A System for Flexible Parallel Execution

Arun Raman

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
by the Department of
Electrical Engineering

Advisor: Professor David I. August

January 2012

Abstract

Exponential growth in transistor density combined with diminishing returns from uniprocessor improvements has compelled the industry to transition to multicore architectures. To realize the performance potential of multicore architectures, programs must be parallelized effectively. The efficiency of parallel program execution depends on the execution environment comprised of workload, platform, and performance goal. In writing parallel programs, most programmers and compilers expose parallelism and optimize it to meet a particular performance goal on a single platform under an assumed set of workload characteristics. In the field, changing workload characteristics, new parallel platforms, and deployments with different performance goals make the programmer's or compiler's development-time or compile-time choices suboptimal.

This dissertation presents Parcae[1], a generally applicable holistic system for platform-wide dynamic parallelism tuning. Parcae includes:

1. the Nona compiler, which applies a variety of auto-parallelization techniques to create *flexible parallel programs* whose tasks can be efficiently paused, reconfigured, and resumed during execution;

2. the Decima monitor, which measures resource availability and system performance to detect change in the environment; and

3. the Morta executor, which cuts short the life of executing tasks, replacing them with other functionally equivalent tasks better suited to the current environment.

Parallel programs made flexible by Parcae outperform original parallel implementations in a variety of interesting scenarios.

[1]According to Roman mythology, the Parcae (pronounced \ˈpär-ˌkī\) are the Three Fates that control the metaphorical thread of life of each mortal. They are Nona, who creates the thread of life; Decima, who measures the thread of life; and Morta, who cuts the thread of life.

Acknowledgments

First and foremost, I sincerely thank my advisor, Prof. David August, for enabling my chosen career path. David has taught me most of what I know about the various aspects of conducting research in computer engineering, from ways to alleviate pressure on load-store units in a processor pipeline to the art of writing good abstracts and creating insightful presentations. I particularly appreciate his faculty for making insightful and encouraging suggestions, which have proved to be of immense use over the years. I am also grateful for his efforts to champion my cause by talking favorably about me to others and providing opportunities to give presentations at various research summits, especially in my early years in graduate school. On the topic of early years, I thank him immensely for believing in me.

I sincerely thank Prof. Niraj Jha and Prof. Scott Mahlke for reading this dissertation and providing thorough and insightful comments. The dissertation is significantly better on account of their feedback. I also thank them for their quick turnaround that facilitated the scheduling of the public defence of this dissertation. I thank Prof. Ruby Lee and Prof. David Walker for serving on my committee. I express my gratitude to them for their willingness to go out of their way to accommodate me in their extremely busy schedules.

I consider myself extremely fortunate to have worked in the Liberty group. One would be hard pressed to find such a group of people who are willing to stay up all night and help address the myriad issues that inevitably crop up just before a paper deadline. I thank Jack, Jialu, Hanjun, Prakash, Nick, Tom, Thomas, and Yun for their help over the years. In particular, I am eternally indebted to Hanjun who has spent many hours working with me on multiple projects, and contributed significantly to the quality of each. His work ethic provided tacit encouragement to improve my own. I thank Prakash for the many interesting discussions on topics ranging from data semantics to the *Ashtavakra Gita*. I thank Nick for fruitful discussions on the compiler implementation. I thank Taewook for volunteering to be a user of a part of the technology described in this dissertation. I thank Tom for his help with system administration matters over the years. I thank Jae and Ayal for their assistance in refining the ideas presented in this dissertation and polishing the presentation. Their help has significantly improved the clarity of this discourse. I thank Easwaran for including me in his project in my second year, affording me the opportunity to watch and learn. I thank Neil for taking the time to comment on early drafts of multiple papers. I thank Guilherme, Matt, and George for providing sagely advice on all-things-grad-school. I thank the previous generations of Liberators for the group culture that is the envy of many, and hope that the next generation, including Deep, Feng, Jordan, Matt, Stephen, and Taewook, will perpetuate and enhance it.

I thank Prof. Sharad Malik for guiding me in my first semester at Princeton. I thank Prof. Martonosi for introducing me to computer architecture in my first year of graduate school and making its study stimulating. I thank Prof. Martonosi and Prof. Peh for serving on my general examination committee. I thank my undergraduate advisors, Prof. Agrawal, Prof. Pillai, and Prof. Anand, for initiating me in electrical engineering and encouraging me to pursue higher studies.

I thank Intel Corporation for supporting my work through a fellowship award. I sincerely thank David, Guilherme, and Olivier Temam for recommending me for the same. I am greatly indebted to Guilherme for being my advocate at Intel.

Having experienced firsthand the terrors of administrative bureaucracy prior to Princeton, I have been fortunate to have had wonderful administrative support. My sincere thanks to Sarah McGovern, Roelie Abdi-Stoffers, Lori Bailey, Stacey Weber, and Melissa Lawson for handling travel grants, fellowship applications, conference organization, and a host of other things.

My stint at Princeton was made immensely enjoyable by all the amazing people that I met and befriended. I promise to strive to make these associations last a lifetime. Abhishek, thanks for your guidance in technical matters, both in my early and later years in graduate school, when needed the most. Aman, Saurabh, and Sushobhan, thanks for your advice on various matters in my early years at Princeton. To all of you, “Respek.” Aditya, thanks for “all ’em laughs”. It was fun to see you always “chillmaxing”. Ajay, thanks for always being a few hundred steps ahead in knowing exactly what needs to be done. Amit, thanks for providing brotherly support whenever I am in the Bay Area. Thanks for guiding me in work matters and in workout matters. Ankit, your expert advice and our engaging conversations helped me sail through my first year at Princeton. Anirudh *Sama*, for being a constant friend during my stint at Princeton, I say *arigatō*. The number of interests we shared was incredible. Aravindan, thank you so much for helping me at various times in the past five years, and for filling any room with laughter. Arjun “*Annathae*” Vijayakumar, thanks for teaching me the ins and outs of social networking, and for treating me to many culinary delights. Arnab, thanks for always reminding me do to my taxes by getting your returns before I even filed. Ashwin “*Ashbhai*” Subramani, five years in Princeton were made a lot of fun in large part due to your friendship and support. Thanks for keeping me grounded when I set off on my flights of fancy. Our collaborative forays into food science ended in success more often than anyone could have guessed. Bhadri “*Nanbaen*” Visweswaran, thanks for “teaching” me Tamil. I encourage you to persist with your *nindō*. Carole and Yu-Yuan, thanks for making me a part of your wedding encore. Volleyball was great fun, as were the many dinners. Deep, it was great fun hanging out, playing basketball, and hearing you rant about the vicissitudes of Manchester United’s fortunes. “Run your socks into the ground.” Easwaran, thanks for your guidance. Your spirituality shines through; I only needed to observe you to understand what the elders talk about. Indraneel, it was a lot of fun hanging out at the movies. You made the quality (lack thereof) of the movies immaterial to having fun. Kostas and Manos, it was amazing to discover how alike Greeks and Indians are culturally, particularly when it came to perpetuating chaos. It was always fun. Niket, thanks for blazing a trail in many ways. Your drive and levelheadedness have always been inspirational. Prakash, the trips to the Whole Earth deli, the not-too-infrequent visits to Mehek, and the late-afternoon fruit salad fiestas were all made memorable, thanks to your knack for engaging conversation. That last game of basketball remains. Prateek, working on linear algebra problems at 2 a.m. in the first year was a lot of fun, as were the tennis games in biting cold. The simple activity of making Maggi in the GC common room always turned out to be anything but mundane. Rajsekar, thanks for the witticisms that you delivered with unfailing regularity. Thanks are also due for your part in preserving for posterity that great cultural artifact; you know what I am talking about. Rohit "Rajgaariyaa", thanks for your brotherly support during our time in Princeton. I always look back fondly upon laid-back weekends enjoying the *pakodas* and *dhoodh-paththi* that you made. Seshadhri, I made a mental note of most of your utterances, which invariably bore the hallmark of a Guru. Entering the parking lot and exiting through the "No Exit" next to Wawa to get from Hibben-Magie to the GC lawns ASAP was always a class move. Srinivas, "woah woah woah". *Kalakku po.* Tushar, thanks for keeping me up-to-date with the happenings in the Liberty group. Even after you moved to Singapore. Vaneet, thanks for showing me the ropes in terms of preparing for the written examination and shopping at the Indian store. Varun "Koomaar", your doings never failed to regale. Our legendary contests, while disdained by some (all), kept things real. Your achievements are a source of inspiration. "*Dosth*" Vijay, I hope I can one day live up to your incredible interpretation of friendship. Bad Tamil movies, awesome home-cooked food, *thayir saadam* at Bridgewater Venky's, moving in the rain (why exactly were we moving that often?), are only a few of the many memories that I shall cherish.

I have been shaped by the love showered upon me by all my aunts, uncles, and cousins. To them, I convey heartfelt gratitude for their kind and encouraging words through this journey in graduate school. In particular, I am indebted to Kumar *Chinnanna*, Deepa *Pinnamma*, Bharat, and Tulasi, and likewise to Shanthi *Atha*, Chinnasamy *Mama*, and Amrutaa, for their support and love and for always making me feel at home. Kumar *Chinnanna*'s and Deepa *Pinnamma*'s help in getting me set up when I had just landed off the boat made the resettlement from across the world seem effortless.

My parents always inspired and instilled values, rather than prescribe a course of action. They gave me the freedom to chart my own course, which has proved priceless in establishing self-confidence. My sister, Aishwarya, has always taken keen interest in my studies, asking questions that forced me to stay on top of my game. My grandparents and parents have surmounted great challenges to increase their children's potential. To *Amma*, *Anna*, Aishwarya, Chellamma *Awwa*, Venku *Thatha*, Papa *Awwa*, and Boocci *Thatha*, whose unconditional love, guidance, and example are responsible for what I am, I dedicate this work.

Contents

List of Figures

Chapter 1

Introduction

Until recently, the computing industry enjoyed a sustained exponential growth in processor performance. Figure 1.1 shows the performance (as measured by execution of the SPEC Integer benchmark suites) of all machine configurations that were reported to SPEC between 1993 and 2011 [84]. The x-axis represents time. A point on the graph represents performance of a real machine at that time. The y-axis measures performance normalized across generations of the SPEC benchmark suite. Note that the y-axis is logarithmic. The trend lines are linear regressions over the best performing machine configurations at each point in time.

The computing industry achieved this growth by scaling clock frequency and enhancing uniprocessor microarchitecture. This improved the performance of a wide range of applications, with programmers remaining blissfully unconcerned with changes in the hardware. Since 2004, however, uniprocessor performance has grown at a much slower pace. Increasing design complexity, power and thermal constraints, and diminishing returns from uniprocessor microarchitectural enhancements are the primary reasons for the slowdown. Meanwhile, Moore's law continues to double the number of transistors per unit area. Processor designers leverage these additional transistors by placing multiple cores on the same die.

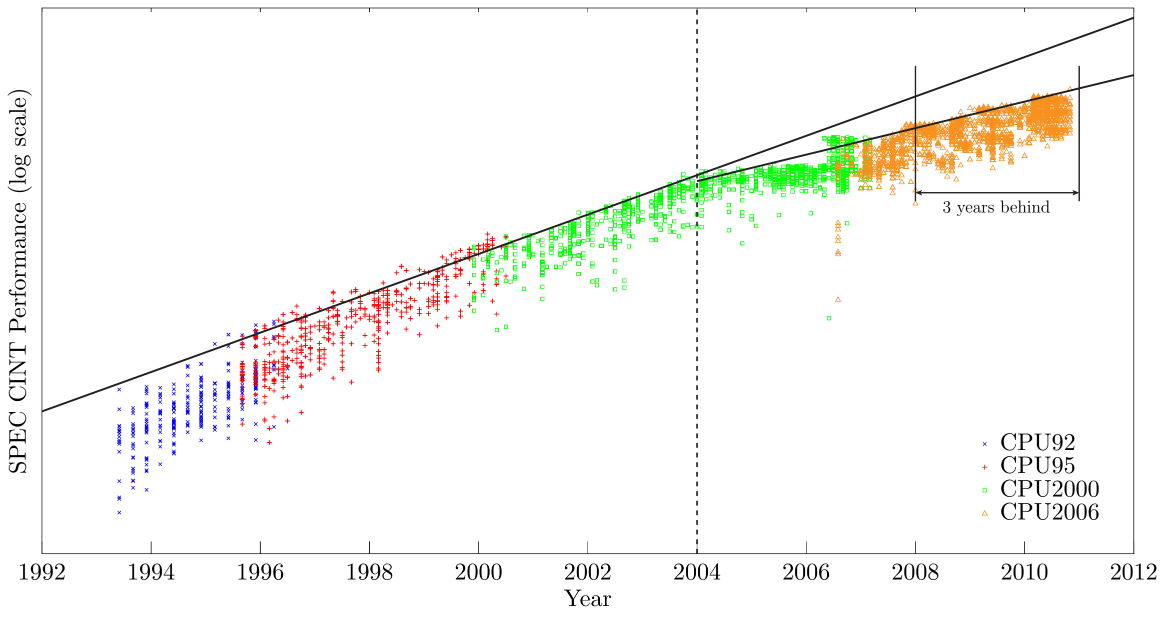


Figure 1.1: Normalized SPEC CINT scores for all reported machine configurations from 1993 to 2011. Since 2004, uniprocessor performance improvement has slowed considerably. Performance levels attained in 2011 would have been achieved in 2008 itself had the pre-2004 growth rate been sustained.

Sequential programs do not benefit from multiple cores. To use the additional cores, programs must be parallel. The hardware extracts instruction-level parallelism within each core, but does not extract thread-level parallelism across multiple cores. Consequently, application developers and compilers are tasked with extracting thread-level parallelism across multiple cores. Much progress has been made in developing tools to extract parallelism, even from seemingly sequential code [18, 24, 70, 72, 92, 104]. Tools such as POSIX threads (Pthreads) [90], Intel Threading Building Blocks (TBB) [77], Cilk++ [17,50], OpenMP [1], and Galois [46] allow application developers to extract thread-level parallelism.

However, parallelism extraction is just one part of the problem of synthesizing well-performing programs that execute efficiently in a variety of execution environments. The other, equally important, part is the tuning and packaging of the extracted parallelism [66, 68]. In the absence of intelligent tuning and packaging, a parallel program may perform worse than the original sequential program [54, 73, 85, 86].

The performance of parallel programs depends on several *run-time* factors. Synchronization and communication overheads are often difficult to predict and may erode the benefits of parallelism [86]. Parallel execution resources available to the program may vary, including number of cores and memory bandwidth [15, 54]. The program's workload characteristics may vary as in the case of web services such as search and video. Furthermore, the performance goals may not be fixed and could be some complex time-varying functions of energy, throughput, etc. Together, these three sources of variability—workload characteristics, platform characteristics, and performance goals—are collectively referred to as the *execution environment* of a program. To execute efficiently, a parallel program must adapt to changes in execution environment. However, the vast majority of parallel programs are produced by programmers or compilers with a single static parallelism configuration encoded at development or compile time. Any single program configuration is likely to become suboptimal with changes in the execution environment [73, 85].

1.1 Shortcomings of Existing Approaches

Fundamentally, the ideal program configuration for a given execution environment can be determined only during the course of execution of the program when information about the execution environment is available. However, once the application has been deployed, it is not possible for the application developer to re-program the application. Consequently, the application code must be prepared ahead of time to facilitate the process of adaptation to change in the execution environment. This split of duties between the code generator (application developer or compiler) and the run-time system is specified by an appropriate interface. The interface is enabled by a run-time system implementation that performs the difficult task of adapting the program to execute optimally as the execution environment changes. The application programming interfaces (APIs), code generation algorithms, and run-time systems of existing parallelization systems all have shortcomings.

1.1.1 Restrictive APIs

Existing parallelization APIs, including the widely used OpenMP [1] and Threading Building Blocks (TBB) [77] APIs, enable parallelizers (application developers or compilers) to encode multiple program configurations implicitly, and have an appropriate configuration be chosen transparently at run-time based on the execution environment [16, 17, 85, 86, 94, 99]. However, these APIs force the parallelizer to either parallelize just a single loop in a loop nest, use a specific type of parallelism (such as task parallelism or pipeline parallelism), or specify a fixed dynamic adaptation mechanism that is tightly coupled to the target parallelism type. This is too restrictive for general-purpose programs. As this dissertation demonstrates, simultaneously parallelizing multiple loops in a nest can provide both scalability and latency-throughput benefits. General-purpose programs tend to have complex dependency patterns that require simultaneous exploitation of different types of parallelism. These programs run on shared commodity platforms with varying constraints such as number of cores available to the program, power, etc., and require adaptation mechanisms that can accommodate all constraints.

1.1.2 Automatic Tuning for Only Array-based Data-parallel Programs

Compilation for flexible execution has typically focused on array-based programs with communication-free data-parallelism, and on parameters such as the degree of parallelism and block size of a loop, because the performance impact of those parameters can be relatively easily modeled for data-parallel array-based programs. Existing compilers and run-times tune these parameters statically or dynamically to match the execution environment, with the goal of optimizing memory access patterns or the number of threads executing a parallel region [5, 10, 32, 43, 96]. However, for general-purpose programs with complex dependency patterns, parallelism is typically non-uniform and involves explicit synchronization or communication. This significantly complicates performance modeling, often resulting in a large space of possibly effective parallelism configurations. Parallelism configurations for such programs consist of the choice of tasks to execute in parallel and the mapping of tasks to threads. Existing compiler-based parallelization algorithms for such programs select a single configuration, typically one deemed most suitable for an unloaded platform [75, 95, 104].

1.1.3 Single Program and Single Objective Optimization

The run-time components of existing parallelization systems optimize execution for a single performance goal (such as throughput), and lack the ability for the system administrator to specify the desired tradeoff between competing desiderata such as latency and throughput or latency and power; such tradeoffs typically arise when general-purpose programs execute on commodity hardware. Furthermore, these run-time systems optimize the execution of individual parallel programs, without exploring platform-wide implications and opportunities. They rely on the operating system scheduler to schedule their threads without responding to system events, such as launch of a new program and consequent reduction in the time-slices available for execution on the platform. As this dissertation shows, the lack of coordination between the OS and the parallel program run-time system results in failure to maximize platform-wide performance.

1.2 Contributions

This dissertation contributes Parcae, a novel API, compiler, and run-time system that effectively delivers performance portability for both array-based programs and general-purpose programs. Parcae enables new functionality in general-purpose programs — automatic adaptation to a changing execution environment. We call such programs *flexible parallel programs.*

The Parcae API enables the separation of the concern of developing a functionally correct parallel program from the concern of optimizing the parallelism for different execution environments. This separation of concerns enables the application developer to focus exclusively on discovering parallelism in the algorithm and expressing the parallelism in all loops in a loop nest, potentially of different types, in a functionally correct and unified manner just once. This frees the application developer from being concerned with optimization and tuning of the parallel program for the duration of its lifetime. The separation of concerns also enables a mechanism developer to specify *mechanisms* that encode the logic to adapt an application's parallelism configuration to meet the performance goals that are set by the administrator of the system on which the program executes.

To enhance existing sequential and parallel programs, Parcae includes the Nona compiler, which transforms those programs into flexible parallel programs by automating the tasks performed by the application and mechanism developers targeting the Parcae API. These tasks include:

- extracting multiple types of parallelism from a program region;
- creating code to efficiently pause, reconfigure, and resume its execution; and
- inserting profiling hooks for a run-time system to monitor its performance.

.

To optimize multiple concurrently executing flexible parallel programs, Parcae includes the Decima monitor and Morta executor—a multi-objective feedback-driven run-time control system incorporating various algorithms and heuristics from control theory. The Parcae run-time system maximizes platform-wide performance by:

- monitoring each program's performance and system events such as launches of new programs;
- quickly determining optimal parallelism configurations for each program; and
- pausing, replacing configurations, and resuming execution of programs efficiently.

.

Specifically, the run-time system automatically and continuously determines for each program:

- which tasks to execute in parallel (e.g., what are the stages of a pipeline);
- how many hardware threads to use (e.g., how many threads to allocate to each stage of the pipeline); and
- how to schedule tasks on to hardware threads (e.g., on which hardware thread should each stage be placed to maximize locality of communication).

Finally, this dissertation evaluates the proposed API, compiler transformations, and parallelization optimization algorithms. On two **real** multicore platforms, this dissertation demonstrates that Parcae provides significant platform-wide and individual-program performance gains running a variety of general-purpose C/C++ programs.

1.3 Dissertation Organization

Chapter 2 contextualizes the need for dynamic adaptation of parallel execution as enabled by Parcae. Chapter 3 describes the overall Parcae architecture and execution model, and sets the stage for detailed description of each component of the system. Chapter 4 explains the novel compiler algorithms that generate flexible parallel programs. Chapter 5 describes the Parcae API for the application developer, mechanism developer, and administrator. Chapter 6 provides details of the novel Parcae run-time system, comprised of the Decima monitor and Morta executor, that optimizes execution of multiple concurrently executing flexible parallel programs. Chapter 7 describes optimizations that reduce the run-time system's overheads. Chapter 8 provides details of the methodology used to evaluate Parcae, and results of performance improvements of several real-world applications and benchmarks on two different platforms. Chapter 9 surveys related work. Chapter 10 discusses future avenues of research and summarizes the conclusions of this dissertation.

Chapter 2

Motivation

This chapter motivates the need for a new parallel-program development and deployment methodology to ensure performance portability. Section 2.1 defines the sources of variability in a program's execution environment. Section 2.2 describes the prevalent methodology employed by programmers to handle the variability. Section 2.3 quantitatively demonstrates the shortcomings of the prevalent methodology. Section 2.4 outlines a novel methodology proposed in this dissertation. Section 2.5 describes limitations of existing systems and specific desirable properties that a system implementing the proposed methodology should possess.

2.1 Sources of Variability in Execution Environment

A parallel program's execution environment consists of the workload characteristics, platform characteristics, and performance goals. Variability along any of these dimensions can necessitate dynamic adaptation of parallelism in order to meet the specified performance goal.

Workload Characteristics A parallel program is typically decomposed into static tasks, with each task being instantiated multiple times at run-time. Each task typically has a workload associated with it. For example, in a pipeline parallelization, the occupancy of the input queues of each stage in the pipeline at any given moment could constitute the workload of that stage at that moment. As we demonstrate later in this chapter, the ideal parallelism configuration of the program depends heavily on the workload.

Platform Characteristics Each new multicore processor has a different (usually larger) number of hardware threads compared to the previous generation. Hardware platforms differ in the number and types of processing threads available, in the core-core and core-memory communication latencies and bandwidths, etc. As more applications move to server clouds, the availability of parallel processing resources itself may change dynamically during the application's execution. A program may have been parallelized with certain assumptions about the target platform. Too tight a coupling between the parallelization and the target platform leads to sub-optimal performance when the assumptions no longer hold.

Performance Goals Computing systems are now called upon to maximize system utility with a variety of optimization goals. One simple example performance goal is maximizing task throughput, energy-delay-squared minimization is another. These performance goals are to be met by the computing system under some constraints such as finite execution resources, power and thermal budgets, etc.

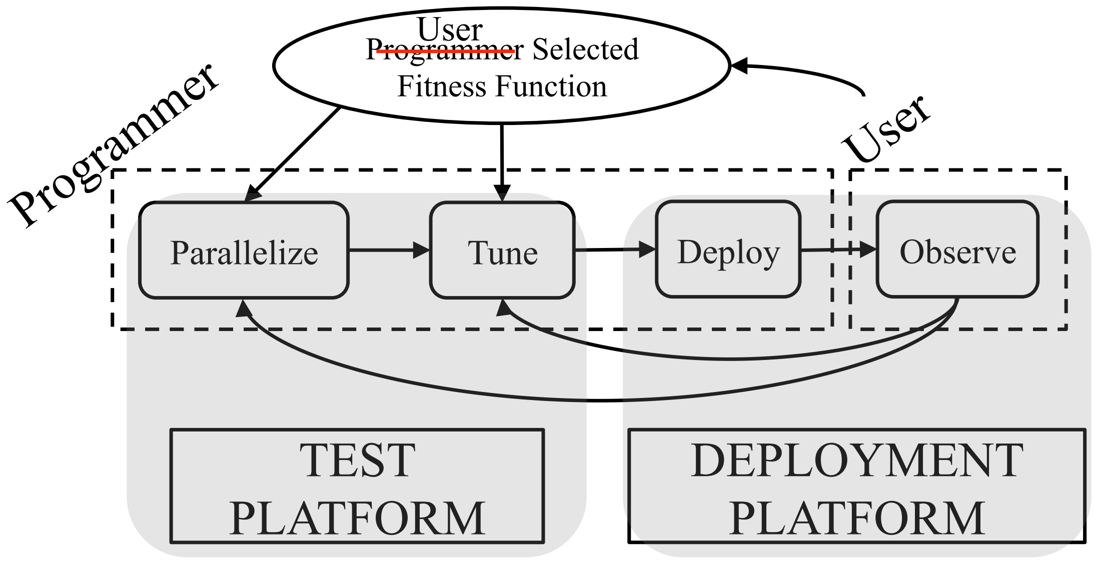


Figure 2.1: A mismatch between the programmer's test platform and the user's deployment platform, or a mismatch between the programmer's and user's notions of performance can result in suboptimal program performance. Performance mismatch feedback from the user to the programmer adds significant human involvement latency to the program optimization process.

2.2 Problems with Prevalent Methodology to Address Variability

Figure 2.1 shows the prevalent parallelization methodology. A programmer implements the user-specified functionality that is desired of the program in a programming model that is most suited to the specifics of the program. The programmer parallelizes the program to improve performance, with some notion of the fitness of the parallelized code. The programmer then tunes the parallel program on his/her development platform, and then ships out the program for deployment. Ideally, the programmer's involvement in the process of optimizing this piece of code would terminate at this juncture, and the programmer would invest time and effort into developing new functionality and providing value addition via new features to the user [87].

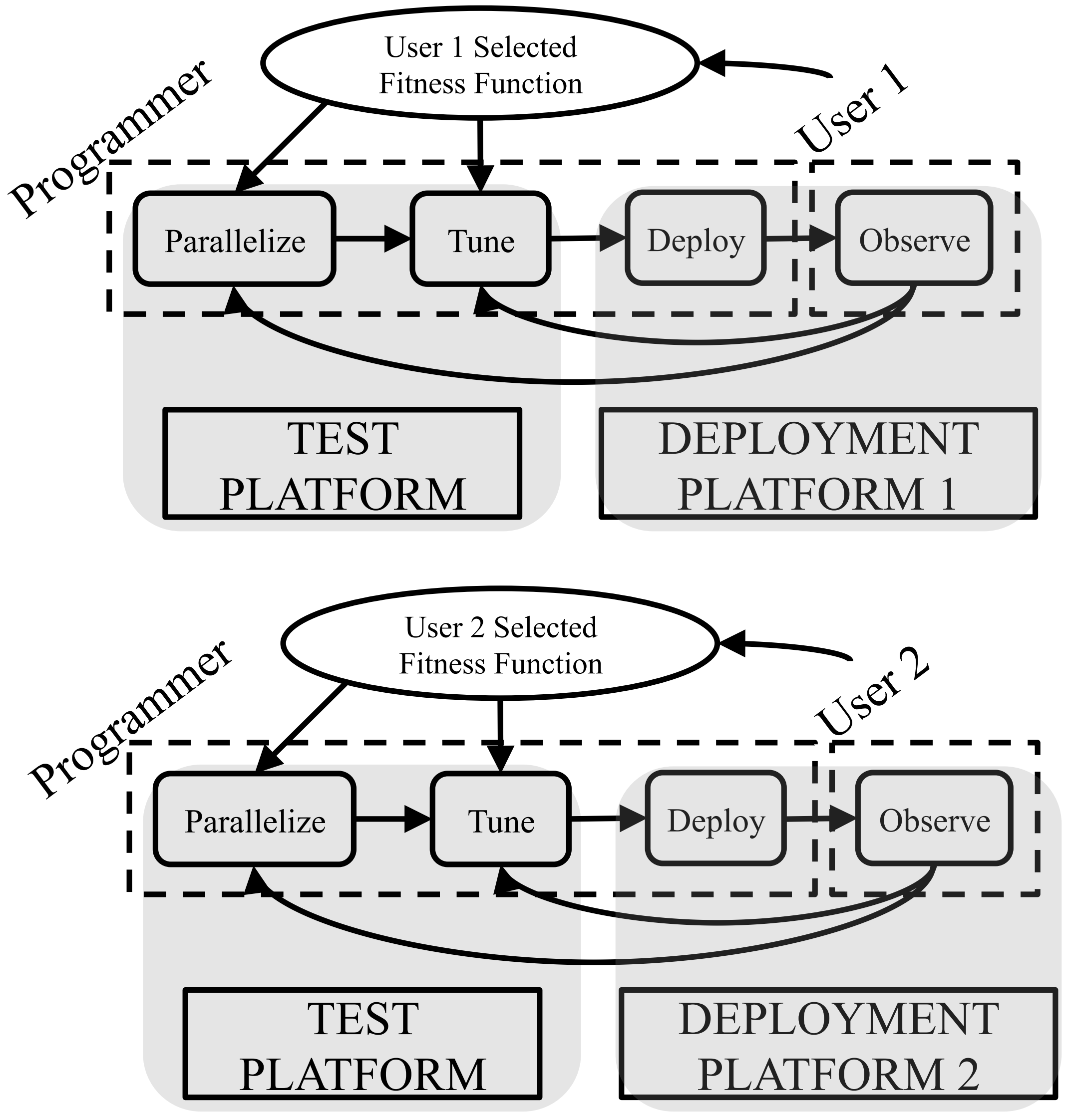


Figure 2.2: As the programmer scales the application to more users, the programmer may need to re-optimize for potentially different platforms and different user expectations of performance.

