Parallelism Orchestration using DoPE: the Degree of Parallelism Executive

Arun Raman Hanjun Kim Taewook Oh Jae W. Lee[†] David I. August

Princeton University
Princeton, NJ
{rarun, hanjunk, twoh, august}@princeton.edu

† Parakinetics Inc.
Princeton, NJ
leejw@parakinetics.com

Abstract

In writing parallel programs, programmers expose parallelism and optimize it to meet a particular performance goal on a single platform under an assumed set of workload characteristics. In the field, changing workload characteristics, new parallel platforms, and deployments with different performance goals make the programmer's development-time choices suboptimal. To address this problem, this paper presents the Degree of Parallelism Executive (DoPE), an API and run-time system that separates the concern of exposing parallelism from that of optimizing it. Using the DoPE API, the application developer expresses parallelism options. During program execution, DoPE's run-time system uses this information to dynamically optimize the parallelism options in response to the facts on the ground. We easily port several emerging parallel applications to DoPE's API and demonstrate the DoPE run-time system's effectiveness in dynamically optimizing the parallelism for a variety of performance goals.

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, Languages, Performance

Keywords parallelization, parallelism, dynamic, run-time, scheduling, task, loop-level, nested, loop nest, pipeline, parametric, optimization

1. Introduction

As multicore processors become ubiquitous, application developers and compilers must extract thread level parallelism (TLP) in order to exploit the execution resources afforded by the hardware. Parallelism of multiple types may exist in an application, such as task parallelism, data parallelism, and pipeline parallelism. Much progress has been made in methodologies and systems to extract parallelism, even from seemingly sequential code [6, 7, 24, 25, 34, 40]. Tools such as POSIX threads (Pthreads) [33], Intel Threading Building Blocks (TBB) [26], Cilk [5], OpenMP [22], and Galois [14] allow application developers to express TLP.

Many applications have parallelism in multiple loops in a loop nest. Each loop may be parallelized by exploiting different types of parallelism to varying extents by allocating a different number of parallel resources (hardware threads) to each loop. The type and extent of each loop parallelization is called the *degree of parallelism* (DoP). Simultaneously parallelizing multiple loops can provide both scalability and latency-throughput benefits.

Unfortunately, determining the right degree of parallelism for even a single loop, let alone multiple loops in a nest, is a complicated task. Application developers or parallel run-times typically fix the degree of parallelism of each loop statically at development-time or run-time. This is often suboptimal in the face of several *run-time* sources of performance variability that could potentially result in leaving hardware resources idle or over-subscribed. The application workload characteristics may vary as in the case of web services such as search and video. The parallel platform characteristics (number of cores, memory bandwidth, etc.) may vary [18, 30]. Furthermore, the performance goals may not be fixed and could be some complex time-varying functions of energy, throughput, etc. Together, these three sources of variability—workload characteristics, platform characteristics, and performance goals—are referred to as the execution environment of an application.

To solve the above problem, the application developer could statically produce multiple versions of code and dynamically select a version that best fits each execution environment. Unfortunately, the number of scenarios resulting from a myriad of platforms, workloads, and performance goals is so large as to preclude the possibility of incorporating all of them into statically-compiled codes. To address the limitations of the static approach, run-time systems have been proposed to map parallel applications to their execution environments. Low-level substrates enable parallel library composition but do not leverage application-specific run-time information [15, 23]. Prior work that performs adaptation by monitoring application features has typically focused on a specific type of parallelism (such as pipeline parallelism or task parallelism) and fixed dynamic adaptation mechanisms that are tightly coupled to the target parallelism type [4, 5, 26, 29, 30, 35, 38]. More importantly, in all known prior work, the adaptation mechanisms are restricted to a single loop in a loop nest.

This paper proposes DoPE, a novel API and run-time system that enables the separation of the concern of developing a functionally correct parallel program from the concern of optimizing the parallelism for different execution environments. The separation of concerns enables a mechanism developer to specify *mechanisms* that encode the logic to adapt an application's parallelism

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. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PLDI'11, June 4–8, 2011, San Jose, California, USA.

Copyright © 2011 ACM 978-1-4503-0663-8/11/06. . . $10.00

Library	Multiple Performance Goals	Parallelism in Loop Nest	Multiple Parallelism Types	Application Feature Monitoring	Multiple Optimization Mechanisms
Pthreads [33]	×	✓	✓	×	×
Intel TBB [26]	×	×	✓	×	×
FDP [29]	×	×	×	✓	×
DoPE [This paper]	✓	✓	✓	✓	✓

Table 1. Comparison of various software-only parallelization libraries for general-purpose applications

configuration to meet the performance goals that are set by the *administrator*. Table 1 highlights DoPE's advantages over other parallelism management libraries. With DoPE, the application developer can expose all the parallelism in an application and express it in a unified manner just once. Then, a run-time system adapts the application's parallelism configuration by monitoring key application features and responding to changes in the application's execution environment. Specifically, the run-time system automatically and continuously determines:

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

We ported several emerging parallel applications to use the DoPE interface. Different performance goals included response time minimization, throughput maximization, and throughput maximization under power constraint. DoPE automatically adapted the application to meet the goals, without necessitating a change in the application code by the developer. To adapt the parallelism, it used new mechanisms proposed in this paper and also mechanisms proposed in prior work [29, 38], demonstrating the robustness of DoPE's interface and the ability for a mechanism developer to implement better mechanisms in the future in a non-disruptive way.

On a 24-core Intel Xeon machine, DoPE improved the response time characteristics of four web service type applications to dominate the characteristics of the best static parallelizations. The throughputs of two batch-oriented applications were improved by 136% (geomean) over their original implementations. For one application (an image search engine), three different goals—involving response time, throughput, and power—were independently specified. DoPE automatically determined a stable and well performing parallelism configuration operating point in all cases.

The primary contributions of this work are:

- An API that separates the concern of correct specification of an application's parallelism from the concern of optimization of the application's execution in a variety of environments
- A smart run-time system that enables the interface and monitors application execution to dynamically adapt the parallelism in program loop nests by means of suitable mechanisms in order to meet specified performance goals

The rest of this paper is organized as follows. The need for dynamic adaptation and separation of concerns is first motivated, followed by a description of DoPE's interfaces for the application developer, mechanism developer, and administrator. Various mechanisms that were implemented are described, followed by an evaluation of the DoPE system and a discussion of related work.

2. Motivation

A parallel application's execution environment consists of the application workload characteristics, platform characteristics, and performance goals. Variability in any of these parameters can necessitate dynamic adaptation of parallelism in order to meet the specified performance goal. The following video transcoding example concretely demonstrates the impact of workload characteristic variability.

