Adaptive, Efficient, Parallel Execution of Parallel Programs

Srinath Sridharan Gagan Gupta Gurindar S. Sohi
University of Wisconsin-Madison
sridhara@cs.wisc.edu gagang@cs.wisc.edu sohi@cs.wisc.edu

Abstract

Future multicore processors will be heterogeneous, be increasingly less reliable, and operate in dynamically changing operating conditions. Such environments will result in a constantly varying pool of hardware resources which can greatly complicate the task of efficiently exposing a program's parallelism onto these resources. Coupled with this uncertainty is the diverse set of efficiency metrics that users may desire. This paper proposes Varuna, a system that *dynamically*, *continuously*, *rapidly* and *transparently* adapts a program's parallelism to best match the instantaneous capabilities of the hardware resources while satisfying different efficiency metrics. Varuna is applicable to both multithreaded and task-based programs and can be seamlessly inserted between the program and the operating system without needing to change the source code of either.

We demonstrate Varuna's effectiveness in diverse execution environments using *unaltered* C/C++ parallel programs from various benchmark suites. Regardless of the execution environment, Varuna always outperformed the state-of-the-art approaches for the efficiency metrics considered.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming - Parallel programming; D.3.4 [*Programming Languages*]: Processors - Run-time environments

General Terms Design, Experimentation, Measurement, Performance

Keywords Autotuning, parallel programming, performance portability, performance tuning, run-time optimization

1. Introduction

Although multiprocessors are now pervasive, efficiently executing parallel programs on them continues to be a challenging problem due to the complex and dynamic interplay between the program and the host system. Efficient execution requires optimum use of resources, as defined by a desired efficiency metric, to perform the program's work. Leaving resources underutilized is inefficient. So is contention-causing overutilization. Matching a program's work to microarchitectural **dynamic resource capabilities** (the resource's capacity to serve a request, given its current load) is non-trivial since neither the program's demands nor the resource capabilities remain constant. A parallel program's demands may change across phases or when its computations vie for resources. Resource capabilities, often specific to a system, may vary due to a variety of reasons. Thus, efficient execution requires dynamically and continuously matching the program's exposed parallelism to the instantaneous resource capabilities.

As the computing landscape evolves, at a phenomenal pace, growing system diversity is likely to pose further challenges to efficient execution. Microarchitectural diversity is growing since computing devices across the spectrum, from the low end (mobile devices) to the high end (servers), employ rapidly evolving role-specific microarchitectures. Within a system, dynamic diversity will arise from a variety of sources, including hardware defects, process variability, dynamic voltage and frequency scaling, dynamic techniques to handle power, etc. The growing popularity of multiprogrammed systems, e.g., mobile devices and cloud services, will increase the diversity of co-located programs. Further, the dynamic diversity is likely to change rapidly during a program's execution. Accounting for the static diversity for portability, and the dynamic diversity for efficiency will be difficult for programmers, and perhaps even for the OS. A diverse range of efficiency metrics, e.g., time, resource consumption and power budgets, will further compound the problem. As a greater number of programmers develop applications for such systems, an automated approach that treats the system as a black box is needed.

Existing Work. Current automated approaches to optimize a program's parallel execution fail to take a comprehensive view of the prevailing programming methods and the system diversity as summarized in Table 1. Some proposals require programs be rewritten from scratch using their own APIs that are not widely adopted [28–30]. Others [10, 11, 33, 34] that work with existing APIs are applicable only to task-based programming models [7, 15, 32]. Most of the approaches [10, 11, 34] can handle only data parallel programs and the ones that propose to tackle arbitrary programs require compiler or programmer support [29, 30, 33]. Importantly, none of these techniques are applicable to arbitrary multithreaded programs. Some proposals prevent only underutilization [7, 15, 32]. Others can also prevent overutilization, but of only some resources, may require compiler support or offline profiling, and may be ineffective in multiprogrammed environments [10, 11, 23, 34]. Further, these approaches use hill-climbing search heuristics to find the right operating point, and hence may fail to react swiftly to changing conditions. Moreover, they optimize only for performance and do not take into account other efficiency metrics, such as resource consumption. We believe that a system that is applicable to both arbitrary multithreaded and task-based programs without altering existing design flows, takes a holistic view of the system, can react swiftly to the changes in dynamic operating conditions, and can optimize for diverse efficiency metrics will find a broader utility and yield higher efficiency.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PLDI'14, June 9-11, 2014, Edinburgh, United Kingdom.
Copyright © 2014 ACM 978-1-4503-2784-8/14/06. . . $15.00.
http://dx.doi.org/10.1145/2594291.2594292

Proposal	Continuous Adaptation	Multi-programming Support	Compiler/ Programming-Model Support	Speed	Arbitrary Programs	Resource-Agnostic	Model-based Search	Diverse Metrics	Code Changes
Parcae[30],DoPE[29]	yes	yes	yes/yes	slow	no	yes	no	no/yes*	yes
PD[33]	yes	yes	no/yes	slow	no	yes	no	no	yes
Curtis-Maury[10, 11]	no	no	yes/yes	slow	no	no	no	no	yes
TT[23]	no	no	yes/yes	slow	no	no	no	no	yes
FDT[34]	no	no	no/yes	rapid	no	no	yes	no	yes
Pthreads[1],TBB[32], Cilk[15],OMP[7]	no	no	no/yes	none	no	no	none	no	yes
Varuna [This paper]	**yes**	**yes**	**no/no**	**rapid**	**yes**	**yes**	**yes**	**yes**	**no**

Table 1. Related work as applied to parallelism adaptation.*DoPE supports different metrics whereas Parcae does not.

Proposed Solution. In this paper, we propose **Varuna**, a runtime system that dynamically, continuously and transparently adapts a program's parallelism to best match the dynamic hardware resource capabilities and the program's characteristics, while optimizing diverse efficiency metrics. Varuna employs a novel, *holistic* and *resource-agnostic scalability model* based on Amdahl's law to estimate changes in efficiency during a program's execution (§ 3). It then uses formulae, derived from the model, to rapidly determine the optimum **degree of parallelism (DoP)**, i.e., the optimum number of hardware threads, to employ for different efficiency metrics and automatically guides the execution to the computed DoP.

Varuna is *compiler* and *programming model independent*. It retains the existing programming abstractions and can be applied to both task-based and multithreaded parallel programs. Further, it requires no changes to the program or the OS, and can *tackle arbitrary parallel programs that use standard APIs*. We demonstrate Varuna for the more widely used parallel programming APIs, Pthreads [1], and Intel Thread Building Blocks (TBB) [32].

To facilitate program/OS-agnostic adaptation, Varuna employs a novel primitive called a *virtual task (vtask)*. Vtasks decouple program-level parallelism, expressed as software threads in multithreaded programs and tasks in task-based programs, from hardware threads. They are progress-aware entities and give Varuna the flexibility needed to transparently regulate a program's parallel execution, without hampering its forward progress (§ 4).

We evaluated Varuna in two different execution environments, isolated and multiprogrammed, using unaltered C/C++ Pthreads and TBB programs from various standard benchmark suites. Two different efficiency metrics, (i) execution time, and (ii) resource consumption, were considered. Two different real hardware platforms with different microarchitectural resource capabilities were used. Experimental results show that Varuna reduced the execution time on an average by 15% in the isolated environment and 33% in the multiprogrammed environment for the execution time metric. The concomitant energy savings were 31% and 32%, respectively. For the resource consumption metric, Varuna saved the consumption cost by 84% and 90% in isolated and multiprogrammed environments, respectively, while reducing the execution time by -1% and 14%, respectively.

Paper Organization. The rest of the paper is organized as follows. § 2 presents an overview of Varuna. § 3 describes Varuna's analytical model. § 4 discusses Varuna's adaptive parallel execution. Then, in § 5, we present our detailed evaluation and results for the two different environments. § 6 reviews related work before § 7 concludes.

2. Varuna: Overview

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

First, they can dynamically increase the latency of a computation, causing **artificial sequentialization**. Artificial sequentialization can lead to slowdowns, sometimes worse than sequential execution, and can arise from a plethora of sources, including contention to software resources (e.g., locks) and shared hardware resources (e.g., last level cache, memory and disk bandwidth, SMT core, etc.), memory effects (e.g., false sharing and processor affinity), cache coherence delays, cache interference due to multiprogramming, TLB misses and page faults, loss of resources due to power and reliability management techniques, load balancing and scheduling overheads, among others.