However, in practice, the programmer is almost always never the end user of the program. The end user may have slightly different expectations of performance, i.e. may impose different fitness functions on the program. The end user may deploy the program on a different platform from the one on which it was developed and tested. The end user may run different workloads on the program. A mismatch in any of these dimensions typically manifests in the program's performance falling short of the user's expectations. The user observes and feeds back the difference in performance to the programmer, who in the best case merely re-tunes the application for the end user's execution environment, and in the worst case ends up re-parallelizing the existing parallel regions or identifying new sources of parallelism. This cyclic process not only adds extra burden on the programmer, but also adds the significant latency of human involvement to the program optimization process.

The problem exacerbates as the programmer scales the number of users (see Figure 2.2). The programmer must simultaneously optimize the program for potentially multiple fitness functions, deployment platforms, and workloads. This leads to a combinatorial explosion in the number of scenarios that the programmer needs to consider.

2.3 Quantifying the Impact of Variability

To put the prior discussion of performance mismatch arising out of differences in platforms, workloads, and user expectations of performance on a quantitative footing, consider the following real-world example of workload variability of a video transcoding server.

Video sharing websites such as YouTube, Google Video, and Dailymotion transcode user submitted videos on their servers. Figure 2.3 shows the parallelism in video transcoding using `x264`, a Pthreads parallel implementation of the popular H.264

standard [101]. Each video may be transcoded in parallel with others. Furthermore, a single video may itself be transcoded in parallel by exploiting parallelism across the frames in the video in a pipelined fashion. $<C_{outer}, C_{inner}>$ represents the parallelization scheme and degree of parallelism (DoP) of the outer (inter-video) and inner (intra-video) loops in the loop nest. Examples of parallelization schemes are data parallel (DOALL) and pipeline parallel (PIPE) [3]. Statically fixing the configuration of each loop may not be optimal for a given performance goal in all execution environments. To demonstrate this, we measured throughput and execution time on a 24-core machine with Intel Xeon X7460 processors. User requests were simulated by a task queueing thread with arrivals distributed according to a Poisson distribution. The average system load factor is defined as the average arrival rate of tasks (videos to be transcoded) divided by the maximum throughput sustainable by the system.

Figure 2.4(a) shows that exploiting intra-video parallelism provides much lower per-video transcoding execution time (T_{exec}) than when only the outer loop is parallelized. T_{exec} is improved up to a maximum of 6.3× on the evaluation platform.

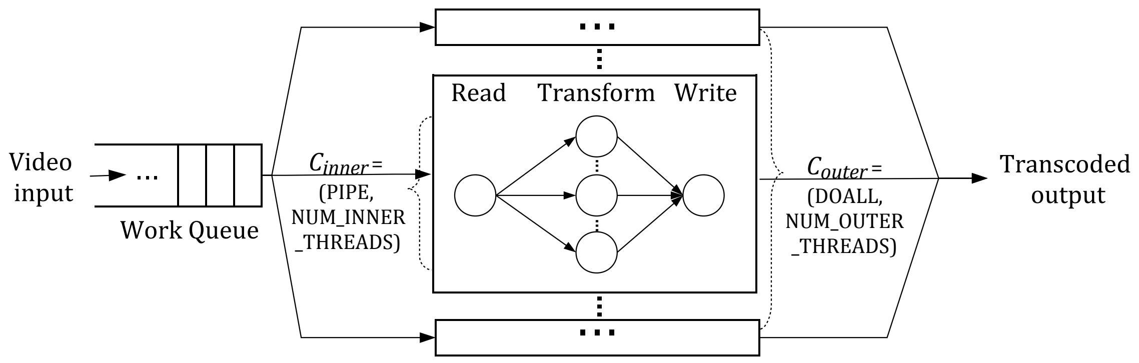
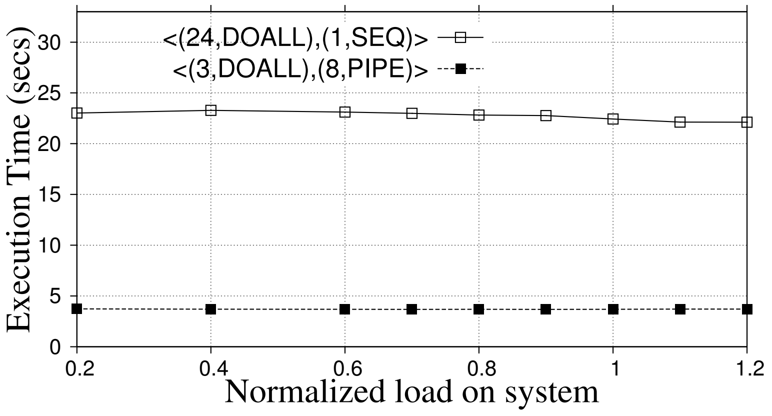
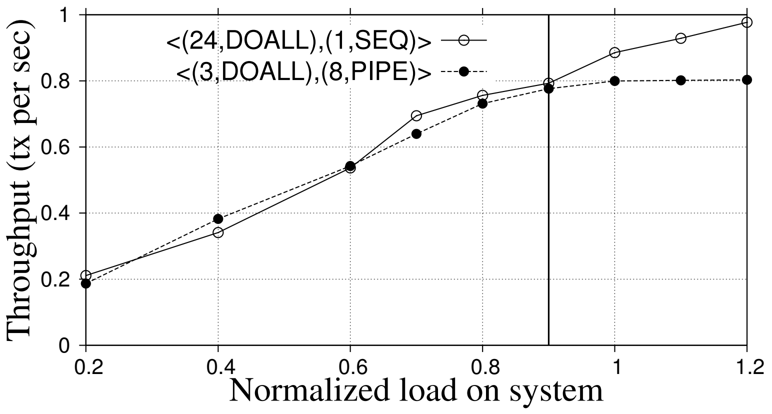

Video input
Work Queue
C_{inner} = (PIPE, NUM_INNER _THREADS)
Read
Transform
Write
C_{outer} = (DOALL, NUM_OUTER _THREADS)
Transcoded output

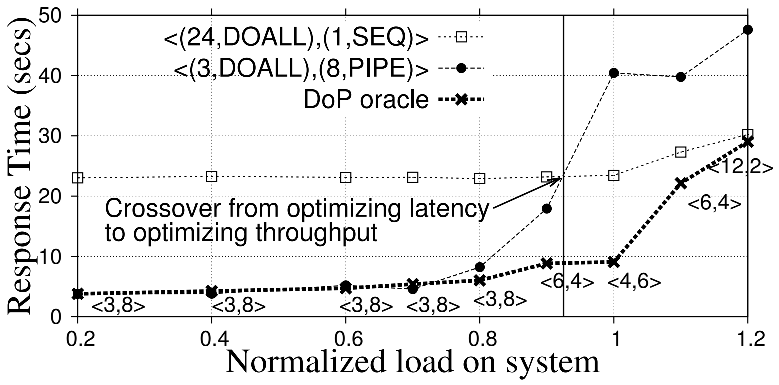
Figure 2.3: A two-level loop nest in video transcoding. Each user-submitted transcoding request gets enqueued in the work queue. Multiple requests are operated upon concurrently; this constitutes the outer level of parallelism. The transcoding of each video itself is done in parallel by a team of threads organized in a pipelined fashion as shown; different stages of the pipeline concurrently operate on different frames of the video. Parallelism Configuration $C = (S, D)$ represents the scheme of parallelization (S) and the number of threads assigned (D, the Degree of Parallelism) to a loop (the parallel region).



(a)



(b)



(c)

Figure 2.4: Variation of (a) execution time and (b) throughput with load factor and parallelism configuration in a video transcoding application on a 24-core Intel Xeon machine. (c) Impact of throughput and execution time on end-user response time; a DoP oracle achieves best response time characteristic by continuously varying DoP with load (ideal parallelism configuration for each load factor is shown). <(k, DOALL), (l, PIPE/SEQ)> indicates that, at any given moment, k iterations of the outer loop are being processed by l threads each, keeping $k \times l$ threads busy. PIPE or SEQ indicates whether each outer-loop iteration is processed in a pipeline-parallel or sequential manner, respectively.

This maximum speedup is achieved when 8 threads are used to transcode each video. Figure 2.4(b), however, shows the dependency of throughput on the application load. At heavy load (load factor 0.9 and above), turning on intra-video parallelism actually degrades throughput. This is due to the inefficiency of parallel execution (a speedup of only about 6$\times$ on 8 threads at load factor 1.0) caused by overheads such as thread creation, communication, and synchronization.

This experiment shows that the usual static choices of parallelism configuration are not ideal across all load factors for *both* execution time and throughput. In other words, there is a tradeoff between the two. This tradeoff impacts end-user response time, which is the primary performance metric of service-oriented applications. Equation 2.1 is helpful to understand the impact of the execution time/throughput tradeoff on response time. The time to transcode a video is the execution time, T_{exec}. The number of videos transcoded per second is the throughput of the system, *Throughput*. The number of outstanding requests in the system's work queue is the instantaneous load on the system, $q(t)$.

$$T_{response}(t) = T_{exec}(C) + \frac{q(t)}{Throughput(C)} \tag{2.1}$$

The response time of a user request, $T_{response}$, is the time interval from the instant the video was submitted for transcoding (at time t) to the instant the transcoded video is output. $T_{response}$ has two components: wait time in the work queue until the request reaches the head of the queue, and execution time, T_{exec}. Note that both T_{exec} and *Throughput* are functions of the parallelism configuration C. At light to moderate load, the average arrival rate is lower than the system throughput. Consequently, the wait time will tend to zero, and $T_{response}$ will be determined by T_{exec}. Assuming reasonably efficient intra-video parallelism, increasing the DoP of the inner loop reduces T_{exec} and in turn $T_{response}$. In other words, in this region of operation, $<C_{outer}, C_{inner}>$ must be optimized for execution time ($C_{inner} = (PIPE, 8)$). At heavy load, $T_{response}$

is dominated by the wait time in the work queue, which is determined by the system throughput. In this region of operation, C_{inner} must be set to a value that optimizes throughput ($C_{inner} = (SEQ, 1)$). Figure 2.4(c) presents experimental validation of the described response time characteristic.

An intelligent programmer can be expected within reason to perform such a trade-off analysis and determine the crossover point (0.9 load factor) when one parallelism configuration becomes better than the other, and rewrite the parallel program to select dynamically at run-time the version that is appropriate for the load regime. However, as Figure 2.4(c) shows, a mere "turn inner parallelism on/off" approach is suboptimal. An oracle that can predict load and change DoP *continuously* achieves significantly better response time. The figure is annotated with the optimal DoP $<D_{outer}, D_{inner}>$ for each load factor obtained by measuring and selecting the best configuration out of all in the parallelism configuration space. The objective of this dissertation is to design a parallel program development methodology and system that can automatically achieve the optimal performance characteristic shown in Figure 2.4(c).

2.4 A Novel Methodology to Tackle Variability

Referring back to Figure 2.1, the fundamental problem with the prevalent parallel program development methodology is that performance observation or monitoring, parallelism tuning, and application re-deployment are all human activities that contribute significant latency to the program optimization process. This additional latency only serves to adversely impact end-user experience. To solve the problem, this dissertation proposes a new methodology shown in Figure 2.5. In the new methodology, the tasks of performance observation or monitoring, parallelism optimization, and program re-deployment are automated by a system called Parcae.

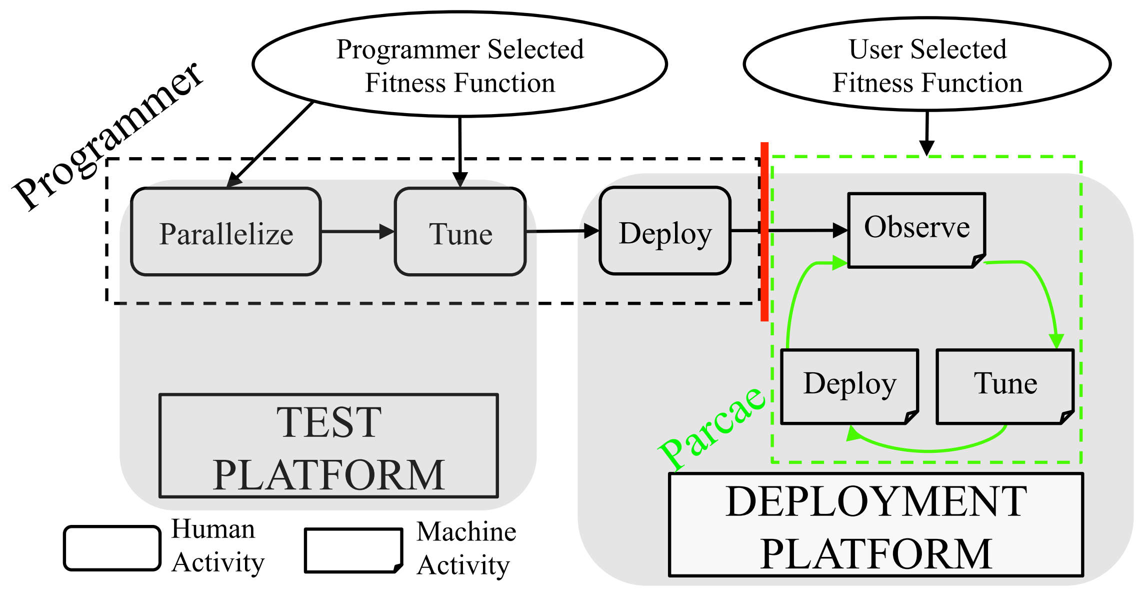


Figure 2.5: Parcae enables separation (indicated by the red vertical bar) of the concern of parallelism discovery and extraction from the concern of optimizing and tuning that parallelism. Parcae automates the tasks of observation, optimization/tuning, and re-deployment to adapt the program to a new execution environment. Automation significantly reduces the latency of optimization and re-deployment of a program.

With Parcae, the problems mentioned before simply do not exist. The programmer can develop a functionally correct parallel program on his/her development platform, and rely on Parcae to adapt the program to the user's deployment platform. The programmer can optionally perform an initial tuning of the parallelism, and rely on Parcae to re-tune if necessary to match the new execution environment. The user now has the flexibility to specify new fitness functions, which Parcae will meet by adapting the program as it it executes. Parcae enables continuous online optimization of the program for the duration of its lifetime. The programmer does not need to do anything special to scale to more users; simply re-deploying on the new platform suffices as Parcae performs the task of re-optimizing to the other users' platforms. In summary, the programmer is freed of the burden of being tied into the process of optimizing a piece of code for its lifetime, and can instead focus on developing new functionality and features.

2.5 Comparison with Existing Systems

Referring to Figure 2.5, development of parallel applications that execute efficiently throughout their lifetime in the face of variability in their execution environment involves:

- correct partitioning of the application into parallel tasks,
- specification of logic for the application to adapt to changes in its execution environment, and
- design of a lightweight run-time system to perform parallelism adaptation.

Most existing systems conflate the above three steps instead of dealing with each separately, resulting in limited portability, extensibility, and reuse of code. We argue that a system that can enable an efficient separation of the steps must have the following properties:

Enables Declarative Parallelism Configuration Expression Libraries such as Pthreads and OpenMP force parallelizers (compilers or programmers) to specify a single fixed configuration of a parallel application—this is an imperative specification. A better approach is for the parallelizers to "declaratively" specify parallelism and not worry about the specific parallelism configuration that the application will adopt since the ideal configuration depends on the application's execution environment.

Performs Dynamic Adaptation Automatically The ideal system should enable the application to adapt automatically to better parallelism configurations without requiring the programmer to manually code the adaptation logic for each application.

Leverages Application Features Prior work has demonstrated the performance benefits of leveraging application level features such as task workload and task execution time [77, 85, 94, 99]. The ideal system should enable the parallelizer to expose such application features.

Leverages Parallelism in Loop Nest The ideal system should enable the expression of all the parallelism in a loop nest since it is crucial for scalability and flexibility to achieve different performance goals (as shown in Section 2.3).

Supports Multiple Parallelism Types The ideal system should enable the expression of multiple parallelism types such as data parallelism, pipeline parallelism, and task parallelism, so that the parallelizer is not constrained to fit the code into a form that is amenable to the system. Rather, the parallelizer should express parallelism in its most natural form for a given application, and the system should adapt the parallelism according to the application's execution environment.

Enables Multiple Performance Goals The ideal system should enable the specification of multiple performance goals including constraints, without necessitating a rewrite of application code. This will ensure the portability of applications across different systems with varying goals and constraints including energy, power, temperature, etc.

Enables Multiple Mechanisms The ideal system should support multiple parallelism optimization mechanisms because the best mechanism for a particular performance goal could be different from the best mechanism for a different performance goal. Additionally, as more constraints, such as power or temperature, are added, the best mechanism will evolve. The system's APIs should be robust to the evolution.

Optimizes Across Multiple Programs General-purpose computing platforms are typically shared by multiple simultaneously executing parallel programs, each of which may be able to utilize all available parallel execution resources. The ideal

Feature \ Library	Pthreads [90]	Intel TBB [77]	FDP [85]	Parcae [This work]
Declarative Parallelism Expression	✗	✓	✓	✓
Automatic Dynamic Adaptation	✗	✓	✓	✓
Leverages Application Features	✗	✗	✓	✓
Supports Parallelism in Loop Nest	✓	✓	✗	✓
Supports Multiple Parallelism Types	✓	✗	✓	✓
Enables Multiple Performance Goals	✗	✗	✗	✓
Enables Multiple Mechanisms	✗	✗	✗	✓
Optimizes Across Multiple Programs	✗	✗	✗	✓

Table 2.1: Comparison of various software-only parallelization libraries for general-purpose applications

system must optimize and efficiently multiplex the execution of all program simultaneously to provide system-wide optimal execution in terms of system-wide throughput, energy consumption, etc.

Table 2.1 evaluates different existing solutions to the dynamic adaptation problem. The columns show how prior systems do not possess the properties discussed above. By contrast, Parcae possesses these properties by design.

Chapter 3

Parcae System Overview

This chapter gives an overview of the proposed Parcae execution model and the Parcae system architecture.

3.1 Parcae Execution Model

In the prevalent execution model, the operating system presents the illusion that a program is running on an unloaded platform. While this abstraction is desirable to reduce the complexity of developing flexible parallel programs, it may result in significant performance degradation when the platform is oversubscribed due to the overheads associated with scheduling the execution of multiple concurrently executing programs. Parcae eliminates the scheduling overheads by re-allocating execution resources (subject to platform constraints such as total number of available cores) when new programs are launched or old programs terminate, and by relying on the programs to adapt themselves to the new resource allocation. Such a co-operative use of resources yields significant performance benefits, as Chapter 8 demonstrates. The primary inhibitor of such execution was the complexity associated with developing flexible parallel programs. Parcae addresses this problem by simplifying the development process for transforming conventional parallel programs into flexible parallel programs through use of the Parcae API, and by automatically transforming sequential programs into flexible parallel programs through the Nona compiler.

Figure 3.1 shows an example of the Parcae execution model on a hypothetical five-core machine. Each parallel region consists of a set of concurrently executing tasks. (The inscription inside a box indicates the task and iteration number of a region; e.g., $M5$ represents the fifth iteration of task M. White space within a core that is not surrounded by the outline of a task is communication or synchronization delay.) At time t_0, program P_1 is launched with a pipeline parallel configuration (PS-DSWP[1]) having three stages corresponding to tasks A, B, and C. A and C are executed sequentially whereas B is executed in parallel by 3 cores as determined by the run-time system. At time t_1, another program P_2 is launched on the same machine. In response, the run-time system signals P_1 to pause at the end of its current iteration (iteration 5). The core receiving this signal (Core 1) acknowledges the signal at time t_2 and propagates the pause signal to the other cores. At time t_3, P_1 reaches a known consistent state, following which the run-time system determines a new allocation of resources to programs P_1 and P_2, say 2 cores to P_1 and 3 cores to P_2. At time t_4, the run-time system launches DOANY[2] execution of both programs P_1 and P_2. For P_1, tasks K and L implement the same functionality as tasks A, B, and C.

In the Parcae execution model, note how a program *reacts* to a change in the resources made available to it. Program P_1 transitions from 5-way PS-DSWP execution to 2-way DOANY execution. General-purpose parallel programs currently lack this capability. Flexible execution, however, does involve certain overheads (such as *Barrier Wait* and *Parallelism Reconfiguration* shown in the figure). Chapter 7 describes the overheads in detail and presents means to alleviate them. Chapter 8 demonstrates that the performance benefits delivered by Parcae far outweigh the overheads.

[1]The PS-DSWP transformation splits a loop body across stages and schedules them for concurrent execution. It enforces dependencies through inter-stage communication channels.

[2]The DOANY transformation schedules loop iterations for parallel execution while synchronizing shared data accesses by means of critical sections.

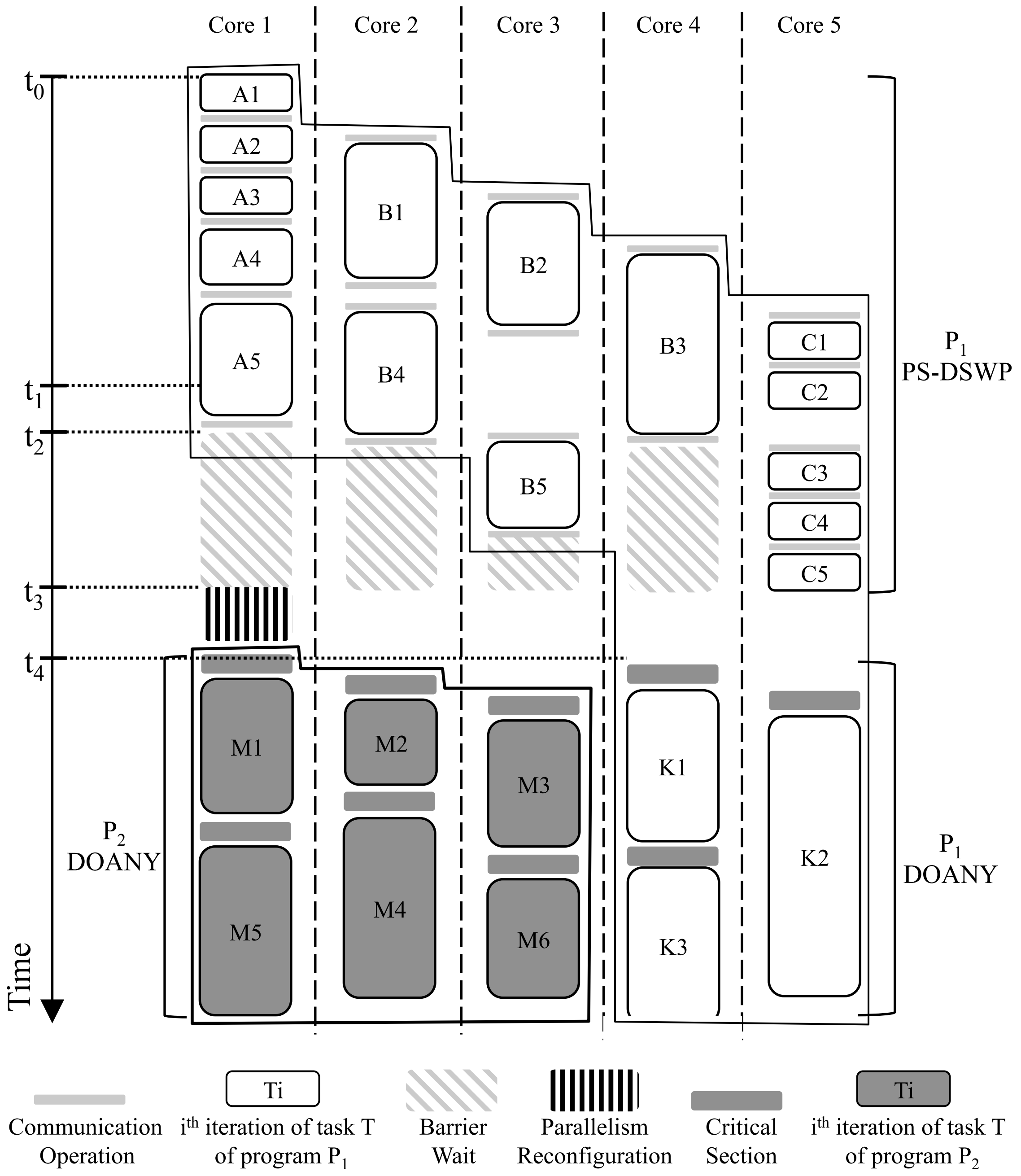


Figure 3.1: Parcae execution model. (t_0) program P_1 is launched; (t_1) program P_2 is launched; (t_2) P_1 acknowledges signal to pause; (t_3) P_1 reaches a known consistent state; (t_4) new resource allocation is determined and parallel execution of P_2 begins, while P_1 switches to a parallelization that is better for two cores.

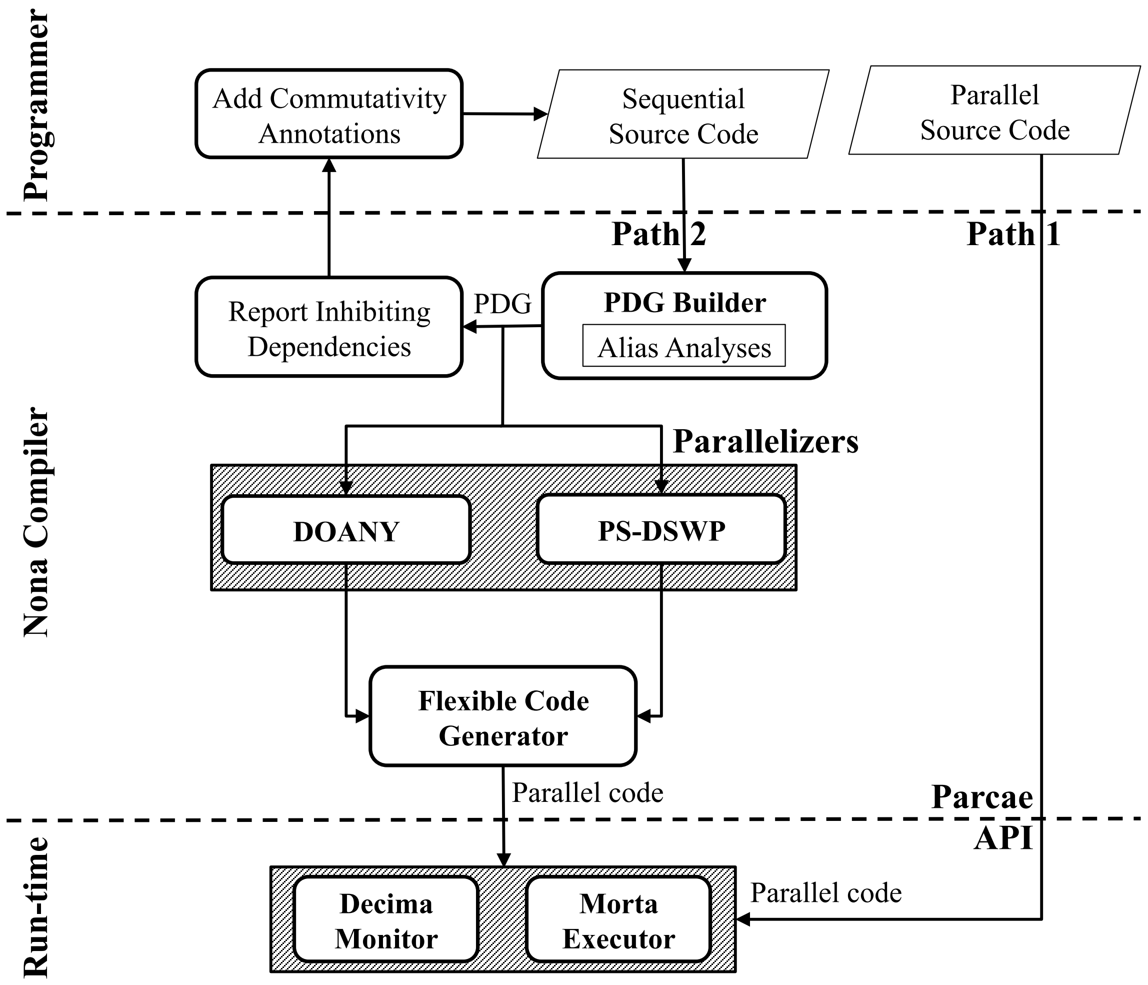


Figure 3.2: Parcae system architecture and parallelization workflow

3.2 Parcae System Architecture

Figure 3.2 shows the Parcae system architecture and parallelization workflow. Chapters that follow will describe each component of the system in detail.

Loops typically constitute the hottest regions of programs, and parallelization efforts are typically targeted at loop nests. Therefore, Parcae targets loop nests. There are two possible workflow paths through the Parcae system, labeled *Path 1* and *Path 2*, respectively, in the figure.

Path 1: The programmer starts from a parallel program or develops a parallel program from scratch, and intends to convert the program into a flexible parallel program. In both cases, the programmer must rewrite the program to use the Parcae API (described in Chapter 5). The Morta run-time system then automatically optimizes/tunes the parallel program at run-time.

Path 2: The programmer starts from a legacy sequential program or writes a new program using the sequential programming model, and intends to convert the program into a flexible parallel program. For this, the Nona compiler automatically rewrites the program into a flexible parallel program.

First, Nona identifies hot loops in the program; such loops are targets of parallelization and are interchangeably called *parallel regions* henceforth.

Second, Nona discovers parallelism by building the program dependence graph (PDG) of the parallel region. The PDG is a graph whose nodes are instructions and edges are dependencies between instructions. Dependencies may be loop-carried or not depending on whether the source and destination of the dependency are on different iterations of a loop or not. Dependencies inhibit parallel execution of the instructions related by them. After constructing the PDG, Nona identifies dependency edges that inhibit parallelization and displays them to the programmer, similar to other parallelizing compilers [71,91,95]. The programmer then inserts commutativity annotations on the source code to relax the sequential order of commuting instructions, or asserts that a dependency does not exist. These annotations serve to remove edges from the PDG, and potentially enable parallelization.