Example: Video Transcoding Video sharing websites such as YouTube, Google Video, and Dailymotion transcode user submitted videos on their servers. Figure 1 shows the parallelism in video transcoding using `x264`, an implementation of the popular H.264 standard [39]. Each video may be transcoded in parallel with others. Furthermore, a single video may itself be transcoded in parallel by exploiting parallelism across the frames in the video in a pipelined fashion. $<DoP_{outer}, DoP_{inner}>$ represents the type of parallelism of, and number of threads assigned to, the outer (inter-video) and inner (intra-video) loops in the loop nest. Examples of types of parallelism are DOALL and pipeline (PIPE) [1]. Statically fixing the DoP assigned to each loop may not be optimal for a given performance goal in all execution environments. To demonstrate this, we measured throughput and execution time on a 24-core machine with Intel Xeon X7460 processors. User requests were simulated by a task queueing thread with arrivals distributed according to a Poisson distribution. The average system load factor is defined as the average arrival rate of tasks (videos to be transcoded) divided by the maximum throughput sustainable by the system.

Figure 2(a) shows that exploiting intra-video parallelism provides much lower per-video transcoding execution time than when only the outer loop is parallelized. T_{exec} is improved up to a maximum of $6.3\times$ on the evaluation platform. This speedup is achieved when 8 threads are used to transcode each video. Figure 2(b), however, shows the dependency of throughput on the application load. At heavy load (load factor 0.9 and above), turning on intra-video parallelism actually degrades throughput. This is due to the inefficiency of parallel execution (a speedup of only about $6\times$ on 8 threads at load factor 1.0) caused by overheads such as thread creation, communication, and synchronization.

This experiment shows that the usual static choices of parallelism configuration are not ideal across all load factors for *both* execution time and throughput. In other words, there is a tradeoff between the two. This tradeoff impacts end user response time which is the primary performance metric of service oriented applications. Equation 1 is helpful to understand the impact of the execution time/throughput tradeoff on response time. The time to transcode a video is the execution time, T_{exec}. The number of videos transcoded per second is the throughput of the system,

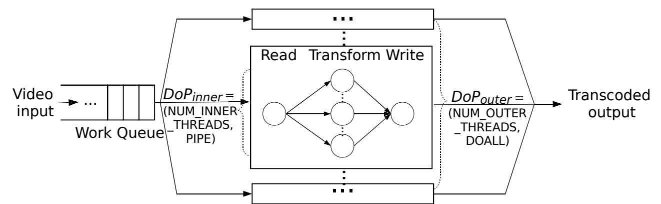


Figure 1. Two-level loop nest in video transcoding: Across videos submitted for transcoding and across frames within each video

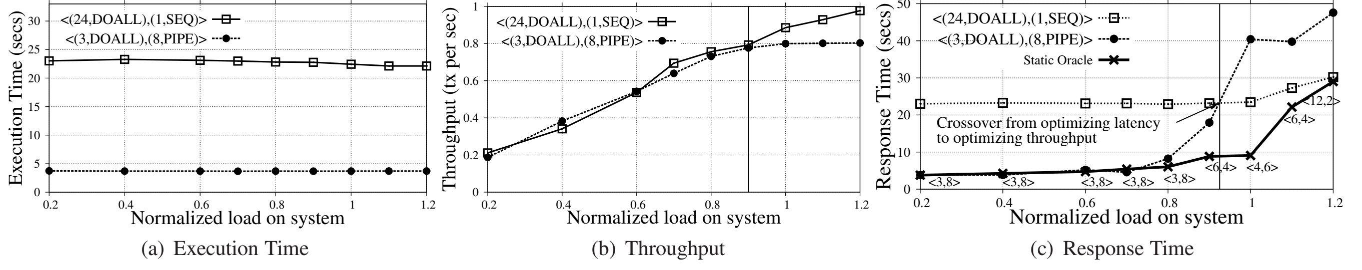


Figure 2. Variation of (a) execution time and (b) throughput with load factor and parallelism configuration in a video transcoding application on a 24-core Intel Xeon machine; (c) impact of throughput and execution time on end user response time; an oracle achieves the best response time characteristic by continuously varying DoP with load (ideal parallelism configuration for each load factor is shown)

Throughput. The number of outstanding requests in the system's work queue is the instantaneous load on the system, $q(t)$.

$$T_{response}(t) = T_{exec}(DoP) + \frac{q(t)}{Throughput(DoP)} \quad (1)$$

The response time of a user request, $T_{response}$, is the time interval from the instant the video was submitted for transcoding (at time t) to the instant the transcoded video is output. $T_{response}$ has two components: wait time in the work queue until the request reaches the head of the queue, and execution time, T_{exec}. At light to moderate load, the average arrival rate is lower than the system throughput. Consequently, the wait time will tend to zero, and $T_{response}$ will be determined by T_{exec}. Assuming reasonably efficient intra-video parallelism, increasing the DoP extent of the inner loop reduces T_{exec} and in turn $T_{response}$. In other words, in this region of operation, $<DoP_{outer}, DoP_{inner}>$ must be optimized for execution time (DoP$_{inner}$ $= (8, PIPE)$). At heavy load, $T_{response}$ is dominated by the wait time in the work queue which is determined by the system throughput. In this region of operation, DoP$_{inner}$ must be set to a value that optimizes throughput (DoP$_{inner}$ $= (1, SEQ)$). Figure 2(c) presents experimental validation of the described response time characteristic. The same figure also shows that a mere "turn inner parallelism on/off" approach is suboptimal; an oracle that can predict load and change DoP continuously achieves significantly better response time.

In addition to workload characteristics, platform characteristics including number of hardware contexts, memory space, etc. may vary. Further, the same system (application+platform) may be called upon to maximize system utility with a variety of performance goals involving energy, throughput, etc. If the application developer were tasked with matching application code to the variety of dynamic execution environments that might arise, there would be a combinatorial explosion in the number of versions of application code. Each version would remain *ad hoc* as the application is deployed on newer platforms and is used in the context of different performance goals.

Separation of Concerns To address the explosion of developer effort and code versions, the task of expressing application parallelism must be separated from the task of adapting that parallelism to specific execution environments. Further, an administrator must be able to specify the current performance goal, and the application must adapt itself to meet the goal. Such a separation of concerns would enable:

- application developers to focus on functional correctness of the parallel application
- administrators to specify arbitrary performance goals involving performance, power, etc.
- mechanism developers to implement parallelism adaptation mechanisms that meet the specified performance goals

Table 1 evaluates existing choices for enabling such a separation of concerns. Using Pthreads, a developer specifies a concrete, unchanging parallelism configuration, or codes an ad hoc adaptation mechanism for every new execution environment. Intel TBB [26] and similar libraries [5, 22] support task parallelism for independent tasks and their schedulers optimize only for throughput. Feedback Directed Pipelining (FDP) implements an adaptation mechanism tied to throughput maximization for a single loop in the application [29]. In summary, these libraries support only a single performance goal, and closely couple the goal with a specific mechanism to adapt parallelism in order to meet the goal.

