Computer Society.

[125] Min Xu, Rastislav Bodik, and Mark D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Proceedings of the 30th International Symposium on Computer Architecture (ISCA)*, pages 122–135, 2003.

[126] Min Xu, Mark D. Hill, and Rastislav Bodik. A regulated transitive reduction (RTR) for longer memory race recording. In *Proceedings of the 12th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 49–60, 2006.

[127] Jun Yan and Wei Zhang. Hybrid multi-core architecture for boosting single-threaded performance. *SIGARCH Comput. Archit. News*, 35(1):141–148, March 2007.

[128] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. Hardware execution throttling for multi-core resource management. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, USENIX'09, pages 23–23, Berkeley, CA, USA, 2009. USENIX Association.