Third, Nona applies multiple parallelizing transforms (parallelizers) to the PDG. In this work, we employ a data-parallel transform with critical sections (DOANY [71, 102]) and a pipeline-parallel transform (PS-DSWP [75, 95]). The system can accommodate additional, new parallelizers. Each parallelizer determines whether it can parallelize the program; if so, it partitions the PDG of the parallel region into a set of tasks. Each task can be sequential (executed by a single thread), or parallel (executed by a team of many threads).

Fourth, the Nona code generator takes as input a partitioning of the PDG nodes into tasks, and outputs multi-threaded code that can be executed flexibly by the Morta run-time system. The Nona code generator is based on the general multi-threaded code generation algorithm (MTCG) described by Ottoni [64], and modifies MTCG for flexible code generation. If multiple parallelizers were able to partition the parallel region, the code generator creates multiple versions of the parallel region. Additionally, the code generator creates a sequential version of the parallel region in case the run-time system deems that sequential execution is most appropriate for a particular execution environment. In summary, the compiler exposes multiple versions of a parallel region corresponding to different execution schemes: SEQ (sequential), DOANY, and PS-DSWP. For DOANY and PS-DSWP, the compiler parameterizes the number of threads in the teams that execute parallel tasks.

Finally, the Parcae run-time system, comprising the Decima monitor and the Morta executor, is responsible for efficient execution of the multi-threaded code output by the compiler. The run-time system optimizes program execution by determining the ideal parallelism configuration from among the choices exposed by the compiler. The run-time system optimizes programs in isolation as well as across the entire platform, as selected by the administrator.

Chapter 4

Compilation for Flexible Execution

The Nona compiler identifies parallel regions in a sequential program and applies a variety of parallelizing transforms to each region, generating multiple parallel versions of code. The generated *flexible code* can be paused during its sequential or parallel execution, reconfigured, and efficiently resumed by the Morta task executor. The compiler also inserts profiling hooks into the generated code for Decima to monitor its behavior. Thus, parallelism opportunities are automatically extracted at compile-time but are monitored and reconfigured at run-time, when information necessary to make informed decisions becomes available.

The parallelizing transforms that we focus on in this work target loop nests as their candidate regions. We therefore describe the methodology and algorithms for the generation of flexible code at this level.

Algorithm 1 summarizes the steps in the compilation process. The following sections describe the steps of the algorithm in detail.

4.1 Parallelism Extraction

The compiler extracts parallelism by building a program dependence graph (PDG) of a loop nest [3, 30]. The PDG is a graph whose nodes are instructions and edges

Algorithm 1: Algorithm to Generate a Flexible Parallel Program

```
Input: Sequential loop nest $L$
Output: Parallel tasks equivalent to $L$
// 1. Extract Parallelism
$G \leftarrow$ buildProgramDependenceGraph($L$)
DAG$_{SCC} \leftarrow$ findStronglyConnectedComponents($G$)
if $|DAG_{SCC}| = 1$ then
    return $L$
// 2. Apply Multiple Parallelizations
tasks $\leftarrow \emptyset$
task$_{\mathrm{SEQ}} \leftarrow L$
tasks $\leftarrow$ tasks $\cup$ task$_{\mathrm{SEQ}}$
parallelizations $\leftarrow$ {PSDSWP, DOANY}
foreach par $\in$ parallelizations do
    $\mathcal{P} \leftarrow$ par.partition(DAG$_{SCC}$)
    tasks$_{\mathrm{par}} \leftarrow$ MTCG($\mathcal{P}$)
    tasks $\leftarrow$ tasks $\cup$ tasks$_{\mathrm{par}}$
// 3. Transform tasks for flexible execution
foreach task $\in$ tasks do
    task $\leftarrow$ changeControlFlow(task)
    task $\leftarrow$ modifyDependencyFlows(task)
    task $\leftarrow$ codegenPauseSignalPropagation(task)
    task $\leftarrow$ insertMonitoringHooks(task)
return tasks
```

are dependencies between instructions. Each dependency is either a data dependency (two instructions access the same register or memory location, not both read-only) or a control dependency (one instruction controls whether the other executes). Register data dependencies are efficiently and precisely computed through data-flow analysis. Memory data dependencies are computed by a pointer analysis suite [49], and control dependencies are computed efficiently based on the *post-dominance* relation [27, 30]. Dependencies inhibit parallel execution of the instructions related by them.

Some data dependency edges in the PDG can be relaxed and their corresponding nodes allowed to execute in parallel by applying special techniques such as privatization and re-association of reduction operations. Nona automatically identifies min, max, and sum reductions. Other dependency edges might be relaxed allowing the corresponding nodes to execute in either order, but not concurrently. Nona processes commutativity annotations provided by the programmer for this purpose [95]. This can be used, for example, to indicate that multiple calls to `rand()` can be legally reordered. The compiler propagates these annotations from the source code to the PDG, relaxing dependencies between commutative operations, and synthesizes the appropriate synchronization to ensure atomicity.

4.2 Multiple Parallelizations

Nona applies multiple parallelizing transforms to the PDG of a loop nest. In this work, we employ a data-parallel transform (DOANY [71,102]) and a pipeline transform (PS-DSWP [75, 95]). The framework can accommodate additional, new transformations. Each transform extracts a distinct form of thread-level parallelism from the given loop nest and produces code packages, called *tasks*, to be executed by concurrent threads. The original, sequential version of the loop is also maintained as a task.

Tasks execute concurrently with each other and communicate or synchronize as necessary to respect program semantics. Each task essentially contains the body of a loop, whose iterations must either execute sequentially or may execute in parallel. A task is labeled as either *sequential* or *parallel* accordingly. Note that parallel execution may involve communication in case of PS-DSWP or synchronization in case of DOANY. A dynamic *instance* of a task refers to any single iteration of a task. Dynamic instances of a sequential task cannot be executed concurrently with each other, whereas dynamic instances of a parallel task can. Note that sequential tasks are also separated into instances; this is done to allow transition from sequential to parallel implementations of a loop, and to support migration of a sequential task due to affinity or load-balancing considerations.

4.3 Task Creation

4.3.1 DOANY

Dependencies may be loop-carried depending on whether the source and destination of the dependency are on different iterations of a loop. DOANY annotates dependencies in the PDG accordingly. Further, the compiler processes commutativity annotations as described earlier, marking appropriate PDG edges as commutative. For DOANY parallelization, such edges are treated as either non-existent or intra-iteration [71]. DOANY treats induction variable updates, min, max, and sum reductions the same way as other commutative operations. DOANY tests the PDG for absence of loop-carried dependencies to determine applicability. If there are no loop-carried dependencies, DOANY extracts the loop into a function. DOANY marshals all loop live-ins into the function via the heap. Finally, DOANY inserts appropriate synchronization operations to ensure atomicity of all commutative operations. A global locking discipline ensures deadlock freedom [71]. The function thus constructed is the DOANY task created by the Nona compiler. The DOANY task is a parallel task, which may be executed by multiple threads at run-time without violating program semantics.

4.3.2 PS-DSWP

PS-DSWP identifies sequential and parallel tasks by building the strongly connected components (*SCCs*) of the PDG. SCCs formed from the PDG form a directed acyclic graph called the DAG_{SCC}. The nodes of the DAG_{SCC} represent groups of instructions that are cyclically dependent on each other; hence, different SCCs constitute tasks that can potentially be executed in parallel provided inter-SCC dependencies are satisfied via appropriate communication. Further, if the dependencies in an SCC are not carried by the loop being transformed, the SCC may be replicated to allow execution as a parallel task.

Nodes of the DAG$_{SCC}$ may consist of only a few instructions. Consequently, nodes are aggregated into larger units before they are exposed as tasks so that the costs of task management at run-time are amortized. To estimate SCC size, Nona employs a heuristic that considers the latency and execution profile weight of each instruction. Only those SCCs of size above a predetermined threshold (SCC_{min}) are exposed as tasks. Other SCCs are coalesced to construct bigger tasks. Various coalescing heuristics have been proposed for pipeline parallel execution (DOACROSS [26], DSWP [65], and PS-DSWP [75, 95]). We have implemented PS-DSWP in our compiler, and use it to construct bigger tasks by coalescing smaller ones.

Coalescence Rules for PS-DSWP

PS-DSWP tries to coalesce the SCCs such that most of the work is performed by a parallel task so as to maximize the potential for independent execution, Additionally, PS-DSWP pipelines the execution of sequential and parallel tasks. PS-DSWP coalesces tasks while maintaining the following invariants:

Invariant 4.3.1 (PS-DSWP Coalescence Invariant) *Tasks $T_1, T_2, ..., T_n$ may be coalesced into tasks $P_1, P_2, ..., P_m$ iff*

1. *$\forall i \in [1, n]$ there exists exactly one $j \in [1, m]$ such that $T_i \in P_j$.*

2. *For every dependency from task T_u to T_v in DAG_{SCC}, with $T_u \in P_i$ and $T_v \in P_j$, we have $i < j$.*

3. *Parallel-tasks T_i and T_j may be coalesced iff there are no tasks $T_k, T_{k+1}, ..., T_{k+l}$ such that there is a dependency chain from T_i to T_j through $T_k, T_{k+1}, ..., T_{k+l}$.*

The first condition preserves correctness by ensuring that operations are not replicated across tasks. The second condition ensures that the resulting tasks form a pipeline. The third condition ensures that coalescing two parallel tasks results in a parallel task.

To create coalesced tasks, PS-DSWP chooses the biggest (in terms of estimated cycles) subset of parallel tasks in the DAG_{SCC} that can be coalesced into a single parallel task without violating the above invariants. It divides the rest of the DAG_{SCC} into a predecessor graph from which dependencies flow into the big parallel task computed previously and a successor graph into which dependencies flow from the big parallel task. Other nodes that do not share dependencies with the big parallel task are distributed to the predecessor and successor subgraphs to balance their weights. The above algorithm is then applied recursively on the predecessor and successor subgraphs to create more coalesced tasks [75].

4.4 Code Generation of Coalesced Tasks

The tasks extracted by each parallelizing transform are initially constructed by applying the multi-threaded code generation (MTCG) algorithm [65,75]. MTCG generates code for each sequential and parallel task including the required inter-task communication and synchronization mechanisms. The algorithm has four steps. First, for each coalesced task, MTCG generates a new control flow graph (CFG) containing the necessary basic blocks for this task. Second, MTCG inserts instructions corresponding to the task in the newly created CFG for that task. Third, MTCG inserts inter-task communication and synchronization instructions. Fourth, MTCG replicates branch instructions into the newly created CFGs as necessary to match the original CFG.

4.5 Flexible Code Generation

The Nona compiler adapts the code generated for each sequential and parallel task in order to support pausing, reconfiguration and resumption of its execution (i.e., its *flexibility*). Originally, the code generated by MTCG targets a fixed number of statically bound threads for each task. However, flexible parallel execution entails dynamic scheduling of task instances across different threads, execution of parallel tasks by a varying number of threads, and pausing a set of tasks followed by resumption of a possibly different set of tasks. Flexible parallel execution is hindered by:

1. Dependencies through private storage: A thread's registers and stack are private. Consequently, cross-iteration dependencies (between one task instance and another) carried by registers and the stack inhibit executing instances across multiple threads.

2. Inter-task communication: MTCG constructs point-to-point communication channels between threads executing tasks, and communicates dependencies in round-robin order across threads. If the number of threads executing each task varies at run-time, the dependencies across the communicating tasks may be reordered, violating sequential semantics.

3. Cross-iteration dependencies: These also inhibit the pausing of executing tasks and relaying of work remaining in the parallel region to a new set of tasks.

To facilitate flexible execution, Nona applies the following changes:

1. Upon completing each iteration, every task yields to the run-time system that determines whether the task should pause or can resume execution on the same thread or a different thread.

2. Registers and stack variables that are live across iterations of sequential tasks are saved and reloaded on the heap at the end and beginning of each iteration, respectively.

3. Parallel tasks avoid having local state across iterations, by sharing cross-iteration data in global memory.

4. When pipelined tasks pause, they flush their communication channels, sending all pending items down the pipeline.

5. Upon resumption, tasks execute an initialization sequence to reload invariant live-in data, and the run-time system resets the communication channels.

These changes are elaborated in the following sub-sections.

4.5.1 Changes to Task Control Flow

Consider the control flow graph CFG_T of an arbitrary task T, as generated by MTCG (Figure 4.1(a)). CFG_T represents a single-entry-single-exit code region containing a loop with a single `tail`→`header` backedge. There is a single entry edge reaching `header` block from outside the loop; there may be multiple exits from the loop, but all reach a single `exit` block, which cannot be reached from outside the loop. The restrictions on CFG_T simplify code generation, but can be relaxed.

Nona modifies CFG_T to support migration (see Figure 4.1(a)). The backedge is redirected from `tail` to a new exit block, which returns `task_iterating` (instead of reaching the `header`). The original `exit` block now returns `task_complete`. Section 4.6 discusses the third (and last) exit block that is reached from a new pre-header block and returns `task_paused`.

Algorithm 2: Control logic for executing task instances

```
// getTaskInstance() blocks until reconfiguration ends or until next region begins. It
   returns NULL when program ends.
while instance ← runtime.getTaskInstance() do
    retVal ← invoke(instance.function, instance.args)
    if retVal == task_iterating then
        taskIterCount[instance.getTaskID()]++
    else
        // retVal == task_paused or task_complete
        region ← instance.getRegion()
        region.waitOnBarrier()
        if (retVal == task_complete) ∧ (isMasterTask(instance)) then
            region.terminate()
```

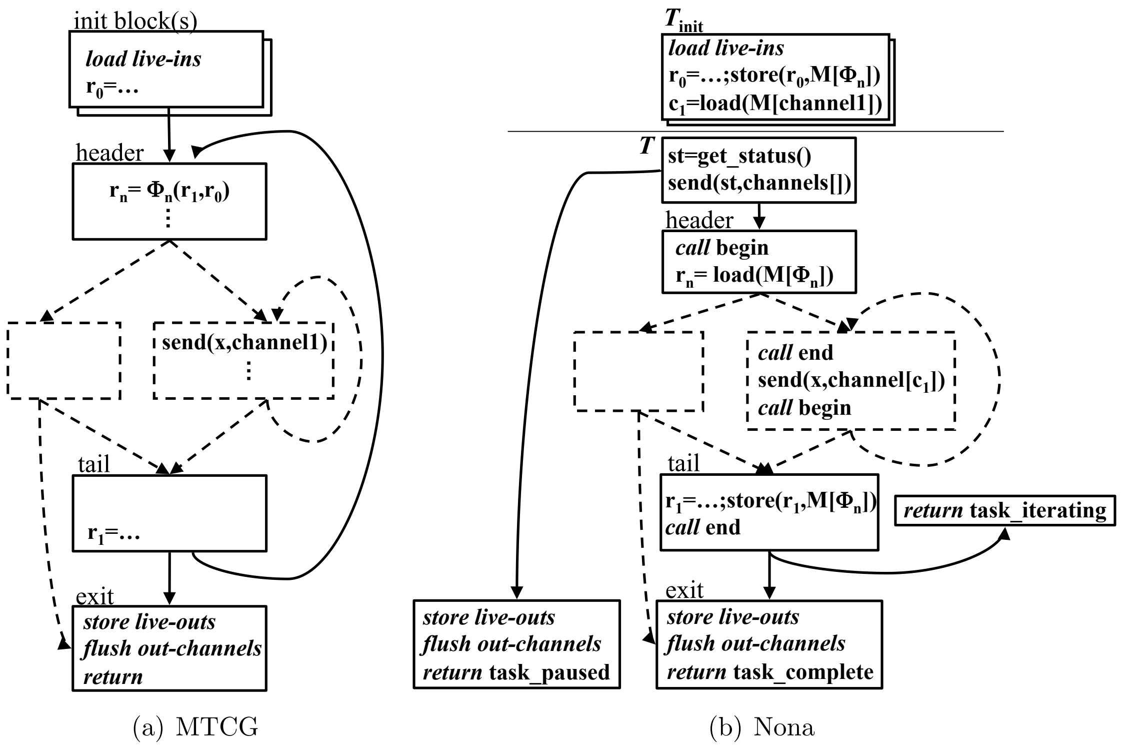


Figure 4.1: Transforming task code produced by MTCG. Dashed blocks/arcs represent arbitrary control flow within the loop.

The control logic to execute task instances is extracted into a loop, shown in Algorithm 2. The Morta executor sets up every worker thread to execute this loop. Upon receiving `task_iterating`, the thread increments a counter that tracks the number of iterations per task. Upon completing or pausing a task, the thread waits for other tasks of the region to complete or pause by means of a barrier, before starting to execute a new task.

4.5.2 Saving and Restoring State

Sequential tasks may have cross-iteration dependencies that flow through registers and variables on the stack, which are local to a thread. To facilitate lightweight migration to another thread, Nona inserts code to copy such variables to the heap at the end of each iteration and reload them at the beginning of each iteration. Note that the amount of information that needs to be copied through the heap is typically much smaller when applied between iterations, than at arbitrary locations, as in general context switches or checkpoints.

Nona uses static single assignment (SSA) form to represent code. In SSA form, loop-carried register dependencies are captured by ϕ nodes in the loop `header`. Figure 4.1 shows how value flows through registers are converted into flows through the heap. Note that a register value need only be stored to the heap at the end of an iteration, not on every write to the register. This minimizes the cost of saving register state. A similar treatment addresses stack variables. The figure also shows how blocks preceding the loop header are extracted into a separate function (T_{init}). This function includes the loading of loop-invariant live-in values, and will be executed at every task activation and resumption. Algorithm 3 summarizes the above changes.

Algorithm 3: Changes to task to enable migration, pausing, and resumption

```
Input: Control flow graph $CFG_T$ of task $T$
Output: Modified code for $T$ that can migrate across threads
// 1. Handle loop-carried register dependencies
$h \leftarrow$ getLoopHeader($CFG_T$)
foreach phinode $\in h$ where phinode $= r_n \leftarrow \phi(r_1, r_2)$ do
    $M \leftarrow$ allocateHeapMemory(getType(phinode))
    loadinst $= r_n \leftarrow$load $[M]$
    replace(phinode, loadinst)
    foreach $t_i \leftarrow$ getPredecessors($CFG_T, h$) do
        storeinst $=$ store $[M], r_i$
        terminst $=$ getTerminator($CFG_T, t_i$)
        insertBefore(terminst, storeinst)
// 2. Execute pre-header once
$S \leftarrow$ getPredecessors($CFG_T, h$) $\setminus \{t\}$
while $b \leftarrow$ extractNewElementFrom($S$) $\neq \emptyset$ do
    $S \leftarrow S \cup$ getPredecessors($CFG_T, b$)
$T_{\text{init}} =$ extractBlocksIntoNewFunction($S$)
```

4.5.3 Inter-task Communication

When extracting pipeline parallelism, dependencies between tasks executing different stages concurrently need to be enforced. MTCG communicates such dependencies by inserting instructions for *send-receive* operations over point-to-point communication channels [65, 75]. Point-to-point communication (e.g., via single-producer-single-consumer queues) has lower contention than point-to-multipoint (or broadcast) communication (e.g., via single-producer-multipler-consumer queues), and allows maintenance of sequential program order.

Communication between two sequential tasks is straightforward; communication between a sequential task (S) and a parallel task (P) involves data arbitration and merge. Consider a dependency edge $u \rightarrow v$ where $u \in S$ and $v \in P$. Let p be the (varying) number of threads that execute P. The value produced by the thread executing S flows to each of the threads executing P in a round-robin fashion: on the i^{th} instance of S, the value is communicated through the $(i \bmod p)^{th}$ channel. MTCG uses the induction variable (i), which is incremented once per instance of S, to identify the channel for a value that flows over the dependency edge. This holds analogously for dependency edges from a parallel task to a sequential task.

In MTCG, the number p of threads that execute a parallel task P is fixed at compile-time. This is not the case for Nona because the value of p may be changed during execution by Morta. Still, the above communication mechanism suffices, provided p and the communication channels are maintained as run-time parameters and task instances are made *relayable*.

4.6 Relayable Task Instances

As described in Section 4.5.2, parallel tasks are constructed to have no local cross-iteration dependencies, and sequential tasks have all their cross-iteration data placed in the global heap between iterations. Thus, every task instance can halt after finishing or before starting an iteration, leaving the program in a known consistent state. Once instances halt, the system can be reconfigured safely and subsequent instances will continue to execute according to the new set of tasks and their thread allocation.

At the beginning of each iteration, every instance checks for a pause signal (see `get_status()` in Figure 4.1(a)), received either directly from Morta or from another instance. If an instance receives a pause signal, it propagates the signal and yields to Morta by returning `task_paused`. These signals are initiated and propagated as follows.

Initiating pause signals: When Morta chooses to reconfigure the program, it sends a pause signal to designated *master* tasks. In the DOANY and sequential versions, the single task is the master task, and in the case of PS-DSWP, only the task executing the first stage is designated as master. The master tasks are designed to query Morta for pause signals at the beginning of each instance, to which Morta responds with either `continue` or `pause`.

Propagating pause signals across pipelined tasks: The master task of a pipeline-parallel region relays the `continue` or `pause` notification to all other tasks in the region, following the structure of the pipeline. This is achieved by placing *send* instructions to all directly connected tasks in the loop header of the master task (before any other instruction, see `send(st,channels[])` in Figure 4.1(a)), and matching *receive* instructions (via `get_status()`) in the corresponding location of each of the connected tasks. Subsequent send-receive messages are placed analogously down the stages of the pipeline. This ensures that all parts of an iteration, scattered across pipeline stages, pause appropriately.

All send-receive operations occur over the same point-to-point communication channels used for dependency flows. Upon receiving a `pause` signal, a task explicitly flushes all outgoing channels, transmitting all pending items down the pipeline. This ensures that all channels are drained properly, relying on the property that at the end of a flushed iteration, all relevant incoming data has been received and processed.

Pausing process: Upon receiving a `pause` signal, a task exits the parallel region by jumping to an exit block that returns `task_paused`. Exiting a parallel region on a pause is identical to exiting the region upon reaching the end of the loop. Indeed, the block returning `task_paused` contains the same instructions as the block returning `task_complete`. These instructions include flushing outgoing channels (see Figure 4.1(a)). After exiting the iteration, the thread waits on a barrier for other threads executing the parallel region to exit as well (see Algorithm 2).

To resume execution, Morta launches the set of tasks determined to be optimal for the new execution environment. Each task first executes T_{init} when launched (the task initialization function, see Figure 4.1(a)).

Chapter 7 discusses pause-resume overhead as well as the means to alleviate it.

4.7 Hooks for Autonomous Monitoring

An important aspect of Parcae is task execution time monitoring by the Decima monitor. Decima distinguishes between the time a task spends computing and the time it spends waiting for communication, possibly across multiple parallel instances. Morta relies on this information to optimize program configurations. To enable such monitoring, Nona inserts `begin` and `end` hooks into the code of each task. These hooks obtain timestamps using the `rdtsc` instruction on x86 platforms (whose overhead is presented in Section 8.3.6). Nona inserts `end` immediately before each *send* and *receive* instruction, and `begin` immediately after each *send* and *receive* instruction. Nona also inserts `begin` into the entry block of a task and `end` into the `task_executing` exit block.

The `begin` and `end` hooks help calculate the total compute-time of an instance by accumulating local time intervals (between consecutive {`begin`, `end`} pairs). These local compute-times of each instance then update a global compute-time counter per task, which feeds Decima, requiring no inter-thread synchronization. Note that the total execute-time of an instance can easily be reported too, as the time elapsed between the first `begin` and the last `end`, from which the communication overhead can be derived. The latter may be important for affinity and allocation optimization purposes; however, these effects were not significant on our evaluation platforms.

Chapter 5

Parcae for the Programmer

A programmer can use the Parcae API to develop a new parallel program, or port an existing parallel program implemented using libraries such as Pthreads, OpenMP, or TBB to the Parcae API. Section 5.1 describes the API in detail and walks through an example illustrating the port of an application parallelized using Pthreads to use the Parcae API. Section 5.2 describes the interface to the system administrator. Finally, Section 5.3 describes the interface that a mechanism developer must implement to encode run-time program optimization logic (the system comes with a default optimizer that the administrator can choose to employ out of the box).

5.1 Application Developer's View

Parcae presents a task-oriented interface akin to Intel Threading Building Blocks [77] to the application developer.

5.1.1 Datatype Definitions

A task consists of a template function that abstracts the control for creating dynamic instances of each task, function objects (functors) that encapsulate the task's func-

```
Task = {control: TaskExecutor, function: Functor, load: LoadCB,
          desc: TaskDescriptor, init: InitCB, fini: FiniCB}

TaskDescriptor = {type: TaskType, pd: ParDescriptor[ ]}

TaskType = SEQ | PAR

ParDescriptor = {tasks: Task[ ]}

TaskStatus = task_iterating | task_paused | task_complete
```

Figure 5.1: Parcae API type definitions

```
1 template<Functor>
2 void TaskExecutor(Functor
3                  Function){
4   ...
5   while(true) {
6     ...
7     TaskStatus status =
8                 Function();
9     ...
10  }
11 }
```

(a) Control flow abstraction

```
1 class Functor{
2   ... //Capture local variables
3
4   ... //Constructor
5
6   TaskStatus operator()(){
7     ... //Task function body
8     return taskstatus;
9   }
10 };
11
```

(b) Functor for task functionality

Figure 5.2: Separation of task's control and functionality in the Parcae API

tionality and expose application level information, and a descriptor that describes the parallelism structure of the task. Figure 5.1 defines the `Task` type and the types from which it is composed.

`TaskExecutor` Parcae provides the control flow abstraction shown in Figure 5.2(a). This is the same as the task instantiation loop shown in Algorithm 2 in Section 4.5.1. Loop exit is determined by `status` (line 7 in Figure 5.2(a)). The abstraction is parameterized on the `Functor` type that encapsulates a task's functionality.

`Functor` The developer must implement a functor that encapsulates the desired functionality of a task. The functor binds the local variables of the original method containing the parallelized loop as member fields (line 2 in Figure 5.2(b)). At run-time, a task could be either iterating normally, paused, or completed. The functor must return the status of the task after each instance of the task (line 8 in Figure 5.2(b)). In particular, when a loop exit branch is to be taken, the functor must return `task_complete`; otherwise, the functor must return `task_iterating`. Combined with the control flow abstraction in Figure 5.2(a), the control flow structure of the original loop is duplicated. The functor can also return `task_paused`—its discussion is deferred until Section 5.1.2.

`LoadCB` Chapter 2 described the importance of application features, such as workload, to determine the optimal parallelism configuration for a given performance goal. To capture the workload on each task, the developer implements a callback functor that, when invoked, returns the current load on the task.

`InitCB` and `FiniCB` To restart parallel execution from a globally consistent program state after the Parcae run-time system reconfigures parallelism, Parcae requires the programmer to implement the `InitCB` (`FiniCB`) functor that is invoked exactly once before (after) the task is resumed (paused). The `InitCB` is the same as the initialization function T_{init} generated by the compiler (refer to Section 4.5.2), while the `FiniCB` contains code that performs the same set of actions as are performed when the task completes (refer to Section 4.6). Figure 4.1(a) in Chapter 4 already illustrated these changes.

`TaskDescriptor` A task can be sequential (`SEQ`) or parallel (`PAR`). A parallel task's functionality can be executed by one or more threads. In other words, the `Functor()` method (lines 6–9 in Figure 5.2(b)) can be invoked concurrently by multiple threads. To enable description of nested parallelism, a task can specify one or more parallelism descriptors (`ParDescriptor`). Specifying more than one descriptor exposes a choice to the Parcae run-time system, which chooses the optimal parallelism configuration (described by the corresponding `ParDescriptor`).

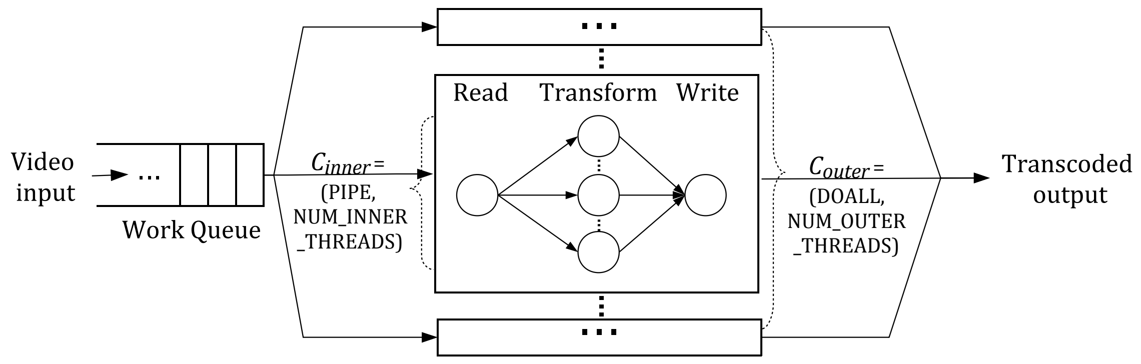

Video input
Work Queue
C_{inner}= (PIPE, NUM_INNER _THREADS)
Read
Transform
Write
C_{outer}= (DOALL, NUM_OUTER _THREADS)
Transcoded output

Figure 5.3: A two-level loop nest in video transcoding. Multiple requests are operated upon concurrently; this constitutes the outer level of parallelism. The transcoding of each video itself is done in parallel by a team of threads organized in a pipelined fashion as shown; different stages of the pipeline concurrently operate on different frames of the video.