DoPE enables an application developer to express common paradigms of nested parallelism in a unified fashion. DoPE enables an administrator to specify different performance goals for the same application. DoPE enables a mechanism developer to implement multiple mechanisms that reconfigure application parallelism to meet a specified performance goal. The application developer needs to write the application just once, and the application executes robustly across multiple scenarios of use, platforms, and workloads.

3. DoPE for the Application Developer

3.1 DoPE API

DoPE presents a task-oriented interface to the application developer. A task consists of a template function that abstracts the control for creating dynamic instances of each task, function objects (functors) that encapsulate the task's functionality and expose application level information, and a descriptor that describes the parallelism structure of the task. Figure 3 defines the `Task` type and the types from which it is composed.

```
1 Task = {control: TaskExecutor, function: Functor,
2           load: LoadCB, desc: TaskDescriptor,
3           init: InitCB, fini: FiniCB}
4 TaskDescriptor = {type: TaskType, pd: ParDescriptor[]}
5 TaskType = SEQ | PAR
6 ParDescriptor = {tasks: Task[]}
7 TaskStatus = EXECUTING | SUSPENDED | FINISHED
```

Figure 3. DoPE type definitions

`TaskExecutor` DoPE provides the control flow abstraction shown in Figure 4(a). Loop exit is determined by `status` (line 7 in Figure 4(a)). The abstraction is templated on the `Functor` type that encapsulates a task's functionality.

Method	Description
`TaskStatus Task::begin()`	Signal DoPE that the CPU intensive part of the task has begun; DoPE returns task status
`TaskStatus Task::end()`	Signal DoPE that the CPU intensive part of the task has ended; DoPE returns task status
`TaskStatus Task::wait()`	Wait until child tasks complete; DoPE returns status of master child task
`DoPE* DoPE::create(ParDescriptor* pd)`	Launch parallel application described by specified parallelism descriptor under the DoPE run-time system
`void DoPE::destroy(DoPE* dope)`	Finalize and destroy the DoPE run-time system; wait for registered tasks to end

Table 2. DoPE API

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

(a) Control flow abstraction

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

(b) Functor for task functionality

Figure 4. Separation of task's control and functionality in DoPE

`Functor` The developer must implement a functor that encapsulates the desired functionality of a task. The functor binds the local variables of the original method containing the parallelized loop as member fields (line 2 in Figure 4(b)). At run-time, a task could be either executing, suspended, or finished. The functor must return the status of the task after each instance of the task (line 8 in Figure 4(b)). In particular, when a loop exit branch is to be taken, the functor must return `FINISHED`; otherwise, the functor must return `EXECUTING`. Combined with the control flow abstraction in Figure 4(a), the control flow structure of the original loop is duplicated. The functor can also return `SUSPENDED`—its discussion is deferred until Section 3.2.

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

`InitCB` and `FiniCB` To restart parallel execution from a globally consistent program state after DoPE reconfigures parallelism, DoPE requires the programmer to implement the `InitCB` (`FiniCB`) functor that is invoked exactly once before (after) the task is executed.

`TaskDescriptor` A task can be sequential (`SEQ`) or parallel (`PAR`). A parallel task's functionality can be executed by one or more threads. In other words, the `Functor()` method (lines 6–9 in Figure 4(b)) can be invoked concurrently by multiple threads. To enable description of nested parallelism, a task can specify one or more parallelism descriptors (`ParDescriptor`). Specifying more than one descriptor exposes a choice to DoPE which at run-time chooses the optimal parallelism configuration (described by the corresponding `ParDescriptor`).

`ParDescriptor` A parallelism descriptor is defined recursively in terms of `Tasks`. A `ParDescriptor` is an array of one or more tasks that execute in parallel and potentially interact with each other (line 6 in Figure 3).

```
1 void Transcode(){
2   Q∗ inq, outq;
3   Video∗ input, ∗output;
4   while(true){
5     ∗input = inq→deque();
6     output = transcode(input);
7     outq→enqueue(∗output);
8   }
9 }
```

Figure 5. Outer loop in x264 video transcoding

Putting it all together Figure 5 shows the outer loop code in x264 video transcoding. Figure 6 shows the transformation of the loop by instantiation of the DoPE types discussed above. In Figure 6(a), duplicated code from the original loop in Figure 5 is shown in bold. Referring to Figure 1, the outer loop task can itself be executed

```
1  class TranscodeFunctor{
2   //Capture local variables
3    Queue∗& inq;
4    Queue∗& outq;
5    ... //Constructor
6    TaskStatus operator()(){
7      Video* input, *output;
8      *input = inq→deque();
9      output = transcode(input);
10     outq→enqueue(*output);
11     return EXECUTING;
12   }
13 };
```

(a) Functionality

```
1  class TranscodeLoadCB{
2    //Capture local variables
3    Queue∗& inq;
4    Queue∗& outq;
5    ... //Constructor
6    float operator()(){
7      //Return occupancy
8      return inq→size();
9    }
10 };
11
12
13
```

(b) Workload

```
1  TaskDescriptor
2   ∗readTD(SEQ, NULL),
3   ∗transformTD(PAR, NULL),
4   ∗writeTD(SEQ, NULL);
5   ...//Create tasks
6     //using descriptors
7  ParDescriptor
8   ∗innerPD({readTask,
9                transformTask,
10               writeTask});
11 TaskDescriptor
12  ∗outerTD(PAR, {innerPD});
13
```

(c) Descriptor

```
1  void Transcode(){
2    Queue∗ inq, ∗outq;
3    Task∗ task
4     (TranscodeFunctor(inq, outq),
5      TranscodeLoadCB(inq, outq),
6      outerTD);
7    //TaskExecutor<OuterLoopFunctor>
8    //is used automatically by DoPE
9  }
10
11
12
13
```

(d) Task

Figure 6. Task definition using DoPE

in a pipeline parallel fashion. Figure 6(c) shows the definition of the outer loop task descriptor in terms of the inner loop parallelism descriptor. Note that the process of defining the functors is mechanical—it can be simplified with compiler support.

3.2 Using the API: A Video Transcoding Example

A developer uses the types in Figure 3 and associated methods in Table 2 to enhance a parallel application using DoPE. Figure 7 describes the port of a Pthreads based parallelization (column 1) of the video transcoding example from before to the DoPE API (column 2). Code that is common between the Pthreads version and the DoPE version is shown in bold.