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

In practice, these two effects often occur in unison and the optimum point at which a program must operate depends on which of the two dominates and by how much. Statically determining this point is hard because these effects can occur at different times and due to different reasons. For example, they can take hold for the entire program, or only during parts of it when the program changes phases, or when co-located applications occupy or release resources, or when the resources go offline or come online. Further, the combination of these effects are different for different microarchitectures. Within a microarchitecture, the combination can vary dynamically and differently across runs, impacting the program's

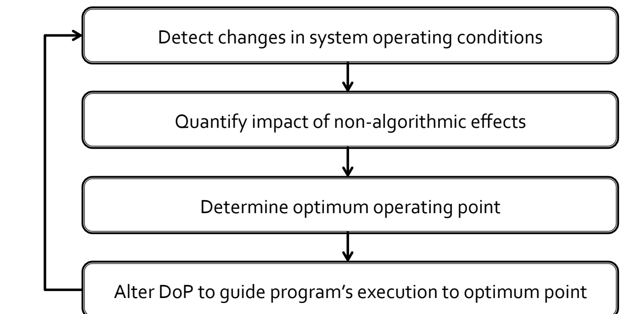


Figure 1. Varuna's strategy.

efficiency unintuitively, especially in multiprogrammed environments. Therefore, to achieve efficient parallel execution, a dynamic and responsive parallelism optimization strategy, which takes advantage of artificial acceleration and mitigates artificial sequentialization, is needed.

Implementing such a strategy requires: (i) detecting changes in the system's operating conditions, (ii) quantifying the aggregate impact of the non-algorithmic effects, (iii) determining the optimum point of operation, and (iv) guiding the program's execution to that point by altering its degree of parallelism (DoP), as depicted in Figure 1. For overall efficiency, the above process needs to be *repeated periodically*, as a program executes, to assess and react to changes.

Expecting common programmers to deploy such a complex strategy can severely hamper their productivity. Deploying this in the OS will necessitate changes in both the OS and the program, and a tighter integration of the two, something that is going to be challenging in future computing environments, such as cloud computing, with potentially several complex software layers between them.

Varuna is a runtime system that implements the above strategy with no modifications to the program, the OS or any other entity. It comprises two components, shown in Figure 2: an **Analytical Engine**, and a **Parallelism Manager**. The two components and their operations are summarized next.

Analytical Engine. The Analytical Engine (AE) continuously monitors changes in the operating conditions using hardware performance monitoring units, models the program's dynamic execution behavior to estimate the non-algorithmic effects, and determines the optimum DoP. The high-level operations of the AE are as follows:

1. Establish the relationship between the program's instantaneous DoP, instantaneous performance and the non-algorithmic effects.
2. Using this information, determine the optimum DoP, P_{opt}, for a given efficiency metric.
3. Passively monitor the program performance for changes, as the parallelism manager employs P_{opt} parallelism for the program.
4. Go to step 1 if the operating conditions change.

To establish the relationship between the program's instantaneous DoP, instantaneous performance and the non-algorithmic effects, the AE employs a novel, holistic and resource agnostic scalability model. It uses the model to dynamically determine the optimum DoP for two different efficiency metrics, execution time and resource consumption. These details are described in §3.

Parallelism Manager. The Parallelism Manager (PM) automatically regulates the execution of program's parallel computations to match the DoP determined by the AE. To achieve this requires the following four capabilities:

- Ability to decouple program-level parallelism (software threads or tasks) from hardware threads so that the DoP can be changed dynamically without altering the program or the OS,
- Ability to transparently pause long running computations when decreasing the DoP, and resume and/or migrate them when increasing the DoP,
- A mechanism to balance the remaining workload in the program while altering the DoP, and
- A mechanism to ensure that the program's forward progress is not affected while regulating the DoP (critical for programs using synchronization objects, such as locks and conditional variables, to manipulate shared data).

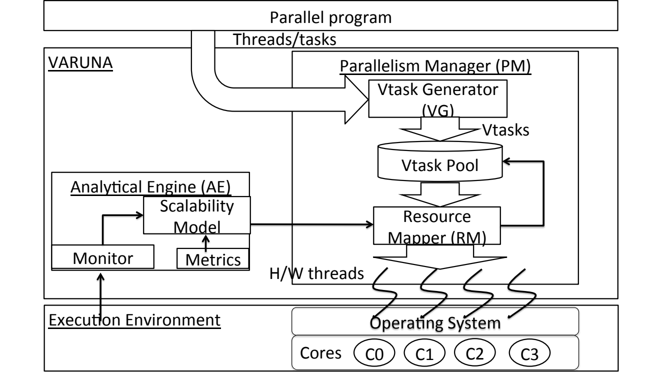


Figure 2. Varuna's system architecture.

To realize the above capabilities, the PM deploys a primitive called a virtual task (vtask). Vtasks abstract hardware threads into logical cooperative tasks [2, 4] to which a program's computations, that comprise the program's software threads/tasks are mapped. The vtasks, in turn, are dynamically scheduled onto hardware threads. Each vtask maintains the state of the current computation mapped on to it using *contexts* (described in § 4), allowing the PM to transparently pause/resume/migrate a computation by saving/restoring its corresponding vtask's context. It also includes the state necessary for the PM to ensure the computations' forward progress even as their execution is regulated. More details on vtasks are given in §4.

Figure 2 summarizes the PM's operations. As a parallel program begins to execute and starts making its `thread_create` or `task_spawn` requests, a Vtask Generator (VG) transparently intercepts these requests, nullifies them and creates vtasks instead. The VG then reassigns the thread's/task's parameters (pointer to actual computation and its arguments) onto vtasks and enqueues them into a vtask pool. A Resource Mapper (RM) then assigns the vtasks from the vtask pool, to the dynamically varying pool of hardware threads, as determined by the Analytical Engine. The RM is also responsible for dynamically controlling the execution of vtasks (suspending, resuming and migrating) as well as ensuring a program's forward progress. These details are described in §4.

3. Analytical Engine

The Analytical Engine (AE) models the non-algorithmic effects, using Amdahl's law, to understand the dynamic relationship between artificial sequentialization, artificial acceleration, and the achieved performance. We present the model, and derive from it the optimum DoP formulae for two efficiency metrics: **MIN(time)**, which minimizes the execution time, and **MIN(consumption)**, which minimizes the CPU consumption-execution time product. These are two popular pricing models employed in cloud-based services. The former, *time-based* pricing, used in Amazon's EC2 and Microsoft's Azure, gives a program a fixed number of cores and charges for how long they are used; thus minimizing execution time is important. In the latter, *consumption-based pricing*, used by VMware, the pricing depends on the average number of cores and the duration of their use.

3.1 Modeling Non-algorithmic Effects

According to Amdahl's law, the speedup $S(P)$ of a program, whose serial execution time is $T(1)$, comprising a parallel region t_p, and a serial region t_s, when employing a DoP of P, is:

$$S(P) = \frac{T(1)}{\frac{t_p}{P} + t_s};\ where\ T(1) = t_p + t_s \tag{1}$$

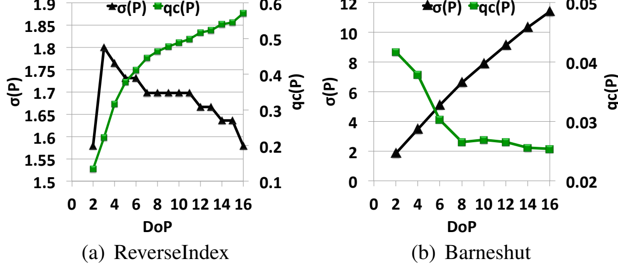


Figure 3. Speedup ($\sigma(P)$) and $qc(P)$ trends of ReverseIndex and Barneshut on Opteron.