`ParDescriptor` A parallelism descriptor is defined recursively in terms of `Tasks`. A `ParDescriptor` is an array of one or more tasks that execute in parallel and potentially interact with each other (line 8 in Figure 5.1).

```
void Transcode(){
  Q∗ inq, outq;
  Video∗ input, ∗output;
  while(true){
    ∗input = inq→deque();
    output = transcode(input);
    outq→enqueue(∗output);
  }
}
```

Figure 5.4: Outer loop in x264 video transcoding

Putting it all together Figure 5.4 shows the outer loop code in x264 video transcoding. Figure 5.5 shows the transformation of the loop by instantiation of the Parcae types discussed above. In Figure 5.5(a), duplicated code from the original loop in Figure 5.4 is shown in bold. As discussed in Chapter 2 and shown in Figure 5.3, the outer loop task in video transcoding can itself be executed in a pipeline parallel fashion. Figure 5.5(c) shows the definition of the outer loop task descriptor in terms of the inner loop parallelism descriptor. Figure 5.5(d) shows the definition

```
class TranscodeFunctor{
 //Capture local variables
  Queue∗& inq;
  Queue∗& outq;
  ... //Constructor
  TaskStatus operator()(){
    Video* input, *output;
    *input = inq→deque();
    output = transcode(input);
    outq→enqueue(*output);
    return EXECUTING;
  }
};
```

(a) Functionality

```
class TranscodeLoadCB{
  //Capture local variables
  Queue∗& inq;
  Queue∗& outq;
  ... //Constructor
  float operator()(){
    //Return occupancy
    return inq→size();
  }
};
```

(b) Workload

```
TaskDescriptor
 ∗readTD(SEQ, NULL),
 ∗transformTD(PAR, NULL),
 ∗writeTD(SEQ, NULL);
 ...//Create tasks
   //using descriptors
ParDescriptor
 ∗innerPD({readTask,
            transformTask,
            writeTask});
TaskDescriptor
 ∗outerTD(PAR, {innerPD});
```

(c) Descriptor

```
void Transcode(){
  Queue∗ inq, ∗outq;
  Task∗ task
   (TranscodeFunctor(inq, outq),
    TranscodeLoadCB(inq, outq),
    outerTD);
  ParDescriptor∗ outerPD({task});
  Parcae.launch(system, outerPD);
}
```

(d) Task

Figure 5.5: Task definition using Parcae

Method	Description
`TaskStatus Task::getStatus()`	Check for pause signal; Parcae returns `continue` or `pause`
`void Task::begin()`	Inform Parcae that the CPU intensive part of the task has begun
`void Task::end()`	Inform Parcae that the CPU intensive part of the task has ended
`void Task::wait()`	Wait until child tasks complete; Parcae returns status of master child task
`Parcae* Parcae::create()`	Create and initialize the Parcae run-time system
`void Parcae::destroy(Parcae* system)`	Finalize and destroy the Parcae run-time system
`void Parcae::launch(Parcae* system, ParDescriptor* pd)`	Launch parallel region described by specified parallelism descriptor under the Parcae run-time system; wait for parallel region to end

Table 5.1: Parcae API

of the outer loop parallelism descriptor and how the outer loop is launched for execution under the control of the Morta executor. Note that the process of defining the functors is mechanical; it can be simplified with compiler support such as that for lambdas in Intel's C++0x compiler [38].

5.1.2 Using the API: A Video Transcoding Example

A developer uses the types in Figure 5.1 and associated methods in Table 5.1 to enhance a parallel application. Figure 5.7 describes the port of a Pthreads based parallelization (column 1) of the video transcoding example to the Parcae API (column 2). Code that is common between the Pthreads and Parcae versions is shown in bold.

Step 1: Parallelism Description In the Pthreads parallelization, lines 4–8 create `NUM_OUTER` threads that execute the `Transcode` method. In the `Transcode` method, a thread dequeues work items (videos) from the work queue (line 15), transcodes them (lines 16–26), and enqueues the transcoded items to the output queue (line 27). Each video transcoding can itself be done in parallel in a pipelined fashion. For this, the `Transcode` method spawns `NUM_INNER` threads to execute the pipeline. One thread each executes `Read` and `Write`, and one or more threads execute `Transform`. A common practice is to set both `NUM_OUTER` and `NUM_INNER` statically based on profile information [61]. Chapter 2 already presented the shortcomings of this approach.

In the Parcae parallelization, the application's parallelism is described in a modular and bottom-up fashion. Line 4 gets the task definition of the outer loop by invoking `Transcode_getTask`. To encode nested parallelism, the `Transcode_getTask` method specifies that `Transcode` can be executed in parallel using the parallelism descriptor `pd` (lines 13–18 in `Transcode_getTask`). Line 5 in `transcodeVideos` creates a parallelism descriptor for the outer loop.

Step 2: Parallelism Registration Line 6 in `transcodeVideos` initializes the Parcae run-time system. Line 7 registers the parallelism descriptor for execution by Morta by invoking `Parcae::launch`. `Parcae::launch` waits for the parallel region to finish. Finally, line 8 frees up execution resources by invoking `Parcae::destroy`.

(a) Parallelization using POSIX threads | **(b) Parallelization using Parcae**

(1) Run-time initialization

(a)
```
 1 #include <pthread.h>
 2 void transcodeVideos() {
 3    Q* inq, *outq;
 4    pthread_t threads[NUM_OUTER];
 5    for (i = 0 ; i < NUM_OUTER ; i++) {
 6       pthread_create(threads[i], attr, Transcode,
 7                               new ArgT(inq, outq));
 8    }
 9    ... // Join threads
10 }
```

(b)
```
 1 #include <dope>
 2 void transcodeVideos() {
 3    Q* inq, *outq;
 4    Task* outerTask = Transcode_getTask(inq, outq);
 5    ParDescriptor* outerPD = new ParDescriptor({outerTask});
 6    Parcae* system = Parcae::create();
 7    Parcae::launch(system, outerPD); // Wait for parallel region to finish
 8    Parcae::destroy(system);
 9 }
10
```

(2) Transcoding of an individual video clip

(a)
```
11 void* Transcode(void* arg) {
12    Q* inq = (ArgT*)arg->inq;
13    Q* outq = (ArgT*)arg->outq;
14    for(;;) {
15       *input  = inq->dequeue();
16       ... //Initialize q1 and q2
17       pthread_t threads[NUM_INNER];
18       pthread_create(threads[0], attr, Read,
19                               new ArgR(input, q1));
20        for (i = 1 ; i < NUM_INNER – 1 ; i++) {
21          pthread_create(threads[i], attr,
22                   Transform, new ArgTr(q1, q2));
23       }
24       pthread_create(threads[NUM_INNER-1],
25             attr, Write,  new ArgW(q2, output));
26       ... // Join threads
27       outq->enqueue(*output);
28    }
29 }
```

(b)
```
11 class TranscodeFunctor {
12    Task* task; //This functor's task
13    ... //Capture local variables
14    ... //Constructor
15    TaskStatus operator()() {
16       *input = inq->dequeue();
17       ... //Initialize q1 and q2
18        status = task->wait();
19        if (status == task_paused)
20           return task_paused;
21        outq->enqueue(*output);
22       return task_iterating;
23    }
24    friend Task* Transcode_getTask(...);
25 };
26
27
28
29
```

```
11 Task* Transcode_getTask(Q* inq, Q* outq) {
12    TranscodeFunctor* func = new TranscodeFunctor(inq,outq);
13    ParDescriptor* pd = new ParDescriptor
14       ({Read_getTask(func->q1),
15          Transform_getTask(func->q1, func->q2),
16          Write_getTask(func->q2)});
17    // Note hierarchical description of parallelism
18    TaskDescriptor* td = new TaskDescriptor(PAR, {pd});
19    Task* task = new Task(func, new TranscodeLoadCB(inq),
20                                  td, NULL, NULL);
21    func->task = task;
22    return task;
23 }
24 class TranscodeLoadCB {
25    Q* inq;
26    TranscodeLoadCB(Q* inq) : inq(inq) {}
27    double operator()() {return inq.size();}
28 };
29
```

Figure 5.6: Comparison of parallelization using POSIX threads and Parcae—continued on next page

(3) Stages of pipeline to transcode an individual video clip

```
30 void* Read(void* arg) {
31   … //Get input and q1 from arg
32   for(;;) {
33       frame = readFrame(*input);
34       if (frame == NULL) break;
35           q1->enqueue(frame);
36   }
37   q1->enqueue(NULL);
38 }
39
40
41
42
43
44
45
46
47
48 void* Transform(void* arg) {
49    … //Get q1 from arg
50    for(;;) {
51        frame = q1->dequeue();
52        if (frame == NULL) break;
53        frame = encodeFrame(frame);
54        q2->enqueue(frame);
55    }
56    q2->enqueue(NULL);
57 }
58
59
60
61
62
63
64
65
66 void* Write(void* arg) {
67    … //Get q2 and output from arg
68    for(;;) {
69        frame = dequeue(q2);
70        if (frame == NULL) break;
71        writeFrame(output, frame);
72    }
73 }
74
75
76
77
78
79
80
```

```
30 class ReadFunctor {
31    Task* task; //This functor's task
32    ... //Capture local variables
33    ... //Constructor
34    TaskStatus operator()() {
35        status = task->getStatus();
36        if (status == task_paused)
37            return task_paused;
38        task->begin();
39        frame = readFrame(*input);
40        if (frame == NULL)
41            return task_complete;
42        task->end();
43        q1->enqueue(frame);
44        return task_iterating;
45    }
46    friend Task* Read_getTask(...);
47 };
48 class TransformFunctor {
49    Task* task; //This functor's task
50    ... //Capture local variables
51    ... //Constructor
52    TaskStatus operator()() {
53        frame = q1->dequeue();
54        if (frame == null)
55            return task_complete;
56        task->begin();
57        frame = encodeFrame(frame);
58        task->end();
59        q2->enqueue(frame);
60        return task_iterating;
61    }
62    friend Task* Transform_getTask(...);
63 };
64
65
66 class WriteFunctor {
67    Task* task; //This functor's task
68    ... //Capture local variables
69    ... //Constructor
70    TaskStatus operator()() {
71        frame = q2->dequeue();
72        if (frame == null)
73            return task_complete;
74        task->begin();
75        writeFrame(output, frame);
76        task->end();
77        return task_iterating;
78    }
79    friend Task* Write_getTask(...);
80 };
```

```
30 Task* Read_getTask(Q* q1) {
31     ReadFunctor* func = new ReadFunctor(q1);
32     TaskDescriptor* td = new TaskDescriptor(SEQ, NULL);
33     Task* task = new Task(func, NULL, td, NULL,
34                                          new ReadFiniCB(q1));
35     func->task = task;
36     return task;
37 }
38
39 class ReadFiniCB {
40     Q* q1;
41     ReadFiniCB(Q* q1) : q1(q1) {}
42     void operator()() {q1->enqueue(NULL);};
43 };
44
45
46
47
48 Task* Transform_getTask(Q* q1, Q* q2) {
49     TransformFunctor* func = new TransformFunctor(q2);
50     TaskDescriptor* td = new TaskDescriptor(PAR, NULL);
51     Task* task = new Task(func, new TransformLoadCB(q1),
52                           td, NULL, new TransformFiniCB(q2));
53     func->task = task;
54     return task;
55 }
56 class TransformFiniCB {
57     Q* q2;
58     TransformFiniCB(Q* q2) : q2(q2) {}
59     void operator()() {q2->enqueue(NULL);}
60 };
61 class TransformLoadCB {
62     Q* q1;
63     TransformLoadCB(Q* q1) : q1(q1) {}
64     double operator()() {return q1.size();}
65 };
66 Task* Write_getTask(Q* q2) {
67     WriteFunctor* func = new WriteFunctor(q1);
68     TaskDescriptor* td = new TaskDescriptor(SEQ, NULL);
69     Task* task = new Task(func, new WriteLoadCB(q2), td,
70                                        NULL, NULL);
71     func->task = task;
72     return task;
73 }
74 class WriteLoadCB {
75     Q* q2;
76     WriteLoadCB(Q* q2) : q2(q2) {}
77     double operator()() {return q2.size();}
78 };
79
80
```

Figure 5.7: Comparison of parallelization using POSIX threads and Parcae

Step 3: Application Monitoring Each task marks the begin and end of its CPU intensive section by invoking `Task::begin` and `Task::end`, respectively. The run-time system records application features such as task execution time in between invocations of these methods. To monitor per-task workload, the developer implements `LoadCB` for each task to indicate the current workload on the task. The callback returns the current occupancy of the work queue in the case of the outer task (line 27), and the input queue occupancies in the case of `Transform` (line 64) and `Write` (line 77). The callbacks are registered during task creation time.

Step 4: Task Execution Control If a task returns `task_iterating`, Morta continues the execution of the loop. If a task returns `task_complete`, Morta waits for other tasks that are at the same level in the loop nest to also return `task_complete`. A task can explicitly wait on its children by invoking `Task::wait`. Recall from Section 4.6 that exactly one task in each parallelized loop is assigned the role of the *master task* (the first task in the array of tasks registered in the `ParDescriptor`). In the running example, the task corresponding to `Transcode` is the master task for the outer loop and the task corresponding to `Read` is the master task for the inner loop. Invoking `Task::wait` on `task` (line 18) returns the status of the master child task.

Step 5: Task Yielding for Reconfiguration By default, Morta returns `task_iterating` when `Parcae::getStatus` is invoked (line 35 in `ReadFunctor`). When Morta decides to reconfigure parallelism, it returns `task_paused`. The application should check this condition (lines 35–36 in `ReadFunctor`), and then enter a globally consistent state prior to reconfiguration. The `FiniCB` callbacks are used for this purpose. In this particular example, `Read` notifies `Transform` (via the `ReadFiniCB` callback), which in turn notifies `Write` (via the `TransformFiniCB` callback). The notifications are by means of enqueuing a sentinel `NULL` token to the in-queue of the next task. Note, by comparing the Pthreads (lines 37 and 56) and Parcae versions (lines 42 and 59), that the developer was able to reuse the thread termination mechanism from the Pthreads parallelization to implement the `FiniCB`s. `InitCB` callbacks are used symmetrically for ensuring consistency before the parallel region is re-entered after reconfiguration. The video transcoding example does not require any `InitCB` callbacks to be defined.

5.1.3 Summary

In the Pthreads based parallelization, the developer is forced to implement a concrete, unchanging configuration of parallelism. In the Parcae parallelization, the developer declares the parallelism structure of the program, while deliberately not specifying the exact parallelism configuration. This underspecification allows Parcae to determine and enforce the optimal parallelism configuration at run-time.

5.2 System Administrator's View

Parcae presents a control panel to the system administrator. The administrator uses the panel to specify a performance goal, which includes an objective and a set of resource constraints under which the objective must be met. Examples of performance goals are "minimize response time" and "maximize throughput under a peak power constraint". The administrator may also invent more complex performance goals such as minimizing the energy-delay product [28], or minimizing electricity bills while meeting minimum performance requirements [56]. Parcae aims to meet the performance goals by dynamically adapting the configuration of program parallelism through the use of optimization mechanisms.

A *mechanism* is an optimization routine that takes an objective function such as response time or throughput, a set of constraints including number of hardware threads and power consumption, and determines the optimal parallelism configuration. The administrator provides values to a mechanism's constraints. An example specification by the administrator to a mechanism that maximizes throughput could be "24 threads, 600 Watts" thereby instructing the mechanism to optimize under those constraints. In the absence of a suitable mechanism, the administrator can play the role of a mechanism developer and add a new mechanism to the library.

5.3 Mechanism Developer's View

The Decima monitor observes both the application and the execution platform. Section 5.1.2 already described the methods that enable Decima to monitor application features such as task execution time and task load. To enable Morta to monitor platform features such as number of hardware threads, power, temperature, etc., the mechanism developer registers a feature with an associated callback that Morta can invoke to get a current value of the feature. Figure 5.8 shows the registration API. For example, the developer could register "`SystemPower`" with a callback that queries the power distribution unit to obtain the current system power draw [6].

```
//Application features
double Parcae::getExecTime(Task* task);
double Parcae::getLoad(Task* task);
//Platform features
void Parcae::registerCB(string feature, Functor* getValueOfFeatureCB);
void* Parcae::getValue(string feature);
```

Figure 5.8: Parcae mechanism developer API

The primary role of the mechanism developer is to implement the logic to adapt a parallelism configuration to meet a performance goal by using the information obtained via monitoring. For this, Parcae exposes the query API shown in Figure 5.8. Figure 5.9 shows an example mechanism that can enable a "Maximize Throughput with N threads" performance goal. Every mechanism must implement the `reconfigureParallelism` method. The method's arguments are the descriptor of the current parallelism configuration and the maximum number of threads that can be used to construct a new configuration. The new parallelism configuration is returned to the run-time system, which initiates execution according the new configuration.

The intuition encoded by the mechanism in Figure 5.9 is that tasks that take longer to execute should be assigned more resources. In step 1, the mechanism computes

```
 1 ParDescriptor∗ Mechanism::reconfigureParallelism (ParDescriptor∗ pd, int nthreads){
 2   float total_time = 0.0;
 3   // 1. Compute total time
 4   foreach (Task∗ task: pd→tasks) {
 5     total_time += Parcae::getExecTime(task);
 6   }
 7   // 2. Assign DoP proportional to execution time;
 8   // recurse if needed
 9   foreach (Task∗ task: pd→tasks) {
10     task→dop = nthreads ∗ (Parcae::getExecTime(task)/total_time);
11     ParDescriptor∗ innerPD = task→pd;
12     if (innerPD) {
13       task→pd[0] = reconfigureParallelism(innerPD, task→dop);
14     }
15   }
16   ... // 3. Construct new configuration − Omitted
17   return newPD;
18 }
```

Figure 5.9: Mechanism to maximize throughput—Assigns a degree of parallelism (DoP) to each task proportional to task's execution time

total execution time (lines 3–6) so that each task's execution time can be normalized. In step 2, the mechanism assigns a degree of parallelism (DoP) that is proportional to the normalized execution time of each task (line 10). `reconfigureParallelism` is recursively invoked to assign DoPs to the inner loops in the loop nest. For each loop, a new configuration is constructed with the new task descriptors and returned to the parent descriptor. For the sake of brevity, this last step is omitted.

Chapter 6

Online Monitoring and Optimization

We give an overview of the Parcae run-time system, comprised of the Decima monitor and the Morta executor, followed by a walk-through of the steps involved in re-configuring parallelism. Then, we describe the set of mechanisms implemented by a mechanism developer for the purposes of this study, highlighting the advantages and disadvantages of each mechanism and the particular performance goals that each enables. Following that, we describe a mechanism that addresses the shortcomings of the other mechanisms and is installed as the default optimization mechanism in Morta. Finally, we describe how the Parcae run-time system optimizes across the entire system by re-configuring multiple simultaneously executing flexible parallel programs.

6.1 Overview

The goal of the Parcae run-time system is to rapidly find and enforce parallelism configurations that are optimal for the execution environment. A parallelism configuration consists of:

1. a parallelization scheme (prepared by the Nona compiler or the programmer), which maps each loop to one of the following: $\mathcal{S} = \{\text{DOANY}, \text{PS-DSWP}, \text{SEQ}\}$; and

2. a degree of parallelism (DoP) D, the varying number of threads allocated to every parallel task of DOANY or PS-DSWP schemes.

A configuration also contains the assignment of threads to cores, but this aspect was not significant on our evaluation platforms.

Figure 6.1 shows the Parcae run-time system architecture. The Morta-*Executive* is responsible for directing the interactions between the various system components. Morta maintains a *Thread Pool* with as many threads as constrained by the performance goals. Morta uses *mechanisms* to adapt parallelism in order to meet the specified goals. One advantage of the separation of concerns enabled by the Parcae API is that a mechanism developer can implement new mechanisms and add them to the library in order to better support existing performance goals or to enable new ones, without changing the application code. The separation of concerns also enables reuse of mechanisms across many parallel applications.

There are two main information flows when an application is launched. First, the application registers its parallelism descriptors (expressed by the compiler or application developer). Second, the administrator specifies the performance goals. Morta then starts application execution. During execution, Decima monitors and Morta adapts the parallelism configuration to meet those goals.

6.2 Morta Operation Walk-through

Once a mechanism is selected, Morta uses it to reconfigure parallelism. The Executive triggers a parallelism reconfiguration in response to changes in the execution

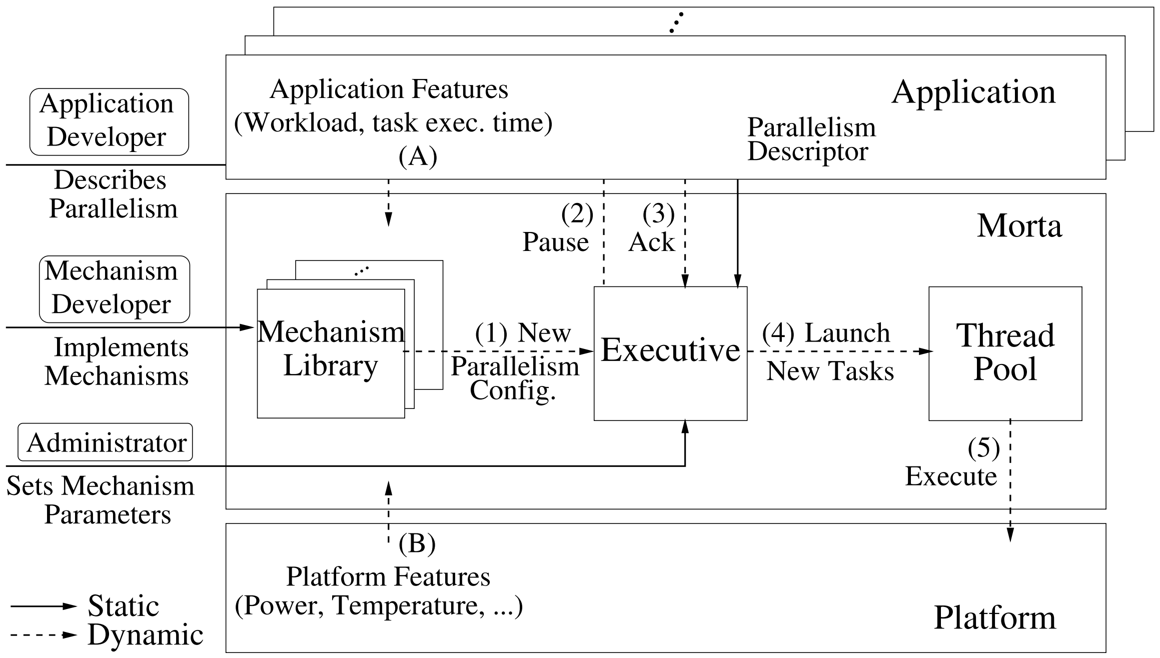


Figure 6.1: Interactions of three agents around Parcae. The application developer describes parallelism using the Parcae API just once. The mechanism developer implements mechanisms to transform the parallelism configuration. The administrator sets the constraint parameter values of the mechanism. Morta optimizes execution of multiple applications on the shared platform. (A) and (B) represent continuous monitoring of application and platform features by Decima. (1)–(5) denote the sequence of events that occurs when parallelism reconfiguration is triggered.

environment, such as an increase in workload. When reconfiguration is triggered, the following sequence of events occurs (refer to Figure 6.1):

1. The Mechanism determines the optimal parallelism configuration, which it conveys to the Executive.

2. The Executive returns `task_paused` to invocations of `Task::getStatus` in order to convey to the application Morta's intent of reconfiguration.

3. In response, the application and Morta steer execution into a suspended state by invoking the `FiniCB` callbacks of all the tasks.

4. The Executive then schedules a new set of tasks determined by the Mechanism for execution by the Thread Pool.

5. The Thread Pool executes the new tasks on the Platform.

6.3 Performance Goals and Mechanisms Tested

We tested three different goals of system use, and multiple mechanisms to achieve them. For each performance goal, there is a best mechanism that Morta uses by default. In other words, a human need not select a particular mechanism to use from among many. Multiple mechanisms are described for each performance goal in order to demonstrate the power of the Parcae API. Table 6.1 lists the implemented mechanisms and the number of lines of code for implementing each. Two of the mechanisms are proposed in prior work for a fixed goal-mechanism combination.

Mechanism					
WQT-H	WQ-Linear	TBF	FDP [85]	SEDA [99]	TPC
28	9	89	94	30	154

Table 6.1: Lines of code to implement tested mechanisms

6.3.1 Goal: "Minimize Response Time with N threads"

For systems serving online applications, the system utility is often maximized by minimizing the average response time experienced by the users, thereby maximizing user satisfaction. In the video transcoding example of Chapter 2, the programmer used an observation to minimize response time: If load on the system is light, a configuration that minimizes execution time is better, whereas if load is heavy, a configuration that maximizes throughput is better. This observation informs the following mechanisms:

Mechanism: Work Queue Threshold with Hysteresis (WQT-H)

WQT-H captures the notion of “latency mode” and “throughput mode” in the form of a 2-state machine that transitions from one state to the other based on occupancy of the work queue. Initially, WQT-H is in the *SEQ* state in which it returns a DoP of 1 (sequential execution) to each task. When the occupancy of the work queue remains under a threshold T for more than N_{off} consecutive tasks, WQT-H transitions to the PAR state in which it returns a DoP of $d_{P_{max}}$ (DoP above which parallel efficiency[1] drops below 0.5) to each task. WQT-H stays in the PAR state until the work queue threshold increases above T and stays like that for more than N_{on} tasks. The hysteresis allows the system to infer a load pattern and avoid toggling states frequently. The hysteresis lengths (N_{on} and N_{off}) can be weighted in favor of one state over another. For example, one extreme could be to switch to the PAR state only under the lightest of loads ($N_{off} \gg N_{on}$).

One advantage of WQT-H is that it is extremely lightweight as a control mechanism. However, it has multiple disadvantages. First, it exposes a very limited space of parallelism configurations, due to the binary nature of its DoP assignment. Each parallel task can have a DoP of 1 or $d_{P_{max}}$. Second, WQT-H is open-loop control; there is no online feedback as to whether its control decisions are beneficial or not.

Mechanism: Work Queue Linear (WQ-Linear)

A more graceful degradation of response time with increasing load may be achieved by varying the DoP continuously in the range $[d_{P_{min}}, d_{P_{max}}]$, rather than just toggling between two DoP values. WQ-Linear assigns a DoP according to Equation 6.1.

$$d_P = max(d_{P_{min}}, d_{P_{max}} - k \times WQo) \tag{6.1}$$

[1]Parallel efficiency equals the speedup of parallel execution over sequential execution divided by the number of cores used to achieve the speedup.

WQo is the instantaneous work queue occupancy. k is the rate of DoP reduction ($k > 0$). k is set according to Equation 6.2.

$$k = \frac{d_{P_{max}} - d_{P_{min}}}{Q_{max}} \tag{6.2}$$

Q_{max} in Equation 6.2 is derived from the maximum response time degradation acceptable to the end user and is set by the system administrator taking into account the service level agreement (SLA), if any. The degradation is with respect to the minimum response time achievable by the system at a load factor of 1.0. The threshold value T in the WQT-H mechanism is obtained similarly by a back-calculation from the acceptable response time degradation. A variant of WQ-Linear could be a mechanism that incorporates the hysteresis component of WQT-H into WQ-Linear.

Like WQT-H, WQ-Linear is extremely responsive due to its relatively simple control equations. Compared to WQT-H, WQ-Linear exposes a much larger space of parallelism configurations. However, like WQT-H, WQ-Linear is also open-loop control, and does not have means to understand the impact of its decisions.

6.3.2 Goal: "Maximize Throughput with N threads"

Many applications can be classified as throughput-oriented batch applications. Figure 6.2(a) shows the parallelism configuration of an image search engine called `ferret` [13]. The parallel tasks (`load`, `seg`, `extract`, `vec`, `rank`, `out`) interact in a pipelined fashion. Each pipeline stage may be *sequential* (with a DoP = 1) or *parallel* (with a DoP $\geq$ 1).

The overall application throughput is limited by the throughput of the slowest parallel task. By observing the in-queue occupancies of each task and task execution time, throughput-limiting tasks can be identified and resources can be allocated accordingly. This informs the following mechanisms:

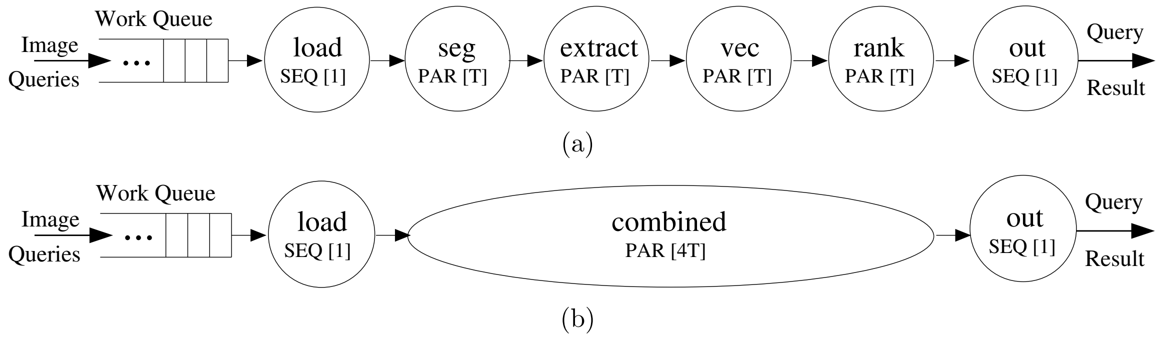


Figure 6.2: Image search engine `ferret` (a) Original pipeline (b) Pipeline with parallel stages collapsed

Mechanism: Throughput Balance with Fusion (TBF)

TBF records a moving average of the throughput (inverse of execution time) of each task. When reconfiguration is triggered, TBF assigns each task a DoP that is inversely proportional to the average throughput of the task. If the imbalance in the throughputs of different tasks is greater than a threshold (set to 0.5), TBF fuses the parallel tasks to create a bigger parallel task. The rationale for fusion is that if a parallel loop execution is heavily unbalanced, then it might be better to avoid the inefficiency of pipeline parallelism. Our current implementation requires the application developer to implement and register the desired fused task via the `TaskDescriptor` API that allows expression of choice of `ParDescriptor`s. Creating fused tasks is easy and systematic: Pipelined task execution and associated unidirectional inter-task data transfer should be changed to method invocations and data transfer via method arguments. Some of the applications that we studied already had pre-existing code for fusing tasks in the original Pthreads-parallelized source code. These were originally included to improve sequential execution in case of cache locality issues. Once registered, Morta will automatically spawn the fused task if task fusion is triggered by the mechanism.