Step 1: Parallelism Description In the Pthreads parallelization, lines 4–7 create `NUM_OUTER` threads that execute the `Transcode` method. In the `Transcode` method, a thread dequeues work items (videos) from the work queue (line 14), transcodes them (lines 15–25), and enqueues the transcoded items to the output queue (line 26). Each video transcoding can itself be done in parallel in a pipelined fashion. For this, the `Transcode` method spawns `NUM_INNER` threads to execute the pipeline. One thread each executes `Read` and `Write`, and one or more threads execute `Transform`. A common practice is to set both `NUM_OUTER` and `NUM_INNER` statically based on profile information [21]. Section 2 already presented the shortcomings of this approach—to operate optimally, an application must dynamically change its parallelism configuration as the execution environment changes.

In the DoPE parallelization, the application's parallelism is described in a modular and bottom-up fashion. Line 4 gets the task definition of the outer loop by invoking `Transcode_getTask`. To encode nested parallelism, the `Transcode_getTask` method specifies that `Transcode` can be executed in parallel using the parallelism descriptor `pd` (lines 12–17 in `Transcode_getTask`). Line 5 in `transcodeVideos` creates a parallelism descriptor for the outer loop.

Step 2: Parallelism Registration Line 6 in `transcodeVideos` registers the parallelism descriptor for execution by DoPE by invoking `DoPE::create`. Line 7 waits for the parallel phase of the application to finish before freeing up execution resources by invoking `DoPE::destroy`.

Step 3: Application Monitoring Each task marks the begin and end of its CPU intensive section by invoking `Task::begin` and `Task::end`, respectively. DoPE records application features such as task execution time in between invocations of these methods. To monitor per-task workload, the developer implements `LoadCB` for each task to indicate the current workload on the task. The callback returns the current occupancy of the work queue in the case of the outer task (line 26), and the input queue occupancies in the case of `Transform` (line 62) and `Write` (line 75). The callbacks are registered during task creation time.

Step 4: Task Execution Control If a task returns `EXECUTING`, DoPE continues the execution of the loop. If a task returns `FINISHED`, DoPE waits for other tasks that are at the same level in the loop nest to also return `FINISHED`. A task can explicitly wait on its children by invoking `Task::wait`. Exactly one task in each parallelized loop is assigned the role of the *master task* (the first task in the array of tasks registered in the parallelism descriptor). In the running example, the task corresponding to `Transcode` is the master task for the outer loop and the task corresponding to `Read` is the master task for the inner loop. Invoking `Task::wait` on `task` (line 17) returns the status of the master child task.

Step 5: Task Yielding for Reconfiguration By default, DoPE returns `EXECUTING` when either `Task::begin` or `Task::end`

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

(1) Run-time initialization

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

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

(2) Transcoding of an individual video clip

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

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

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

Figure 7. Comparison of parallelization using POSIX threads and DoPE—continued on next page

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

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

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

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

Figure 7. Comparison of parallelization using POSIX threads and DoPE

is invoked. When DoPE decides to reconfigure, it returns `SUSPENDED`. The application should check this condition (lines 35–36 in `ReadFunctor`), and then enter a globally consistent state prior to reconfiguration. The `FiniCB` callbacks are used for this purpose. In this particular example, `Read` notifies `Transform` (via the `ReadFiniCB` callback) which in turn notifies `Write` (via the `TransformFiniCB` callback). The notifications are by means of enqueuing a sentinel `NULL` token to the in-queue of the next task. Note by comparing the Pthreads (lines 36 and 54) and DoPE versions (lines 41 and 57) that the developer was able to reuse the thread termination mechanism from the Pthreads parallelization to implement the `FiniCB`s. `InitCB` callbacks are used symmetrically for ensuring consistency before the parallel region is reentered after reconfiguration. The video transcoding example does not require any `InitCB` callbacks to be defined.

3.3 Summary

In the Pthreads based parallelization, the developer is forced to implement a concrete, unchanging configuration of parallelism. In the DoPE based parallelization, the developer declares the parallelism structure of the program, while deliberately not specifying the exact parallelism configuration. This underspecification allows DoPE to adapt parallelism to determine the optimal configuration at runtime. While the API has been described for use by a developer, a parallelizing compiler could also target the API in the same way as it targets Pthreads.

4. DoPE for the Administrator

The administrator specifies a performance goal that includes an objective and a set of resource constraints under which the objective must be met. Examples of performance goals are "minimize response time" and "maximize throughput under a peak power con-

straint". The administrator may also invent more complex performance goals such as minimizing the energy-delay product [9], or minimizing electricity bills while meeting minimum performance requirements [19]. DoPE aims to meet the performance goals by dynamically adapting the configuration of program parallelism.

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

5. DoPE for the Mechanism Developer

Figure 8 shows the DoPE system architecture. The DoPE-*Executive* is responsible for directing the interactions between the various system components. DoPE maintains a *Thread Pool* with as many threads as constrained by the performance goals. DoPE uses *mechanisms* to adapt parallelism in order to meet the specified goals.

There are two main information flows when an application is launched. First, the application registers its parallelism descriptors (expressed by the application developer). Second, the administrator specifies the performance goals. The DoPE run-time system then starts application execution. During execution, it monitors and adapts the parallelism configuration to meet those goals.

Referring to Figure 8, DoPE monitors both the application (A) and platform (B). Section 3.2 already described the methods that enable DoPE to monitor application features such as task execution time and task load. DoPE uses per thread timers (updated using calls to `clock_gettime`) to obtain task execution time. To enable DoPE to monitor platform features such as number of hardware contexts, power, temperature, etc., the mechanism developer registers a feature with an associated callback that DoPE can invoke to get a current value of the feature. Figure 9 shows the registration API. For example, the developer could register "`SystemPower`" with a callback that queries the power distribution unit to obtain the current system power draw [2].

