Holistic Run-time Parallelism Management for Time and Energy Efficiency

Srinath Sridharan Gagan Gupta Gurindar S. Sohi

Computer Sciences Department
University of Wisconsin-Madison
{sridhara, gagang, sohi}@cs.wisc.edu

ABSTRACT

The ubiquity of parallel machines will necessitate time- and energy-efficient parallel execution of a program in a wide range of hardware and software environments. Prevalent parallel execution models can fail to be efficient. Unable to account for dynamic changes in the execution environment, they may create non-optimum parallelism, leading to under-utilization of, or contention for, resources. We propose ParallelismDial (PD), a model to dynamically, continuously and judiciously adapt a program's degree of parallelism to the prevailing dynamic execution conditions. PD uses a holistic metric to measure system efficiency. The metric is used to systematically optimize the program's execution.

We incorporated PD in two different parallel programming models: Intel TBB, an industry standard, and Prometheus, a recent research proposal. Two prototypes were implemented and evaluated on stock multicore machines. Dedicated and multiprogrammed environments were considered. Experimental results show that the prototypes outperform the state-of-the-art approaches, on an average, by 15% on time and 31% on energy efficiency, in the dedicated environment. In the multiprogrammed environment, the savings are to the tune of 19% and 21% in time and energy, respectively.

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

ICS'13, June 10–14, 2013, Eugene, Oregon, USA.

1. INTRODUCTION

Trends in computing systems have put a spotlight on efficient execution of parallel programs [14]. Multicores are now standard in computing devices, ranging from mobile handsets to HPC servers. While time efficiency was traditionally the primary goal, energy efficiency is now equally significant. System designers have turned to parallel programs to achieve both objectives.

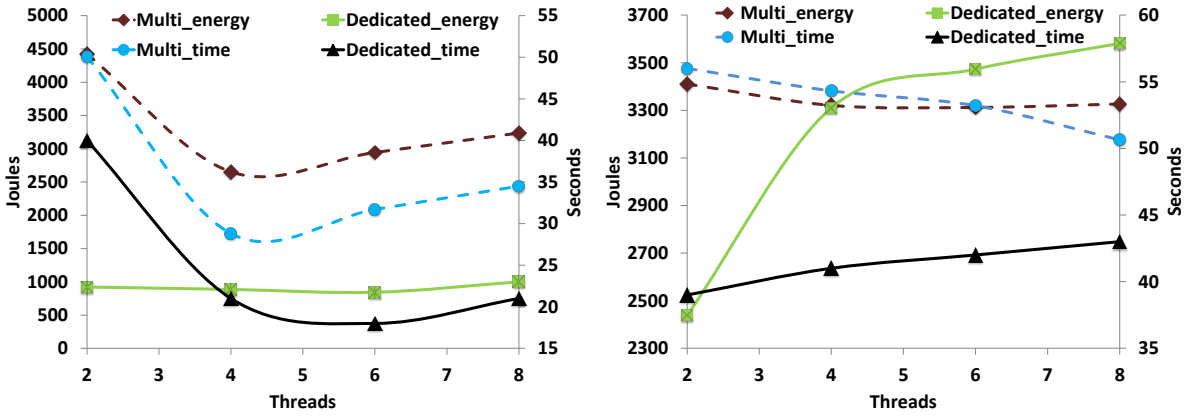
Multiple factors influence a parallel program's efficiency. In addition to the customary system artifacts (e.g., cache sizes), they include: (i) the parallelism exposed by the developer, and (ii) the interactions of the program's concurrent computations with each other and the host system. Too little parallelism can underutilize the system. Excess parallelism, on the other hand, can create contention for resources, such as processors, caches, main memory, bus, disk, other I/O, etc. Severe contention can lead to inefficient program execution, even worse than sequential execution. Hence performance optimization requires matching a program's parallelism to the execution environment.

Challenges to efficient execution in future systems will be further compounded by the dynamically changing operating conditions. Multiprogrammed platforms, executing a statically unknown mix of programs, will make continuously varying resources available to programs. Increasingly unreliable hardware will further exacerbate the variability [7].

In the past, efficiency was a concern for a select few parallel-program developers, but in the future it will be for many more. Until recently a small set of experts developed parallel programs for a very small set of machines. They tuned programs using intimate knowledge of the host, e.g., its microarchitecture. Often, they could assume the host was entirely available to them, and expect little or no interference from other programs. Going forward, we expect a multitude of developers to program commodity parallel systems, e.g., the vast array of cell phones, laptops, desktops, etc. We expect programs to be written without the detailed knowledge of the hardware, software, and operating conditions, e.g., resource demands of programs co-scheduled on an unknown system in a data center. Further, given the possibly large number of diverse target hosts, portability will be highly desired.

To achieve efficient program execution, it will be daunting for common programmers to account for the often complex and non-intuitive factors related to program behavior, operating conditions and system characteristics.

Therefore, we believe, optimizing performance of a parallel program on future systems will require automatic, dy-



(a) RE (b) Stream

Figure 1: Execution time and energy trends of two programs, RE and Stream, in dedicated and multiprogrammed environments on a Core i7-2600 workstation.

namic and continuous harmonization of its parallelism with the execution environment, while being agnostic to the underlying platform. Both, under- and over-subscription to various resources will need to be avoided.

Prevalent approaches take a limited view of the efficiency issues programmers face. The traditional multithreaded models put the onus of efficient execution largely on programmers. Task-based programming models, e.g., Intel Thread Building Blocks (TBB) [42], ease some of the burden. They automate load-balancing of tasks to prevent resource underutilization. However, they do not address overutilization, nor account for co-scheduled programs. Although recent proposals take a broader view, some address contention in only a subset of resources [45], while others rely on offline profiling [9, 10, 26], and yet others may not optimize utilization of resources [39, 40] (§5).

We propose a model that takes a comprehensive and automated approach to efficient parallel program execution (§2). The model dynamically and continuously adapts a task-based program's demands to the currently available capacity of resources. It exploits inter-task dependence information to arbitrarily control the execution. The model defines a simple metric to measure the system's efficiency periodically. The metric effectively captures the changes in resource utilization, the operating environment, and the program's demands. Changes cause the model to search for the optimum operation point under the new conditions. We develop a hill climbing heuristic, based on the Tabu search [17], to locate the optimum point in the search space.

We applied the model to two different approaches to parallel programming: TBB, a well-established task execution model, and Prometheus, a sequentially-determinate model [3]. Two prototypes were developed (§3) and evaluated on two stock multicore machines (§4). Standard benchmarks were developed using both models and tested in a dedicated as well as a multiprogrammed environment. Experimental results show that the prototypes outperform a state-of-the-art proposal on an average by 15% on time and 31% on energy efficiency, in the dedicated environment. In the multiprogrammed environment, the savings are to the tune of 19% and 21% in time and energy, respectively.

2. THE ParallelismDial MODEL

2.1 Motivation

The key to efficient program execution lies in accounting for the execution environment's characteristics at run-time. A program's execution environment is defined by the program's behavior, resources provided by the system to the program, and the utilization of resources by all co-located programs in the system. Prevalent parallel programming models primarily focus on expressing parallelism and exposing it to the system. Although necessary, only exposing parallelism is inadequate for efficient execution. The dynamic execution environment also influences the program's efficiency.

Figure 1 shows how different aspects of an execution environment impact a program's efficiency. It plots the energy consumed by and the execution time of two benchmarks, *Stream* and *RE*, written using TBB, in a dedicated and an example multiprogrammed scenario, on an Intel Core i7 2600 workstation[1].

RE is a networking redundancy elimination application [4]. In the dedicated environment, at two threads the resources are underutilized (Figure 1(a)). Increasing the threads to up to six improves performance without consuming additional energy. Clearly, insufficient parallelism can lead to inefficiency.