Amdahl's law ignores the non-algorithmic effects and assumes that the parallel region is perfectly parallelizable, i.e., speedup in the parallel region is P. In reality, the parallel region can incur artificial sequentialization and benefit from artificial acceleration. Further, both effects vary wrt P. To account for these effects, let $t_q(P)$ be the additional time incurred due to artificial sequentialization and $t_c(P)$ be the time saved due to artificial acceleration, when employing a DoP of P. Then the new speedup, $S'(P)$, is:

$$S'(P) = \frac{t_p + t_s + t_q(1) - t_c(1)}{\frac{t_p}{P} + t_s + t_q(P) - t_c(P)} \quad (2)$$

Since we are concerned with the net impact of the non-algorithmic effects, we combine the two quantities, $t_q(P)$ and $t_c(P)$ into $t_{qc}(P) = t_q(P) - t_c(P)$. Equation 2 captures the impact of non-algorithmic effects on a program's overall performance, which includes both parallel and serial regions. However, we are only concerned with the impact of these effects when the program is executing in parallel, i.e., $P > 1$, since they arise only in the parallel region. Therefore, $t_{qc}(1)$ is zero. Further, the term t_s can be eliminated, since it is applicable only to the serial region (when $P = 1$). Accordingly, for the speedup, $\sigma(P)$, obtained in the parallel region, Equation 2 becomes:

$$\sigma(P) = \frac{t_p}{\frac{t_p}{P} + t_{qc}(P)} = \frac{1}{\frac{1}{P} + \frac{t_{qc}(P)}{t_p}},\ where\ P > 1 \quad (3)$$

Let $qc(P) = \frac{t_{qc}(P)}{t_p}$. From Equation 3 it follows that:

$$qc(P) = \frac{1}{\sigma(P)} - \frac{1}{P} \quad (4)$$

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

Figures 3(a) and 3(b) illustrate these aspects. They plot the measured speedup, σ(P) (primary vertical axis), and the computed $qc(P)$ (secondary vertical axis) for two of our programs, ReverseIndex and Barneshut, respectively, with varying DoP (P), on one of our experimental platforms, Opteron (details are provided in §5). ReverseIndex processes files and places significant demands on the disk bandwidth. Even modest attempt at parallel execution results in disk contention, indicated by the higher and increasing values of $qc(P)$. Between $P = 2$ and $P = 3$, the program scales even if $qc(P)$ increases, because $qc(P)$ is less than $\frac{1}{P}$. But when P exceeds 3, $qc(P)$ exceeds $\frac{1}{P}$, resulting in slowdown.

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

By computing $qc(P)$ from measured $\sigma(P)$ (speedup of the parallel region) and using these observations, we can determine the optimum DoP for the above-mentioned efficiency metrics, as described next.

3.2 Optimizing for MIN(time)

We can obtain the optimum DoP, P_{opt_t} that minimizes the execution time (the inverse of $\sigma(P)$) of the parallel region by simply differentiating Equation 3 wrt to P and equating it to zero as follows:[1]

$$\frac{d\frac{1}{\sigma(P)}}{dP} = -\frac{1}{P^2} + \frac{dqc(P)}{dP} = 0\ ; P_{opt_t} = \sqrt{\frac{1}{\frac{dqc(P)}{dP}}} \quad (5)$$

where $\frac{dqc(P)}{dP}$ is the rate of change of $qc(P)$ or the gradient of the $qc(P)$ curve at a given P. Note that Equation 5 is applicable only when $\frac{dqc(P)}{dP}$ is positive, as in the case of ReverseIndex (Figure 3(a)). A negative or zero $\frac{dqc(P)}{dP}$, however, as in the case of Barneshut (Figure 3(b)), indicates that the program is benefiting from more parallelism. Hence, as many resources as possible, P_{max}, may be allocated to the program. Amending Equation 5 with boundary conditions we get:

$$P_{opt_t} = \begin{cases} \sqrt{\frac{1}{\frac{dqc(P)}{dP}}} & if\ \frac{dqc(P)}{dP} > 0 \\ P_{max} & if\ \frac{dqc(P)}{dP} \leq 0 \end{cases} \quad (6)$$

To apply Equation 6 as an online metric, the AE needs to compute $qc(P)$, $\frac{dqc(P)}{dP}$, and P_{max} dynamically.

To determine $qc(P)$, AE needs to compute $\sigma(P)$ empirically (Equation 4), and this requires computing a baseline performance for the parallel region. To do this, whenever the program (re-)enters a parallel region (indicated by either `thread(task)_create` or `thread(task)_barrier` calls), the AE sets $P = 1$ for a predefined time period ($100ms$ in all our experiments[2]), monitors its execution and establishes a baseline performance ($Perf(1)$). The exact quantity to represent performance depends on the type of the program. For example, for mobile class programs, Instructions per Second (IPS) is a good measure. For server class programs, Requests Per Second (RPS) is a suitable measure. In this paper, we use IPS to represent performance. We make a fair assumption that spin-locks in program code are rare and that users use standard synchronization interfaces to access their critical sections. Spin-

[1] For the purpose of this derivation, we assume that P is a continuous variable, although, in reality, it is discrete.

[2] We chose this interval to also capture OS context switching overheads.

locks can occur in the OS and we avoid this issue by not counting the OS instructions.

Once the baseline performance is measured, AE switches the DoP to P, allows the program to run for $100ms$ and measures its performance ($Perf(P)$). $\sigma(P)$ can then be obtained by dividing $Perf(P)$ by $Perf(1)$. The AE substitutes this value in Equation 4 to obtain $qc(P)$.

To obtain estimates of $\frac{dqc(P)}{dP}$, the AE uses linear regression, based on the ordinary least squares estimation, on a subset of $qc(P)$ values. The alternative, sweeping through all the values of P, to obtain the corresponding $qc(P)$, and from these to compute $\frac{dqc(P)}{dP}$, may not be an effective solution, especially when the operating conditions change frequently. Although linear regression may lead to errors, our experiments (§5) and residual analysis [24] show that it is adequate and leads to better results than the state-of-the-art adaptive methods, which resort to time-consuming iterative search strategies.

Based on our experiments for $1 < P \leq 24$ (our experimental platforms have a maximum of 24 hardware contexts), linear regression using data for three parallelism configurations (in addition to $P = 1$ for which $qc(1) = 0$) gave sufficiently accurate estimates of $\frac{dqc(P)}{dP}$[3]. Hence, we restrict our measurement to three points (P_1, P_2 and P_3) in order to make quick decisions. In §5, we demonstrate that this approach is sufficient to make informed decisions to arrive at the optimum configuration.

The AE computes performance at three different DoPs, $P_1 = 2$, $P_2 = \frac{N}{2}$ and $P_3 = N$, where N is the maximum number of processing resources in the system. $\sigma(P_1/P_2/P_3)$ can then be obtained by dividing the corresponding performance measures by the baseline performance. The AE verifies that these values are indeed greater than one. Otherwise, it will switch to sequential execution. These values are substituted in Equation 4 to get $qc(P_1/P_2/P_3)$, which are then used to obtain $\frac{dqc(P)}{dP}$ by applying the least square method. The AE then computes P_{opt_t} by substituting $\frac{dqc(P)}{dP}$ in Equation 6. If $\frac{dqc(P)}{dP}$ is negative, P_{max} is simply set to N.

3.3 Optimizing for MIN(consumption)

To find the optimum parallelism, P_{opt_c}, that minimizes the resource consumption cost, we want to minimize the product $P \times \frac{1}{\sigma(P)}$. Similar to the first metric, P_{opt_c} can be obtained by simply differentiating $P \times \frac{1}{\sigma(P)}$ wrt P, and then equating it to zero. In the interest of space, we present the final equation below:

$$P_{opt_c} = \begin{cases} -\frac{qc(P)}{\frac{dqc(P)}{dP}} & if\ \frac{qc(P)}{\frac{dqc(P)}{dP}} < 0 \\ P_{min} & if\ \frac{dqc(P)}{dP} > 0\ \&\ qc(P) > 0 \\ P_{max} & if\ \frac{dqc(P)}{dP} \leq 0\ \&\ qc(P) \leq 0 \end{cases} \quad (7)$$

A negative $\frac{qc(P)}{\frac{dqc(P)}{dP}}$ indicates net artificial acceleration and hence efficient resource consumption (Figure 3(b)). In this case P_{opt_c} is computed using the formula $-\frac{qc(P)}{\frac{dqc(P)}{dP}}$. $\frac{qc(P)}{\frac{dqc(P)}{dP}}$ will yield multiple values depending on the value of P. The AE picks the one with minimum value of $\frac{qc(P)}{\frac{dqc(P)}{dP}}$ since it signifies the least contention and hence most efficient consumption of resources. If both $\frac{dqc(P)}{dP}$ and $qc(P)$ are positive, the program is not scaling due to artificial sequentialization, and hence the resources are not being consumed efficiently (Figure 3(a)). In this case, minimum resources, P_{min}, are allocated to the program. If both $\frac{dqc(P)}{dP}$ and $qc(P)$ are zero or negative, the program is scaling linearly or superlinearly and hence, as many resources as possible, P_{max}, may be allocated to it.