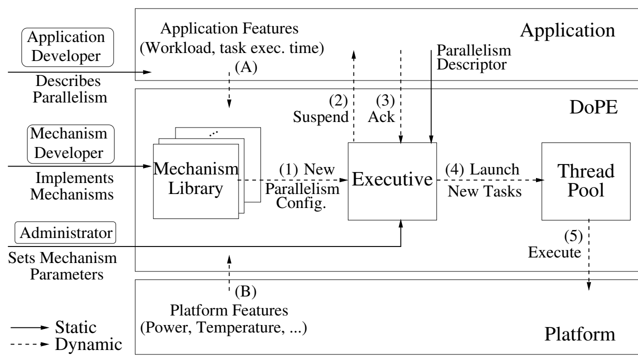


Figure 8. Interactions of three agents around DoPE. The application developer describes parallelism using DoPE just once. The mechanism developer implements mechanisms to transform the parallelism configuration. The administrator sets the constraint parameter values of the mechanism. (A) and (B) represent continuous monitoring of application and platform features. (1)–(5) denote the sequence of events that occurs when parallelism reconfiguration is triggered.

```
1 //Application features
2 double DoPE::getExecTime(Task∗ task);
3 double DoPE::getLoad(Task∗ task);
4 //Platform features
5 void DoPE::registerCB(string feature, Functor∗ getValueOfFeatureCB);
6 void∗ DoPE::getValue(string feature);
```

Figure 9. DoPE Mechanism Developer API

The primary role of the mechanism developer is to implement the logic to adapt a parallelism configuration to meet a performance goal by using the information obtained via monitoring. For this, DoPE exposes the query API shown in Figure 9 to the mechanism developer. Figure 10 shows a mechanism that can enable a "Maximize Throughput with N threads" performance goal. Every mechanism must implement the `reconfigureParallelism` method. The method's arguments are the descriptor of the current parallelism configuration and the maximum number of threads that can be used to construct a new configuration. The new configuration is returned to the caller (DoPE-Executive).

```
 1 ParDescriptor∗ Mechanism::reconfigureParallelism
 2                          (ParDescriptor∗ pd, int nthreads){
 3   float total_time = 0.0;
 4   // 1. Compute total time
 5   foreach (Task∗ task: pd→tasks) {
 6     total_time += DoPE::getExecTime(task);
 7   }
 8   // 2. Assign DoP proportional to execution time;
 9   // recurse if needed
10   foreach (Task∗ task: pd→tasks) {
11     task→dop = nthreads ∗ (DoPE::getExecTime(task)/total_time);
12     ParDescriptor∗ innerPD = task→pd;
13     if (innerPD) {
14       task→pd = reconfigureParallelism(innerPD, task→dop);
15     }
16   }
17   ... // 3. Construct new configuration − Omitted
18   return newPD;
19 }
```

Figure 10. Mechanism to maximize throughput—Assigns DoP to each task proportional to task's execution time

The intuition encoded by the mechanism in Figure 10 is that tasks that take longer to execute should be assigned more resources. In step 1, the mechanism computes total execution time (lines 4–7) so that each task's execution time can be normalized. In step 2, the mechanism assigns a DoP that is proportional to the normalized execution time of each task (line 11). `reconfigureParallelism` is recursively invoked to assign DoPs to the inner loops in the loop nest. For each loop, a new configuration is constructed with the new task descriptors and returned to the parent descriptor. For brevity, this last step is omitted.

6. DoPE Operation Walk-through

Once a mechanism is selected, DoPE uses it to reconfigure parallelism. The Executive triggers a parallelism reconfiguration in response to changes in the execution environment such as increase in workload. When reconfiguration is triggered, the following sequence of events occurs (refer to Figure 8):

1. The Mechanism determines the optimal parallelism configuration, which it conveys to the Executive.
2. The Executive returns `SUSPENDED` to invocations of `Task::begin` and `Task::end` in order to convey to the application DoPE's intent of reconfiguration.

3. In response, the application and DoPE steer execution into a suspended state by invoking the `FiniCB` callbacks of all the tasks.
4. The Executive then schedules a new set of tasks for execution by the Thread Pool—the task set is defined by the new parallelism configuration specified by the Mechanism.
5. The Thread Pool executes the new tasks on the Platform.

7. Performance Goals and Mechanisms Tested

One advantage of the separation of concerns enabled by the DoPE interface is that a mechanism developer can implement new mechanisms and add them to the library in order to better support existing performance goals or to enable new ones, without changing the application code. The separation of concerns also enables reuse of mechanisms across many parallel applications. This separation allowed us to implement and test three different goals of system use, with multiple mechanisms to achieve them. *For each performance goal, there is a best mechanism that DoPE uses by default. In other words, a human need not select a particular mechanism to use from among many.* Multiple mechanisms are described for each performance goal in order to demonstrate the power of DoPE's API. Table 3 lists the implemented mechanisms and the number of lines of code for implementing each. Two of the mechanisms are proposed in prior work for a fixed goal-mechanism combination.

Mechanism					
WQT-H	WQ-Linear	TBF	FDP [29]	SEDA [38]	TPC
28	9	89	94	30	154

Table 3. Lines of code to implement tested mechanisms

7.1 Goal: "Min Response time with N threads"

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

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

Mechanism: Work Queue Linear (WQ-Linear) A more graceful degradation of response time with increasing load may be achieved by varying the DoP extent continuously in the range $[M_{min}, M_{max}]$, rather than just toggling between two DoP extent values. WQ-Linear assigns a DoP extent according to Equation 2.

$$DoP_{extent} = max(M_{min}, M_{max} - k \times WQo) \quad (2)$$

WQo is the instantaneous work queue occupancy. k is the rate of DoP extent reduction ($k > 0$). k is set according to Equation 3.

$$k = \frac{M_{max} - M_{min}}{Q_{max}} \quad (3)$$

Q_{max} in Equation 3 is derived from the maximum response time degradation acceptable to the end user and is set by the system administrator taking into account the service level agreement (SLA), if any. The degradation is with respect to the minimum response time achievable by the system at a load factor of 1.0. The threshold value T in the WQT-H mechanism is obtained similarly by a back-calculation from the acceptable response time degradation. A variant of WQ-Linear could be a mechanism that incorporates the hysteresis component of WQT-H into WQ-Linear.

7.2 Goal: "Max Throughput with N threads"

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

Mechanism: Throughput Balance with Fusion (TBF) TBF records a moving average of the throughput (inverse of execution time) of each task. When reconfiguration is triggered, TBF assigns each task a DoP extent that is inversely proportional to the average throughput of the task. If the imbalance in the throughputs of different tasks is greater than a threshold (set to 0.5), TBF fuses the parallel tasks to create a bigger parallel task. The rationale for fusion is that if a parallel loop execution is heavily unbalanced, then it might be better to avoid the inefficiency of pipeline parallelism. Our current implementation requires the application developer to implement and register the desired fused task via the `TaskDescriptor` API that allows expression of choice of `ParDescriptor`s (see lines 4 and 6 in Figure 3). Creating fused tasks is easy and systematic: Unidirectional inter-task communication should be changed to method argument communication via the stack. Some of the applications that we studied already had pre-existing code for fusing tasks in the original Pthreads-parallelized source code. These were originally included to improve sequential execution in case of cache locality issues. Once registered, DoPE will automatically spawn the fused task if task fusion is triggered by the mechanism. Other mechanisms for throughput maximization that we tested are:

Mechanism: Feedback Directed Pipelining (FDP) FDP uses task execution times to inform a hill climbing algorithm to identify parallelism configurations with better throughput [29].