Beyond six threads, RE's performance and energy degrade, slightly, in the dedicated environment, indicating resource contention. In the multiprogrammed environment, as more programs occupy the machine, the performance degrades beyond four threads. When the total number of threads exceeds the total number of hardware contexts, and the OS deschedules RE threads, including at times when they hold what is otherwise a very low-contention lock, the efficiency drops.

In contrast to RE, Stream exhibits an opposite trend (Figure 1(b)). Stream, designed to stress the memory bandwidth, has ample parallelism and is indicative of numerical vector kernels whose datasets are larger than the cache capacity [30]. Even a modest attempt at parallel execution in a dedicated environment results in excessive parallelism,

[1]Details of the applications, machine, and operating environments are presented in §4.

creating memory contention, and thus degrades execution time and energy. However, in the multiprogrammed environment, when it is contending with other programs for other resources (e.g., cache capacity or issue slots in the SMT processor), it naturally slows down, thereby slowing down the creation of parallelism and hence the memory contention. Thus, higher occupancy of the machine proves to be beneficial rather than detrimental in the case of Stream.

Other applications in different dynamic operation scenarios may oversubscribe other system resources. Examples include processing cores, caches, memory, I/O device (e.g., disk) operating system data structures (e.g. page tables), synchronization primitives, among others.

Contention arises because the existing programming models assume that resources (hardware and software) are *isolated*, *static* and *unlimited* (primarily due to the illusion provided by the OS). However, resources are *shared*, *dynamic* and *limited*. As seen above, oversubscribed resources may lead to artificial serialization, either causing, or easing contention. Thus, as programs and tasks execute, they can interact with each other and the system unpredictably. Furthermore, a program's own demands may vary, e.g., due to phase changes, and a system's resources may vary, e.g., due to hardware failures.

Therefore, to achieve efficient parallel execution, the challenge is to take a holistic, system-wide view, and to dynamically and continuously control the parallelism to match the instantaneous capacity of resources, without user-intervention.

2.2 The Model Architecture

We present ParallelismDial (PD), a model that optimizes a program's execution efficiency by dynamically and continuously adapting the program's parallelism to the execution environment.

To dynamically adapt a program's parallelism, PD: (i) assesses the efficiency of the system, (ii) detects contention, changes in the program, and changes in the available resources, (iii) responds to the changes by finding the optimum degree of parallelism (DoP) under the new conditions, and (iv) controls the execution to move to the desired DoP.

To adapt continuously, PD periodically repeats the above steps during a program's execution. Changes in successive measures of efficiency indicate changes in the execution environment. As we see below, responding to the changes requires arbitrary control of the execution.

To arbitrarily control a program's execution, we propose to use a task-based execution model that is also *dependence-aware*. We view a program to comprise appropriately-sized concurrent tasks. Tasks may be viewed as computations that compose a thread in a conventional multithreaded program. They are decoupled from the execution contexts (threads) and thus provide the flexibility needed to manage the program's execution, typically at task boundaries. The finer granularity of tasks permits distribution of tasks to contexts as soon as contexts become available, helping prevent resource underutilization. These aspects are also exploited by modern task-based models [13]. Further, as shown below, awareness of dependences between tasks permits the adaptability needed to prevent resource overutilization, while ensuring a program's forward progress.

PD consists of four main components: the *Spooler*, the *Regulator*, the *Scheduler* and the *Monitor*. It assumes an underlying system that provides a pool of worker threads, common in a modern OS, and a low-overhead means to measure an execution environment's characteristics, e.g., via performance counters provided by modern processors.

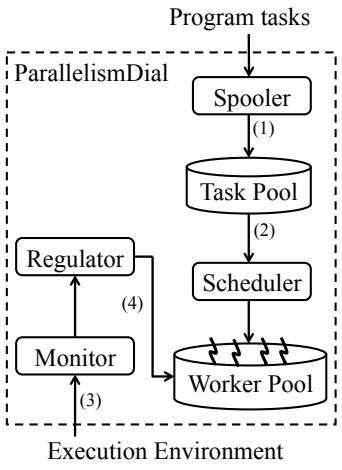


Figure 2: ParallelismDial model architecture.

Figure 2 summarizes the model's operations. The Spooler gathers the tasks exposed by the programmer and creates a task pool (1). The Scheduler assigns tasks from the task pool to threads, if available, from the worker pool (2). Once assigned, the threads execute the tasks. The Monitor periodically probes the system and measures the necessary parameters to compute the instantaneous efficiency of the program (3). The Regulator assesses the changes in the operating efficiency. It responds by continuously seeking the optimum point of operation by controlling the number of workers allotted to the program (4). The number of workers executing tasks directly corresponds to the maximum exposed parallelism of the program.

Each aspect of PD is further described below.

2.2.1 Controlling the Execution of Programs

Regulating parallelism requires the ability to arbitrarily block work from being performed when resources are oversubscribed, and introduce work into the execution environment when resources are undersubscribed. For example, in Figure 1(a), in the multiprogrammed environment, increasing the parallelism from four to six degrades RE's execution efficiency. In this case, it is ideal to execute no more than four threads. Doing so requires preventing additional work from being assigned to more threads. However, this can lead to unintended consequences in canonical parallel programs, especially in those with arbitrary dependence patterns.

Challenges in Independence-based Programs

Canonical parallel programs are *independence-based*; the programmer ensures independence amongst concurrently executing computations. Using appropriate synchronization primitives the programmer enforces an arbitrary order between the computations when they access shared data. However, in certain instances the enforced order is specific, e.g., to execute the consumer before a producer in a produce-consumer style program. Conventional parallel programming assumes that once introduced in the execution environment, work is guaranteed to receive resources. Not granting resources can violate this guarantee. Controlling the execu-

tion, without being aware of the programmer's intentions, e.g., blocking the producer, can hinder forward progress, or worse, deadlock the execution.

Consider the popular Pthreads implementation of Pbzip2 [16]. It reads data blocks serially from an input file, one at a time, compresses the blocks in parallel, and writes the results to an output file. The computations are organized into threads, as shown in Figure 3(a). Block-read operations are grouped into one thread, e.g., R0 to R3 in thread TH0. Compress operations are grouped across multiple *compress* threads (to achieve concurrency), e.g., C0 to C3 in threads TH1 and TH2. Block-write operations are grouped into one thread, e.g., W0 and W1 in thread TH3. The threads are expected to be co-scheduled.

When the program executes, say three processors, U0 to U2, are allotted to it. Threads TH0, TH1 and TH2 execute in epoch t0, (Figure 3(b))[2]. R0 and R1 in TH0 read blocks which are compressed by C0 and C1, respectively. Now, say, at the start of epoch t1, U0 is taken away from the program to regulte parallelism, and TH0 is blocked. TH1 and TH2 complete C0 and C1, and advance to execute C2 and C3. C2 and C3, in turn, wait for R2 and R3 in TH0 to read the blocks from the input file. However, if no more processors are available, TH0, which can make progress, remains blocked, stalling the program's progress, or even deadlocking it, while available resources are occupied by tasks that cannot make progress, defeating the very purpose of controlling the execution. Although the example is from a Pthreads program, a task-based implementation with a similar structure would be equally vulnerable.

Therefore, PD requires that either a program's concurrent tasks not enforce a specific order among themselves, or they are independent, or the dependence information is supplied to PD. Note that this condition does not preclude use of PD in independence-based programs. It is possible to write such programs without creating a specific order between tasks. For example, we rewrote Pbzip2, using TBB, without resorting to producer-consumer style parallelism. The program is divided into three non-overlapping phases: read, compress and write. Read and write phases are sequential. All tasks in the compress phase are independent. We show successful application of PD to both independence-based and dependence-aware program execution (§3). How PD uses dependence information to ensure forward progress is discussed next.