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

$\frac{dqc(P)}{dP}$ and $qc(P)$ are computed dynamically using the same methodology described in § 3.2. If $\frac{dqc(P)}{dP} > 0$ & $qc(P) > 0$, P_{min} is simply set to 1. If $\frac{dqc(P)}{dP} \leq 0$ & $qc(P) \leq 0$, P_{max} is set to N.

3.4 Monitoring, Recalibrating and Periodic Diversification

Once P_{opt} is determined for the desired efficiency metric, the AE conveys the DoP value to the PM. It then enters into a *passive monitoring mode* where it periodically monitors the performance with P_{opt} parallelism until its value changes by more than a predefined threshold (10% in our experiments). At this point, the AE switches the DoP to 1 and repeats the process described in § 3.2 to recalibrate P_{opt}. To ensure that the parallel execution is not trapped in a local optimum, the AE, while in passive monitoring mode, periodically diversifies (at 3s granularity in our experiments) by switching the DoP to 1 and repeating the search for a new P_{opt} with different, randomly chosen, P_1 and P_2 values. Our experiments showed this scheme to be adequate.

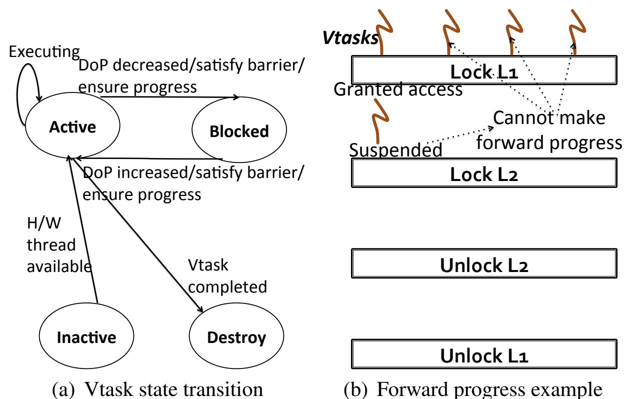
4. Parallelism Manager

The Parallelism Manager (PM) receives the P_{opt} value from the AE and uses it to control the number of inflight computations. To be able to continuously and transparently alter the number of concurrent computations, the PM maps units of computation designated for parallel execution, a task in task-based parallel programs, or a thread in multithreaded programs, to *vtasks*. Vtasks are closest in spirit to *fibers* [2], an implementation of cooperative tasks in the Windows OS, but with three key differences. First, vtasks preserve the same programming semantics as that of threads/tasks and require no additional programming effort to create and manage their scheduling and context switches, providing the flexibility to transparently control their execution, whereas fibers must be explicitly created and managed by the program. Second, the primary use of vtasks is to control program-level parallelism to match the varying number of hardware threads, whereas, the primary use of fibers is to avoid excessive OS thread context switching. Finally, vtasks are *progress-aware* entities. Their contexts include additional state needed to ensure forward progress of programs while controlling their execution. Using fibers as is for Varuna's objectives can hamper a program's forward progress (§ 4.3).

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

4.1 Vtask Context

To enable suspension and resumption of a vtask, similar to an OS thread, each vtask contains a *vtask context block (VCB)*. The VCB contains the following state: (1) a *call stack* to maintain the *activation records* of functions invoked from the vtask, (2) a *Program Counter (PC)* that specifies the address of the next instruction in the vtask control flow to be executed, (3) *user-level registers* that contain data generated by the vtask computation, (4) a *Stack Pointer (SP)* that points to the next entry in the vtask's *call stack*, and (5) a *user mode mutex counter* that contains the number of mutex variables currently acquired by the vtask computation when executing in user mode.



(a) Vtask state transition (b) Forward progress example

Figure 4. Vtask operations.

The first four quantities are the standard elements necessary to save an execution's state. The *user mode mutex counter* is needed to ensure forward progress of programs when varying the number of inflight vtasks (§4.3). To implement the call stack, we employ an approach inspired from Goldstein, lazy threads [17], originally proposed to reduce the overheads of nested parallel function calls. The strategy supplies each vtask with a *stacklet* — a linear stack that stores the activation records of functions invoked by computations mapped on to the vtask. To suspend a vtask, the PM saves all user-level registers and the PC on the vtask's stacklet, so that the stacklet is a self-contained record of the context state. To resume the vtask, the PM restores registers from the stacklet.

4.2 Managing Vtasks

Figure 4(a) depicts the state transition diagram the PM uses to manage vtasks, from their creation to destruction. When a vtask is created by the Vtask Generator (VM) (Figure 2), it is in an *inactive* state. A vtask moves to an *active* state when the Resource Mapper (RM) assigns it for execution, either when the DoP is increased or when a current hardware thread becomes free. To decrease the DoP, the RM pre-empts excess vtasks when they reach a *safe point* (discussed in § 4.3), transitions them to a *blocked* state, and moves them back to the vtask pool. Safe point pre-emption is necessary to ensure a vtask's forward progress (§ 4.3). When decreasing DoP, the RM moves vtasks to a *blocked* state in the order in which they arrive at the safe point. A vtask is also moved to a *blocked* state when it arrives at a barrier and the barrier condition is not satisfied. When assigning vtasks for execution, the RM prioritizes vtasks that have waited the longest, to ensure fairness. Finally, if a vtask finishes its assigned quota of work, it transitions to a *destroy* state before its state is destroyed.

To efficiently schedule vtasks, the RM employs a Cilk-style work stealing scheduler [15]. At the start of the program execution, the RM creates a pool of hardware threads, one per hardware context allocated to it by the OS. A double-ended work queue (deque) is then assigned to each thread in the system. A thread schedules vtasks for execution by queuing them in its work deque. Each thread seeks vtasks from its own deque, failing which it steals from someone else's deque. The RM uses lazy task creation [26] to avoid memory explosion when creating vtasks. It also uses a randomized task-stealing policy to balance the load across the executing threads.

4.3 Ensuring Forward Progress

Vtasks are cooperative tasks and are not pre-emptively scheduled, unlike threads. Tasks in task-based programs are usually independent, and can run to completion without communicating with each other. This, however, may not be the case with threads in multithreaded programs. Hence, arbitrarily pausing/resuming a vtask's execution in a multithreaded program can potentially affect the forward progress of other vtasks. A vtask's progress can be affected when it is waiting on: (1) a mutex lock held by a *blocked* vtask, or (2) a signal from a *blocked* vtask (e.g., a consumer computation waiting to receive data from a producer computation, in a producer-consumer style program).

Handling Blocked Mutexes. To avoid forward progress issues due to blocked mutexes, the RM pauses a vtask only when it reaches a *safe point* in its execution. A safe point is a control point in the vtask execution flow at which the vtask is currently executing in user mode and does not hold any user mode mutex locks.

Ensuring that a vtask is executing in user mode is necessary to avoid suspending the vtask after acquiring a kernel-level (spin)lock. Utilizing the fact that the OS will release all kernel locks before returning to user-level, the RM can monitor all switches between user-level and kernel-level. However, such monitoring is hard as it requires modifying the OS calls or requires prediction mechanisms based on monitoring privileged instructions [35]. Instead, the RM takes a much simpler approach. It suspends a vtask only if it reaches one of the following synchronization points in the control flow: before or after a mutex lock or unlock, respectively, after reaching a barrier, and before or after a conditional wait or signal, respectively. These points are guaranteed to be free of kernel-level locks. If none of the above points are reached, the RM simply waits until the vtask completes its execution.

Ensuring that a vtask is not suspended when holding any user-level mutex locks is necessary to avoid unsafe pre-emptions when in user mode. For example (Figure 4(b)), if a vtask holds a user-level lock, L1, and is suspended when it was attempting to acquire a second user-level lock, L2, other vtasks waiting for L1 cannot make any forward progress. To address this problem, each vtask maintains a count of the number of mutexes it has currently acquired in a *user mode mutex counter*. It increments the counter when acquiring a user-level lock and decrements it when releasing the same. The RM leverages this information and ensure that a vtask is never suspended until its counter becomes zero.

Many multithreaded programs have few safe points since these programs seldom synchronize. Hence they provide few opportunities to control the parallel execution of vtasks. In such cases, more safe points can be created by spawning more vtasks than the actual number of hardware contexts. (Note that task-based programs already create many more tasks than actual hardware threads, naturally achieving this.) We observe that many multithreaded programs are generally written to take the number of threads as an argument, which is used to divide the work into as many independent portions at run-time. Hence spawning more threads, similar to tasks, is as simple as altering the command line argument to the program. As we demonstrate in §5, spawning a high number of vtasks in Varuna does not have the same overheads as spawning a high number of threads, due to the following reasons: (1) vtasks are userspace objects and hence the cost of switching between them is extremely low, (2) the RM employs lazy vtask creation [26], which avoids memory explosion.