Mechanism: Stage Event-Driven Architecture (SEDA) SEDA assigns a DoP extent proportional to load on a task [38].

7.3 Goal: "Max Throughput with N threads, P Watts"

Mechanism: Throughput Power Controller (TPC) The administrator might want to maximize application performance under a system level constraint such as power consumption. DoPE enables the administrator to specify a power target, and uses a closed-loop controller to maximize throughput while maintaining power consumption at the specified target. The controller initializes each task with a DoP extent equal to 1. It then identifies the task with the least throughput and increments the DoP extent of the task if throughput improves and the power budget is not exceeded. If the power budget is exceeded, the controller tries alternative parallelism configurations with the same DoP extent as the configuration prior to power overshoot. The controller tries both new configurations

Application	Description	Lines of Code					Number of Loop Nesting Levels	Inner DoP_{min} extent for speedup
		Added	Modified	Deleted	Fused	Total		
x264	Transcoding of yuv4mpeg videos [3]	72	10	8	-	39617	2	2
swaptions	Option pricing via Monte Carlo simulations [3]	85	11	8	-	1428	2	2
bzip	Data compression of SPEC `ref` input [6, 28]	63	10	8	-	4652	2	4
gimp	Image editing using oilify plugin [10]	35	12	4	-	1989	2	2
ferret	Image search engine [3, 17]	97	15	22	59	14781	1	-
dedup	Deduplication of PARSEC `native` input [3]	124	10	16	113	7546	1	-

Table 4. Applications enhanced using DoPE. Columns 3–7 indicate the effort required to port the original Pthreads based parallel code to the DoPE interface. Where applicable, column 6 indicates the number of lines of code in tasks created by fusing other tasks. Column 8 indicates the number of loop nesting levels in each application that were exposed for this study. Where applicable, the last column indicates the minimum DoP extent of the inner loop at which the execution time of a transaction is improved.

and configurations from recorded history in order to determine the configuration with best throughput. The controller monitors power and throughput continuously in order to trigger reconfiguration if needed.

8. Evaluation

Table 4 provides a brief description of the applications that have been enhanced using DoPE. All are computationally intensive parallel applications.

8.1 The DoPE Interface

Columns 3–7 in Table 4 are indicative of the effort required to port existing Pthreads based parallel applications to the proposed DoPE API. The nature of the changes has already been illustrated in Section 3. The number of additional lines of code written by the application developer could be significantly reduced with compiler support for functor creation and variable capture in C++ and task fusion.

8.2 The DoPE Run-time System

The DoPE run-time system is implemented as a user-land shared library built on top of Pthreads. The performance overhead (compared to the Pthreads parallelizations) of run-time monitoring of workload and platform characteristics is less than 1%, even for monitoring each and every instance of all the parallel tasks. While we have explored more combinations, for all but one benchmark in Table 4, we present results on one performance goal. For one benchmark—an image search engine (`ferret`)—we present results on all the tested performance goals.

All improvements reported are over the baseline Pthreads based parallelizations. All evaluations were done natively on an Intel Xeon X7460 machine composed of 4 sockets, each with a 6-core Intel Core Architecture 64-bit processor running at 2.66GHz. The total number of cores (and hardware contexts) is 24. The system is equipped with 24GB of RAM and runs the 2.6.31-20-server Linux kernel. Applications were compiled using *gcc* 4.4.1 with the `-O3` optimization flag. Reported numbers are average values over three runs. In the case of applications with online server behavior, the arrival of tasks was simulated using a task queuing thread that enqueues tasks to a work queue according to a Poisson distribution. The average arrival rate determines the load factor on the system. A load factor of 1.0 corresponds to an average arrival rate equal to the maximum throughput sustainable by the system. The maximum throughput is determined as N/T where N is the number of tasks and T is the time taken by the system to execute the tasks in parallel (but executing each task itself sequentially). To determine the maximum throughput for each application, N was set to 500.

8.2.1 Goal: "Min Response time with N threads"

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

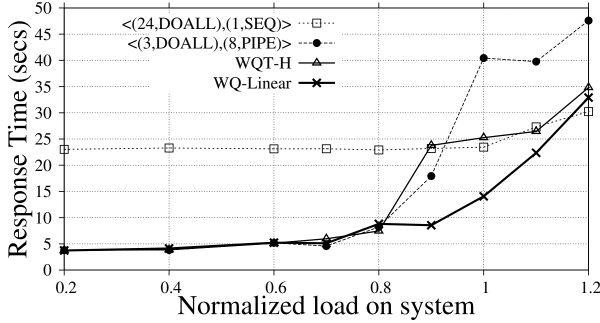
Figure 11 shows the performance of the WQT-H and WQ-Linear mechanisms compared to the static $<DoP_{outer}, DoP_{inner}>$ configurations of DoP = $<(24, DOALL), (1, SEQ)>$ and DoP = $<(N/M_{max}, DOALL), (M_{max}, PIPE \mid DOALL)>$. Here, M_{max} refers to the extent of DoP_{inner} above which parallel efficiency drops below 0.5.

Interestingly, WQT-H outperforms both static mechanisms at certain load factors. For example, consider the response times at load factor 0.8 in Figure 11(b). Analysis of the work queue occupancy and DoP assignment to tasks reveals that even though the load factor is on average equal to 0.8, there are periods of heavier and lighter load. DoPE's dynamic adaptation of the DoP between DoP = $<(24, DOALL), (1, SEQ)>$ and DoP = $<(N/M_{max}, DOALL), (M_{max}, PIPE \mid DOALL)>$ results in an average DoP somewhere in between the two, and this average DoP is better than either for minimizing response time. This provides experimental validation of the intuitive rationale behind WQ-Linear, which provides the best response time characteristic across the load factor range. In the case of data compression (Figure 11(c)), the minimum extent of DoP_{inner} at which speedup is obtained over sequential execution is 4 (see Table 4). This results in two problems for WQ-Linear. First, WQ-Linear may give unhelpful configurations such as $<(8, DOALL), (3, PIPE)>$. Second, the number of configurations at WQ-Linear's disposal is too few to provide any improvement over WQT-H.

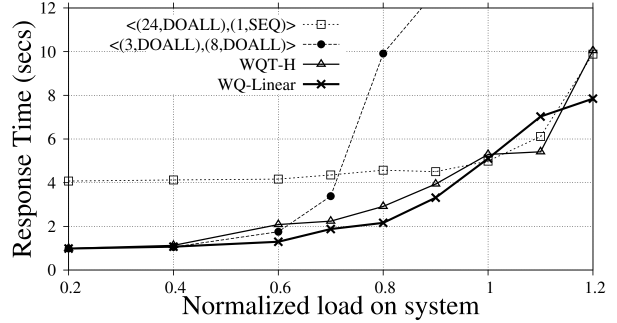
Figure 12 shows the response time characteristic of `ferret`. The figure shows the static distribution of threads to each pipeline stage. For example, ($<1, 6, 6, 6, 6, 1>, PIPE$) indicates a single loop parallelized in a pipelined fashion with 6 threads allocated to each parallel stage and 1 thread allocated to each sequential stage. Oversubscribing the hardware resources by allocating 24 threads to each parallel task results in much improved response time compared to a static even distribution of the 24 hardware threads. DoPE achieves a much better characteristic by allocating threads proportional to load on each task.