Dependence-aware Regulated Execution

The Spooler in the proposed model, accepts tasks exposed by the programmer, but admits only independent tasks into the task pool; dependent tasks, when identified, are suspended. Once their dependences have resolved, they are introduced into the task pool and become candidates for the Scheduler to assign to worker threads. (Two example implementations of the Spooler are presented in §3.)

The Regulator changes the DoP by changing the number of workers in the worker pool. The change is effected without pausing the program. It is straightforward to add idle or new workers to the pool. Workers are removed from the pool only after they complete their currently assigned tasks.

The Scheduler operates independently from the Regulator. Given a pool of workers, it assigns tasks from the task pool, if available, to a worker, as soon as the worker is free.

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

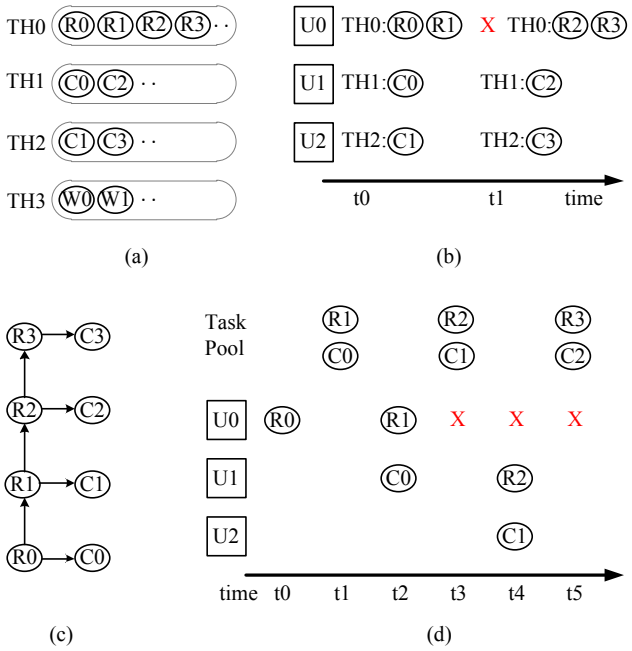


Figure 3: Comparison of independence-based and dependence-aware execution of Pbzip2

It also ensures that tasks eventually receive resources, either by way of a predefined or an aged-based priority. A decoupled Scheduler helps maximize resource utilization without waiting for the Regulator to optimize the DoP.

Now consider the same Pbzip2 example in our model. Figure 3(c) shows its computations formulated as tasks, R1, R2, C0 - C3, and the dependences between them[3]. Figure 3(d) shows the execution as per our model in finer time steps. At time t0, U0, U1 and U2 are alloted to the program. R0 executes. Since no other independent computations exist, the task pool is empty. When R0 completes in t1, dependences of R1 and C0 have resolved, and hence they are added to the task pool. The Scheduler assigns them to U0 and U1 in t2. In t2 the task pool is once again empty since R2 and C1 are suspended. R2 and C1 are added to the task pool once R1 completes in t3. In t3, say U0 is taken away. The Scheduler assigns R2 and C1 to U1 and U2 in t4. Note that unlike in Figure 3(b), C2 does not get scheduled unless R2 completes. Thus the model ensures dependent tasks do not occupy execution resources. The program can make forward progress as long as it is allotted at least one worker.

Recent research proposals also exploit task-dependence information, albeit for a different purpose, to execute parallel programs. SMPSs [38] and Gupta et al. [18] leverage this philosophy to discover distant parallelism. Prometheus [3] and DPJ [6] execute only independent tasks concurrently to achieve determinacy. We also note that TBB presents a "pipeline" abstraction that would capture the dependence information in the above Pbzip2 example, if used.

2.2.2 *Assessing Execution Efficiency*

To continuously regulate parallelism, the Regulator needs a metric to assess: (i) the instantaneous efficiency of the program for a given DoP, and (ii) changes in the execution

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

environment without the involvement of any external entity such as the OS. We note that any change in the efficiency of the program or the execution environment ultimately manifests as a change in the total system energy expended and/or the program's instruction execution rate. Although changes in the system energy and the instruction rate may be used individually, we propose a simple, holistic, unified metric, *Joules Per Instruction (JPI)* that is affected by changes to both. As we demonstrate in our experiments (§ 4), JPI serves as an adequate metric.

In a program composed of N instructions, let JPI(P_1) be the system energy expended per instruction when parallelism P_1 is deployed. The total system energy expended to execute the entire program, TE(P_1) is

$$TE(P_1) = JPI(P_1) * N \quad (1)$$

If the parallelism is changed from P_1 to P_2, the total system energy expended, TE(P_2) is

$$TE(P_2) = JPI(P_2) * N \quad (2)$$

For the given program, to determine whether P_2 is more efficient than P_1, we define *Efficiency (Eff)* as follows:

$$Eff(P_1, P_2) = \frac{TE(P_2)}{TE(P_1)} = \frac{JPI(P_2)}{JPI(P_1)} \quad (3)$$

Eff(P_1,P_2) is a measure of relative JPI; a value less than 1 indicates P_2 is more efficient and is preferred over P_1, whereas a value greater than 1 indicates parallelism P_1 is preferred.

Equation (3) defines *Eff* for the entire program. In order to use *Eff* as an online metric, the Regulator computes *Eff* and adapts the parallelism at frequent intervals. Given two such intervals D_1 and D_2, and the respective parallelism points, P_1 and P_2, *Eff* is computed as follows.

$$Eff_{online}((P_1, D_1), (P_2, D_2)) = \frac{JPI(P_2, D_2))}{JPI(P_1, D_1)} \quad (4)$$

When P_1 is the same as P_2, Eff_{online} is expected to be around 1. Any other value would suggest a change in the execution environment indicating that the optimum parallelism point has shifted. We demonstrate how this observation is utilized, in the next subsection.

2.2.3 Optimizing Parallelism

Once the Regulator determines that the program efficiency can be improved, it employs a heuristic to optimize the DoP. The heuristic is based on the commonly used hill-climbing search algorithm. It locates the optimum DoP by systematically exploring the direction in the search space that yields higher efficiency.

Hill climbing algorithms are known to get "stuck" in a local optimum and fail to reach the global optimum. Hence the heuristic, developed using the guidelines due to Gendreau [15], incorporates *Tabu search* [17] to escape local optima. The Tabu search maintains a fixed-size *tabu_list* to log previously searched points along with their JPIs. The heuristic periodically "diversifies" the search from the current optimum by exploring unvisited points. Any time a point is searched, it is logged in the *tabu_list*. The oldest point in the list is evicted to make room for a new point, if needed. The heuristic, divided into five steps, is as follows:

Step 1: Establish Sequential Measure: Before beginning the search, the Regulator establishes a notion of *Sequential JPI* (JPI_{seq}) to ensure that the optimum point it finds is indeed profitable. Hence, whenever the program encounters new conditions, the Regulator alters the DoP to one (sequential) and executes the program for a pre-defined duration until the Monitor measures the baseline JPI_{seq}.

Step 2: Establish Initial Search Direction: Next, the Regulator establishes the search direction for the optimization process. It gathers JPIs corresponding to three DoPs, the mid-point, P_{mid}, of the predetermined maximum DoP (often, total hardware contexts in the system), and P_{mid} +/- 1. Tasks are executed for a pre-defined duration in each configuration. The better of the three points, P_{eff}, sets the search direction. The points are added to the *tabu_list* and the execution is switched to P_{eff}.