Handling Blocked Signals. Signaling is another way threads in multithreaded programs communicate with each other (task-based

	#S	#HC	SMT	Freq.	Cache	Mem.	LK
Opteron-8350	4	16	no	2.1GHz	16M	16G	3.4.4
Xeon E5-2420	2	24	yes	1.9GHz	15M	32G	2.6.32

Table 2. Machine configurations. S: Sockets, HC: Hardware Contexts, SMT: Simultaneous Multi-Threading, and LK: Linux Kernel.

Program	Characteristics	Opteron			Xeon			Xeon-Time(s)	
		T/V#	DoP (time)	DoP (con)	T/V#	DoP (time)	DoP (con)	Isolated	Multi-programmed
1	2	3	4	5	6	7	8	9	10
Barneshut[22]	Barriers	10K/10K(16)	16(16)	10(10)	10K/10K(24)	24(24)	20(20)	16.2	75.4
Canneal[8]	Atomics & barriers	96/96(16)	16(16)	10(10)	96/96(24)	24(24)	5(5)	89.1	165
Dedup[8]	Pipeline-parallel	16/16(16)	16(16)	8(8)	24/24(24)	24(24)	16(16)	14.2	55.1
Fluidanimate[8]	Locks & barriers	64/64(16)	16(16)	10(10)	64/64(24)	24(24)	16(16)	62.5	94.5
Histogram[31]	OS locks	1024/1024(16)	1(1)	1(1)	1024/1024(24)	1(1)	1(1)	15	23
Bzip2[16]	Pipeline-parallel	1566/16(16)	16(16)	11(11)	1566/24(24)	24(24)	20(20)	23.2	56.2
RE[5]	Locks	100K/100K(16)	8(8)	1(1)	100K/100K(24)	12(12)	1(1)	33.1	65.6
ReverseIndex[31]	Disk-intensive	78371/16(16)	3(3)	1(1)	78371/24(24)	16(16)	1(1)	72.3	130.1
Swaptions[8]	Data-parallel	384/384(16)	16(16)	10(10)	384/384(24)	24(24)	23(23)	86.7	135.2
WordCount[31]	Parallel-reduction	256/256(16)	16(16)	9(9)	256/256(24)	24(24)	20(20)	3.2	6.4
X264[8]	Pipeline-parallel	512/512(16)	16(16)	9(9)	512/512(24)	24(24)	19(19)	89.1	127.3
Blackscholes[8]	Data-parallel	10K/10K(16)	16(16)	9(9)	10K/10K(24)	24(24)	22(22)	36.3	87.5

Table 3. Programs used in experiments and key operational data for different multithreaded versions. *T/V#*=Thread/vtask count for PT_FG/Varuna configurations with default PT_CG count in parantheses. *DoP(time/con)*=Optimum DoP dynamically chosen for MIN(time/consumption) by Varuna with the best static DoP in parantheses.

programming models usually do not support signaling). If a computation's execution is dependent on another, there are two generic approaches to mitigate starvation among computations which are not executing concurrently: (1) avoid pre-empting a computation that is responsible for producing the signal/data to other computations, or (2) pro-actively pre-empt an executing computation in favor of executing a more productive computation. While the first approach is more efficient, it requires precise information about the producer, which is not readily available from the existing multithreaded abstractions. Hence, to avoid starvation issues due to blocked signals, the RM takes the second approach. Everytime a vtask invokes a `cond_wait` call, if there are more vtasks in the vtask pool waiting to be scheduled for execution, the RM suspends the current vtask, enqueues it at the tail of the vtask pool and schedules the oldest vtask in the vtask pool in the former's place. In this way, each communicating vtask gets a slice of the resource to execute, avoiding starvation and potential deadlock situations.

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

5. Evaluation and Results

To evaluate Varuna's efficacy we applied it to threaded and task-based programs, optimizing them for execution time and resource consumption. We tested under two execution environments, isolated and multiprogrammed, on two stock multiprocessor machines with different microarchitectures. We report the total execution time, the energy consumed and the resource consumption cost for each program, along with the harmonic mean (HM) for the entire benchmark set, when optimizing for both the metrics. In the experiments, we sought to assess the following: (1) Varuna's overheads, (2) benefits of applying vtasks to unmodified threaded and task programs, (3) further benefits of applying adaptive optimization to them, (4) effectiveness in highly dynamic operating conditions, and (5) agility in responding to changes. We present key results that highlight the major trends from our extensive experiments.

Machines, Benchmarks and Baselines. Table 2 provides the details of the two machines used in the evaluation. To demonstrate Varuna's generality, we present results of select programs from different suites that exhibit different characteristics. Table 3 shows the list of threaded programs (column 1) we used along their characteristics (column 2). We used large input sizes for each application obtained from their respective suites (not shown). The baseline threaded versions use the fast NPTL Pthreads library (provided with the Linux kernels). To test Varuna with task-based programs, we applied Varuna to five TBB programs (Barneshut, Histogram, Bzip2, RE, and ReverseIndex).

We also compared Varuna to two recent proposals: Feedback Diven Threading (FDT) [34] and Parcae [30]. FDT and Parcae are adaptive approaches applicable only to task-based programs. FDT can adapt to contention for locks and memory bandwidth. Parcae is more general, but optimizes for only one metric, execution time. It uses a hill climbing search method to adapt to dynamic changes. We faithfully implemented FDT mechanisms and a Parcae-like search heuristic in the TBB runtime. Note that neither Parcae nor FDT can be applied to threaded programs. Hence we compare them with Varuna only for task-based programs.

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

Configurations. The results to follow in the next section show data for the configurations listed in Table 4. All data for a given experiment are normalized to PT_CG, which serves as the base case for comparison, and hence is not shown in the figures. Columns 3 and 6 in Table 3 list the thread and vtask count for the PT_FG and Varuna configurations, for the two platforms, Opteron and Xeon, respectively. The higher thread/vtask values are chosen based on the input size and are spawned by giving a different parameter to the command line; the source code is left untouched. The task count for the TBB versions are same as PT_FG. For multithreaded versions of Dedup, Bzip2 and ReverseIndex we did not spawn a higher number of vtasks as they are already written with enough periodic safe points and dynamic load balancing capabilities. To measure the instantaneous IPS needed to compute *qc(P)* in the scalability model, we used the PAPI library APIs [27]. Energy was measured using a Wattsup meter to which the experimental machines were connected.

Results Exposition. The exposition of the results is grouped along the lines of the execution environments, isolated and multipro-

Config.	Description
PT_CG	Pthreads programs compiled with `-lpthread` with the default number of threads
PT_FG	Pthreads programs compiled with `-lpthread` executing with a higher number of threads (spawned by changing the command line argument)
TBB	TBB programs compiled with `-ltbb` with default number of tasks and threads
Parcae [30]	State-of-the-art adaptive scheme implemented in TBB
FDT [34]	State-of-the-art adaptive scheme implemented in TBB
V_base	Pthreads programs compiled with `-lvaruna` with adaptation capability disabled (number of threads fixed to default value and never varied during runtime)
V_PT_T	Pthreads programs compiled with `-lvaruna` optimized for MIN(time)
V_PT_C	Pthreads programs compiled with `-lvaruna` optimized for MIN(consumption)
V_TBB_T	TBB programs compiled with `-lvaruna` optimized for MIN(time)
V_TBB_C	TBB programs compiled with `-lvaruna` optimized for MIN(consumption)

Table 4. Different configurations used in experiments.

grammed. The first tests Varuna's basic capabilities (§ 5.1). The second stress tests Varuna in a range of highly dynamic, multiprogrammed operating conditions (§ 5.2).

5.1 Isolated Environment

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

Varuna overheads.