Like the previous two mechanisms, TBF is also lightweight and responsive. The mechanism's average-case (and worst-case) execution-time complexity is $\Theta(n)$ where

n is the number of parallel tasks. TBF optimizes DoPs of all parallel tasks under the global constraint of number of threads (i.e., $\sum_{i=1}^{n} d_{P_i} \leq N$). In other words, it has a global view of resource allocation across all parallel tasks. On the downside, since TBF allocates all threads to tasks according to their respective throughputs, it assumes that assigning more threads to a task does not degrade performance. This assumption is not generally true [85]. Furthermore, by assigning tasks to all threads at all times, TBF keeps all cores on at all times, precluding the possibility of turning off some cores to save power and energy. However, if TBF's assumption about lack of performance degradation with increasing threads holds true for an application, then TBF can determine near-optimal configurations extremely quickly within a few iterations.

To demonstrate the ease of incorporating optimization mechanisms proposed in other contexts into the Parcae system, we implemented two high-quality mechanisms from prior work. These are described below.

Mechanism: Feedback Directed Pipelining (FDP) [85]

FDP initially allocates a single thread to each task. FDP then measures average task execution times, and ranks tasks from lowest to highest throughput by dividing a task's iteration count by its measured average execution time. The task with the least throughput is identified as the LIMITER task. If free threads are available, the number of threads allocated to the LIMITER task (i.e., the DoP of the LIMITER task) is incremented by one. Task throughputs are measured again to determine whether performance improved. If performance did improve, the DoP of the LIMITER task is incremented by one again; this process repeats until performance does not improve any further, or there are no more free threads. If there are no more free threads available, FDP schedules the two tasks with highest throughput for execution on a single thread, thereby freeing up a thread for potential use by the LIMITER task. The process of incrementing the DoP of the LIMITER task and measuring performance is repeated, until a stable parallelism configuration is reached.

Compared to the other mechanisms, FDP employs proportional closed-loop control. This gives it a considerable advantage in directing its search of the optimal configuration, and in converging to a configuration. However, the response time or the number of iterations of the optimization routine to converge to a good configuration can be quite large, especially as the dimensionality of the search space (corresponding to the number of parallel tasks) increases. FDP assumes a hill-shaped throughput versus DoP characteristic; consequently, it can limit the number of threads to the number corresponding to the peak, unlike TBF. This is advantageous both to avoid poorer performing configurations beyond the peak, and to save power and energy by turning off unused cores. Compared to TBF, FDP only simulates task fusion via time-multiplexing of tasks on the same thread; TBF executes real fused tasks exposed by the parallelizer thereby avoiding the overheads of communicating data between the fused tasks.

Mechanism: Stage Event-Driven Architecture (SEDA) [99]

The SEDA mechanism periodically samples the input work queue of a task and allocates an additional thread (increments the DoP) for that task when the queue length exceeds a threshold, up to a maximum number of threads per stage.

Each task in SEDA changes its DoP locally without coordinating with other tasks. This is in contrast to both TBF and FDP, which have a global view of resource allocation. Also, SEDA uses open-loop control unlike FDP.

6.3.3 Goal: "Maximize Throughput with N threads, P Watts"

Mechanism: Throughput Power Controller (TPC)

The administrator might want to maximize application performance under a system-level constraint such as power consumption. Morta enables the administrator to specify a power target, and uses a closed-loop controller to maximize throughput while maintaining power consumption at the specified target. The controller initializes each task with a DoP equal to 1. It then identifies the task with the least throughput and increments the DoP of the task if throughput improves and the power budget is not exceeded. If the power budget is exceeded, the controller tries alternative parallelism configurations with the same DoP as the configuration prior to power overshoot. The controller tries both new configurations and configurations from recorded history in order to determine the configuration with best throughput. The controller monitors power and throughput continuously in order to trigger reconfiguration if needed.

Like FDP, TPC employs closed loop control. Unlike FDP, TPC employs closed loop control for both power and throughput. This enables the specification of target power or throughput values, and the control mechanism will change parallelism configurations to meet the specified target.

6.4 Closed-loop Platform-wide Mechanism

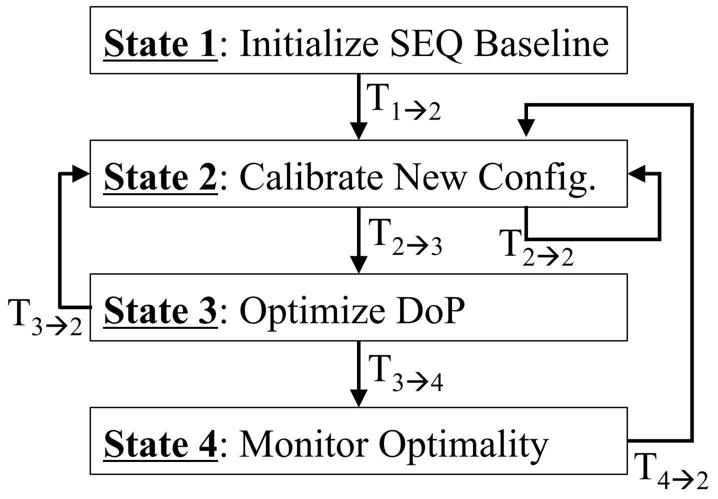
The mechanisms described previously have shortcomings such as slow response time and lack of platform-wide optimization. This section describes a novel, comprehensive control system to address these shortcomings. In this work, we focus on the following optimization objective: Minimize total execution time, and subject to that, minimize energy consumption. Morta achieves this objective by maximizing iteration throughput (number of iterations processed per second) and saving idle threads, as explained in this section. The system design allows additional goals, as desired.

Morta uses the following schema to identify optimal configurations: establish a baseline performance metric; identify an optimal configuration by repeatedly searching for a better configuration, pausing execution to change configurations, and measuring the performance of the new configuration relative to the baseline; monitor optimality of current configuration and trigger a new search if the dynamic execution environment changes. By virtue of employing a closed loop optimization schema, Morta can target other performance goals described by means of fitness functions, provided the parameters of a fitness function can be directly measured or indirectly computed from other measurements. As an example, Morta could be re-targeted at minimizing the energy delay squared product, since delay can be measured directly and energy can be indirectly computed from running power and elapsed execution time measurements. The finite-state machine shown in Figure 6.3 implements the above schema.

6.4.1 Finite-state Machine

State 1: Initialize Sequential Baseline. When a program enters a parallel region, Morta selects the sequential scheme ($S' = \text{SEQ}$, $D' = \{1\}$) and monitors its execution to establish a baseline throughput T_{seq}. After completing a fixed number of N_{seq} iterations (set to 10 in the current implementation), Morta reconfigures the program to execute in an initial parallel scheme $S' = S_{\text{par}}$ and default DoP $D' = D_{\text{par}}$, and transitions to State 2. Note that D' is a vector, every element of which represents the DoP of a single task. Also, the initial value of D' need not be 1 (explained in Section 6.4.2).

State 2: Calibrate New Configuration. In State 2, Morta treats the current parallel configuration with its scheme and DoP (S', D') as being new. It gathers initial


State 1: Initialize SEQ Baseline
T1→2
State 2: Calibrate New Config.
T2→3
T2→2
T3→2
State 3: Optimize DoP
T3→4
State 4: Monitor Optimality
T4→2

(a) Finite-state Machine Structure

Transition	Description
$T_{1\to2}$	Measured SEQ baseline, reconfigure to parallel scheme.
$T_{2\to3}$	Feed configuration profile.
$T_{3\to2}$	Reconfigure to next scheme.
$T_{3\to4}$	Reconfigure to optimal DoP.
$T_{4\to2}$	Detected change in workload, re-calibrate configuration.
$T_{2\to2}$ $T_{3\to2}$ $T_{4\to2}$	Detected change in resource allocation, re-calibrate configuration.

(b) Description of Transitions

Figure 6.3: Run-time controller

timing information for scheme S' while repeatedly reconfiguring the system to $D' \pm 1$ and executing each configuration for a number of iterations N_{par}[2]. This is done to guide a local search for an optimum DoP in the next state. After completing the calibration iterations, Morta restores the original (S', D') configuration and moves to State 3.

State 3: Optimize Degree of Parallelism (DoP). Based on the information collected in State 2, Morta performs a local monotonic search for an optimal DoP

[2]The number of iterations N_{par} is dynamically set to $\max(N_{\text{seq}}, 2.d_P)$, where d_P is the current DoP of the parallel task being optimized.

for each task, repeatedly reconfiguring the system to different values of DoP and executing a number of N_{par} iterations to measure its performance. Specifically, it uses a finite difference gradient ascent control law [34, 83], as described in detail in Section 6.4.2. This process converges to an optimal DoP (D'') for the given scheme S', with associated throughput T''. If T'' is better than the best throughput T^* achieved so far for the region, Morta updates T^* with T'' and records S', D''. ($T^* = T_{\text{seq}}$ initially.) Morta then selects the next scheme S' from $\mathcal{S}$, resets DoP to its default $D' = D_{\text{par}}$, reconfigures the system accordingly, and returns to State 2. If all schemes in $\mathcal{S}$ have been explored, Morta retrieves the configuration S', D'' that achieved the best throughput T^*, reconfigures the system accordingly, and moves to State 4.

State 4: Monitor Configuration Optimality. In this state, the Decima monitor performs a passive monitoring of the system (i.e., no reconfigurations) to detect changes in either the resources allocated to the program or in the workload of the program itself. For the former, Decima interacts with the platform-wide run-time system (described in Section 6.4.3). For the latter, Decima monitors the throughput of the parallel region; if the throughput changes by more than a preset threshold, the workload is deemed to have changed. If any such change is detected, the configuration is suspected to have become suboptimal, and control returns to State 2 retaining the current scheme. If the change corresponds to an increase in resources, the current DoP is retained (hopefully as a good starting point); otherwise, if resources decreased or the throughput of the workload itself decreased, the DoP is reset to its initial value D_{par}.

Note that all reconfigurations that modify the DoP while keeping the scheme intact (such as those carried out by State 2, State 3, and potentially upon transitions $\text{T}_{3\to4}$ and $\text{T}_{4\to2}$) do not require changing the code distributed among the worker threads. Reconfigurations that do modify the parallelization scheme, upon transitions $\text{T}_{1\to2}$ and $\text{T}_{3\to2}$, do involve replacing this code.

6.4.2 Optimizing the Degrees of Parallelism

A given parallelization scheme may comprise both sequential $\{S_1, S_2, ...S_m\}$ and parallel tasks $\{P_1, P_2, ...P_n\}$. The DoP d_{S_i} for every sequential task is inherently 1. The objective of optimizing DoP (State 3 of Figure 6.3(a)) is therefore to find a DoP d_{P_i} for each parallel task that maximizes the overall performance of the region. The problem can be formulated as follows:

$$\text{Maximize overall throughput}: \ T = f(d_{P_1}, d_{P_2}, \ldots, d_{P_n})$$
$$\text{subject to}: \ d_{P_i} \geq 1 \ \ \forall i, \quad \sum_{i=1}^{n} d_{P_i} \leq N$$

where N denotes the total number of threads made available to the program minus those used by its sequential tasks (m). Morta optimizes each d_{P_i} separately, in turn, according to the relative throughputs of the tasks in ascending order (see Algorithm 4). This is done in order to prioritize slower tasks, which are typically bottlenecks in pipelined networks of sequential-parallel tasks. This order is updated after optimizing a task.

Morta computes the optimal DoP d_{P_i} for a parallel task P_i by using a fast iterative gradient ascent technique [83], based on the assumption that function f is unimodal with a single (local) optimum (see Figure 6.4). Section 8.3.5 discusses the impact of this assumption. On each iteration of the optimization routine, the current throughput is compared with the previous throughput, and this difference either establishes the gradient of the change to the next DoP or concludes the search. First, an upper bound $\overline{d_{P_i}}$ on the allowed DoP is calculated, which is equal to the maximum number of threads currently available for P_i. Initially, every parallel task is assigned half of its fair share of threads: $\frac{N}{2n}$. Thus, $\overline{d_{P_i}} = N - \sum_{j \neq i} \frac{N}{2n} = \frac{(n+1)N}{2n}$. The search for an optimal $d_{P_i} \in [1, 2, \ldots, \overline{d_{P_i}}]$ starts at the midpoint of this range: $d_{P_i}(0) = D_{\text{par}} = \frac{1}{2}\overline{d_{P_i}}$. Morta then sets $d_{P_i}(1) = d_{P_i}(0) \pm 1$ according to whichever achieves greater throughput. We call the search *increasing* or *decreasing*, respectively.

Algorithm 4: Optimizing multiple DoPs in a region

```
Input: Calibrated parallel region, thread budget $N$
Output: Optimized DoP for region ($\leq N$)
totalDoP $\leftarrow$ computeTotalDoP(region.parallelTasks())
foreach $P_i \in$ region.parallelTasks() do
    $P_i$.opt $\leftarrow$ false; $P_i$.sat $\leftarrow$ false
repeat
    optimize_a_task $\leftarrow$ false
    $\mathcal{P} \leftarrow$ sortInAscendingThroughput(region.parallelTasks())
    for $P_i \in \mathcal{P}$ while $\neg$optimize_a_task do
        $\overline{d_{P_i}} \leftarrow N -$ totalDoP $+ d_{P_i}$
        if $(\neg P_i.\text{opt}) \vee ((d_{P_i} < \overline{d_{P_i}}) \wedge (\neg P_i.\text{sat}))$ then
            $d_{P_i} \leftarrow$ gradientAscent($P_i, d_{P_i}, \overline{d_{P_i}}$)
            $P_i$.opt $\leftarrow$ true; $P_i$.sat $\leftarrow (d_{P_i} < \overline{d_{P_i}})$
            totalDoP $\leftarrow$ updateTotalDoP(totalDoP, $d_{P_i}$)
            optimize_a_task $\leftarrow$ true
until $\neg$optimize_a_task
if parThroughput(region) > (0.9*totalDoP)*seqThroughput(region) then
    return totalDoP        // current parallel scheme profitable, keep it
else if bumpToNextScheme(region) then
    return CalibrateAndOptimize(region) // try next scheme recursively
else return 1 // no parallel scheme is profitable, revert to sequential
```

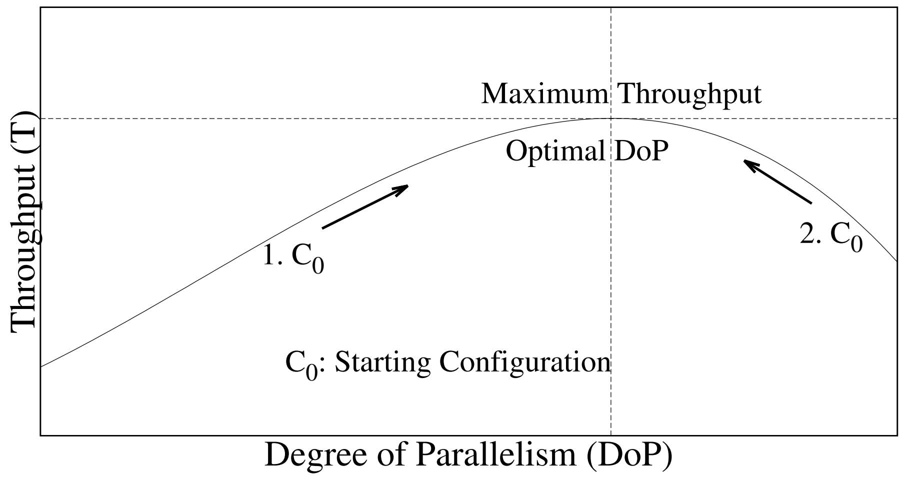


Figure 6.4: Assumed throughput characteristic as a function of the i^{th} component of the DoP vector, with other components fixed. Gradient-ascent is used to determine the optimal DoP.

Next, the gradient between the current $(k + 1)$ solution and the previous (k) solution is calculated, starting from $k = 0$:

$$\delta(k+1) = T(d_{P_i}(k+1)) - T(d_{P_i}(k)). \tag{6.3}$$

If $\delta(k+1) < 0$, the optimal solution has been passed and Morta terminates the search taking $d_{P_i}(k)$ to be the solution. The gradient ascent approach is designed to take small steps near the summit, so interpolation is typically not needed. Recall that we seek to maximize the throughput, and subject to that, minimize the number of threads (thereby saving energy). The case $\delta(k + 1) = 0$ is treated similar to case $\delta(k + 1) < 0$ above if the search is increasing, and similar to case $\delta(k + 1) > 0$ below if the search is decreasing. If $\delta(k + 1) > 0$, the next solution is calculated according to the gradient-ascent formula:

$$d_{P_i}(k+2) = d_{P_i}(k+1) + \alpha\delta(k+1) \tag{6.4}$$

where α is positive if the search is increasing, and negative if the search is decreasing. The search continues by incrementing k and evaluating Equations 6.3 and 6.4 repeatedly.

After optimizing d_{P_i}, the optimization process repeats by sorting the tasks according to their throughputs, and selecting the next task P_j to optimize. The process terminates after all d_{P_i}'s have been optimized, possibly more than once, and cannot be further improved.

Once the search terminates at an optimal DoP, Morta measures the overall throughput of the region $T = f(d_{P_1}, \ldots, d_{P_n})$ and compares it with the baseline sequential throughput T_{seq} measured in State 1. The parallel configuration is deemed profitable only if its efficiency is significantly better than that of the sequential configuration. Otherwise, Morta repeats the optimization process considering an alternative parallel scheme exposed by Nona, if any (corresponding to transition $T_{3\to2}$ in Figure 6.3). When all available schemes have been considered, Morta chooses the most efficient scheme (possibly even SEQ) for execution, and enters State 4. To accelerate the optimization process, Morta caches previously optimized configurations and reuses them, if feasible, as initial configurations upon future entry into a parallel region.

The controller caches optimized configurations of regions, including their scheme, DoPs, and number of cores N used. On future entry into a parallel region, previously recorded configurations are used if applicable, according to the number of cores available N': if $N' = N$, the configuration is reused; if $N' > N$, the configuration serves to initialize the optimization process (i.e., setting D_{par}); and if $N' < N$, the configuration is not used (similar to transition $T_{4\to2}$ in Figure 6.3). This strategy potentially speeds up the optimization process.

6.4.3 Platform-wide Control

The description of the Morta run-time system in Section 6.4.2 applies to a single program executing a parallel region. However, the same control system used to optimize the parallel execution of one program easily extends and generalizes to multiple programs running concurrently, thereby achieving platform-wide execution optimization. Indeed, each executing program p can be considered a collection of parallel and sequential tasks, executed in parallel and independent of other co-scheduled programs, as if all programs belong to one parallel region with multiple parallel tasks. At this outer level, Morta must decide how to partition the total number of threads $\mathcal{N}$ available in the system across the different co-scheduled programs: $\mathcal{N} = \sum_p N_p$. Given its budget of threads N_p, a program is optimized and monitored by its dedicated controller. Algorithm 5 summarizes the platform-wide control logic.

The platform-wide Morta run-time system is implemented as a daemon, launched upon system boot. When a flexible parallel program p is launched, its controller registers itself with the daemon and acquires its set of resources N_p; initially $N_p = \mathcal{N}/\mathcal{P}$ where $\mathcal{P}$ is the number of flexible parallel programs. Each controller proceeds

Algorithm 5: Platform-wide optimization

```
Input: platform-wide thread budget $\mathcal{N}$
Output: Optimized DoP for platform ($\leq \mathcal{N}$)
slack ← $\mathcal{N}$
activePrograms ← platform.programs
while $slack > 0$ do
    foreach $P \in activePrograms$ do
        $N_{P_{slack}}$ ← slack/activePrograms.size
        slack ← slack - $N_{P_{slack}}$
        $N_P \leftarrow N_{P_{new}} + N_{P_{slack}}$        // $N_{P_{new}} \leftarrow 0$ initially
        $N_{P_{new}}$ ← P.optimizeDoP(P.region, $N_P$)
        if $N_{P_{new}} < N_P$ then
            slack ← slack + $(N_P - N_{P_{new}})$
            activePrograms.extract(P)
```

to initialize, optimize, and monitor its program, as explained (through States 1,2,3 and 4), and reports its optimal amount of resources $N_p' \leq N_p$ to the daemon (upon transition $T_{3\to4}$).

On receiving optimization results N_p' from the controllers, the daemon distributes slack resources $\mathcal{N} - \sum_p N_p'$ if any, among controllers with $N_p' = N_p$. Finally, the daemon monitors changes in system resources (launch and termination of programs) and re-partitions resources across executing programs when they occur.

Note that this implementation distributes most of the work across the parallel controllers; not only are the programs profiled concurrently and independently, but the processes of optimizing them are distributed as well. This is done in order to load-balance and speed-up the process of reaching an optimal platform-wide configuration. All aspects of the run-time control system are implemented in shared memory, thus making operations such as status query, execution monitoring, metadata updates, etc. extremely lightweight. Chapter 8 evaluates the overhead of each recurring operation and demonstrates the effectiveness of the run-time control system in making efficient and accurate decisions about reconfiguration.

Chapter 7

Reducing Run-time Overheads

Figure 7.1 highlights the overheads of Parcae execution. The figure does not show all sources of Morta overhead; specifically, it does not contain the overhead of status query (whether the program should pause) and the overhead of monitoring execution time and workload. Both overheads can be reduced by simply reducing the frequency at which the corresponding functions are invoked. Of course, this may impact the responsiveness of the run-time control system. Monitoring overhead can be reduced by leveraging performance monitoring hardware [40, 47]. We found these two overheads to be so low on our evaluation platforms that we do not bother to reduce them. The overheads on which we focus are:

1. **Task Migratability:** consisting of (i) Task Activation, yielding to and returning from the task activation loop; and (ii) Data Management, loading and saving cross-iteration dependency data; both occurring per iteration.

2. **Pause-Resume**: on receiving a pause signal, all threads synchronize by means of a barrier; threads that reach the barrier early waste cycles waiting for the slower threads to catch up (shown as Barrier Wait in Figure 7.1). Tasks must be re-started on threads on resumption.

3. **Parallelism Reconfiguration**: time spent executing core optimization routine to determine new parallelism configurations (shown as Parallelism Reconfiguration in Figure 7.1).

4. **Critical Section Corresponding to Reduction:** For the sake of illustration, assume that the critical sections shaded darker correspond to a reduction operation (e.g., max-reduction[1]). Recall that parallel tasks avoid having local state across iterations by sharing cross-iteration data—in this example, the variable containing the maximum value (`max`)—in global memory. Consequently, there may be contention for atomically performing a read-modify-update operation on the `max` variable on each and every iteration.

In this chapter, we describe optimizations that almost completely eliminate each of the above overheads. Figure 7.2 shows how the optimizations enable significantly improved utilization of the cores; in the same amount of time, optimized Morta completes two full rounds of reconfiguration compared to only one round in the unoptimized case.

7.1 Reducing Data Management Overhead

Figure 7.3 illustrates the optimization. Rather than returning to the task invocation loop on every iteration (a), the optimized version retains the original loop backedge (b). This makes it possible to hoist the load out of the loop header into the preheader, and to push the corresponding store into the `pause` exit block. Consequently,

[1]Max reduction is used to determine the maximum item in a set of items in a distributed fashion. Computing the maximum item normally involves initializing a variable `max` to one of the items, comparing `max` with each of the other items, and updating `max` in case the other item is greater than the current item stored in `max`. At the end of the operation, the `max` variable contains the maximum item in the set. Such operations are both commutative and associative, and hence can be performed in any order. Using this property, max reduction initializes `max` to one of the items, then each thread computes a local maximum on its subset of the item-set, and finally the global maximum is computed over the local maxima.

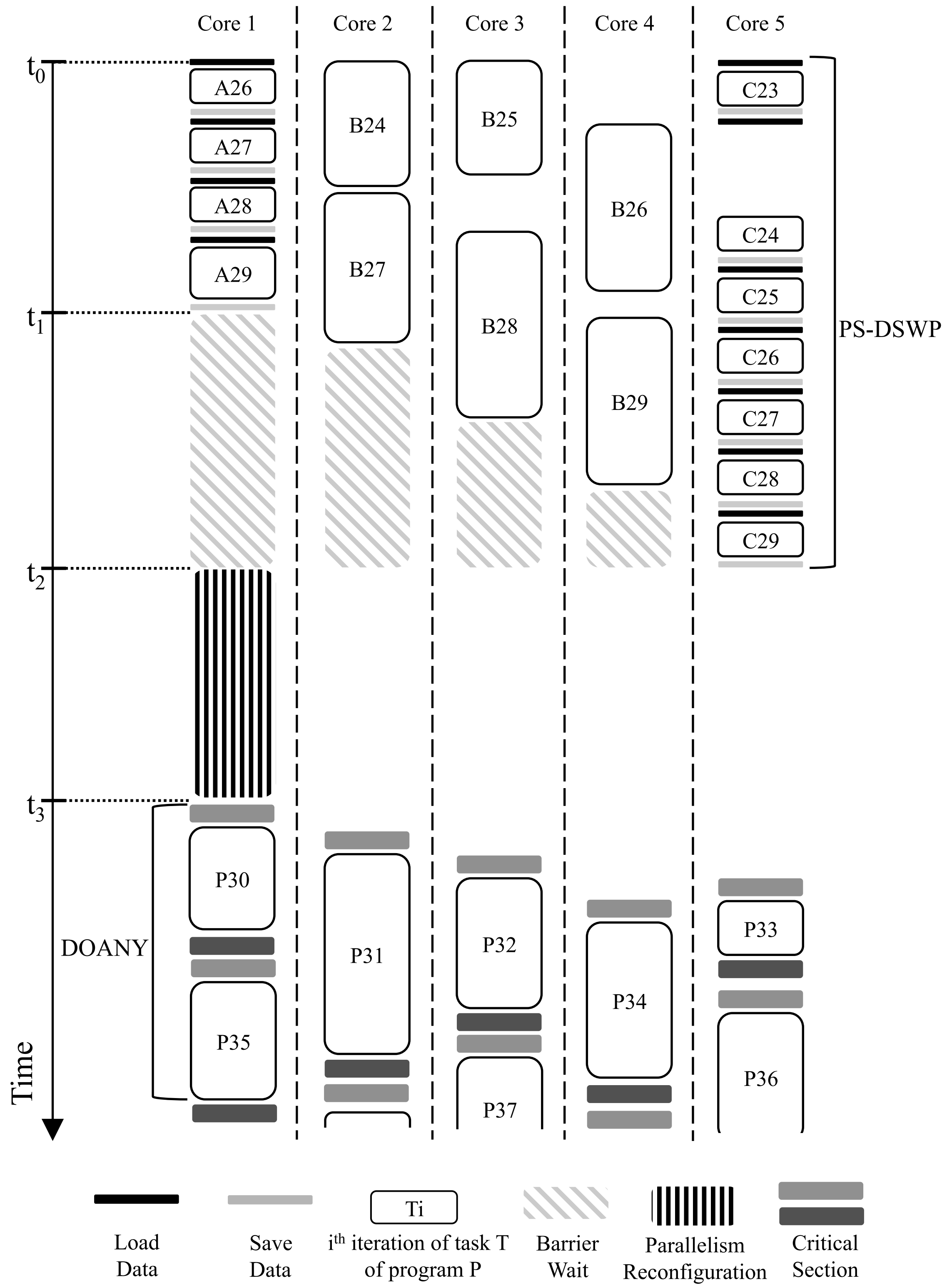


Figure 7.1: Morta execution overheads. (t_0) Program P is in the steady state of PS-DSWP execution; (t_1) program P is signaled to pause by Morta; (t_2) P's pipeline is drained; (t_3) new parallelism configuration is determined and P begins DOANY execution.