Step 3: Search for Optimum Parallelism: If P_{eff} is the same as P_{mid}, we already have the local optimum, else the Regulator linearly searches in the established direction [4]. It picks new unvisited points (not in the *tabu_list*) and executes tasks in each of these points for a pre-defined duration. Visited points are added to the *tabu_list* until a local optimum, P_{lopt}, is found. If JPI_{lopt} is not significantly better than JPI_{seq}, the Regulator starts over from step 1, otherwise it fixes the current configuration to P_{lopt}, and enters a "Sleep" mode.

Step 4: Sleep Mode: The Regulator continues to sleep until either: (i) the value of JPI_{lopt} changes by more than a pre-defined threshold, which may indicate phase change or introduction of another program in the system or change in the operating conditions, or (ii) the diversification threshold is reached. In case of (i), the Regulator clears the *tabu_list* and begins the search once again from step 1. In case of (ii), the Regulator saves the current JPI_{lopt} and moves to step 5.

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

3. THE PROTOTYPES

We evaluated PD by applying it to independence-based and dependence-aware task-based parallel programming models. For the former we picked TBB due to its ubiquity and its rich programming APIs. For the latter we chose Prometheus, from among the different research proposals, for its TBB-like APIs. TBB details can be found in [42] and Prometheus details are described in [2,3]. Both models provide C++ runtime libraries. The runtimes provide the APIs to programmers, and employ state-of-the-art dynamic load balancing schedulers to avoid resource underutilization.

Two PD run-time library prototypes, one each for TBB and Prometheus, were developed. We incorporated the TBB and Prometheus runtimes into our prototypes to leverage

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

their capabilities. Their runtime engines were modified and plugged into the PD runtime. The PD runtimes expose the respective APIs to programmers, and provide the efficiency mechanisms (common to both prototypes) described in §2. We describe the prototypes in this section and discuss their evaluation in §4.

Independence-based Prototype (PD_TBB)

Since TBB programs do not always convey dependence information, the Spooler in this prototype simply passes all dynamically discovered tasks into the task pool. For reasons described in §2, we ensured that the TBB-programs we developed did not explicitly enforce a specific order between their tasks.

Dependence-aware Prototype (PD_PM)

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

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

Given the above mechanisms in the two prototypes, to discover tasks in a program and expose them for execution, we now describe how they are efficiently executed.

3.1 Scheduling Execution

Both Prometheus and TBB use Pthreads as abstractions to create OS threads. Upon initialization, they create a fixed global pool of threads (by default, one per hardware context) and associate a worker with each thread. Each worker maintains a separate double ended queue (deque). Tasks are scheduled for execution by enqueuing them in the deques. Both TBB and Prometheus employ a Cilk-style work stealing algorithm to balance the load [13]. A worker seeks tasks first from its own deque, failing which it steals from someone else's deque. PD_TBB and PD_PM retain the schedulers from the respective runtimes, but replace the worker pool component with the PD worker pool design as described in §3.2.

3.2 Regulating Parallelism

To find the optimum operating point, the Regulator changes the DoP by controlling the size of the worker pool. Instead of a fixed pool of workers, it maintains two variable-sized disjoint pools: an *active pool* and a *suspended pool*. A worker can be associated either with the active pool or the suspended pool, but not both. Only the workers in the active pool are seen by the Scheduler. Workers in the suspended pool are put to sleep until the Regulator awakens and moves them to the active pool. The core algorithm for each worker remains the same as described in §3.1. An increase/decrease in the number of workers in the active pool results in an increase/decrease in the program's exposed DoP.

Table 1: Machine configurations

Machine Parameters	Xeon E5-2420	Core i7-2600
# Sockets	2	1
# Hardware Contexts	24	8
Clock Speed	1.9 GHz	3.4 GHz
Total Cache	15 MB	8 MB
Memory	32 GB	16 GB
Linux Kernel	2.6.32	2.6.32
Memory Controllers/Channels	1/2	2/6

Table 2: Benchmarks and input sizes

Benchmarks	Description	Input
Histogram [41]	Image analysis	1.4 GB bitmap file
Hash Join [11]	In-memory DB join	28 MB tables
Pbzip2 [16]	Compression	1.3 GB file
Reverse Index [41]	HTML Analysis	1.3 GB directory
Barneshut [24]	N-Body Simulation	100000 bodies
RE [4]	Packet dedup	1300000 packets
Stream [30]	Memory Streaming	915MB of data

3.3 Monitoring System Parameters

The Monitor is a system-specific component of the prototypes. It relies on information available from the system to measure *Eff* described in §2.2.2. At present, the monitor assumes an Intel SandyBridge-based microarchitecture which exports energy and performance counters. It can as easily work with other platforms with similar capabilities.

To measure processor energy, we use the RAPL (Running Average Power Limit) power management interface provided in the Intel SandyBridge microarchitecture [1]. To measure the number of instructions completed, we use the PAPI library APIs [32]. We do not count instructions that may cause potential side effects when determining the efficiency of the program. For example, we omit instructions used for synchronization, and OS mode instructions.

4. EVALUATION AND RESULTS

In this section, we describe our experimental setup, the hardware and the benchmarks used in our experiments, and the results obtained. Experiments were conducted on two 64-bit x86 machines, described in Table 1.

Table 2 lists the benchmarks and the input data sizes we selected from different benchmark suites to evaluate PD. The baseline parallel versions use the fast NPTL Pthreads library (which comes along with the 2.6.32 Linux kernel that both the platforms run) to parallelize their algorithms. We compiled all the benchmarks with GCC 4.4.3 using -O3 optimization and architecture flag (-march=core2).

We present results of PD for two different environments: dedicated, where a benchmark is the only program running on the machine (§4.1), and multiprogrammed, where other applications run simultaneously with the benchmarks (§4.2). We evaluated PD for performance and energy efficiency. We measured the execution time and energy of the entire program, including the overheads associated with any thread/task creation, rather than just the parallel regions. We used the RAPL interface to measure the total energy consumed. We fixed the measurement time interval for the Regulator at 100ms and the diversification threshold to 5s. PD sought a new DoP when JPI changed by more than 10%.

4.1 Dedicated Environment

To demonstrate the effectiveness of PD in a dedicated environment, we provide four points of comparison. The first

Table 3: Dedicated environment: Optimum DoP (execution time, energy consumed).

#	Benchmarks	i7-2600	E5-2420
1	Histogram	1 (*12*s,*93*J)	1 (*11*s,*102*J)
2	Hash Join	4 (*9.4*s,*353*J)	11 (*24*s,*898*J)
3	Pbzip2	8 (*36*s,*1608*J)	24 (*23*s,*999*J)
4	Reverse Index	4 (*31*s,*384*J)	10 (*38*s,*853*J)
5	Barneshut	8 (*26*s,*1163*J)	24 (*18*s,*919*J)
6	RE	5 (*15*s,*806*J)	13 (*32*s,*1401*J)
7	Stream	2 (*32*s,*562*J)	7 (*17*s,*821*J)

point is the Pthreads baseline. The second and third points are the Prometheus (PM) and TBB implementations. The fourth point of comparison is a recently proposed technique, Feedback Driven Threading (FDT) [45], which dynamically adjusts the number of threads for homogeneous loops and considers contention to only two types of resources, locks and memory bandwidth. The original version of FDT is implemented in OpenMP. We faithfully implemented FDT in the PM runtime to provide a fair comparison against PD. For PD, we present the results for the two prototypes: PD_PM and PD_TBB.

Figures 4 and 5 present the performance and energy results for all our benchmarks on both the platforms for the dedicated environment. Every data point in our results is an average of 10 runs and normalized to the Pthreads baseline (not shown). Table 3 shows the optimum DoP that PD reaches for each benchmark along with its execution time and energy consumed.