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

Figure 5 shows the results of threaded programs on the Xeon (although not shown, trends on the Opteron are similar). V_base incurs no noticeable overheads as compared to PT_CG despite creating a large number of vtasks for several programs (Table 3, column 6). This is because vtasks are userspace objects and have negligible creation and preemption overheads. For some of the programs (Barneshut, Dedup, Swaptions and Wordcount), V_base actually improved performance. This is because these programs exhibit irregular memory access patterns and V_base is able to improve their efficiency by applying: (i) lazy vtask creation (which avoids memory explosion), and (ii) fine-grained dynamic load balancing via randomized work-stealing. V_base reduces the execution time (Figure 5(a)) and energy consumption (Figure 5(b)) as compared to PT_CG on average (HM) by 6% and 3%, respectively, on the Xeon. On the Opteron, it reduces execution time and energy on average (HM) by 5% and 2%, respectively (not shown).

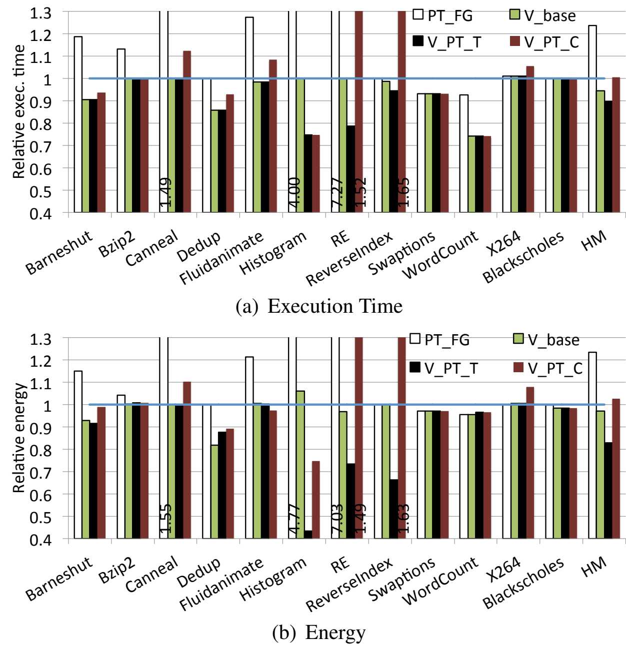
Result 2. *PT_FG does not benefit programs as does V_base.*

Although PT_FG creates fine-grained work like V_base, it degrades performance in most cases as compared to PT_CG as shown in Figure 5. This is due to the high overheads involved in creating a large number of OS threads. Since each thread seeks OS resources, large number of threads can create contention, e.g., in Barneshut due to frequent barrier synchronization, in Histogram due to page table lock contention [9] and in RE due to memory exhaustion.

Optimizing MIN(time).

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

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



(a) Execution Time

(b) Energy

Figure 5. Execution time and energy comparison of threaded programs on the Xeon.

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

Three of our threaded benchmark, ReverseIndex, Histogram and RE exhibit contention for shared resources. For them, as the DoP(time) columns 4 and 7 in Table 3 show, Varuna (V_PT_T) chooses a DoP far less than the maximum (e.g., 3, 1 and 8, respectively on the 16-core Opteron) to alleviate the contention.

ReverseIndex exhibits significant disk activity and hence its performance depends on the parallelism that the disk bandwidth can handle. The best parallelism point for ReverseIndex is different on different machines. On the Xeon, it scales up to 16 threads, whereas on the Opteron it does not scale beyond 3 (Figure 3(a)). As shown in Figure 5, V_PT_T reduces ReverseIndex's execution time and energy by 6% and 34%, respectively, on the Xeon, and 2% and 2%, respectively, on the Opteron (not shown), as compared to PT_CG. Reductions in execution time are modest because the performance of this benchmark does not degrade significantly for higher thread counts wrt the best DoP. The energy savings are greater on the Xeon since with V_PT_T, some processors are idle and are put into deep sleep states, a feature that is not available in the Opteron.

Histogram scales poorly due to contention to the page table. Anything beyond the sequential execution degrades its perfor-

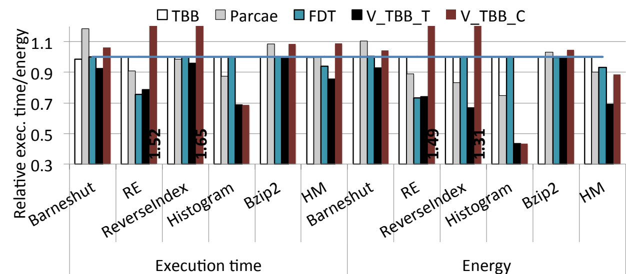


Figure 6. Execution time and energy comparison of task-based Varuna, TBB, Parcae and FDT on the Xeon.

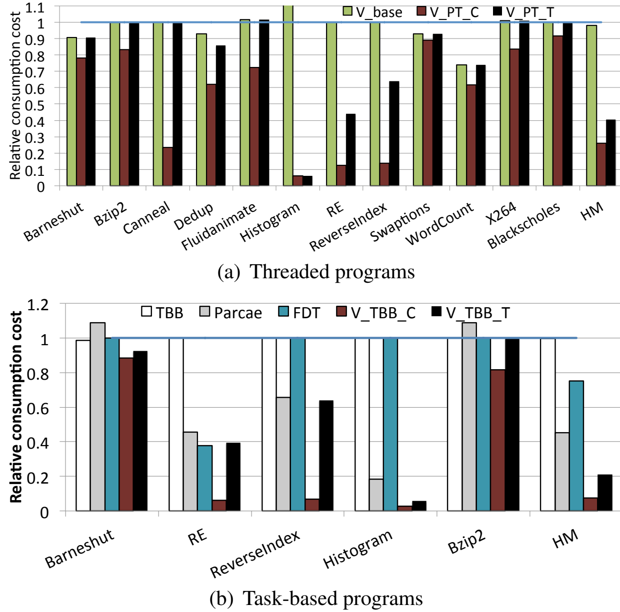


Figure 7. Resource consumption cost on the Xeon.

mance. It is an example of a likely future scenario, where an external software component causes contention, over which the programmer has no control. By dynamically adapting the number of hardware threads, V_PT_T is able to reduce its execution time and energy by 26% and 57%, respectively, on the Xeon and 20% and 20%, respectively, on the Opteron (not shown).

RE, a packet de-duplicating program, has abundant packet-level parallelism but has a lock protected hash table that each packet must access serially. The program incurs degradation both in execution time and energy beyond a thread count of 8 on the Opteron and 12 on the Xeon (Table 3, columns 4 and 7). V_PT_T reduces the execution time and energy consumption by 22% and 27%, respectively, on the Xeon and by 8% and 12%, respectively, on the Opteron (not shown). Thus, Varuna can handle contention to different resources.

For non-contending programs, V_PT_T incurs negligible performance degradation (less than 1%), primarily on the Opteron (not shown), due to the overheads associated with varying the number of threads.

V_PT_T reduces the execution time and energy consumption as compared to PT_CG on average (HM) by 11% and 18%, respectively, across all the programs on the Xeon. On the Opteron, it reduces execution time and energy on average (HM) by 9% and 10%, respectively.

Handling task programs.

Result 6. *Varuna is as effective for task programs as it is for threaded programs.*

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

Figure 6 shows Varuna's (V_TBB_T) time and energy efficiency for the MIN(time) metric when applied to the unmodified TBB programs, on the Xeon. It also compares the results with Parcae and FDT, two of the recent adaptive approaches. FDT employs a resource-specific mechanism to detect and avert contention. It can detect contention to locks (RE), however, it cannot detect contention to either the disk bandwidth (ReverseIndex) or the page table (Histogram). While Parcae improves performance and energy consumption of all contending programs (RE, ReverseIndex, Histogram), its slow hill climbing search-based approach degrades time and energy efficiency of non-contending programs (Barneshut and Bzip2) over PT_CG baseline. Varuna, on the other hand, due to its holistic and quick adaptation, improves over PT_CG on an average (HM) by 15%, and outperforms FDT by 8% and Parcae by 14%. The average energy savings are even higher, 31% over PT_CG, 23% over FDT and 21% over Parcae.

Optimizing MIN(consumption).

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

Recall that resource consumption is the product of the average number of hardware threads used by the program and its total execution time. Figure 7 shows the resource consumption cost of both threaded and task programs. On an average (HM), V_PT_C reduces the consumption cost by 84% for multithreaded programs over PT_CG and outperforms V_PT_T by 15% (Figure 7(a)). For task programs, V_TBB_C, on an average, reduces the consumption cost by 93% and outperforms V_TBB_T by 14% (Figure 7(b)). FDT and Parcae are unable to optimize for this metric. Parcae, similar to V_TBB_T, applies its MIN(time) adaptation, which incidentally also improves resource consumption, but only to some extent. V_TBB_C outperforms Parcae by 38% and FDT by 68%.