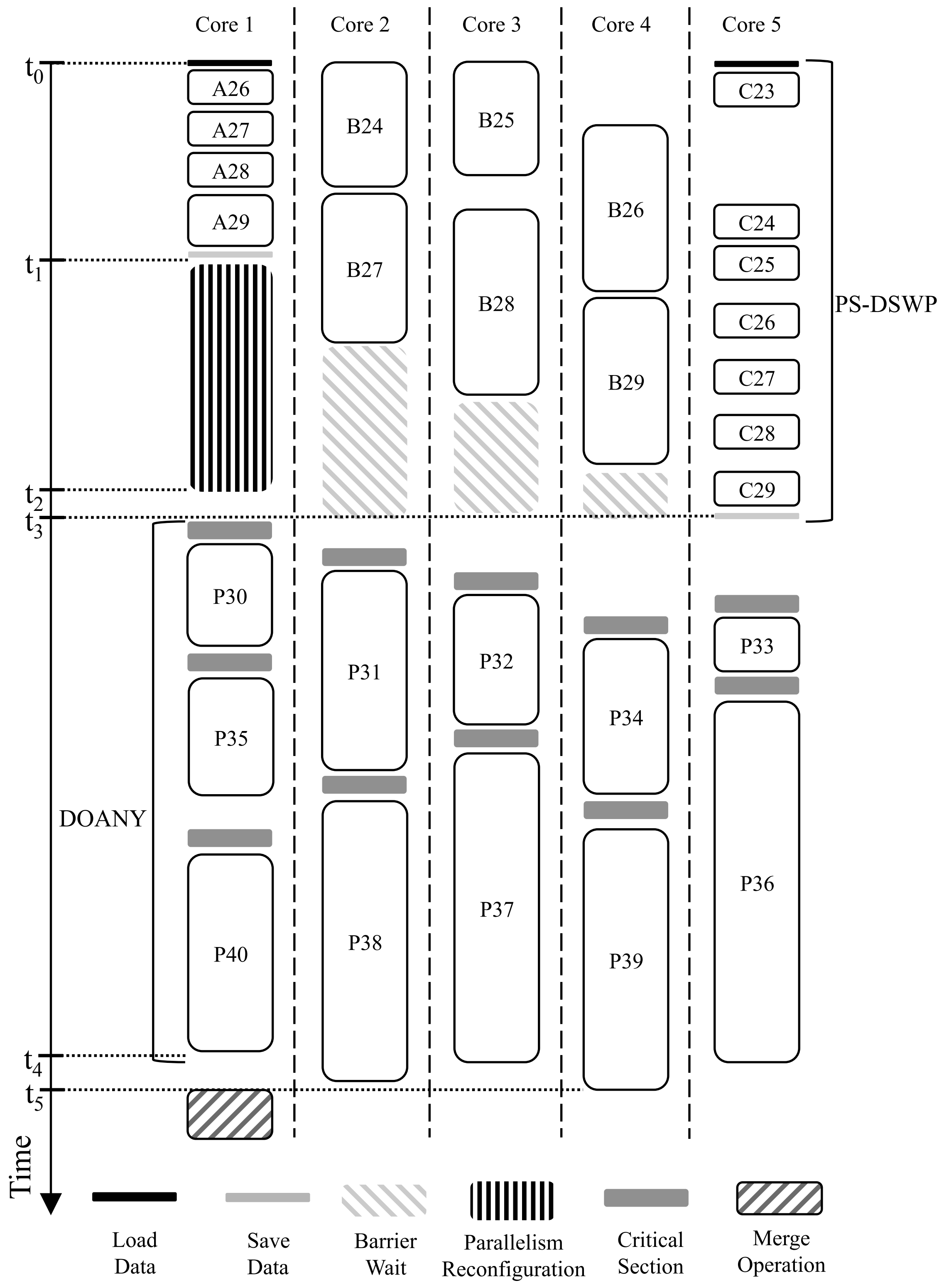
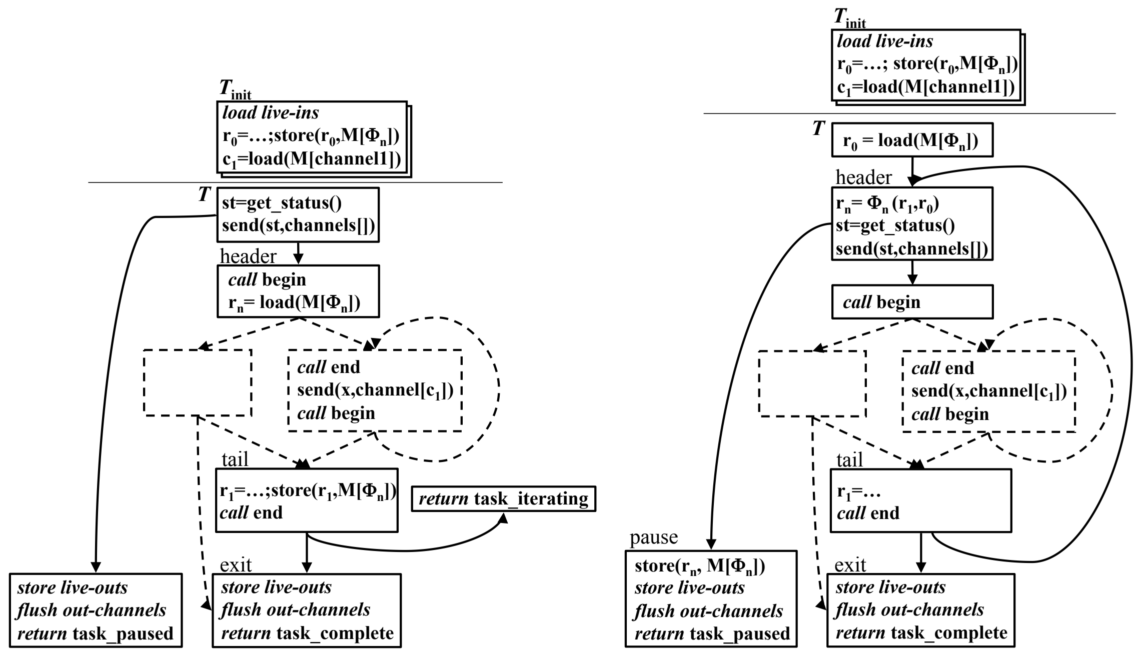


Figure 7.2: Morta execution after optimization. In the same amount of time that unoptimized Morta finishes one round of reconfiguration (shown in Figure 7.1), optimized Morta finishes **two** rounds of reconfiguration. Barrier wait overhead reduction is described later in this chapter.



(a) Unoptimized Morta (b) Optimized Morta

Figure 7.3: Reducing cross-iteration dependency load/save overhead by hoisting the corresponding operations out of the loop

the cross-iteration dependency values need to be saved and loaded only once per parallelism reconfiguration.

Note that this strategy imposes a constraint on tasks with dependencies to execute in the same thread between reconfigurations, i.e., migration of sequential tasks is disabled. The correct choice of retaining the ability to migrate sequential tasks versus reducing their load/save overhead is platform-specific; Morta can execute benchmarks to determine the (constant) overhead on a given platform and make the appropriate choice.

7.2 Reducing Barrier Wait Overhead

Figure 7.4 shows how the full barrier on pausing for reconfiguration results in pipeline drain stalls in case of pipelined execution. The most common scenario in which reconfiguration happens is when a scheme is being optimized, by means of the gradient-

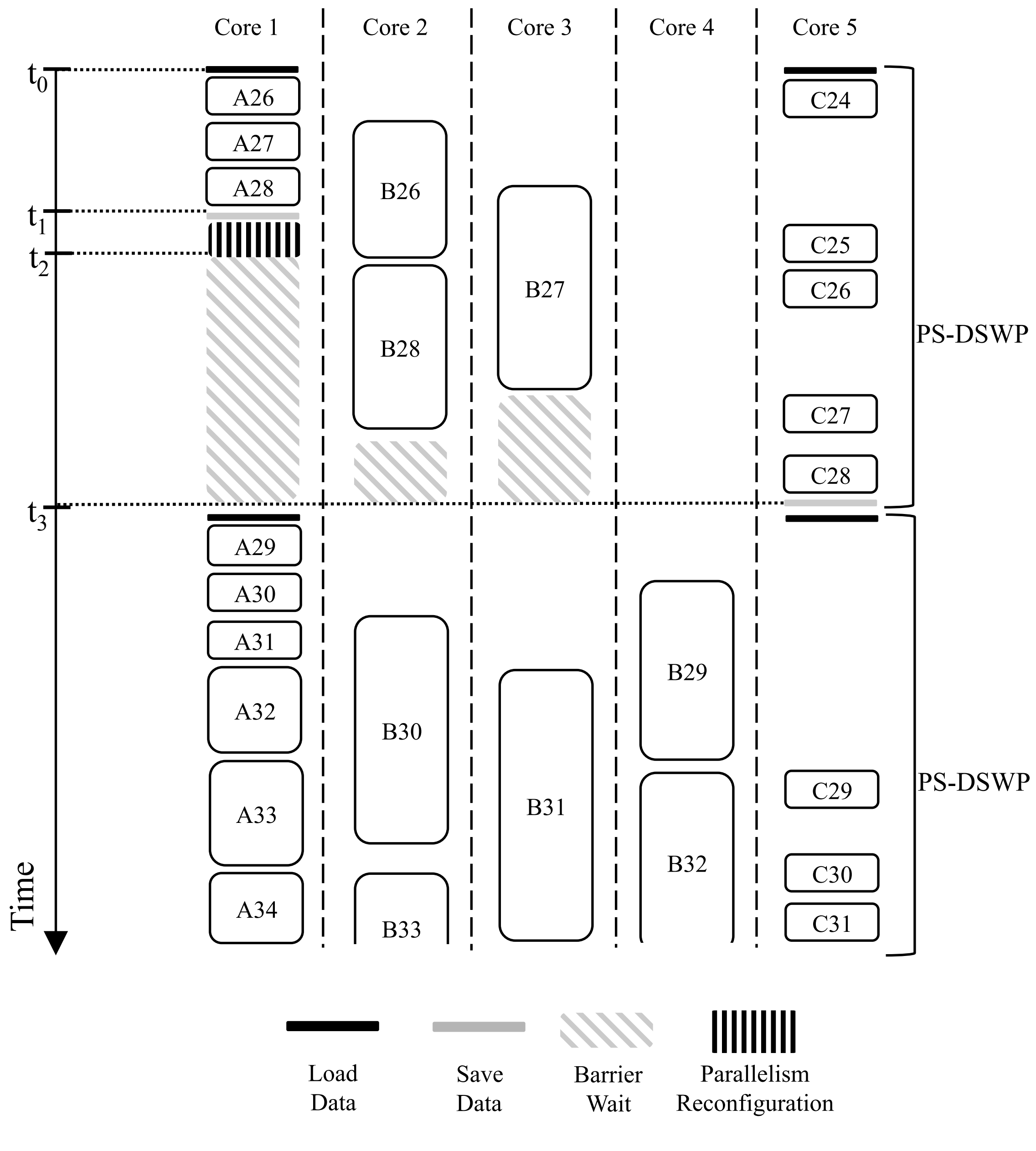


Figure 7.4: Strategy of waiting at a barrier for all tasks to pause incurs the pipeline drain overhead

ascent algorithm. During gradient ascent, the degree of parallelism (DoP) of a parallel task is increased or decreased (according to whether the ascent is increasing or decreasing) in step sizes proportional to the gradient. Since the run-time merely has to increase the DoP of a task, the "Reconfigure Parallelism" overhead in the figure is very small. Recall that a parallel task is explicitly constructed such that it either does not have any loop-carried dependencies or updates global memory with appropriate synchronization. As a consequence, additional threads may start executing the parallel task without violating program consistency, eliminating the need for a barrier wait. In the presence of communication with other tasks, however, other actions must be taken. Below, we work through the complexities from a specific case to the general.

7.2.1 Single Parallel Task

If a parallel region is executed by a single parallel task, as in the case of DOANY, then the run-time system merely needs to change the DoP of the task, and enforce as many additional threads as the difference between the new and old DoPs to execute the parallel task.

7.2.2 Sequential-Parallel-Sequential Tasks

Consider a set of three communicating tasks. PS-DSWP typically employs a round-robin iteration distribution policy. In other words, the first sequential task hands out iterations to the threads executing the parallel task in a round-robin order, and the last sequential task consumes iterations from the threads executing the parallel task in the same round-robin order. Let the parallel task have an initial DoP of m. Then, the first (last) sequential task sends (receives) the i^{th} iteration dependencies to (from) the $(i \bmod m)^{th}$ thread executing the parallel task, where i is a simple induction variable maintained by both sequential tasks independently.

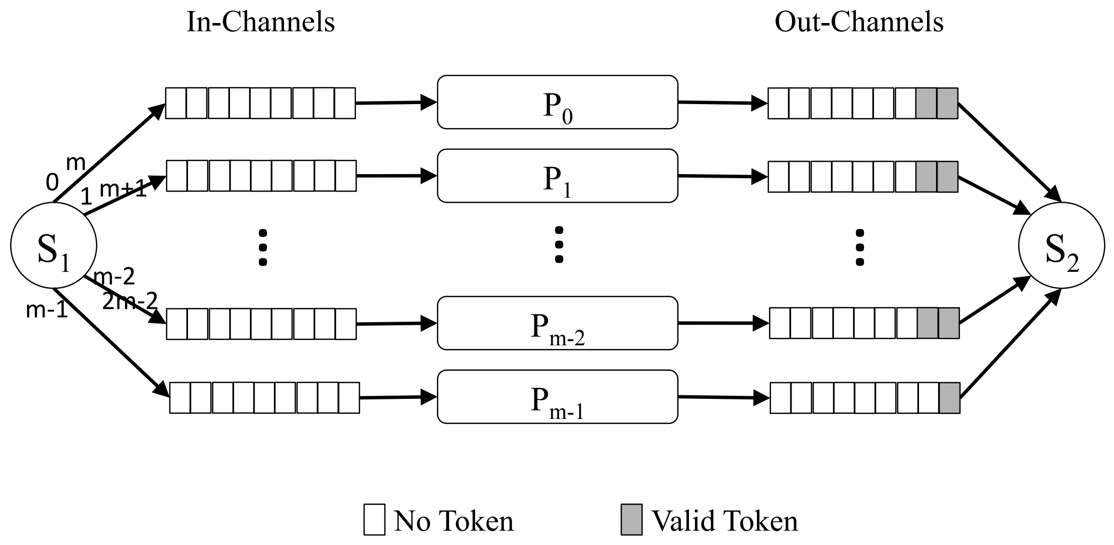
Now, assume that Morta decides to increase the parallel task's DoP to n as part of the gradient-ascent routine. As in the case of the single parallel task previously, the run-time system enforces $n - m$ additional threads to execute the parallel task. The additional threads will wait on their input queues; however, the sequential tasks will continue to produce and consume data only on channels 0 through $m - 1$. So, the run-time system

1. pauses only the sequential tasks (by sending the pause signal to the first sequential task that is the master task, which then propagates the signal to the other sequential task),
2. updates their view of the DoP of the parallel task with which they are communicating to n, and
3. resumes their execution.

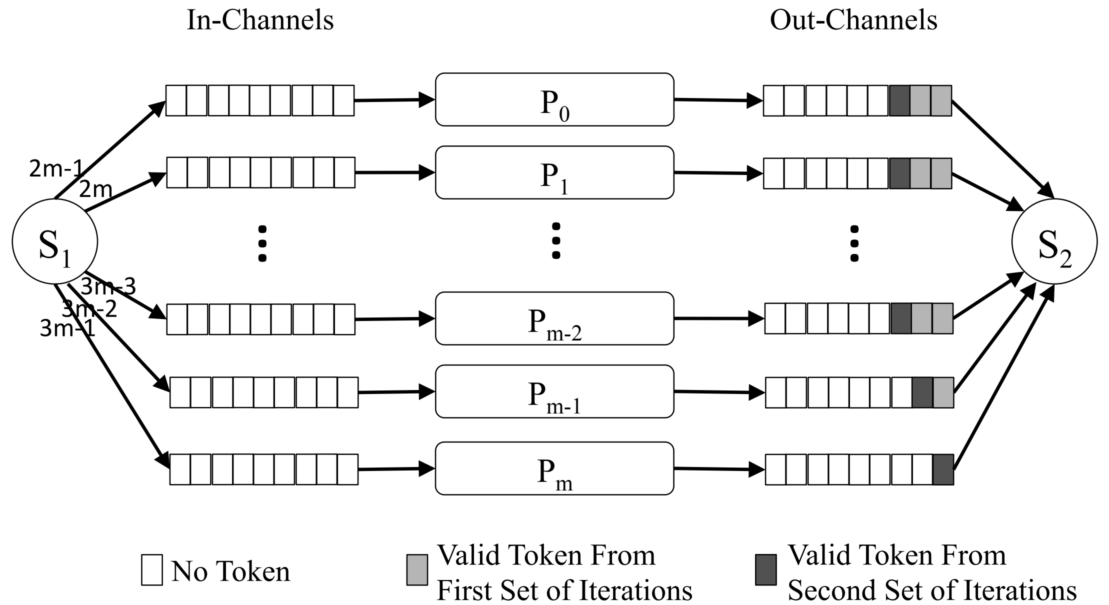
Merely performing the above actions may result in a race condition that leads to violation of program semantics due to out-of-order processing of iterations. Figure 7.5 shows why. For simplicity, assume that the first sequential task (S_1) produces a single token on each iteration to the parallel task, and the parallel task (P) produces a single token on each iteration to the last sequential task (S_2). Thus, we use the word token to denote a single iteration.

Assume that S_1 receives the pause signal on iteration $i_{pause} = 2m - 2$. Assume also that the threads executing P have processed these tokens and enqueued the corresponding out-tokens on their out-channels. (In reality, P usually lags behind S_1, but this is not pertinent to the discussion.) Finally, assume that S_2 has not started processing its tokens. Figure 7.5(a) shows the system in this state, with $2m-1$ tokens in the out-channels of the P task.

Figure 7.5(b) shows the situation when Morta increases the DoP of task P to $n = m+1$, resulting in S_1 producing iteration i's token on the $(i \bmod (m+1))^{th}$ channel.



(a) Round-robin iteration allocation is shown near S_1


In-Channels
Out-Channels
P0
P1
Pm-2
Pm-1
Pm
S1
S2
2m-1
2m
3m-3
3m-2
3m-1
No Token
Valid Token From First Set of Iterations
Valid Token From Second Set of Iterations

(b) Race condition may result in order violation

Figure 7.5: Barrier optimization. Naïve optimization may result in violation of program semantics. (b) shows how increase in DoP from m to $m+1$ without appropriate synchronization between S_1 and S_2 can result in S_2 processing iterations out of order.

This results in P processing the tokens and producing the corresponding out-tokens on the $m+1$ out-channels. This new set of tokens is shown in a darker shade in the figure. Now, when S_2 processes these tokens, since its channel width has been set by the run-time to $n = m+1$, it mistakenly processes token $3m-1$ on channel m before processing tokens m through $2m-2$ from the first set of iterations and tokens $2m-1$ through $3m-2$ from the second set of iterations. This out-of-order processing can result in violation of semantics, if for instance, S_2 prints values to an output device.

The problem with the above naïve optimization is fairly straightforward. S_2 must process all tokens up to the iteration at which S_1 was paused, before increasing the width of the channel from m to $m+1$. In other words, the pipeline must be drained by S_2 before expanding the channel width, without impacting either S_1 or P.

To solve the problem, Morta maintains a count of the number of iterations that have been executed by the master sequential task since the parallel region was invoked. When the run-time system signals the master sequential task S_1 to pause, it reads the value of the counter (call this I), and communicates I to the other sequential task S_2. To determine which channel to consume from, S_2 compares the value of its induction variable i (that tracks current iteration count) with I. If $i < I$, then S_2 consumes tokens from the $(i \bmod m)^{th}$ channel; otherwise, it updates its channel-width to $n = m+1$ and consumes tokens from the $(i \bmod n)^{th}$ channel.

Figure 7.6 shows the improvement in performance with the above barrier optimization. “Barrier Synch.” represents the additional overhead of communicating the I counter to task S_2.

7.2.3 Network of Sequential-Parallel Tasks

The barrier wait optimization described for sequential-parallel-sequential tasks can be generalized to a pipeline network of sequential-parallel tasks, such as the one shown in Figure 7.7, even when the DoPs of multiple parallel tasks are simultaneously changed.

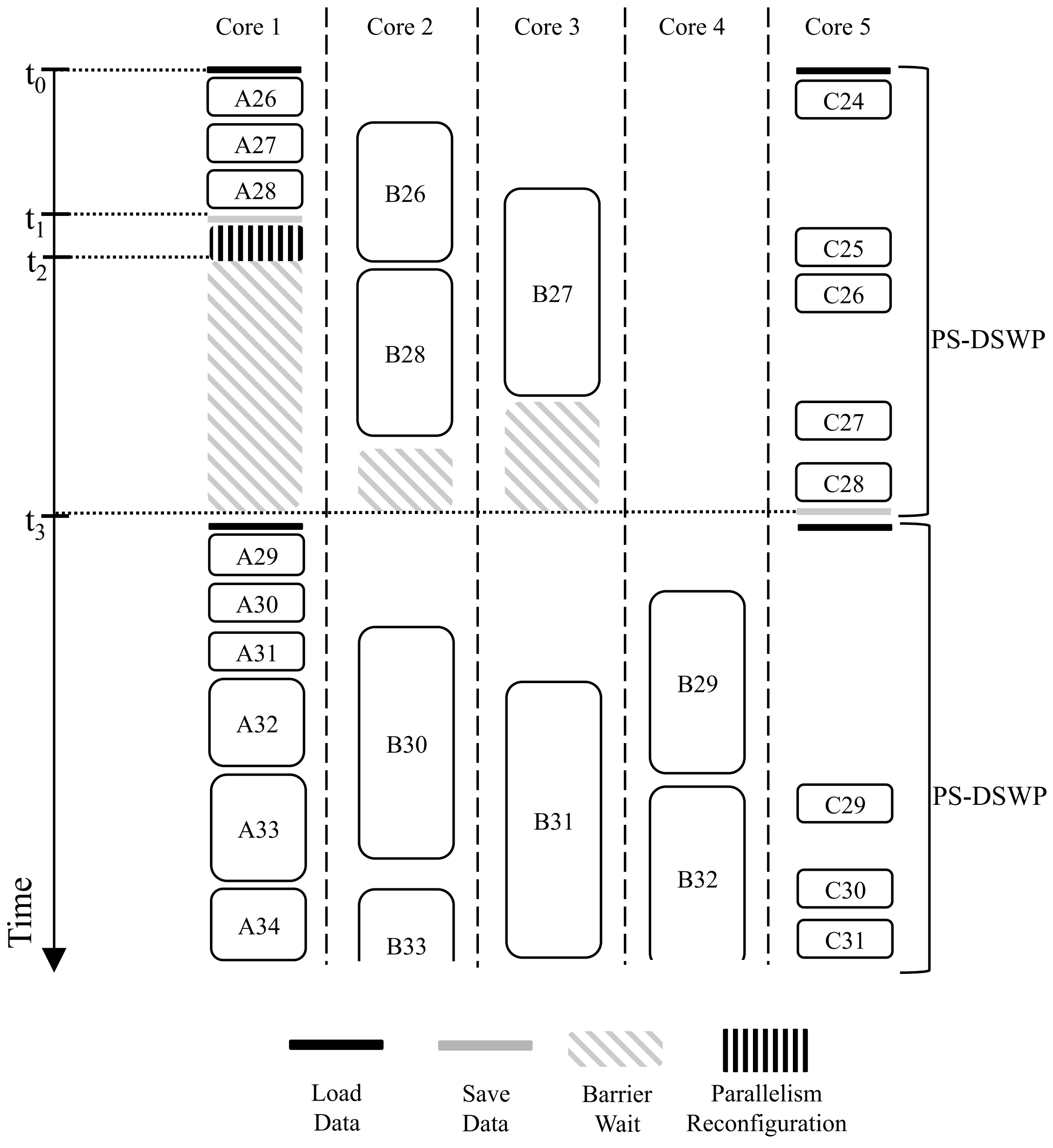


Figure 7.6: Morta optimization to eliminate barrier wait stall saves time and enables additional iterations to complete

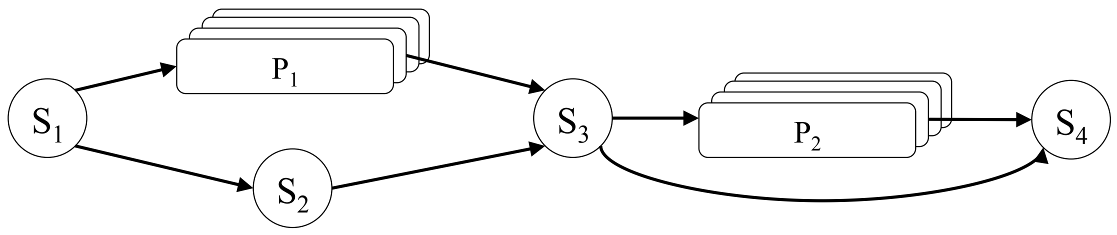


Figure 7.7: Morta barrier wait optimization works for arbitrary sequential-parallel pipeline networks

To see why, note that the optimization only hinges upon informing all sequential tasks (except the master task) of the number of iterations *after* which they must adjust the width of their respective communication channels, following a parallelism reconfiguration involving change of DoPs of the parallel tasks. Hence, the run-time is modified in a straightforward manner to communicate, upon a pause, the iteration count I of the master task to all other sequential tasks in the parallel region. Multiple parallel tasks are handled naturally; the run-time system returns the appropriate channel width (equal to the DoP of the corresponding parallel task) when queried by the sequential tasks that communicate with each parallel task.

Finally, while the above discussion assumed increasing gradient ascent, where the channel width increases, the same barrier wait optimization works for decreasing gradient ascent as well.

7.3 Reducing Parallelism Reconfiguration Overhead

The thread executing the master task receives the pause signal from the run-time system. This thread can immediately start executing the optimization routine that determines the next parallelism configuration, rather than waiting for other threads to hit the barrier before executing the optimization routine. This enables an overlap of the time spent executing the optimization routine and the difference between the times of the last and the first thread to reach the barrier.

7.4 Reducing Critical Sections Corresponding to Reductions

To reduce contention due to concurrent read-modify-update actions on reduction variables by threads executing a parallel task, Parcae employs privatize-and-merge [3].

Privatization involves allocating per-thread local storage for the reduction variable. After each reconfiguration, threads executing the parallel task copy the reduction variable (that exists in global memory) into their local storage; the copy operation is performed in the initialization function T_{init} (see Figure 7.3). Each thread executing the parallel task then performs a local reduction. As an example, for max-reduction, each thread computes the local maximum over the set of iterations that it executes. When reconfiguration is triggered, the threads executing the parallel task communicate the local reduction values to the sequential master task, which *merges* them by executing an appropriate merge operation. For max-reduction, the merge operation involves initializing the global `max` variable with one of the local values, iterating over other local values and comparing each with `max`, and updating `max` if the local value is greater than the current value stored in `max`.

Chapter 8

Evaluation

Table 8.1 describes the two **real** platforms used to evaluate Parcae. The Nona compiler is built on LLVM [49] revision 129859. Nona generates optimized assembly, which is then assembled and linked by GCC version 4.4.1. The Parcae API and run-time system are compared against Pthreads-based expertly-tuned manual parallelizations. The Nona compiler is compared against an LLVM compiler also built on revision 129859, applying fixed high-quality parallelization schemes (PS-DSWP [75, 95] and DOANY [71,102]). Both the manual parallelization and the LLVM parallelizing compiler target an unloaded parallel platform, and use a standard thread-pool with load balancing by the Linux scheduler [71,95]. The Morta run-time system is implemented on top of the same thread-pool, facilitating fair comparison.

Parcae performance is also compared with that of the optimal static configuration for each workload, which is determined via exhaustive search. The execution time of the sequential program produced by the same set of optimizations, excluding the parallelization transformations, serves as the baseline for speedup computation of all parallel runs. All execution times used for speedup computation are averages over N runs, where N is the larger of three and the minimum number of runs required for a standard deviation of less than 5% of the geomean execution time.

Feature	Platform 1	Platform 2
Brand Name	Intel Xeon E5310	Intel Xeon X7460
Processor	Intel Core, 64-bit	Intel Core, 64-bit
Clock Speed	1.60 GHz	2.66 GHz
Total # Threads	8	24
Total L2 Cache	16 MB	36 MB
Total L3 Cache	-	64 MB
Total RAM	8 GB	24 GB
OS	Linux 2.6.32	Linux 2.6.31

Table 8.1: Hardware platforms used for evaluation

In the case of applications with online server behavior, the arrival of tasks was simulated using a task queuing thread that enqueues tasks to a work queue according to a Poisson distribution. The average arrival rate determines the load factor on the system. A load factor of 1.0 corresponds to an average arrival rate equal to the maximum throughput sustainable by the system. The maximum throughput is determined as M/T where M is the number of tasks and T is the time taken by the system to execute the tasks in parallel (but executing each task itself sequentially). To determine the maximum throughput for each application, M was set to 500.

8.1 Chapter Organization

We present details of evaluation of the Parcae API. Specifically, we discuss ease of use of the API, and performance of the optimization mechanisms implemented by a mechanism developer. Then, we present details of evaluation of the Nona compiler and the closed-loop platform-wide optimization mechanism. Finally, we present details of the overheads of various aspects of Morta.

8.2 Parcae API

Table 8.2 provides a brief description of the applications that have been enhanced using Parcae. All are computationally intensive parallel applications. These are

Application	Description
x264	Transcoding of yuv4mpeg videos (PARSEC) [13]
swaptions	Option pricing via Monte Carlo simulations (PARSEC) [13]
bzip	Data compression of SPEC `ref` input (SPEC CPU2006) [18, 84]
gimp	Image editing using oilify plugin (GIMP) [31]
ferret	Image search engine (PARSEC) [13, 53]
dedup	Data deduplication (PARSEC) [13]

Table 8.2: Real-world applications enhanced using Parcae.

Application	Lines of Code				
	Added	Modified	Deleted	Fused	Total
x264	72	10	8	-	39617
swaptions	85	11	8	-	1428
bzip	63	10	8	-	4652
gimp	35	12	4	-	1989
ferret	97	15	22	59	14781
dedup	124	10	16	113	7546

Table 8.3: Columns 2–6 indicate the effort required to port the original Pthreads based parallel code to the Parcae API. Where applicable, column 6 indicates the number of lines of code in tasks created by fusing other tasks.

full real-world applications, not small kernels. The first column in the table shows the names that will be used to refer to the benchmarks in the remainder of this dissertation.

Columns 2–6 in Table 8.3 are indicative of the effort required to port existing Pthreads based parallel versions of the applications to the Parcae API. The nature of the changes has already been illustrated in Chapter 5. The number of additional lines of code written by the application developer could be significantly reduced with compiler support for functor creation and variable capture in C++, such as the support for lambdas in Intel's C++ compiler [38]. Anecdotally, a single programmer was able to port a production-quality Pthreads parallel version of `bzip2` to use the Parcae API, in the course of an afternoon. The programmer was unfamiliar with both the Parcae API and the original Pthreads code base.

Table 8.4 shows the mechanisms that were implemented by a mechanism de-

Mechanism					
WQT-H	WQ-Linear	TBF	FDP [85]	SEDA [99]	TPC
28	9	89	94	30	154

Table 8.4: Lines of code to implement tested mechanisms

veloper. While we have explored more combinations, for all but one application in Table 8.3, we present results on one performance goal. For one application—an image search engine (`ferret`)—we present results on all the tested performance goals.

8.2.1 Goal: "Minimize Response Time with N Threads"

The applications studied for this goal are video transcoding, option pricing, data compression, image editing, and image search. All applications studied for this goal have online service behavior. Minimizing response time is most interesting in the case of applications with nested loops due to the potential latency-throughput tradeoff described in Chapter 2. The outermost loop in all cases iterates over user-transactions. The amount of parallelism available in this loop-nesting level varies with the load on the servers.