The results demonstrate the following: (i) in situations where there is resource contention, PD always results in time- and energy-efficient execution when compared to Pthreads, PM, TBB and FDT; (ii) PD is able to handle contention to different resources using a single holistic metric, unlike FDT, which can handle only synchronization and memory bandwidth issues; (iii) in situations where parallelism is not excessive and contention does not occur, PD is unlikely to be beneficial but does not incur overheads; and (iv) the optimum DoP depends on the benchmark and the underlying platform, and can range between one and the maximum number of hardware contexts. We organize the results into "contention" and "no contention" scenarios. We will refer to Figures 4 and 5 in the discussions to follow.

4.1.1 Contention Scenarios

Our benchmarks exhibit five different contention cases:

Contention in external software components

Histogram is a data parallel benchmark that carries out computations on a 1.4GB bitmap file. The benchmark invokes *mmap* which allocates memory in a region in the kernel space, marks the new region as allocated but defers updating the page table entries. Pages are loaded from the disk only when the application accesses them. When a new page is accessed, a page fault occurs and the kernel locks the entire region with a read lock. If multiple threads try to access different pages in parallel, concurrent page faults can occur, potentially creating contention for the lock [8]. The benchmark scales poorly and incurs degradation in both performance and energy for even a modest increase in the number of threads. As shown in Table 3, Histogram does not scale beyond one thread on either machine.

Histogram is an example of a very likely future scenario: a programmer can work hard to avoid contention, but has little or no control over external software components (e.g., libraries or the OS). By controlling the parallelism appropriately, PD is able to reduce the execution time and energy consumption by 20% and 68% on i7-2600, and 32% and 80% on E5-2420, respectively. FDT cannot detect this contention since it happens within the OS, and hence performs as poorly as Pthreads, PM and TBB.

Contention due to memory bandwidth

Stream mainly moves data from one region of the memory to another. Again, there is ample application-level parallelism, however, there is a significant demand for the memory (bus) bandwidth. i7-2600 has a single memory controller with two memory channels, while E5-2420 has two memory controllers with three channels each. Any parallelism beyond the total number of channels available on these platforms will result in contention leading to performance and energy degradation. Both PD and FDT are able to control the parallelism, alleviating the contention, thus preventing increase in memory latencies. This results in reducing the processor idle time, and thus more time- and energy-efficient execution. PD reduces the execution time and energy consumption by 12% and 64% on i7-2600, and 15% and 17% on E5-2420, respectively.

Contention due to disk bandwidth

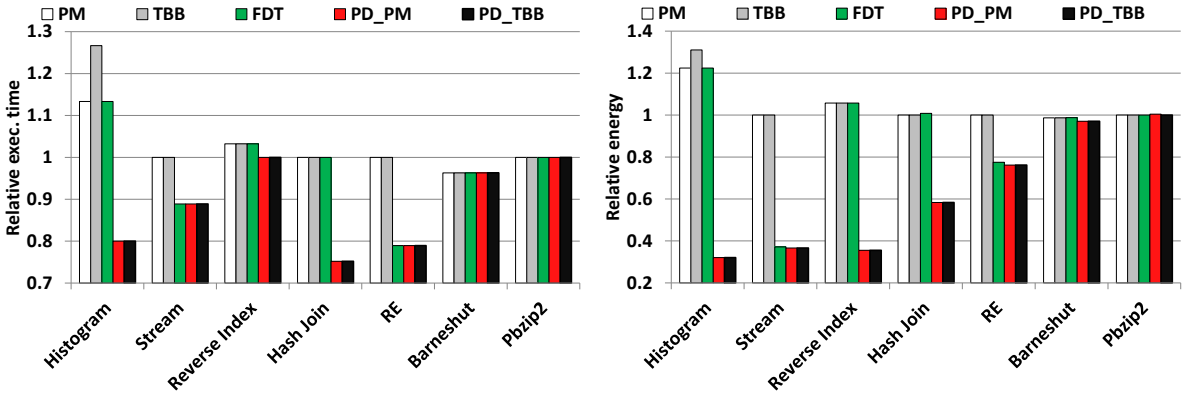
Reverse Index is a benchmark whose parallel execution makes excessive demands for another resource, disk bandwidth. Each parallel computation in Reverse Index does the following: (i) open an HTML file, (ii) parse the file contents to identify links to other pages, (iii) extract the links and update a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

As in the case of Histogram, Pthreads, PM, TBB and FDT have no mechanism in place to detect contention to disk bandwidth. On the other hand, PD's heuristic is able to detect and alleviate this contention by appropriately controlling parallelism. This demonstrates the effectiveness of the *Eff* metric that PD employs. PD is able to reduce the execution time and energy consumption by 0% and 65% on i7-2600 and 8% and 51% on E5-2420 respectively. Reductions in execution time are modest because the performance of this benchmark does not degrade significantly for higher thread counts *wrt* the best operating point. But not using unwanted processors (Table 3) significantly reduced energy consumption.

Contention due to cache capacity

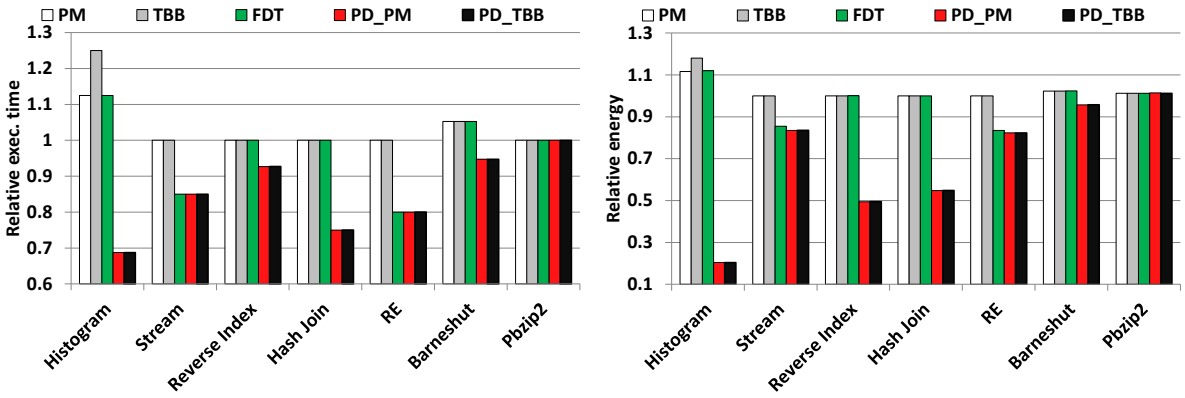
In Hash Join there is potential for contention in a shared cache. The working set size of each thread is over 2MB. i7-2600 has a shared L3 cache capacity of 8 MB, whereas E5-2420 has L3 cache capacity of 15MB per socket and hence contention is likely if the working set size exceeds the respective capacities.

PD controls the parallelism to ensure that the working set does not exceed the available cache capacity, thereby reducing both energy consumption and execution time. FDT cannot detect contention to shared caches. While a more sophisticated technique like Thread Tailor [25] (TT) may detect contention to shared caches and control parallelism accord-



(a) Execution Time (b) Energy

Figure 4: Performance and energy comparison of PM, TBB, FDT and PD on i7-2600 relative to Pthreads.



(a) Execution Time (b) Energy

Figure 5: Performance and energy comparison of PM, TBB, FDT and PD on E5-2420 relative to Pthreads.

ingly, TT cannot detect other types of contention, such as disk bandwidth and in external components, whereas PD's holistic approach easily can. PD reduces the execution time and energy consumption by 25% and 42% on i7-2600, and 25% and 46% on E5-2420, respectively.