Figures 5 and 6 also show the time and energy efficiency when Varuna (V_PT_C and V_TBB_C) optimizes for MIN(consumption) metric. When compared to V_PT_T (V_TBB_T), V_PT_C (V_TBB_C) degrades the execution time and energy consumption of several programs. This is because the MIN(consumption) metrics essentially permits use of a resource only if it is effectively utilized. For example, it picked a DoP of 10 for Barneshut on the Opteron because the program speeds up linearly up to 10 threads, beyond which the gains are only sub-linear (Figure 3(b)). Even a trivial increase in parallelism of histogram, RE and ReverseIndex increases the contention to resources. When applied for this metric, Varuna throttles back their DoP to 1 (Table 3, columns 5 and 8). For threaded programs, on an average, V_PT_C degrades the execution time and energy consumption by 12% and 18%, respectively, as compared to V_PT_T. However, the average degradation is significantly lower than PT_CG, 1% in execution time and 2% in energy. For task programs, V_TBB_C incurs 9% degradation in execution time over PT_CG. However, it saves the average energy consumption by 12%.

Parallelism Determination Accuracy.

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

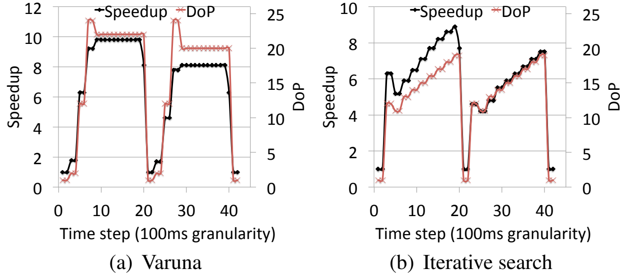
When adaptive optimization is applied, Varuna determines the DoP as per the MIN(time) and MIN(consumption) metrics. Table 3, columns 4 and 5 show the DoP for the Opteron, and columns 7 and 8 for the Xeon, for the two metrics, respectively. For both the machines and metrics, Varuna does as well as the best static DoP (shown in parantheses), which is determined by performing a full static thread sweep.

5.2 Multiprogrammed Environment

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

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

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



(a) Varuna (b) Iterative search

Figure 8. Comparison of search heuristics

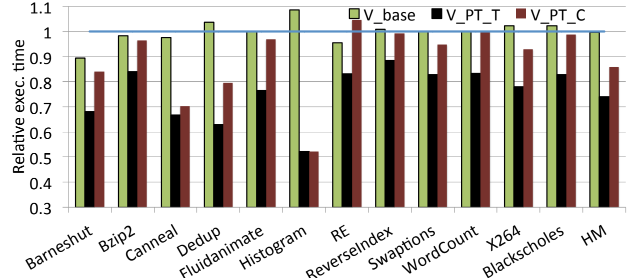


Figure 9. Execution time of threaded programs when scheduled with dynamically varying instances of *mcf* on the Xeon, relative to PT_CG running in the same environment (Table 3, column 10).

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

PT_FG results for these experiments are not shown since they were poor (as was also the case in §5.1).

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

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

Note that in Figure 8(a), there is no single best operating point for Barneshut unlike in the isolated environment. FDT cannot handle this scenario as it assumes static operating conditions and does not have the ability to continuously adapt. It identifies the optimum DoP, typically once at the beginning of the program or at the inception of every user-defined phase, and fixes that value for the rest of the program/phase.

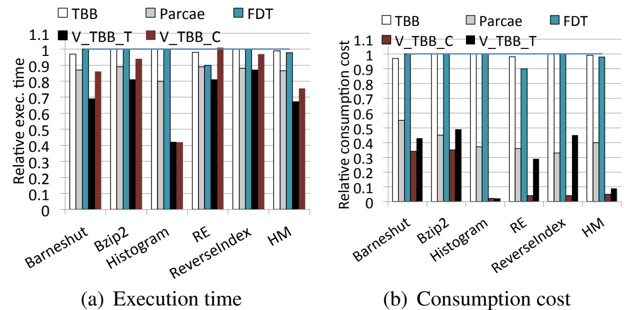
Figure 8(b) shows the adaptation using a Parcae-like search for the same scenario. It begins to determine the optimum DoP at *t=5*, like Varuna, but since it searches for the optimum by iteratively trying different DoPs, it is unable to find the optimum immediately. In this case, at *t=20* it is still searching, when the operating conditions change (due to the new mcf instance), causing it to restart the search to adapt to the new conditions. As the figure shows, an iterative search strategy can take longer to adapt to the conditions, and if the conditions change rapidly, they may be far less effective than Varuna.

Table 3 (columns 9 and 10) presents the execution times achieved by the PT_CG programs in this environment against the isolated environment, on the Xeon. As it can be seen, all the PT_CG programs in this environment incur significant degradation (average of 1.93x). This is because of the additional contention caused by the co-scheduled *mcf* instances not only to the shared resources, but also to processing cores and private caches.

Figures 9 and 10(a) present the execution times achieved for threaded and task-based programs, respectively, by Varuna relative to PT_CG for this multiprogrammed scenario (Table 3, column 10). Energy savings are not presented since their trends looked similar to the corresponding execution times. V_PT_T (V_TBB_T) reduces the execution time of threaded (task) programs on an average by 26% (33%) as compared to PT_CG. As compared to PT_CG in the isolated environment (Table 3, column 9), V_PT_T limits the average degradation to 1.47x. V_TBB_T outperforms FDT and Parcae by 30% and 20%, respectively. Unlike in the isolated environment, V_PT_C (V_TBB_C) reduces the average execution time by 14% (25%) over PT_CG and is only 12% (12%) slower than V_PT_T (V_TBB_T). This is because, in this environment, a program receives fewer resources, due to sharing of resources with other programs, and hence the optimum DoPs computed by these metrics are not far apart.

Figures 11 and 10(b) present the resource consumption cost achieved by Varuna for threaded and task-based programs, respectively, relative to PT_CG. V_PT_C (V_TBB_C) reduces the average (HM) consumption cost for threaded (task) programs by 90% (95%) over PT_CG. However, it outperforms V_PT_T (V_TBB_T) only by 6% (5%) since their optimum DoPs are similar. As in the isolated environment, Parcae's MIN(time) adaptation incidentally improved the resource consumption cost by 60% over PT_CG. However, Varuna (V_TBB_C) outperforms FDT and Parcae by 92% and 35%, respectively.

Adapting to Contention Due to Excessive Threads. In this scenario, we successively launched 8 instances of our benchmark programs, with the same input, on the Opteron. Each instance creates



(a) Execution time (b) Consumption cost

Figure 10. Execution time and resource consumption cost of task-based programs when scheduled with dynamically varying instances of *mcf* on the Xeon.

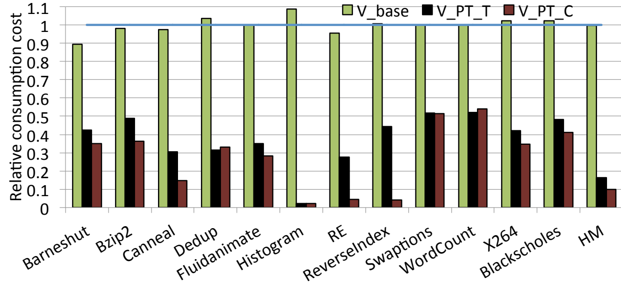


Figure 11. Resource consumption cost of threaded programs when scheduled with dynamically varying instances of *mcf* on the Xeon.

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

Figure 12 presents the execution time achieved by Varuna for the threaded programs, when optimized for MIN(time) metric, relative to PT_CG. Unlike PT_CG, which tries to allocate resources to all 128 threads at the same time, V_PT_T reduces the number of threads employed for each instance individually, thereby avoiding unnecessary context switches and hence its negative impact. It reduces the execution time on an average by 11% over PT_CG. Unlike the previous scenarios, V_base degrades the execution time for some of the programs as compared to PT_CG. This is due to the interference caused by excessive context switching to its runtime data structures.

Benchmarks with Different Resource Demands. Next we evaluate Varuna's effectiveness when different benchmarks with different resource needs are co-scheduled. In the interest of space, we present two case studies, without any figures.