8.2.2 Goal: "Max Throughput with N threads"

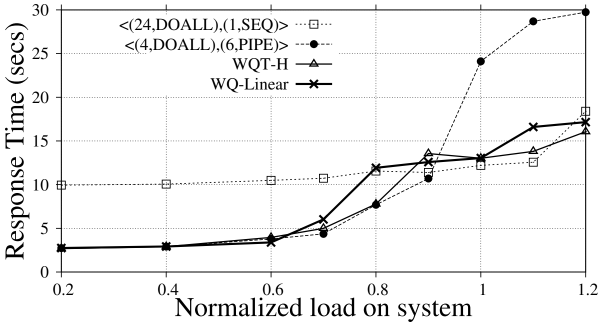
For batch processing applications, a desirable performance goal is throughput maximization. DoPE uses the mechanisms described in Section 7.2 to improve the throughput of an image search engine and a file deduplication application.



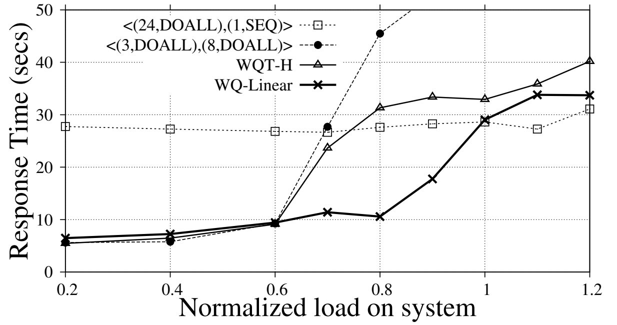
(a) Video transcoding



(b) Option pricing



(c) Data compression



(d) Image editing

Figure 11. Response time variation with load using Static, WQT-H, and WQ-Linear mechanisms

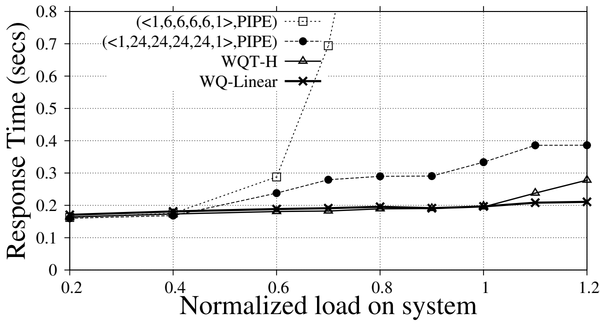


Figure 12. `ferret` Response Time

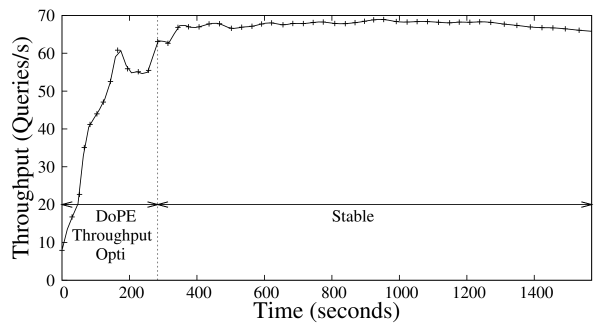


Figure 13. `ferret` Throughput

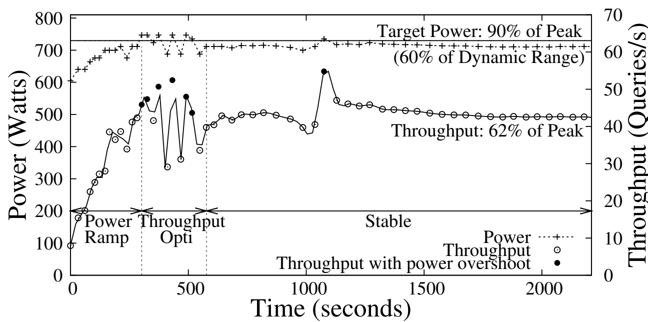


Figure 14. `ferret` Power-Throughput

Apps.		ferret	dedup
Pthreads	Baseline	1.00×	1.00×
	OS	2.12×	0.89×
DoPE	SEDA [38]	1.64×	1.16×
	FDP [29]	2.14×	2.08×
	TB	1.96×	1.75×
	TBF	2.35×	2.36×

Figure 15. Throughput improvement over static even thread distribution

Table 15 shows the throughput improvements for `ferret` and `dedup` using different mechanisms. Pthreads-Baseline is the original Pthreads parallelization with a static even distribution of available hardware threads across all the parallel tasks after assigning a single thread to each sequential task. (This is a common practice [21].) The Pthreads-OS number shows the performance when each parallel task is initialized with a thread pool containing as many threads as the number of available hardware threads in the platform, and the operating-system's scheduler is called upon to do load balancing. The remaining numbers represent the performance of the DoPEd applications using the mechanisms described in Section 7.2. DoPE-TB is the same as DoPE-TBF but with task fusion turned off, in order to demonstrate the benefit of task fusion.

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

Figure 13 shows the dynamic throughput characteristic of `ferret`. DoPE searches the parallelism configuration space before stabilizing on the one with the maximum throughput under the constraint of 24 hardware threads.

8.2.3 Goal: "Max Throughput with N threads, P Watts"

Figure 14 shows the operation of DoPE's power-throughput controller (TPC) on `ferret`. For a peak power target specified by the administrator, DoPE first ramps up the DoP extent until the power budget is fully used. DoPE then explores different parallelism configurations and stabilizes on the one with the best throughput without exceeding the power budget. Note that 90% of peak total power corresponds to 60% of peak power in the dynamic CPU range (all cores idle to all cores active). DoPE achieves the maximum throughput possible at this setting. Fine-grained per core power control can result in a wider dynamic range and greater power savings [27]. Full system power was measured at the maximum sampling rate (13 samples per minute) supported by the power distribution unit (AP7892 [2]). This limited the speed with which the controller responds to fluctuations in power consumption. Newer chips have power monitoring units with higher sampling rates and clock gating per core. They could be used to design faster and higher performance controllers for throttling power and parallelism. The transient in the Stable region of the figure shows how constant monitoring enables DoPE to respond to system events.