Contention due to user-level locks

RE has abundant packet-level parallelism but uses a lock protected shared hash table that each concurrent packet must access. Only one packet (thread) can update the hash table at a given time. As the number of concurrent packets increases, updates to the hash table increase, which increases the contention to the critical section. The benchmark incurs degradation both in execution time and energy beyond a thread count of 5 and 13 on i7-2600 and E5-2420 respectively. Both PD and FDT avoid this degradation by accurately estimating the best number of threads, in turn reducing the execution time and energy up to 22% and 24% on i7-2600, and 20% and 18% on E5-2420, respectively.

4.1.2 No Contention Scenarios

We consider two benchmarks, Barneshut and Pbzip2, to illustrate no contention scenarios. Both have abundant parallelism and few contention concerns in common circumstances. PD neither incurs any degradation nor provides any savings. For PD_TBB, we wrote Pbzip2 such that we avoided the ordering issues described in §2.2.1.

4.2 Multiprogrammed Environment

We now consider the effectiveness of PD in a multiprogrammed environment. We consider two different scenarios. In the first, we launch our benchmark alongside other instances of non-adaptive, resource-intensive programs. In the second, we co-locate pairs of our benchmarks that have different resource constraints. We present data for Pthreads, PM, FDT and PD_PM (TBB and PD_TBB trends are similar to PM and PD_PM, respectively).

The results from these experiments demonstrate: (i) PD *continuously* adapts the parallel execution when the resources allocated to the program dynamically and continuously change due to the resource demands of other co-located programs; (ii) PD adapts the DoP to the instantaneous subscription of resources; and (iii) creating contention to one resource can indirectly alleviate contention to another resource, potentially improving a program's scalability.

4.2.1 Scenario 1

In this scenario, multiple instances of (non-adaptive) *mcf*, a highly cache- and memory-intensive single-threaded application, are executed simultaneously with our benchmarks.

Figure 6 shows PD adapting the DoP for Barneshut in re-

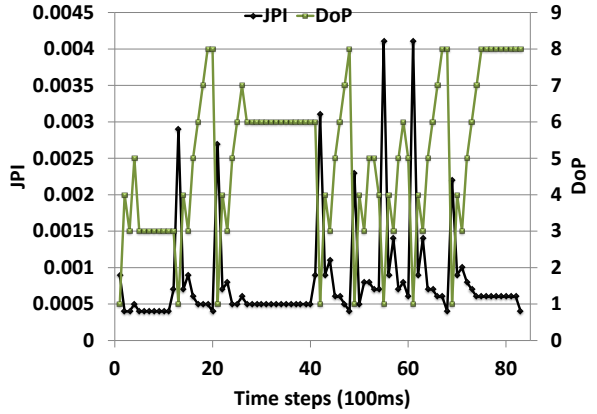


Figure 6: PD's continuous adaptation ability for Barneshut.

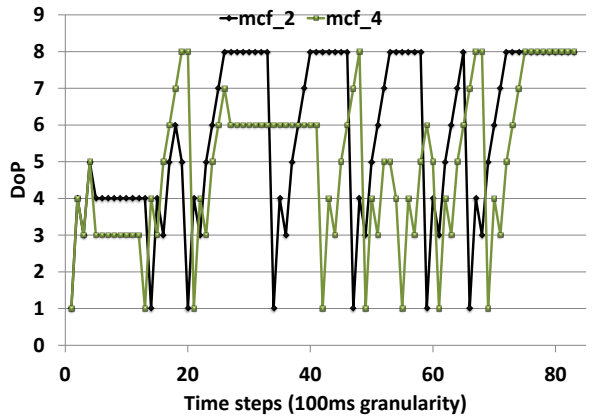


Figure 7: PD's heuristic under different degrees of contention for Barneshut.

sponse to the demands placed by four instances of *mcf*, on i7-2600. The X-axis shows time incremented in 100ms. There are two vertical axes: the primary axis shows the instantaneous JPI of the benchmark and the secondary axis shows the corresponding DoP predicted by PD. From *t=3* to *t=12*, Barneshut executes at a stable DoP of 3. A change in JPI, at *t=12*, indicates change in the resource demands of the co-scheduled *mcf* instances. At *t=13*, the PD runtime reacts by waking up from the sleep mode and restarting the search from sequential execution (§2.2.3) to find the new operating point. At *t=19*, PD arrives at the new optimum (DoP=8) after establishing the search direction and performing the necessary search, and re-enters the sleep mode. PD takes a similar course of action whenever JPI changes, e.g., at *t=41*, *t=68*, etc. Thus, PD continuously alters the parallelism to best suit the variations in the execution environment. Note that in Figure 6, there is no single best operating point for Barneshut unlike in the dedicated environment. Techniques that assume static operating conditions, e.g., FDT, are unable to handle this scenario. They identify the optimum degree of parallelism, typically once at the beginning of the program or at the inception of every user-defined phase, and fix that value for the rest of the program/phase. Techniques like Parcae [40] that propose to continuously adapt, employ search heuristics that can get stuck in local optima.

PD adapts a program's DoP to match the available capacity of resources. Figure 7 shows the DoP for Barneshut when co-scheduled with two (*mcf_2*) and four (*mcf_4*) instances of *mcf* on i7-2600. In general, two instances of *mcf* utilize fewer resources than four instances, and over time PD exposes higher DoP when Barneshut is co-scheduled with two rather than four instances of *mcf*.

Figure 8 shows the summary of execution time and the energy consumed by all the benchmarks normalized to Pthreads, when co-scheduled with four instances of *mcf* on i7-2600. PD outperforms its Pthreads, PM and FDT counterparts for all the benchmarks on average, 19% in time and 21% in energy. FDT measures a program's characteristics, when it begins, to establish the DoP. It can neither account for system-wide conditions nor run-time variations in the conditions. Hence it is inadequate in a multiprogrammed environment. Even for the two instances in which FDT performed as well as PD in the dedicated environment, Stream and RE, FDT fares poorly in the multiprogrammed environment since it makes the same decisions as it made in the dedicated environment.

Stream, in this environment, shows particularly interesting results as compared to the results shown in Figure 4 for the same platform, i7-2600. Unlike in the dedicated environment, in which it scales poorly due to memory bandwidth contention, Stream scaled up to the maximum number of hardware contexts in this environment (Figure 1(b)). This is because the contention created by *mcf* instances to cores and caches slowed down Stream's demand for resources, thus avoiding the memory contention in the first place. Note that FDT is unable to take advantage of this fact since it does not account for the co-located *mcf* instances in its metric. It picks the same DoP as in the dedicated environment and hence incurs significant degradation in both performance and energy consumption. This demonstrates the effectiveness of PD's holistic approach to controlling parallelism as opposed to taking a limited view of the execution environment.