Figures 8.1–8.4 show the performance of the WQT-H and WQ-Linear mechanisms compared to the static $<C_{outer}, C_{inner}>$ configurations $<(DOALL, 24), (SEQ, 1)>$ and

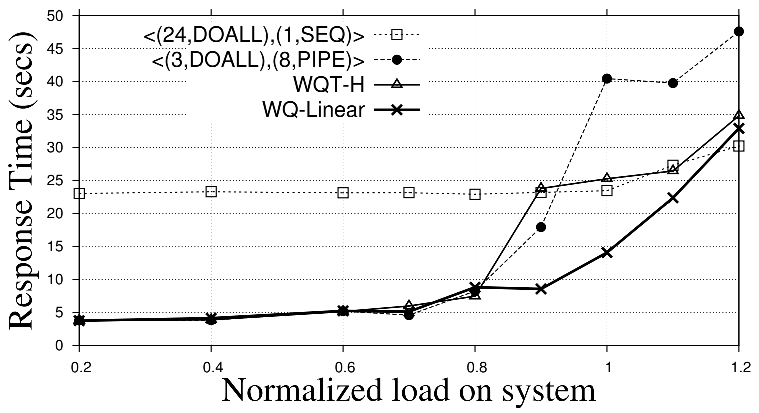


Figure 8.1: Video transcoding: Response time variation with load using Static, WQT-H, and WQ-Linear mechanisms

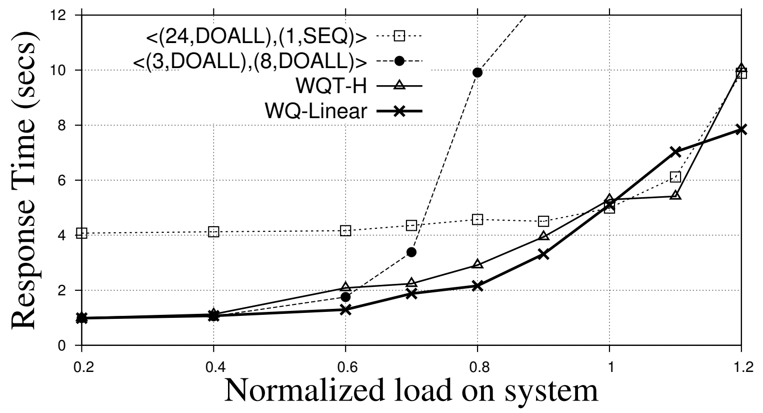


Figure 8.2: Option pricing: Response time variation with load using Static, WQT-H, and WQ-Linear mechanisms

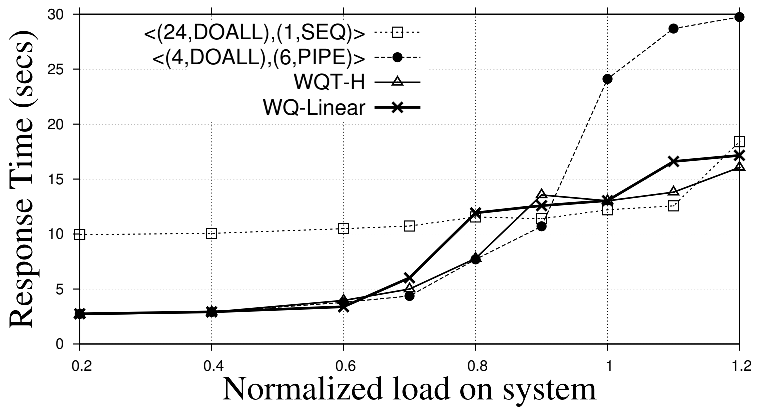


Figure 8.3: Data compression: Response time variation with load using Static, WQT-H, and WQ-Linear mechanisms

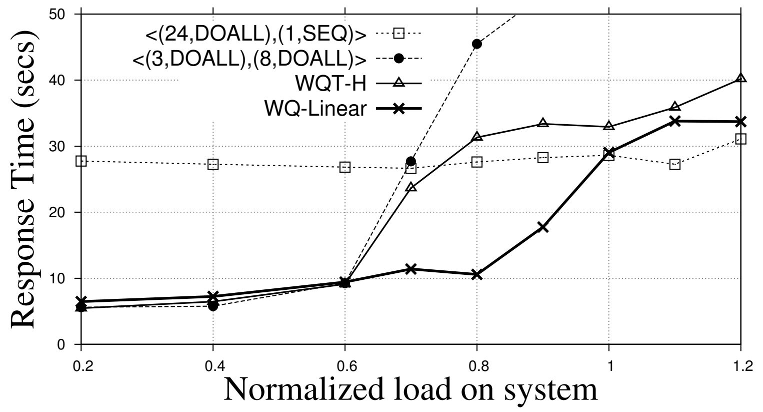


Figure 8.4: Image editing: Response time variation with load using Static, WQT-H, and WQ-Linear mechanisms

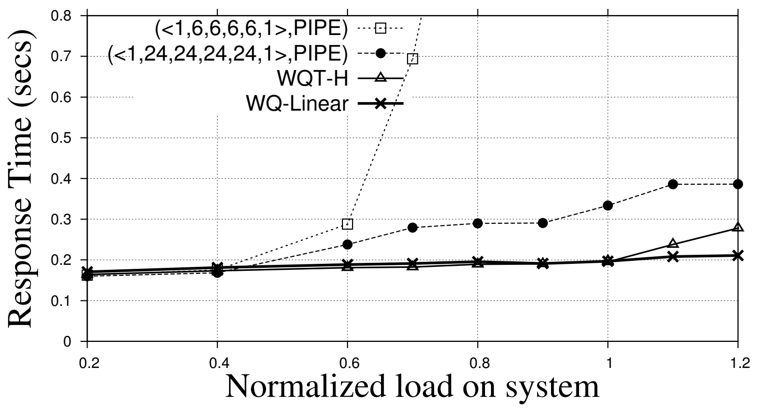


Figure 8.5: Image search engine: Response time variation with load using Static, WQT-H, and WQ-Linear mechanisms

$<(DOALL, N/d_{P_{max}}), (PIPE \mid DOALL, d_{P_{max}})>$. Here, N refers to the total number of threads available on the platform (24) and $d_{P_{max}}$ refers to the degree of parallelism of the inner loop above which parallel efficiency drops below 0.5.

Interestingly, WQT-H outperforms both static mechanisms at certain load factors. For example, consider the response times at load factor 0.8 in Figure 8.2. Analysis of the work queue occupancy and DoP assignment to tasks reveals that even though the load factor is on average equal to 0.8, there are periods of heavier and lighter load. Morta's dynamic adaptation of the DoP between $C = <(DOALL, 24), (SEQ, 1)>$ and $C = <(DOALL, N/d_{P_{max}}), (PIPE \mid DOALL, d_{P_{max}})>$ results in an average DoP somewhere in between the two, and this average DoP is better than either for minimizing response time. This provides experimental validation of the intuitive rationale behind WQ-Linear, which provides the best response time characteristic across the load factor range. In the case of data compression (Figure 8.3), the minimum DoP $d_{P_{inner}}$ at which speedup is obtained over sequential execution is four. This results in two problems for WQ-Linear. First, WQ-Linear may explore unhelpful configurations such as $<(DOALL, 8), (PIPE, 3)>$. Second, the number of configurations at WQ-Linear's disposal is too few to provide any improvement over WQT-H.

Figure 8.5 shows the response time characteristic of `ferret`. The figure shows the static distribution of threads to each pipeline stage. ($PIPE$,<$1, 6, 6, 6, 6, 1$>) indicates a single loop parallelized in a pipelined fashion with six threads allocated to each parallel stage and one thread allocated to each sequential stage. Similarly, ($PIPE$,<$1, 24, 24, 24, 24, 1$>) indicates the same loop parallelized in a pipelined fashion with 24 threads allocated to each parallel stage and one thread allocated to each sequential stage. Oversubscribing the hardware resources by allocating 24 threads to each parallel task results in much improved response time compared to a static even distribution of the 24 hardware threads. WQ-Linear achieves a much better characteristic by allocating threads proportional to load on each task.

8.2.2 Goal: "Maximize Throughput with N Threads"

For batch-processing applications, a desirable performance goal is throughput maximization. Parcae uses the mechanisms described in Section 6.3.2 to improve the throughput of an image search engine and a file deduplication application.

Table 8.5 shows the throughput improvements for `ferret` and `dedup` using different mechanisms. Pthreads-Baseline is the original Pthreads parallelization with a static even distribution of available hardware threads across all the parallel tasks after assigning a single thread to each sequential task. (This is a common practice [61].) The Pthreads-OS number shows the performance when each parallel task is initialized with a thread pool containing as many threads as the number of available hardware

Apps.		ferret	dedup
Pthreads	Baseline	1.00×	1.00×
	OS	2.12×	0.89×
Parcae	SEDA [99]	1.64×	1.16×
	FDP [85]	2.14×	2.08×
	TB	1.96×	1.75×
	TBF	2.35×	2.36×

Table 8.5: Throughput improvement over static even thread distribution

threads in the platform, and the operating-system's scheduler is called upon to do load balancing. The remaining numbers represent the performance of the applications enhanced using Parcae. Parcae-TB is the same as Parcae-TBF but with task fusion turned off, in order to demonstrate the benefit of task fusion.

Parcae-TBF outperforms all other mechanisms. OS scheduling causes more context-switching, cache pollution, and memory consumption. In the case of `dedup`, these overheads result in virtually no improvement over the baseline. The overheads may become prominent even in the case of `ferret` on a machine with a larger number of cores. In addition, this mechanism is still a static scheme that cannot adapt to run-time events such as more cores becoming available to the application. Each task in SEDA resizes its thread pool locally without coordinating resource allocation with other tasks. By contrast, both FDP and TBF have a global view of resource allocation and are able to redistribute the hardware threads according to the throughput of each task. Additionally, FDP and TBF are able to either fuse or combine tasks in the event of very uneven load across stages. Compared to FDP, which simulates task fusion via time-multiplexed execution of tasks on the same thread, TBF has the additional benefit of avoiding the overheads of forwarding data between tasks by enabling the developer to explicitly expose the appropriate fused task.

Figure 8.6 shows the dynamic throughput characteristic of `ferret`. Morta

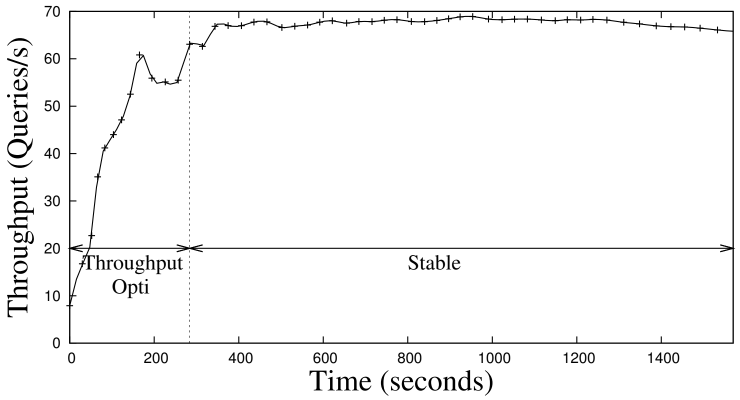


Figure 8.6: Image search engine: Throughput variation using TBF mechanism

searches the parallelism configuration space before stabilizing on the one with the maximum throughput under the constraint of 24 hardware threads.

8.2.3 Goal: "Maximize Throughput with N Threads, P Watts"

Figure 8.7 shows the operation of Parcae's power-throughput controller (TPC) on `ferret`. For a peak power target specified by the administrator, Parcae-TPC first ramps up the degree of parallelism until the power budget is fully used. Parcae-TPC then explores different parallelism configurations and stabilizes on the one with the best throughput without exceeding the power budget. Note that 90% of peak total power corresponds to 60% of peak power in the dynamic CPU range (all cores idle to all cores active). Morta achieves the maximum throughput possible at this setting. Fine-grained per-core power control can result in a wider dynamic range and greater power savings [82]. Full system power was measured at the maximum sampling rate (13 samples per minute) supported by the power distribution unit (AP7892 [6]). This limited the speed with which the controller responds to fluctuations in power consumption. Newer chips have power-monitoring units with higher sampling rates and clock gating per core. They could be used to design faster and higher performance

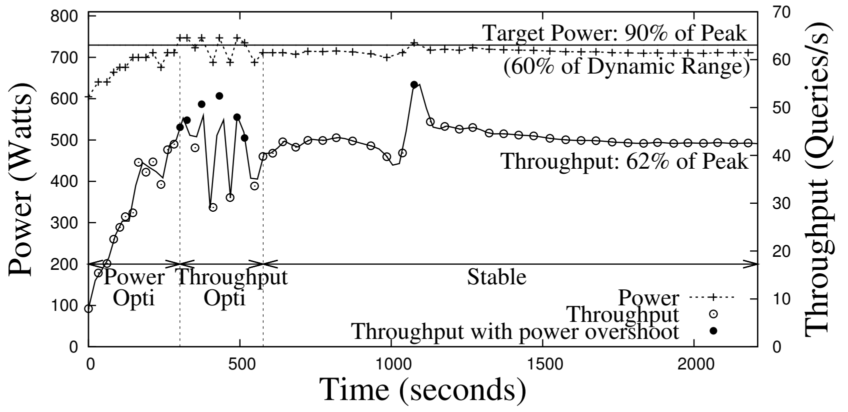


Figure 8.7: Image search engine: Power-throughput variation using TPC mechanism

controllers for throttling power and parallelism. The transient in the Stable region of the figure shows how constant monitoring by Decima enables Parcae to respond to system events.

8.2.4 Summary

The experiments highlight fulfillment of the framework's design objective of enabling change of optimization goals and mechanisms without changing the application code. Referring back to Table 8.4, the number of lines of code required to implement the mechanisms evaluated above is quite modest, considering the performance benefits that they deliver. We now present evaluation results of the Nona compiler and the closed-loop platform-wide mechanism that is designed based on the advantages and disadvantages of the mechanisms evaluated thus far.

8.3 Nona Compiler

Table 8.6 describes the seven benchmarks selected to evaluate the Nona compiler and the Parcae run-time system. These include benchmarks parallelized by the baseline state-of-the-art compiler [71], allowing direct comparison. Table 8.7 gives detailed information about each benchmark including the targeted code region, fraction of total execution time spent in parallelized regions, and the number of invocations and iterations of the parallelized regions. Finally, Table 8.8 provides a summary of the execution time and energy improvements delivered by Parcae. These improvements are discussed in detail below.

8.3.1 Power/Energy Measurement Methodology

Processors will have interfaces to turn cores on or off at fine time granularities [14]. We evaluated the energy savings that Parcae could deliver by leveraging such capabilities.

Application	Description
blackscholes	Portfolio pricing with the Blackscholes PDE (PARSEC) [13]
geti	Data mining of frequent itemsets (MineBench) [60]
kmeans	K-means clustering (STAMP) [58]
ks	Kernighan-Lin graph partitioning algorithm (LLVM Test Suite) [51]
md5sum	md5sum computation (Open Source) [7]
potrace	Vectorization of bitmap images (Open Source) [81]
url	URL based packet switching (NetBench) [57]

Table 8.6: Sequential applications automatically enhanced using the Nona compiler.

Program	Main Loop	Parallel Coverage	Total number of	
			Invocations	Iterations
blackscholes	`worker`	92.9%	1	1000
geti	`FindSomeETIs`	100.0%	24	12242
kmeans	`work`	99.30%	501	131334144
ks	`FindMaxGpAndSwap`	100.0%	7750	488250
md5sum	`main`	100.0%	1	384
potrace	`main`	100.0%	1	200
url	`main`	100.0%	1	40000

Table 8.7: Sequential programs transformed, targeted code region, fraction of total execution time spent in parallelized regions, and number of invocations and iterations of parallelized regions.

In Section 8.2, power was measured online in spite of the low sampling rate of the power distribution unit (AP7892 [6]). This was feasible because the applications and workloads were real-world and long-running. For evaluating the compiler, however, we use benchmarks from standard benchmark suites. These are short-running, and power measurements need to be more frequent than the supported rate. Hence, we employ an alternative methodology to determine power at different points during program execution. We measured the power consumption characteristic of each platform by executing a power-virus that maintains 100% CPU utilization of each core to measure power draw when a number of cores are busy, and by leaving the system unused to measure power draw when those cores are idle. The processor's dynamic power range is then the difference between active power and idle power. Active processor power consumption as a fraction of whole platform power consumption is 18.0% for

Program	Parcae improvement relative to		
	Baseline		Optimal
	Exec. Time	Energy	Exec. Time
blackscholes	-1.39%	45.35%	-1.39%
geti	12.55%	23.90%	-18.56%
kmeans	41.94%	83.89%	-62.98%
ks	3.78%	35.11%	-0.83%
md5sum	34.41%	58.51%	-3.55%
potrace	9.95%	42.67%	-2.97%
url	-0.18%	36.22%	-2.40%

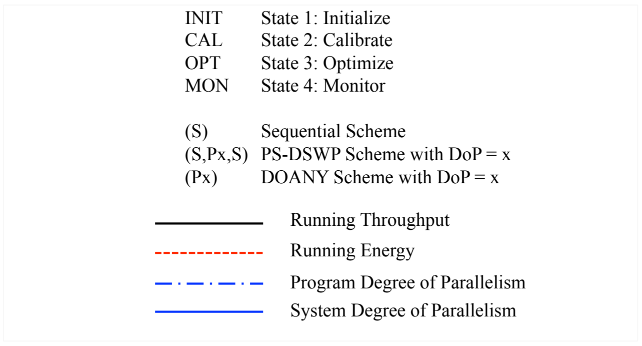
Table 8.8: Improvements in execution time and energy delivered by Parcae. The improvements are on a single input on Platform 2, and are relative to baseline parallel and optimal parallel versions of these programs. Optimal versions are determined via offline exhaustive search. Positive numbers indicate improvement.

Platform 1 and 26.7% for Platform 2. Processor energy is then computed as the integral of the piece-wise product of time spent actively using a number of cores and the average power draw of those many active cores. Recent work uses a similar methodology [14].

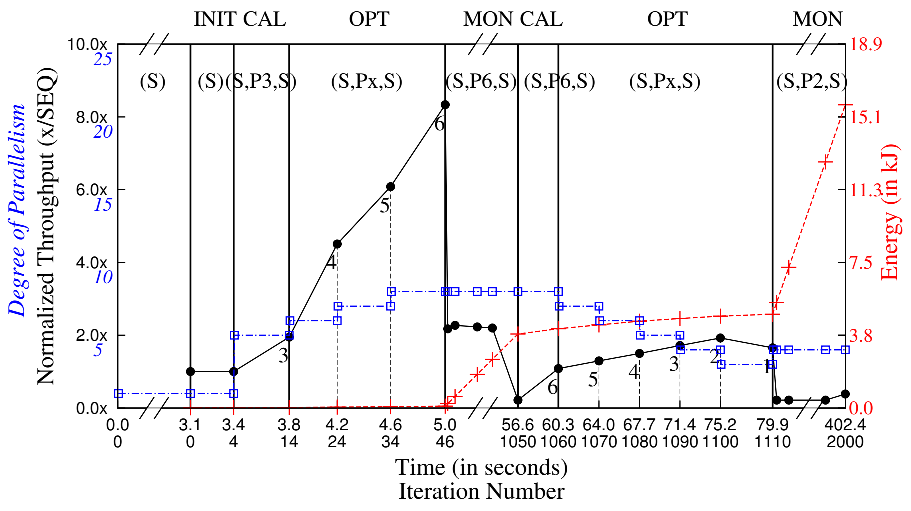
We present case studies using one application for three interesting scenarios, each highlighting a different aspect of Parcae. Table 8.8 provides a summary of the execution time and energy improvements delivered by Parcae across all benchmarks.

8.3.2 Parcae adapts execution to workload change

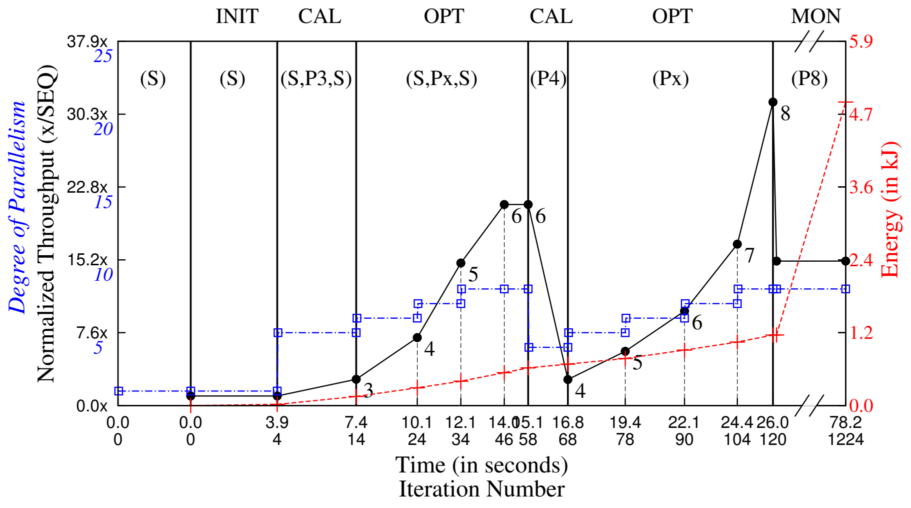
`blackscholes` calculates the prices for a portfolio of European options using the Black-Scholes partial differential equation [13]. To study workload variation, the source code is modified to enable pricing of multiple portfolios. Nona is able to apply the PS-DSWP parallelization scheme, in addition to SEQ. Figure 8.8(b) shows Morta and Decima in action. Morta starts in State 1 to initialize the sequential baseline. It then enters State 2 to set the initial configuration and determine the direction for gradient ascent. In State 3, it performs gradient ascent until peak throughput is achieved, at which point ($t = 5.0s$) it enters State 4 to monitor the execution of this



(a) Legend



(b) `blackscholes`: Controller re-optimizes when workload changes



(c) `geti`: Controller determines optimal execution scheme from many

Figure 8.8: Parcae run-time control. Solid vertical lines indicate state transition times. All throughput values are normalized to the throughput measured in the INIT state. States shown at the top of each figure are the states of the program's controller.

configuration. During the optimization phase, the re-configuration interval constantly varies according to the optimization logic; this results in compute throughput measurements that are accurate relative to each other, but are less accurate with respect to the baseline throughput. Hence, once the peak is reached, the system stabilizes for additional iterations to obtain a more accurate throughput measurement. Decima's workload change detection logic kicks in after this throughput measurement. At $t = 56.6s$, Decima detects workload change via drop in throughput, and signals Morta to re-optimize in response. In State 3, Morta reduces DoP by performing decreasing gradient ascent, finally to reach State 4 at $t = 79.9s$; Decima monitors the remaining execution. Parcae achieves performance within 1.4% of optimal (due to overheads) while consuming 45.4% less energy.

8.3.3 Parcae optimizes across multiple parallelization schemes

`geti` is a data mining program that computes a list of frequent itemsets using a vertical database. The set semantics of the output operations enables order relaxation via commutativity, which is expressed using 11 annotations in 889 lines of source code. The resulting dependence graph is then amenable to PS-DSWP and DOANY parallelization schemes (in addition to SEQ). Figure 8.8(c) shows the execution trace. As with `blackscholes`, Morta starts in State 1 to initialize the sequential baseline. In State 2, it calibrates an initial configuration of the PS-DSWP scheme, and starts an increasing gradient ascent in State 3. Morta determines an optimal configuration for the PS-DSWP scheme (at $t = 14.0s$), and compares throughput with that of the SEQ scheme. As the comparison is below the preset threshold, Morta returns to State 2 in order to evaluate the next parallel scheme: DOANY. Optimizing the DoP for DOANY then takes place in State 3 (from $t = 16.8s$ until $t = 26.0s$). The optimal DOANY configuration is found to be of adequate throughput, so Morta moves to

State 4 and Decima monitors the remaining execution. Parcae improves performance over the baseline by 12.6%, and achieves performance within 18.6% of optimal. Parcae reduces energy consumption by 23.9%.

8.3.4 Parcae adapts execution to resource availability change

Figure 8.9 illustrates Morta's platform-wide control. Eight copies of `blackscholes` are launched in succession on Platform 2 having 24 cores. For brevity, the execution of only one copy P_0 is shown (labeled *Program*). The figure also shows platform-wide dynamic thread utilization as determined by the platform-wide daemon (labeled *System*). P_0 begins execution on an unloaded machine, so following the initialization of State 1, State 2 calibrates the parallel task of the PS-DSWP scheme starting from a DoP of $d_{P_0} = \frac{1}{2}\overline{d_{P_0}} = 11$. (There are two additional sequential tasks resulting in a whole program DoP of 13.) P_0's controller then transitions to State 3 and performs an increasing gradient ascent. By $t = 3.6s$, all eight program copies have launched, and the daemon reallocates threads across them, assigning a DoP of three to each copy. In response, P_0's controller transitions to State 2 and measures the parallel performance of configuration `(S,P1,S)`, which is now the only feasible parallel con-

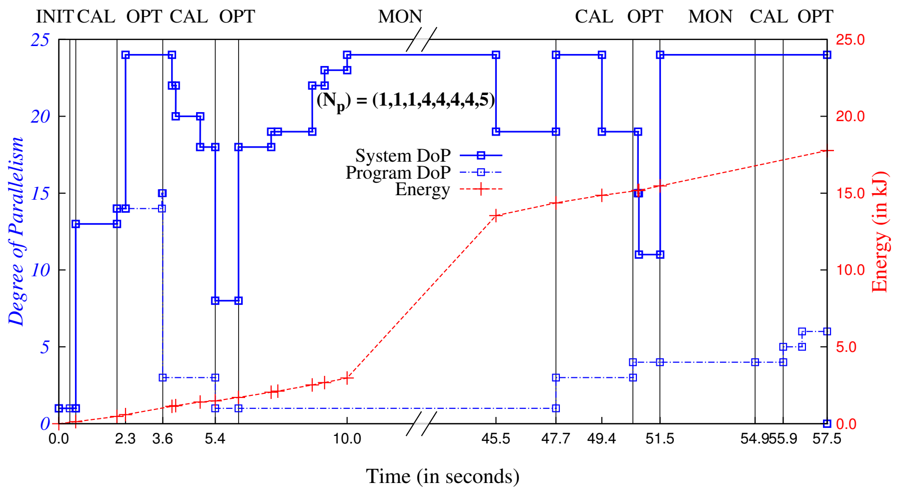


Figure 8.9: Controller optimizes multiple programs simultaneously

figuration. It then computes the configuration's efficiency and compares the efficiency with that of sequential execution, determining at time $t = 5.4s$ that sequential execution is preferred, and returning two threads back to the daemon. Spare threads are redistributed by the daemon among the copies. By $t = 10.0s$, the platform-wide optimization algorithm converges (at $\{N_p\} = \{1, 1, 1, 4, 4, 4, 4, 5\}$), all copies enter their stable configurations, and the controllers enter State 4. At $t = 45.5s$, the fastest copy terminates, yielding its (five) threads back to the daemon, which in turn initiates another round of optimization. The figure shows how, barring the short optimization phases, Parcae enables full use of all threads on the platform. Compared to OS scheduling of the eight copies of the baseline 24-thread parallel version, Parcae reduces total execution time by 12.8%, and energy consumption by 29.1%.

Interestingly, Parcae benefits increase as the number of concurrently executing programs increases. With 16 copies, execution time and energy improve by 18.3% and 52.7%, respectively. With 24 copies, improvements reach 22.4% and 22.4%, respectively. The co-operative release and acquisition of resources by Parcae reduces contention among concurrently executing programs. Scheduling and memory contention significantly overwhelm the benefits of parallel execution when copies of the baseline parallelized programs execute each with 24 threads.

8.3.5 Optimality: A Closer Look

While Parcae delivers significant performance improvements, its achievements are still short of optimal execution determined via offline exhaustive search (see Table 8.8) for two main reasons. For `blackscholes`, `geti`, `md5sum`, and `potrace`, Parcae identifies the optimal configuration, but converges to that solution after first spending multiple iterations in `SEQ` and other sub-optimal schemes. In addition to the above overhead, for `kmeans`, `ks`, and `url`, the assumption about unimodal performance characteristic with increasing DoP (discussed in Section 6.4.2) does not hold; Parcae identifies a local optimum, which is worse than the global optimum. The Parcae gradient-ascent algorithm can be enhanced to escape from local optima by employing techniques such as simulated annealing. We leave this to future work.

8.3.6 Morta and Decima Overheads

The execution time improvements reported in Table 8.8 include all overheads of the system. Table 8.9 shows each overhead independently. The first three overheads are recurring (incurred on each instance of each task). To quantify them, we executed microbenchmarks designed to exercise each overhead independently. All aspects of the run-time control system are implemented in shared memory, thus making the run-time operations extremely lightweight. In the common case, task instances are executed on the same core, resulting in low-latency access to cross-iteration dependency data. Timestamp acquisitions are not serializing instructions; their latencies may be masked naturally via hardware exploiting instruction-level parallelism. The recurring overheads are less than 0.1% of task execution times for all benchmarks. "Pause-Resume" and "Parallelism Reconfiguration" overheads are application-specific and DoP-specific. Overheads shown are for `blackscholes` with maximum DoP.

Operation	**Overhead**		
	Platform 1	Platform 2	Unit
Task Migratability			
- Task Activation	11	7	nanoseconds per task instance
- Data Management	3	2	nanoseconds per dependency per task instance
Status Query	8	5	nanoseconds per query
Monitoring	44	15	nanoseconds per timestamp
Pause-Resume	33	4	milliseconds per pause-resume
Parallelism Reconfiguration	35	2	milliseconds per reconfiguration

Table 8.9: Recurring run-time overheads

Chapter 9

Related Work

Prior work falls into four categories corresponding to the level of the system stack at which changes for flexible execution are proposed.