9. Related Work

Parallelization Libraries Several interfaces and associated run-time systems have been proposed to adapt parallel program execution to run-time variability [4, 8, 9, 13, 20, 26, 29, 35, 37, 38]. However, each interface is tied to a specific performance goal, specific mechanism of adaptation, or a specific application/platform domain. OpenMP [22], Cilk [5], and Intel TBB [26] support task parallelism for independent tasks and their schedulers optimize only for throughput. DoPE enables the developer to express parallelism in loop nests involving interacting tasks, and enables administrators to specify different performance goals. Navarro et al. developed an analytical model for pipeline parallelism to characterize performance and efficiency of pipeline parallel implementations [21]. Suleman et al. proposed Feedback Directed Pipelining (FDP) [29]. Moreno et al. proposed a technique similar to FDP called Dynamic Pipeline Mapping (DPM) [20]. We implemented FDP as a throughput maximization mechanism.

Domain-specific Programming Models Traditionally, multiple levels of parallelism across tasks and within each task has been investigated in the database research community for SQL queries [7, 12, 31, 32]. DoPE extends these works by providing dynamic adaptation to general-purpose applications that typically involve other forms of parallelism like pipeline parallelism, task parallelism, etc. DoPE also allows the administrator to specify different performance goals, and optimizes accordingly. For network service codes, programming models such as the Stage Event-Driven Architecture (SEDA) [38] and Aspen [35] have been proposed. We implemented the SEDA controller as a throughput maximization mechanism. Compared to these models, DoPE is applicable to programs with loop nests, and supports multiple performance goals. The mechanisms proposed for different performance goals in the context of DoPE could form a valuable part of the respective run-time schedulers of SEDA and Aspen. Blagojevic et al. propose user-level schedulers that dynamically "rightsize" the loop nesting level and degree of parallelism on a Cell Broadband Engine system [4]. Unlike DoPE, they exploit only one form of intra-task parallelism—loop-level DOALL parallelism.

Auto-tuning Wang et al. use machine learning to predict the best number of threads for a given program on a particular hardware platform [37]. They apply their technique on programs with a single loop. Luk et al. use a dynamic compilation approach and curve fitting to find the optimal distribution of work between a CPU and GPU [16]. Hall and Martonosi propose to increase or decrease threads allocated to compiler parallelized DOALL loops at run-time as the measured speedup exceeds or falls short of the expected speedup [11]. The ADAPT dynamic optimizer applies loop optimizations at run-time to create new variants of code [36]. Some of these sophisticated machine learning techniques could be used to improve DoPE's mechanisms.

10. Conclusion

Parallel applications must execute robustly across a variety of execution environments arising out of variability in workload characteristics, platform characteristics, and performance goals. For this, a separation of concerns of parallel application development, its optimization, and use, is required. The Degree of Parallelism Executive (DoPE) enables such a separation. Using DoPE, the application developer can specify all of the potential parallelism in loop nests just once; the mechanism developer can implement mechanisms for parallelism adaptation; and the administrator can select a suitable mechanism that implements a performance goal of system use. As a result of DoPE, they can be confident that the specified performance goals are met in a variety of application execution environments.

Acknowledgments

We thank the entire Liberty Research Group for their support and feedback during this work. We thank Alexander Wauck for help with the data compression application. We also thank the anonymous reviewers for their valuable feedback. This material is based on work supported by National Science Foundation Grants

0964328 and 1047879, and by United States Air Force Contract FA8650-09-C-7918. Arun Raman is supported by an Intel Foundation Ph.D. Fellowship.

References

[1] R. Allen and K. Kennedy. *Optimizing compilers for modern architectures: A dependence-based approach*. Morgan Kaufmann Publishers Inc., 2002.

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

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

[4] F. Blagojevic, D. S. Nikolopoulos, A. Stamatakis, C. D. Antonopoulos, and M. Curtis-Maury. Runtime scheduling of dynamic parallelism on accelerator-based multi-core systems. *Parallel Computing*, 2007.

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

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

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

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

[9] Y. Ding, M. Kandemir, P. Raghavan, and M. J. Irwin. Adapting application execution in CMPs using helper threads. *Journal of Parallel and Distributed Computing*, 2009.

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

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

[12] N. Hardavellas, I. Pandis, R. Johnson, N. Mancheril, A. Ailamaki, and B. Falsafi. Database servers on chip multiprocessors: Limitations and opportunities. In *Proceedings of the Third Biennial Conference on Innovative Data Systems Research (CIDR)*, 2007.

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

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

[15] R. Liu, K. Klues, S. Bird, S. Hofmeyr, K. Asanovi, and J. Kubiatowicz. Tessellation: Space-time partitioning in a manycore client OS. In *Proceedings of the First USENIX Workshop on Hot Topics in Parallelism (HotPar)*, 2009.

[16] C.-K. Luk, S. Hong, and H. Kim. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009.

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

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

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

[20] A. Moreno, E. César, A. Guevara, J. Sorribes, T. Margalef, and E. Luque. Dynamic Pipeline Mapping (DPM). In *Proceedings of the International Euro-Par Conference on Parallel Processing (Euro-Par)*, 2008.

[21] A. Navarro, R. Asenjo, S. Tabik, and C. Cascaval. Analytical modeling of pipeline parallelism. In *Proceedings of the Eighteenth International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2009.

[22] The OpenMP API specification for parallel programming. `http://www.openmp.org`.

[23] H. Pan, B. Hindman, and K. Asanović. Composing parallel software efficiently with Lithe. In *Proceedings of the ACM SIGPLAN 2010 Conference on Programming Language Design and Implementation (PLDI)*, 2010.

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

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

[26] J. Reinders. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2007.

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

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

[29] M. A. Suleman, M. K. Qureshi, Khubaib, and Y. N. Patt. Feedback-directed pipeline parallelism. In *Proceedings of the Nineteenth International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2010.

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

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

[32] J. Tellez and B. Dageville. *Method for computing the degree of parallelism in a multi-user environment*. United States Patent No. 6,820,262. Oracle International Corporation, 2004.

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

[34] C. Tian, M. Feng, V. Nagarajan, and R. Gupta. Copy or discard execution model for speculative parallelization on multicores. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2008.

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

[36] M. J. Voss and R. Eigenmann. ADAPT: Automated De-Coupled Adaptive Program Transformation. In *Proceedings of the 28th International Conference on Parallel Processing (ICPP)*, 1999.

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

[38] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. *ACM SIGOPS Operating Systems Review*, 2001.

[39] T. Wiegand, G. J. Sullivan, G. Bjontegaard, and A. Luthra. Overview of the H.264/AVC video coding standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 2003.

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