4.2.2 Scenario 2

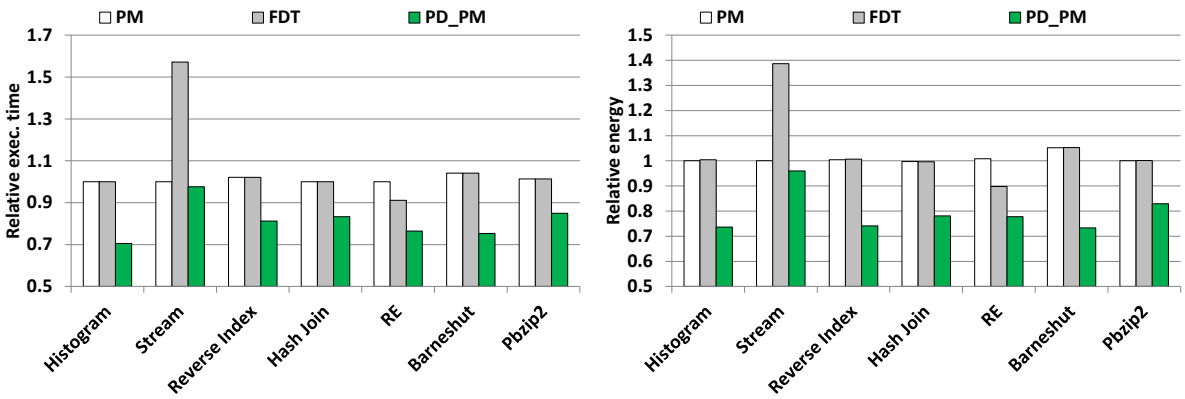
In the next scenario, we co-schedule selected pairs of our benchmarks on E5-2420. We compare the time and energy efficiency of PD with Pthreads and PM. Table 4 presents the results. The first main column shows the pairs of co-scheduled benchmarks. The next three main columns, one for each runtime, show the execution time of each benchmark and the total energy consumed by the pair. The last main column gives the stable DoP that PD reaches for each benchmark.

Pthreads and PM create as many threads as hardware contexts (24) to execute the benchmarks. As Table 3 shows, 24 contexts are already too many for most of them. Depending on their type, simultaneously executing Pthreads/PM benchmarks can make things worse. Note that PD ensures that only as many threads as the hardware contexts are created (Table 4, DoP column). We describe the other results using three types of benchmark pairs.

Contention-prone and contention-free

In this case, we co-schedule a benchmark prone to contention in the dedicated environment, e.g., Reverse Index and Hash Join, with one that is not, e.g., Barneshut. Row #1 and row #2 from Table 4 are examples of this case.

As shown in Table 3, Reverse Index and Hash Join do



(a) Execution Time (b) Energy

Figure 8: Performance and energy comparison of PM, FDT and PD on i7-2600 relative to Pthreads, when scheduled with 4-instances of mcf.

not scale beyond 10 and 11 threads, respectively, when executed in isolation (row #4 and row #2 in Table 3). When their Pthreads and PM versions are co-scheduled with Barneshut, the contention created by Barneshut to cores and private caches further degrade their performance by a factor of 1.7x and 3x, respectively, as compared to the dedicated environment (not shown).

By using only the required number of contexts and returning the rest to the OS, PD versions of Reverse Index and Hash Join reduce contention not only to shared resources (disk and shared cache), but also to cores and private caches, reducing their execution times by 42% and 60% as compared to their Pthreads and PM counterparts. When compared to their dedicated versions (row #4 and row #2 in Table 3), PD limits the degradation in execution times of Reverse Index and Hash Join by 1.07x and 1.5x, respectively, as compared to 1.7x and 3x degradation incurred by Pthreads and PM.

PD is also able to reduce the execution time of Barneshut by more than a factor of 2 when compared to Pthreads and PM. Reducing the number of contexts to execute both Reverse Index and Hash Join allows Barneshut to take advantage of the free resources available to execute its computations in an uninterrupted fashion, thereby improving its performance.

Contention-prone and contention-prone

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

When two instances of Hash Join (Pthreads/PM versions) are co-scheduled (Table 4, row #3), they create more contention to the shared L3 cache, degrading the performance (by a factor of 4 compared to the dedicated environment (not shown)), and energy consumption. PD alleviates this problem by dynamically decreasing the parallelism in each instance close to the best DoP (11). The execution time is almost comparable to the one in the dedicated environment (Table 3, row #2). The total energy reduces by a factor of 4 as compared to its Pthreads and PM counterparts.

The result for Hash Join co-scheduled with Stream (row #4, Table 4), reiterates the inference made in §4.2.1 for Stream. The interference caused by Hash Join slows down Stream's demand for memory bandwidth, thereby reducing the negative impact on performance for all its versions (Pthreads, PM and PD). However, the Pthreads/PM versions of Hash Join slowed down by a factor of 2 due to Stream's interference as compared to dedicated environment (not shown). PD improves the performance and energy of Hash Join by 20% and 27% respectively against its Pthreads/PM counterparts by reducing its DoP to 10 (Table 4, row #4, column DoP-B1).

Contention-free and contention-free

In the final case, both benchmarks were scalable, e.g., Barneshut and Pbzip2 (row #5, Table 4). PD has only marginal impact since neither suffers from contention. It simply divides the resources evenly amongst the benchmarks. For all the versions, the execution times are worse than the dedicated times since both benchmarks can benefit from higher number of resources which they are not provided in the multiprogrammed environment.

4.3 Summary

As above results show, PD provides the best time and energy efficiency in every case. Thus PD frees programmers to focus on composing functionally correct programs rather than optimizing/re-writing them for different execution environments. PD dynamically optimizes the program's parallel execution. Without PD, this is an arduous task.

5. RELATED WORK

Efficient parallel execution using hardware and software techniques is a much studied topic. We summarize the work here using broad categories.

OS scheduling techniques have been proposed to control parallelism in programs [5, 29, 31, 44]. They mainly focus on controlling the resources allocated to a program. PD focuses on controlling the program's parallelism and the resources it occupies. It does not require OS support and can be applied at the user level. Lithe [37], a user level approach, controls the resources alloted to a heterogeneous mix of runtime systems, e.g., TBB alongside OpenMP. PD focuses within a runtime and can be deployed on top of Lithe.

Several recent proposals dynamically vary a program's degree of parallelism [9, 10, 25, 26, 39, 40, 45]. While [9, 10, 26] dynamically control parallelism, they require offline analysis and learning. It is unclear how their runtimes will adapt the program on a different machine with different character-

Table 4: Performance and energy comparison of Pthreads, PM and PD on E5-2420 when benchmarks with different resource constraints are co-scheduled. S(B*): B*'s execution time in seconds; J: Total energy consumed by B1 and B2 in joules; DoP: stable parallelism points for B1 and B2.

#	Benchmarks		Pthreads			PM			PD			DoP	
	B1	B2	S(B1) s	S(B2) s	J	S(B1) s	S(B2) s	J	S(B1) s	S(B2) s	J	B1	B2
1	Reverse Index	Barneshut	71	63	3245	73	68	3502	41	24	1552	9	15
2	Hash Join	Barneshut	97	68	3644	99	69	4764	38	21	1845	11	13
3	Hash Join	Hash Join	101	103	5404	102	104	5603	25	28	1352	13	11
4	Hash Join	Stream	46	19	2544	46	19	2565	37	19	1856	10	14
5	Barneshut	Pbzip2	25	36	1870	22	38	1889	23	38	1872	12	12

istics. Importantly, they cannot continuously and dynamically vary the degree of parallelism. Suleman's approach [45] employs a resource specific parallelism adaption mechanism which is not holistic. Thread Tailor [25] requires support from a static compiler tool chain to identify parallel interactions and is not purely dynamic. DoPE [39] and Parcae [40] are compiler-based approaches to controlling parallelism. Both pause a program's execution when altering its parallelism, potentially underutilizing resources. PD can be applied to existing programming models. It adapts the parallelism without pausing the execution. PD adapts to the dynamic changes and any contention in the system, whereas DoPE does not. Parcae's search for optimum DoP may stop at a local optima whereas PD searches for global optima.

Several papers have considered multiprogrammed environments. They have investigated a range of techniques, both architectural/microarchitectural and system software, to address contention in hardware resources such as on-chip caches [20, 21, 23, 33–36]. These proposals mostly partition the shared resources amongst the different programs, though recent work reduces the demand for a shared resource (and thus the resulting contention) by slowing down the execution speed of a program [12, 19]. PD simply views such scenarios as inefficiencies due to excessive parallelism and handles them without additional support.