In the first study, one instance each of Barneshut and Blackscholes are launched simultaneously on the Xeon, each spawning 24 threads. Both the benchmarks can scale up to the maximum number of hardware contexts on this platform and have no contention to any of the resources when executed in isolation. But when these benchmarks are co-scheduled, they have to contend for processing cores and their corresponding private caches. Compared to the baseline which relies on OS scheduling, Varuna reduces the execution time of Barneshut instances and Blackscholes by 5% and 18%, respectively, and the overall energy consumption by 18% for MIN(time). For MIN(consumption), Varuna reduces the resource consumption cost of Barneshut and Blackscholes by 11% and 29%, respectively. In the baseline case, the OS time multiplexes the total 48 threads in an arbitrary fashion. Varuna avoids this multiplexing by ensuring that the total number of threads that the benchmark employs does not exceed the number of available cores.

In the second study, we successively launched one instance of ReverseIndex and one instance of Barneshut on the Xeon. As seen in §5.1, ReverseIndex does not scale beyond 16 threads (Table 3, column 7) when executed in isolation on the Xeon, due to disk contention. Co-scheduling such a program with other programs can degrade the execution efficiency of all, due to non-optimum use of resources. This is what we observed with PT_CG. Unlike the baseline case in which ReverseIndex attempts to use all the cores, Varuna uses only 16 cores for MIN(time) and 1 core for MIN(consumption), while leaving the remaining unused. The co-located Barneshut is then free to use the unused resources, thereby improving its efficiency. This judicious sharing of resources helps Varuna to reduce the execution time of Barneshut and ReverseIndex by 67% and 10%, respectively, and the overall energy consumption by 14% for MIN(time). For MIN(consumption), Varuna reduces the consumption cost of Barneshut and ReverseIndex by 21% and 71%, respectively.

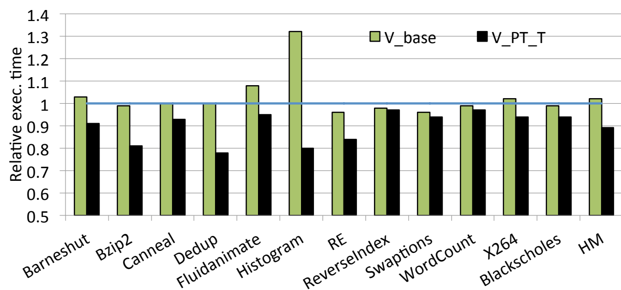


Figure 12. Execution time when 8 instances of the same thread program are scheduled together on the Opteron.

6. Related Work

Several recent proposals dynamically vary the degree of parallel execution from within the program. We have summarized these proposals in § 1(Table 1). We discuss some more related work in this section.

Dynamically adapting a program's parallelism has been widely studied in the OS community [3, 6, 12, 14, 25]. However, these techniques require a two-level scheduler necessitating changes in both the OS and the program. Moreover, these techniques primarily focus on preventing processor underutilization due to blocking I/O or synchronization and have not dealt with contention to microarchitectural resources, such as, caches and memory bandwidth.

Several papers have proposed microarchitectural techniques to alleviate the negative impacts of contention to shared hardware resources, such as shared cache and memory, in a multiprogrammed environment. These techniques [19–21] mostly rely on partitioning the shared resources amongst the different programs or slow down the execution speed of the program [13, 18]. They are orthogonal to Varuna and can be deployed with Varuna.

7. Conclusions and Future Work

In this paper, we proposed Varuna, a system that provides a comprehensive solution to optimize a program's parallel execution. Varuna takes a principled approach to modeling a program's scalability and uses it to dynamically, continuously and rapidly adapt a program's parallelism in dynamically changing conditions. Varuna is compiler/programming model independent. It requires no source code/OS modifications and is applicable to arbitrary threaded and task-based programs, due to its vtask mechanism. Further, it can optimize a program's execution for two different metrics, MIN(time) and MIN(consumption). For the MIN(time) metric, Varuna reduced the execution time on an average by 15% in the isolated environment and 33% in the multiprogrammed environment. The concomitant energy savings are 31% and 32%, respectively. For the MIN(consumption) metric, Varuna saved resource consumption cost by 84% and 90% in isolated and multiprogrammed environments, respectively. While Varuna was evaluated in the context of two optimization metrics, we believe it can be easily extended to support additional metrics, such as Watts, perf/W, perf/J, etc. and can also be extended to model contention to heterogeneous systems

comprising of accelerators and I/O devices. We plan to investigate these aspects as part of our future work.

8. Acknowledgments

We thank Michael Swift for his feedback on early drafts of the paper. We thank the anonymous reviewers for their insightful comments and feedback. This material is based upon work supported, in part, by the National Science Foundation under Grant CCF-0963737. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] Posix threads programming. https://computing.llnl.gov/tutorials/pthreads/.

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

[3] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for unix development. pages 93–112, 1986.

[4] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative task management without manual stack management. In *Proceedings of the USENIX ATC*, pages 289–302, 2002.

[5] A. Anand, C. Muthukrishnan, A. Akella, and R. Ramjee. Redundancy in network traffic: findings and implications. In *SIGMETRICS*, pages 37–48, 2009.

[6] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: effective kernel support for the user-level management of parallelism. In *SOSP*, pages 95–109, 1991.

[7] G. J. Barbara Chapman and R. van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*. The MIT Press, 2007.

[8] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. In *PACT*, pages 72–81, 2008.

[9] S. Boyd-Wickizer, A. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An analysis of linux scalability to many cores. In *OSDI*, 2010.

[10] 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 *ICS*, pages 157–166, 2006.

[11] M. Curtis-Maury, A. Shah, F. Blagojevic, D. S. Nikolopoulos, B. R. de Supinski, and M. Schulz. Prediction models for multi-dimensional power-performance optimization on many cores. In *PACT*, pages 250–259, 2008.

[12] K. Dussa, B. Carlson, L. Dowdy, and K.-H. Park. Dynamic partitioning in a transputer environment. *SIGMETRICS Perform. Eval. Rev.*, 18(1):203–213, Apr. 1990.

[13] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt. Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems. In *ASPLOS*, pages 335–346, 2010.

[14] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: an operating system architecture for application-level resource management. In *SOSP*, pages 251–266, 1995.

[15] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI*, pages 212–223, 1998.

[16] J. Gilchrist. Parallel data compression with bzip2. In *PDCS*, pages 559–564, 2004.

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

[18] R. Illikkal, V. Chadha, A. Herdrich, R. Iyer, and D. Newell. Pirate: Qos and performance management in cmp architectures. *SIGMETRICS Perform. Eval. Rev.*, 37:3–10, March 2010.

[19] R. Iyer. Cqos: a framework for enabling qos in shared caches of cmp platforms. In *ICS*, pages 257–266. ACM, 2004.

[20] R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt. Qos policies and architecture for cache/memory in cmp platforms. In *SIGMETRICS*, pages 25–36. ACM, 2007.

[21] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *PACT*, pages 111 – 122, 2004.

[22] M. Kulkarni, M. Burtscher, K. Pingali, and C. Cascaval. Lonestar: A suite of parallel irregular programs. In *ISPASS*, pages 65–76, 2009.

[23] J. Lee, H. Wu, M. Ravichandran, and N. Clark. Thread tailor: Dynamically weaving threads together for efficient, adaptive parallel applications. In *ISCA*, 2010.

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

[25] C. McCann, R. Vaswani, and J. Zahorjan. A dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 11(2):146–178, May 1993.

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

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

[28] H. Pan, B. Hindman, and K. Asanović. Composing parallel software efficiently with lithe. In *PLDI*, pages 376–387, 2010.

[29] A. Raman, H. Kim, T. Oh, J. W. Lee, and D. I. August. Parallelism orchestration using DoPE: the degree of parallelism executive. In *PLDI*, pages 26–37, 2011.

[30] A. Raman, A. Zaks, J. W. Lee, and D. I. August. Parcae: a system for flexible parallel execution. In *PLDI*, pages 133–144, 2012.

[31] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor systems. In *HPCA*, pages 13–24, 2007.

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

[33] S. Sridharan, G. Gupta, and G. S. Sohi. Holistic run-time parallelism management for time and energy efficiency. In *ICS*, pages 337–348, 2013.

[34] M. A. Suleman, M. K. Qureshi, and Y. N. Patt. Feedback-driven threading: power-efficient and high-performance execution of multithreaded workloads on cmps. In *ASPLOS*, pages 277–286, 2008.

[35] V. Uhlig, J. LeVasseur, E. Skoglund, and U. Dannowski. Towards scalable multiprocessor virtual machines. In *USENIX VM*, 2004.