9.1 Parallel Programming Models

Programming models for parallelization and flexible execution can be categorized into general-purpose models and domain-specific models.

9.1.1 General-purpose Parallel Programming Models

Several interfaces and associated run-time systems have been proposed to adapt parallel program execution to run-time variability [16, 25, 28, 45, 59, 77, 85, 90, 94, 97, 99]. However, each interface is tied to a specific performance goal, specific mechanism of adaptation, or a specific parallelism type. Most run-time systems enabling the parallel programming interfaces are based on work stealing [2, 8, 17, 50, 77, 93]. They support task parallelism for independent tasks (parallel function calls) and their schedulers optimize only for throughput. The Cilk++ and Intel TBB schedulers support DOALL [3] loop parallelism, again targeting the only goal of throughput maximization. Cilk++ requires the programmer to specify the degree of parallelism (DoP) explicitly by trying out various values, thus limiting performance portability and increasing programming effort since this needs to be done for each loop of each application [50,93]. Intel TBB supports two partitioners—the Static Partitioner (SP), which requires the programmer to explicitly specify the degree of parallelism [79]; and the Auto Partitioner (AP), which frees the programmer from choosing the degree of parallelism, but is not run-time adaptive [93]. The Lazy Splitting (LS) scheduler [12,93] is run-time adaptive. There are several points of difference between Parcae and LS: (i) Parcae handles conditional loops whereas LS handles only counted loops (loops with static trip counts, e.g., `for i in 1:N do stmt`); (ii) Parcae can optimize multiple loop-level parallelism types whereas LS handles only DOALL parallel loops; (iii) the Parcae run-time system, in conjunction with Nona and the Parcae API, optimizes across multiple parallelizations of the same parallel region, whereas Intel TBB with LS has no such capability; and (iv) Morta can optimize for different performance goals specified by the administrator, whereas LS can optimize only for throughput. Navarro et al. developed an analytical model for pipeline parallelism to characterize performance and efficiency of pipeline parallel implementations [61]. Suleman et al. proposed Feedback Directed Pipelining (FDP) [85]. Moreno et al. proposed a similar technique called Dynamic Pipeline Mapping (DPM) [59]. These run-time systems optimize a single program in isolation. By contrast, Parcae optimizes multiple programs running concurrently.

9.1.2 Domain-specific Parallel Programming Models

Traditionally, multiple levels of parallelism across tasks and within each task have been investigated in the database research community for SQL queries [24,33,88,89]. For network service codes, programming models, such as the Stage Event-Driven Architecture (SEDA) [99] and Aspen [94], have been proposed. Blagojevic et al. have proposed user-level schedulers that dynamically "rightsize" the loop nesting level and extent of parallelism on a Cell Broadband Engine system [16]. All these systems are typically restricted to a single form of parallelism (either loop-level DOALL, task parallelism, or pipeline parallelism) that is dominant in the domain. They do not support multiple performance goals and constraints. General-purpose applications tend to have multiple types of parallelism in loop nests, and different goals and constraints.

9.2 Compiler and User-level Run-time Systems

Most work on automatic parallelization at compile-time (e.g., [3, 55, 75, 95, 104]) or run-time [18, 20, 21, 23, 35, 46, 68, 70, 72, 74, 76, 80, 92, 103] is primarily concerned with parallelism discovery and extraction. Parcae addresses the complementary problem of parallelism optimization.

Little attention has been given so far to compiling general-purpose applications with flexible execution capabilities in mind. Ko et al. describe a system that uses a simple heuristic to find the right DoP for each level (inter-node, intra-node) in a cluster environment, focusing on applications with MPI/OpenMP parallelizations [45]. To reduce search time, their compiler creates a program slice, a small code segment isolated from a program that represents its performance behavior. Wang et al. use machine learning to predict the best number of threads for a given program on a particular hardware platform [97]. They apply their technique to a single loop in isolation. Luk et al. use a dynamic compilation approach and curve fitting to find the optimal distribution of work between a CPU and GPU [52]. Adaptive Thread Management throttles the number of threads allocated to compiler-parallelized DOALL loops at run-time, as their measured speedup exceeds or falls short of the expected speedup [32]. The ADAPT dynamic optimizer applies loop optimizations at run-time to create new variants of code [96]. Parcae employs multiple parallelization schemes prepared at compile-time to support efficient optimization at run-time, and optimizes multiple programs co-scheduled on a system. Diniz and Rinard implemented "dynamic feedback" in the context of a parallelizing compiler to determine the best synchronization policy from among those exposed by the compiler [29]. Using commutativity analysis, the compiler parallelizes a code region, applies multiple synchronization schemes with varying costs, and exposes each code version with a particular synchronization scheme to the run-time system. At run-time, the system measures the performance of each synchronization scheme during the "sampling phase", and uses the best performing scheme during the "production phase". Parcae differs from this system in various respects. First, Parcae addresses the more general problem of determining optimal parallelism configurations. The Parcae compiler parallelizes a code region in multiple ways (employing different forms of parallelism such as pipeline parallelism and data parallelism, potentially using different synchronization schemes) and exposes them to the run-time system. Second, "dynamic feedback" employs fixed, user-specified durations of the sampling and production phases, and employs a single sampling phase and a single production phase. This strategy fails to detect phase changes. By contrast, the Parcae run-time system periodically monitors execution to detect phase changes. Third, "dynamic feedback" operates on a single program in isolation, whereas Parcae optimizes multiple programs co-scheduled on a system. Fourth, "dynamic feedback" uses synchronous switching between program configurations, which can be quite expensive (as Diniz and Rinard remark [29]). Parcae addresses this by a novel mechanism for efficient pausing of one set of communicating tasks, followed by the resumption of the parallel region's execution by a different set of communicating tasks.

Auto-tuning is an approach for optimizing performance-impacting parameters. In linear algebra algorithms, a very important parameter is the blocking or tiling factor, which directly effects the data cache utilization. Such parameters can be provided as compile-time variables in order to support performance portability across different platforms [5, 10, 43]. However, not all important architectural variables can be handled by such parameterized adaptation (e.g. choice of combined or separate multiply and add instructions); some choices require changing the underlying source code. This type of adaptation involves generating distinct implementations for the same operation [5]. Frameworks such as PetaBricks [4] and Elastin [62] leverage alternative code implementations exposed by programmers. Parcae automatically generates multiple versions of a parallel region from standard source code, and selects and tunes them at run-time based on performance feedback.

9.3 Operating Systems

Bird and Smith outline the need to revamp operating system schedulers for the multicore era [15]. Specifically, they too emphasize the need for performance aware convex optimization for resource allocation. Parcae is a real implementation of a parallelism optimization and resource allocation manager that uses convex optimization concepts. The number of threads active in an application's thread-pool is determined across runs of an application by the operating system [42]. Parcae optimizes parallelism as applications are executing, not across invocations alone. Peter et al. outline design principles for future operating systems for multicores including the need for tighter integration between parallel application run-time systems and operating system schedulers [67]. Parcae demonstrates the benefits of such an integration. Parcae's monitoring service could serve as the application and platform monitoring module, while its resource management daemon could serve as the parallel resource allocation and management module in operating systems for manycore processors such as Barrelfish [11] and fos [100].

9.4 Hardware

CoreFusion [39] consists of a hardware design that can dynamically adjust itself to best match the parallelism configuration of the program that it executes. In this approach, a group of relatively thin cores dynamically morphs into a larger processor with more micro-architectural resources to balance throughput and latency. Wells et al. propose hardware techniques for multicore virtualization to enable resource management applications for improving performance and reliability [98]. Ding et al. propose use of helper threads that monitor hardware events and use curve fitting to determine the ideal number of cores to employ in order to minimize the energy-delay product (EDP) when the availability of cores varies [28]. Curtis-Maury et al. employ an IPC prediction scheme to determine the right balance between the number of SMT threads/core and number of cores to be used by an application for near-optimal energy-efficient performance [25]. Chen et al. proposed network-driven processing (NDP) for chip multiprocessors that dynamically maps threads to cores at run-time using hardware thread management mechanisms [22].

Chapter 10

Future Directions and Conclusions

This dissertation introduced a framework for enabling performance portability of parallel programs. This chapter discusses potential avenues of future research opened by the research presented in this dissertation, and summarizes the conclusions of the research described in this dissertation.

10.1 Avenues of Future Research

1. Port to restricted shared-memory processors: Future multicore or manycore processors may have restricted shared-memory domains [44, 100]. The data structures and optimization algorithms used by the Morta run-time system may need to be re-factored appropriately for non-shared memory execution in order to use Morta to control execution of parallel programs on such processors.

2. Orchestrate execution on heterogeneous processors: Future processors may be heterogeneous with a few powerful cores and many smaller cores [36, 48]. They may share the same instruction set architecture. Morta can be extended relatively easily to schedule heavyweight tasks (determined via application feature monitoring) on to powerful cores.

Cooperative CPU-GPU parallelizations have demonstrated significant performance gains over CPU-only or GPU-only parallelization [41]. For best performance across CPUs and GPUs that have differing strengths, applications should be parallelized at multiple granularities. Depending on the cluster architecture, different parallelizations will be more or less efficient. Adapting to varying resources dynamically is especially important for large scientific clusters, where the number of nodes available for computation varies dramatically over time. Parcae can be extended to manage execution on such CPU-GPU clusters. More generally, on large-scale datacenters, Parcae can be scaled up to maximize utilization and minimize power consumption of clusters of multicore processors.

3. Enable flexible parallel execution on embedded platforms: Morta can be used as a lightweight run-time system for embedded multicore processors. On devices such as smartphones, Parcae with its ability to optimally trade performance and power can switch parallel program execution between performance-optimal and energy-optimal according to the state of the battery. Code-size optimizations that involve sharing parts of code across multiple parallel versions can be useful to reduce the negative performance impact on the instruction cache behavior of multi-versioned code.

4. Optimize for more complicated user-specified fitness functions: Interactive applications can have sophisticated fitness functions [15]. As an example, consider image search. Humans perceive system response as “instantaneous” provided top search results are returned within 100 milliseconds [19, 63]. User satisfaction begins to suffer when the latency increases above the threshold of 100 milliseconds. In other words, the fitness function is constant up to a latency of 100 milliseconds and falls, perhaps linearly, beyond latency of 100 milliseconds.

Morta can be made aware of such complicated fitness functions that may give it more slack to re-allocate resources.

5. Incorporate quality of service (QoS) metric: QoS can be added as another dimension in the performance-power-energy trade-off space in which Parcae determines the optimal operating point, given specific performance goals. Existing QoS auto-tuning frameworks can be incorporated into Parcae [9, 37].

6. Orchestrate speculative parallel execution: Speculation is a key enabler of parallelism in general-purpose programs [18, 69, 72, 92, 104]. Previous studies have shown that excessive speculation can lead to wasted computation as more transactions are aborted, resulting in energy-inefficient execution [78]. Speculation can be throttled dynamically at run-time, depending on observed misspeculation rates. Parcae can be extended such that the Nona compiler exposes misspeculation frequency as an application feature that Morta can monitor, and Morta can throttle the DoP to reduce misspeculation. This may improve energy efficiency since the amount of computation that is discarded is reduced. In the extreme, Parcae could choose a parallel version that employs pessimistic synchronization using locks, for example, as opposed to a parallel version that employs optimistic synchronization via transactions.

10.2 Conclusions

Parallel applications must execute robustly across a variety of execution environments arising out of variability in workload characteristics, platform characteristics, and performance goals. For this, a separation of concerns of parallel application development, its optimization, and use, is required. Parcae, introduced in this dissertation, enables such a separation. Using the Parcae API, 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. As a result of Parcae, they can be confident that the specified performance goals are met in a variety of application execution environments. Parcae improved the response time characteristics of four web service type applications to dominate the characteristics of the best static parallelizations. The throughputs of two batch-oriented applications were improved by 136% (geomean) over their original implementations. Throughput was also maximized under both number of cores and power consumption constraints.

Applications developed in the sequential programming model (including legacy applications) are automatically enhanced to execute flexibly on multicore platforms by the Nona compiler. The Parcae run-time system, comprising the Decima monitor and the Morta executor, optimizes the execution of multiple flexible programs running on a shared parallel platform. The Parcae run-time system determines optimal degrees of parallelism for each parallelization scheme, as well as selects the optimal parallelization scheme from among multiple schemes prepared by the Nona compiler or programmer for each parallel region. Compared to conventional parallel execution of seven benchmarks, Parcae reduced execution time by -1.39% (a small slowdown) to 41.94% with concomitant energy savings of 23.9% to 83.9%.

Use of the Parcae API and consequent performance improvements suggest that the ability to expose and optimize parallelism across multiple levels in a loop nest, the ability to express multiple types of parallelism simultaneously, and the ability to expose application features to the run-time system are all crucial to maximize parallel program performance. An optimization schema consisting of trying a new parallelism configuration, measuring its fitness vis-à-vis the specified performance goal, and triggering reconfiguration if the current configuration is unfit, enables the specification of different, complex performance goals. The process of online parallelism reconfiguration is an essential component of performance optimization in the face of variability in the execution environment, and is complicated by inter-task communication. This dissertation described means for low-overhead online parallelism reconfiguration while respecting original program semantics. For performance guarantees, such as minimum throughput requirement or maximum power consumption constraint, this dissertation demonstrated that online closed-loop control is key. Finally, platform-wide experiments with the Parcae run-time system suggest that cooperative use of parallel resources, as orchestrated by a platform-wide resource manager in concert with each application's run-time system, is essential to maximize platform utilization. Tighter integration between the application-level run-time system and the operating system's scheduler yields significant performance gains compared to when the two operate in isolation.

The Parcae software system is designed to support additional, new parallelization optimizations and different, potentially adaptive platform-level optimization objectives. With the introduction of architectures with more cores, heterogeneous cores, complex memory hierarchies and communication fabrics, and sophisticated performance goals demanded by users, a holistic, automatic, dynamic, and generally applicable parallelism tuning system will only become more relevant.

Bibliography

[1] The OpenMP API specification. http://www.openmp.org.

[2] K. Agrawal, Y. He, and C. E. Leiserson. Adaptive work stealing with parallelism feedback. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 112–120, 2007.

[3] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc., 2002.

[4] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. PetaBricks: A language and compiler for algorithmic choice. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[5] C. W. Antoine, A. Petitet, and J. J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27:2001, 2000.

[6] APC metered rack PDU user's guide. http://www.apc.com.

[7] Apple Open Source. md5sum: Message Digest 5 computation. http://www.opensource.apple.com/darwinsource.

[8] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the 10th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 119–129, 1998.

[9] W. Baek and T. M. Chilimbi. Green: A framework for supporting energy-conscious programming using controlled approximation. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 198–209, 2010.

[10] M. M. Baskaran, N. Vydyanathan, U. K. R. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. Compiler-assisted dynamic scheduling for effective parallelization of loop nests on multicore processors. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 219–228, 2009.

[11] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*, pages 29–44, 2009.

[12] L. Bergstrom, M. Rainey, J. Reppy, A. Shaw, and M. Fluet. Lazy tree splitting. In *Proceedings of the 15th International Conference on Functional Programming (ICFP)*, pages 93–104, 2010.

[13] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008.

[14] O. Bilgir, M. Martonosi, and Q. Wu. Exploring the potential of CMP core count management on data center energy savings. In *Proceedings of the 3rd Workshop on Energy Efficient Design (WEED)*, 2011.

[15] S. L. Bird and B. J. Smith. PACORA: Performance aware convex optimization for resource allocation. In *Proceedings of the 3rd USENIX Workshop on Hot Topics in Parallelism (HotPar: Posters)*, 2011.

[16] F. Blagojevic, D. S. Nikolopoulos, A. Stamatakis, C. D. Antonopoulos, and M. Curtis-Maury. Runtime scheduling of dynamic parallelism on accelerator-based multi-core systems. *Parallel Computing*, 33(10-11):700–719, 2007.

[17] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 1995.

[18] M. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. August. Revisiting the sequential programming model for multi-core. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 69–84, 2007.

[19] J. D. Brutlag, H. Hutchinson, and M. Stone. User preference and search engine latency. In *JSM Proceedings, Quality and Productivity Research Section*, 2008.

[20] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 519–538, 2005.

[21] D.-K. Chen, J. Torrellas, and P.-C. Yew. An efficient algorithm for the run-time parallelization of DOACROSS loops. In *Proceedings of the 1994 ACM/IEEE Conference on Supercomputing (SC)*, pages 518 –527, Nov. 1994.

[22] J. Chen, P. Juang, K. Ko, G. Contreras, D. Penry, R. Rangan, A. Stoler, L.-S. Peh, and M. Martonosi. Hardware-modulated parallelism in chip multiprocessors. In *Proceedings of the 2005 Workshop on Design, Architecture, and Simulation of Chip Multi-Processors*, 2005.

[23] M. K. Chen and K. Olukotun. The Jrpm system for dynamically parallelizing java programs. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA)*, pages 434–446, 2003.

[24] C. B. Colohan, A. Ailamaki, J. G. Steffan, and T. C. Mowry. Optimistic intra-transaction parallelism on chip multiprocessors. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB)*, pages 73–84, 2005.

[25] M. Curtis-Maury, J. Dzierwa, C. D. Antonopoulos, and D. S. Nikolopoulos. Online power-performance adaptation of multithreaded programs using hardware event-based prediction. In *Proceedings of the 20th International Conference on Supercomputing (ICS)*, pages 157–166, 2006.

[26] R. Cytron. DOACROSS: Beyond vectorization for multiprocessors. In *Proceedings of the 1986 International Conference on Parallel Processing (ICPP)*, pages 836–884, 1986.

[27] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[28] Y. Ding, M. Kandemir, P. Raghavan, and M. J. Irwin. Adapting application execution in CMPs using helper threads. *Journal of Parallel and Distributed Computing*, 69(9):790 – 806, 2009.

[29] P. Diniz and M. Rinard. Dynamic feedback: An effective technique for adaptive computing. In *Proceedings of the 18th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1997.

[30] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9:319–349, July 1987.

[31] GNU Image Manipulation Program. http://www.gimp.org.

[32] M. W. Hall and M. Martonosi. Adaptive parallelism in compiler-parallelized code. In *Proceedings of the 2nd SUIF Compiler Workshop*, 1997.

[33] N. Hardavellas, I. Pandis, R. Johnson, N. Mancheril, A. Ailamaki, and B. Falsafi. Database Servers on Chip Multiprocessors: Limitations and Opportunities. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, 2007.

[34] J. L. Hellerstein, V. Morrison, and E. Eilebrecht. Applying control theory in the real world: Experience with building a controller for the .NET thread pool. *Performance Evaluation Review*, 37:38–42, 2010.

[35] B. Hertzberg and K. Olukotun. Runtime automatic speculative parallelization. In *Proceedings of the Annual International Symposium on Code Generation and Optimization (CGO)*, pages 64–73, 2011.

[36] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *IEEE Computer*, 2008.

[37] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard. Dynamic knobs for responsive power-aware computing. In *Proceedings of*

the Sixteenth International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 199–212, 2011.

[38] The Intel C/C++ Compiler. http://www.intel.com/software/products/compilers.

[39] E. Ipek, M. Kirman, N. Kirman, and J. F. Martinez. Core fusion: Accommodating software diversity in chip multiprocessors. *ACM SIGARCH Computer Architecture News*, 35(2):186–197, 2007.

[40] C. Isci, A. Buyuktosunoglu, C.-Y. Cher, P. Bose, and M. Martonosi. An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 347–358, 2006.

[41] T. B. Jablin, P. Prabhu, J. A. Jablin, N. P. Johnson, S. R. Beard, and D. I. August. Automatic CPU-GPU communication management and optimization. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011.

[42] T. Karcher and V. Pankratius. Run-time automatic performance tuning for multicore applications. In *Proceedings of the International Euro-Par Conference on Parallel Processing (Euro-Par)*, pages 3–14, 2011.

[43] A. Kejariwal, A. Nicolau, A. V. Veidenbaum, U. Banerjee, and C. D. Polychronopoulos. Efficient scheduling of nested parallel loops on multi-core systems. In *Proceedings of the 2009 International Conference on Parallel Processing (ICPP)*, pages 74–83, 2009.

[44] H. Kim, A. Raman, F. Liu, J. W. Lee, and D. I. August. Scalable speculative parallelization on commodity clusters. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2010.

[45] W. Ko, M. N. Yankelevsky, D. S. Nikolopoulos, and C. D. Polychronopoulos. Effective cross-platform, multilevel parallelism via dynamic adaptive execution. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS)*, page 130, 2002.

[46] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 211–222, 2007.

[47] A. Kumar, L. Shang, L.-S. Peh, and N. K. Jha. HybDTM: A coordinated hardware-software approach for dynamic thermal management. In *Proceedings of the 43rd ACM/IEEE Design Automation Conference (DAC)*, pages 548–553, 2006.

[48] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In *ISCA '04: Proceedings of the 31st Annual International Symposium on Computer Architecture*, ISCA '04, pages 64–, Washington, DC, USA, 2004. IEEE Computer Society.

[49] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the Annual International Symposium on Code Generation and Optimization (CGO)*, pages 75–86, 2004.

[50] C. E. Leiserson. The Cilk++ concurrency platform. In *Proceedings of the 46th ACM/IEEE Design Automation Conference (DAC)*, pages 522–527, 2009.

[51] LLVM Test Suite Guide. http://llvm.org/docs/TestingGuide.html.

[52] C.-K. Luk, S. Hong, and H. Kim. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proceedings of the 42nd Annual*

IEEE/ACM International Symposium on Microarchitecture (MICRO), pages 45–55, 2009.

[53] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Ferret: A toolkit for content-based similarity search of feature-rich data. *ACM SIGOPS Operating Systems Review*, 40(4):317–330, 2006.

[54] J. Mars, N. Vachharajani, M. L. Soffa, and R. Hundt. Contention aware execution: Online contention detection and response. In *Proceedings of the Annual International Symposium on Code Generation and Optimization (CGO)*, Toronto, Canada, 2010.

[55] M. Mehrara, J. Hao, P.-C. Hsu, and S. Mahlke. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[56] D. Meisner, B. T. Gold, and T. F. Wenisch. PowerNap: eliminating server idle power. In *Proceedings of the Fourteenth International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 205–216, 2009.

[57] G. Memik, W. H. Mangione-Smith, and W. Hu. NetBench: A benchmarking suite for network processors. In *Proceedings of the 2001 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2001.

[58] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, 2008.

[59] A. Moreno, E. Csar, A. Guevara, J. Sorribes, T. Margalef, and E. Luque. Dynamic pipeline mapping (DPM). In *Proceedings of the International Euro-Par*

Conference on Parallel Processing (Euro-Par), volume 5168, pages 295–304. 2008.

[60] R. Narayanan, B. Ozisikyilmaz, J. Zambreno, G. Memik, and A. Choudhary. Minebench: A benchmark suite for data mining workloads. 2006.

[61] A. Navarro, R. Asenjo, S. Tabik, and C. Cascaval. Analytical modeling of pipeline parallelism. In *Proceedings of the 18th International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 281–290, 2009.

[62] I. Neamtiu. Elastic executions from inelastic programs. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, 2011.

[63] J. Nielsen. *Usability Engineering*. Academic Press, 1993.

[64] G. Ottoni. *Global Instruction Scheduling for Multi-Threaded Architectures*. PhD thesis, Department of Computer Science, Princeton University, Princeton, New Jersey, United States, 2008.

[65] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic thread extraction with decoupled software pipelining. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 105–118, 2005.

[66] D. A. Penry. Multicore diversity: A software developer's nightmare. *ACM SIGOPS Operating Systems Review*, 43:100–101, 2009.

[67] S. Peter, A. Schüpbach, P. Barham, A. Baumann, R. Isaacs, T. Harris, and T. Roscoe. Design principles for end-to-end multicore schedulers. In *Proceedings*

of the 2nd USENIX Workshop on Hot Topics in Parallelism (HotPar), pages 10–10, 2010.

[68] C. D. Polychronopoulos. The hierarchical task graph and its use in auto-scheduling. In *Proceedings of the 5th International Conference on Supercomputing (ICS)*, pages 252–263, 1991.

[69] M. K. Prabhu and K. Olukotun. Using thread-level speculation to simplify manual parallelization. In *Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 1–12, 2003.

[70] M. K. Prabhu and K. Olukotun. Exposing speculative thread parallelism in SPEC2000. In *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 142–152, 2005.

[71] P. Prabhu, S. Ghosh, Y. Zhang, N. P. Johnson, and D. I. August. Commutative set: A language extension for implicit parallel programming. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011.

[72] A. Raman, H. Kim, T. R. Mason, T. B. Jablin, and D. I. August. Speculative parallelization using software multi-threaded transactions. In *Proceedings of the Fifteenth International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

[73] A. Raman, H. Kim, T. Oh, J. W. Lee, and D. 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)*, 2011.

[74] A. Raman, G. Yorsh, M. Vechev, and E. Yahav. Sprint: Speculative prefetching of remote data. In *Proceedings of the 26th Annual ACM SIGPLAN Conference*

on Object-Oriented Programming Systems Languages and Applications (OOPSLA), pages 259–274, 2011.

[75] E. Raman, G. Ottoni, A. Raman, M. Bridges, and D. I. August. Parallel-stage decoupled software pipelining. In *Proceedings of the Annual International Symposium on Code Generation and Optimization (CGO)*, 2008.

[76] L. Rauchwerger, N. M. Amato, and D. A. Padua. A scalable method for run-time loop parallelization. *International Journal of Parallel Programming (IJPP)*, 26:537–576, 1995.

[77] J. Reinders. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., 2007.

[78] J. Renau, K. Strauss, L. Ceze, W. Liu, S. R. Sarangi, J. Tuck, and J. Torrellas. Energy-efficient thread-level speculation. *IEEE Micro*, 26:80–91, 2006.

[79] A. Robison, M. Voss, and A. Kukanov. Optimization via reflection on work stealing in TBB. In *Proceedings of the 22nd International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–8, 2008.

[80] J. Saltz, R. Mirchandaney, and R. Crowley. Run-time parallelization and scheduling of loops. *IEEE Transactions on Computers*, 40, 1991.

[81] P. Selinger. potrace: Transforming bitmaps into vector graphics. http://potrace.sourceforge.net.

[82] G. Semeraro, G. Magklis, R. Balasubramonian, D. H. Albonesi, S. Dwarkadas, H. Dwarkadas, and M. L. Scott. Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 29–40, 2002.

[83] J. C. Spall. *Introduction to Stochastic Search and Optimization.* Wiley-Interscience, 2003.

[84] Standard Performance Evaluation Corporation (SPEC). http://www.spec.org.

[85] M. A. Suleman, M. K. Qureshi, Khubaib, and Y. N. Patt. Feedback-directed pipeline parallelism. In *Proceedings of the 19th International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 147–156, 2010.

[86] M. A. Suleman, M. K. Qureshi, and Y. N. Patt. Feedback-driven threading: Power-efficient and high-performance execution of multi-threaded workloads on CMPs. In *Proceedings of the Thirteenth International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 277–286, 2008.

[87] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), 2005.

[88] Sybase adaptive server. http://sybooks.sybase.com/nav/base.do.

[89] J. Tellez and B. Dageville. *Method for Computing the Degree of Parallelism in a Multi-user Environment.* United States Patent No. 6,820,262. Oracle International Corporation, 2004.

[90] The IEEE and the Open Group. *The Open Group Base Specifications Issue 6 IEEE Std 1003.1, 2004 Edition.* 2004.

[91] W. Thies, V. Chandrasekhar, and S. Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in C programs. In *Proceedings*

of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pages 356–369, 2007.

[92] C. Tian, M. Feng, V. Nagarajan, and R. Gupta. Copy or discard execution model for speculative parallelization on multicores. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 330–341, 2008.

[93] A. Tzannes, G. C. Caragea, R. Barua, and U. Vishkin. Lazy binary-splitting: A run-time adaptive work-stealing scheduler. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 179–190, 2010.

[94] G. Upadhyaya, V. S. Pai, and S. P. Midkiff. Expressing and exploiting concurrency in networked applications with Aspen. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 13–23, 2007.

[95] H. Vandierendonck, S. Rul, and K. De Bosschere. The Paralax infrastructure: Automatic parallelization with a helping hand. In *Proceedings of the 19th International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 389–400, 2010.

[96] M. J. Voss and R. Eigenmann. ADAPT: Automated de-coupled adaptive program transformation. In *Proceedings of the 1999 International Conference on Parallel Processing (ICPP)*, pages 163–170, 1999.

[97] Z. Wang and M. F. O'Boyle. Mapping parallelism to multi-cores: A machine learning based approach. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 75–84, 2009.

[98] P. M. Wells, K. Chakraborty, and G. S. Sohi. Dynamic heterogeneity and the need for multicore virtualization. *ACM SIGOPS Operating Systems Review*, 43:5–14, 2009.

[99] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. *ACM SIGOPS Operating Systems Review*, 35(5):230–243, 2001.

[100] D. Wentzlaff and A. Agarwal. Factored operating systems (fos): The case for a scalable operating system for multicores. *SIGOPS Operating Systems Review*, 43:76–85, 2009.

[101] T. Wiegand, G. J. Sullivan, G. Bjontegaard, and A. Luthra. Overview of the H.264/AVC Video Coding Standard. 13(7):560–576, 2003.

[102] M. Wolfe. DOANY: Not just another parallel loop. In *Proceedings of the 4th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 1992.

[103] E. Yardimci and M. Franz. Dynamic parallelization and mapping of binary executables on hierarchical platforms. In *Proceedings of the 3rd ACM International Conference on Computing Frontiers (CF)*, pages 127–138, 2006.

[104] H. Zhong, M. Mehrara, S. Lieberman, and S. Mahlke. Uncovering hidden loop level parallelism in sequential applications. In *Proceedings of the 14th International Symposium on High-Performance Computer Architecture (HPCA)*, 2008.