Li and Martinez [27, 28] propose runtime heuristics for parallelism and DVFS control to find power/performance efficient execution points for specific soft performance/power targets. PD can achieve both time- and energy-efficient execution whereas their approaches can only achieve either time-efficiency or energy-efficiency, but not both. Unlike their approach, PD requires no soft targets to control parallelism; it begins executing from an unknown operating point to arrive at an optimized operating point.

PD may also be viewed as an example of autonomic computing, in which systems self-monitor and self-regulate to achieve desired objectives, without user-intervention [22]. PD's periodic monitoring and control of execution to improve efficiency is analogous to the MAPE cycle of autonomic systems and observer-controller paradigm of organic computing [43]. Invasive computing is another proposal whose one goal is to optimize parallel execution of programs [46]. It requires pervasive changes to all aspects of a system, whereas PD works with existing hardware and prevalent programming methods.

6. CONCLUSION

Parallel systems often consume more time and energy than necessary to execute parallel programs. This results from inefficient utilization of resources. Program behavior and unpredictable dynamic operating conditions can make performance-tuning complex and non-intuitive. As increasingly more programmers develop parallel programs that are portable to a diverse range of systems, an automated approach is necessary. We presented ParallelismDial, a model that automatically tunes a program's performance to the underlying system. It monitors the system efficiency, regulates the degree of exposed parallelism, and continuously navigates the execution to an optimum point of operation.

The model was applied to two different runtime systems: TBB and Prometheus. Programs were developed using both systems. ParallelismDial was used to tune their execution on two stock multicore workstations, in dedicated and multiprogrammed environments. Their efficiency improved on an average by 15% in execution time and 31% in energy in the dedicated environment and by 19% in time and 21% in energy in the multiprogrammed environment.

7. ACKNOWLEDGMENTS

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.

8. REFERENCES

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

[2] M. D. Allen. *Data-Driven Decomposition of Sequential Programs for Determinate Parallel Execution*. PhD thesis, University of Wisconsin, Madison, 2010.

[3] M. D. Allen, S. Sridharan, and G. S. Sohi. Serialization sets: a dynamic dependence-based parallel execution model. In *PPoPP '09*, pages 85–96, New York, NY, USA, 2009.

[4] A. Anand, C. Muthukrishnan, A. Akella, and R. Ramjee. Redundancy in network traffic: findings and implications. SIGMETRICS '09, pages 37–48, New York, NY, USA, 2009.

[5] 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 *Proceedings of the thirteenth ACM symposium on Operating systems principles*, SOSP '91, pages 95–109, New York, NY, USA, 1991. ACM.

[6] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel java. OOPSLA '09, pages 97–116, New York, NY, USA, 2009.

[7] S. Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *MICRO '05*, 25(6):10–16, 2005.

[8] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An analysis of linux scalability to many

cores. OSDI'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.

[9] M. Curtis-Maury, J. Dzierwa, C. D. Antonopoulos, and D. S. Nikolopoulos. Online power-performance adaptation of multithreaded programs using hardware event-based prediction. ICS '06, pages 157–166, New York, NY, USA, 2006.

[10] 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. PACT '08, pages 250–259, New York, NY, USA, 2008.

[11] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. A. Wood. Implementation techniques for main memory database systems. SIGMOD '84, pages 1–8, New York, NY, USA, 1984.

[12] 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 '10*, pages 335–346, New York, NY, USA, 2010.

[13] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI '98*, pages 212–223, 1998.

[14] S. H. Fuller and E. Lynette I. Millett. *The Future of Computing Performance: Game Over or Next Level?* The National Academies Press, 2011.

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

[16] J. Gilchrist. Parallel data compression with bzip2. In *ICPDCS '04*, pages 559–564, 2004.

[17] F. Glover and M. Laguna. *Tabu Search*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[18] G. Gupta and G. S. Sohi. Dataflow execution of sequential imperative programs on multicore architectures. In *MICRO '11*, pages 59–70, New York, NY, USA, 2011.

[19] 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.

[20] R. Iyer. CQoS: a framework for enabling QoS in shared caches of CMP platforms. In *ICS '04*, pages 257–266, New York, NY, USA, 2004.

[21] 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. *SIGMETRICS Perform. Eval. Rev.*, 35(1):25–36, June 2007.

[22] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, Jan. 2003.

[23] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *PACT '04*, pages 111 – 122, 2004.

[24] M. Kulkarni, M. Burtscher, K. Pingali, and C. Cascaval. Lonestar: A suite of parallel irregular programs. In *ISPASS '09*, pages 65–76, April 2009.

[25] J. Lee, H. Wu, M. Ravichandran, and N. Clark. Thread tailor: dynamically weaving threads together for efficient, adaptive parallel applications. In *ISCA '10*, pages 270–279, New York, NY, USA, 2010.

[26] D. Li, B. de Supinski, M. Schulz, K. Cameron, and D. Nikolopoulos. Hybrid MPI/OpenMP power-aware computing. In *IPDPS '10*, pages 1 –12, April 2010.

[27] J. Li and J. Martinez. Power-performance implications of thread-level parallelism on chip multiprocessors. In *ISPASS '05*, pages 124 –134, March 2005.

[28] J. Li and J. Martinez. Dynamic power-performance adaptation of parallel computation on chip multiprocessors. In *HPCA '06*, pages 77 – 87, Feb. 2006.

[29] R. Liu, K. Klues, S. Bird, S. Hofmeyr, K. Asanović, and J. Kubiatowicz. Tessellation: space-time partitioning in a manycore client os. HotPar'09, pages 10–10, Berkeley, CA, USA, 2009.

[30] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *TCCA Newsletter*, pages 19–25, Dec. 1995.

[31] 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.

[32] 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.

[33] O. Mutlu and T. Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *MICRO '07*, pages 146 –160, 2007.

[34] O. Mutlu and T. Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems. In *ISCA '08*, pages 63–74, 2008.

[35] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. Fair queuing memory systems. In *MICRO '06*, pages 208–222, 2006.

[36] K. J. Nesbit, J. Laudon, and J. E. Smith. Virtual private caches. In *ISCA '07*, pages 57–68, New York, NY, USA, 2007.

[37] H. Pan, B. Hindman, and K. Asanović. Composing parallel software efficiently with lithe. In *PLDI '10*, pages 376–387, New York, NY, USA, 2010.

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

[39] A. Raman, H. Kim, T. Oh, J. W. Lee, and D. I. August. Parallelism orchestration using DoPE: the degree of parallelism executive. In *PLDI '11*, pages 26–37, New York, NY, USA, 2011. ACM.

[40] A. Raman, A. Zaks, J. W. Lee, and D. I. August. Parcae: a system for flexible parallel execution. In *PLDI '12*, pages 133–144, New York, NY, USA, 2012.

[41] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. HPCA '07, pages 13–24, Washington, DC, USA, 2007.

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

[43] U. Richter, M. Mnif, J. Branke, C. MÃijller-Schloer, and H. Schmeck. Towards a generic observer/controller architecture for organic computing. In C. Hochberger and R. Liskowsky, editors, *GI Jahrestagung (1)*, volume 93 of *LNI*, pages 112–119. GI, 2006.

[44] H. Sasaki, T. Tanimoto, K. Inoue, and H. Nakamura. Scalability-based manycore partitioning. In *PACT '12*, pages 107–116, New York, NY, USA, 2012.

[45] 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 '08*, pages 277–286, 2008.

[46] J. Teich, J. Henkel, A. Herkersdorf, D. Schmitt-Landsiedel, W. Schroder-Preikschat, and G. Snelting. Invasive computing: An overview. In *Multiprocessor System-on-Chip*, pages 241–268. 2011.