MIT Libraries | DSpace@MIT

MIT Open Access Articles

On-the-fly pipeline parallelism

The MIT Faculty has made this article openly available. **Please share** how this access benefits you. Your story matters.

Citation: I-Ting Angelina Lee, Charles E. Leiserson, Tao B. Schardl, Jim Sukha, and Zhunping Zhang. 2013. On-the-fly pipeline parallelism. In Proceedings of the twenty-fifth annual ACM symposium on Parallelism in algorithms and architectures (SPAA '13). ACM, New York, NY, USA, 140-151.

As Published: http://dx.doi.org/10.1145/2486159.2486174

Publisher: Association for Computing Machinery (ACM)

Persistent URL: http://hdl.handle.net/1721.1/90258

Version: Author's final manuscript: final author's manuscript post peer review, without publisher's formatting or copy editing

Terms of use: Creative Commons Attribution-Noncommercial-Share Alike

Massachusetts Institute of Technology

On-the-Fly Pipeline Parallelism

I-Ting Angelina Lee* Charles E. Leiserson* Tao B. Schardl* Jim Sukha† Zhunping Zhang*

*MIT CSAIL
32 Vassar Street
Cambridge, MA 02139

*{angelee, cel, neboat, jzz}@mit.edu

†Intel Corporation
25 Manchester Street, Suite 200
Merrimack, NH 03054

†jim.sukha@intel.com

ABSTRACT

Pipeline parallelism organizes a parallel program as a linear sequence of s stages. Each stage processes elements of a data stream, passing each processed data element to the next stage, and then taking on a new element before the subsequent stages have necessarily completed their processing. Pipeline parallelism is used especially in streaming applications that perform video, audio, and digital signal processing. Three out of 13 benchmarks in PARSEC, a popular software benchmark suite designed for shared-memory multiprocessors, can be expressed as pipeline parallelism.

Whereas most concurrency platforms that support pipeline parallelism use a "construct-and-run" approach, this paper investigates "on-the-fly" pipeline parallelism, where the structure of the pipeline emerges as the program executes rather than being specified *a priori*. On-the-fly pipeline parallelism allows the number of stages to vary from iteration to iteration and dependencies to be data dependent. We propose simple linguistics for specifying on-the-fly pipeline parallelism and describe a provably efficient scheduling algorithm, the PIPER algorithm, which integrates pipeline parallelism into a work-stealing scheduler, allowing pipeline and fork-join parallelism to be arbitrarily nested. The PIPER algorithm automatically throttles the parallelism, precluding "runaway" pipelines. Given a pipeline computation with T_1 work and T_∞ span (critical-path length), PIPER executes the computation on P processors in $T_P \le T_1/P + O(T_\infty + \lg P)$ expected time. PIPER also limits stack space, ensuring that it does not grow unboundedly with running time.

We have incorporated on-the-fly pipeline parallelism into a Cilk-based work-stealing runtime system. Our prototype Cilk-P implementation exploits optimizations such as lazy enabling and dependency folding. We have ported the three PARSEC benchmarks that exhibit pipeline parallelism to run on Cilk-P. One of these, *x264*, cannot readily be executed by systems that support only construct-and-run pipeline parallelism. Benchmark results indicate that Cilk-P has low serial overhead and good scalability. On *x264*, for example, Cilk-P exhibits a speedup of 13.87 over its respective serial counterpart when running on 16 processors.

This work was supported in part by the National Science Foundation under Grants CNS-1017058 and CCF-1162148. Tao B. Schardl is supported in part by an NSF Graduate Research Fellowship.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SPAA'13, June 23–25, 2013, Montréal, Québec, Canada.
Copyright 2013 ACM 978-1-4503-1572-2/13/07 ...$15.00.

Categories and Subject Descriptors

D.3.3 [**Language Constructs and Features**]: Concurrent programming structures; D.3.4 [**Programming Languages**]: Processors—*Run-time environments*.

General Terms

Algorithms, Languages, Theory.

Keywords

Cilk, multicore, multithreading, parallel programming, pipeline parallelism, on-the-fly pipelining, scheduling, work stealing.

1. INTRODUCTION

Pipeline parallelism[1] [6, 16, 17, 25, 27, 28, 31, 33, 35, 37] is a well-known parallel-programming pattern that can be used to parallelize a variety of applications, including streaming applications from the domains of video, audio, and digital signal processing. Many applications, including the *ferret*, *dedup*, and *x264* benchmarks from the PARSEC benchmark suite [4, 5], exhibit parallelism in the form of a ***linear*** pipeline, where a linear sequence $\mathcal{S} = \langle \mathcal{S}_0, \ldots, \mathcal{S}_{s-1} \rangle$ of abstract functions, called ***stages***, are executed on an input stream $I = \langle a_0, a_1, \ldots, a_{n-1} \rangle$. Conceptually, a linear pipeline can be thought of as a loop over the elements of I, where each loop ***iteration*** i processes an element a_i of the input stream. The loop body encodes the sequence $\mathcal{S}$ of stages through which each element is processed. Parallelism arises in linear pipelines because the execution of iterations can overlap in time, that is, iteration i may start after the preceding iteration $i-1$ has started, but before $i-1$ has necessarily completed.

Most systems that provide pipeline parallelism employ a ***construct-and-run*** model, as exemplified by the pipeline model in Intel Threading Building Blocks (TBB) [27], where the pipeline stages and their dependencies are defined *a priori* before execution. Systems that support construct-and-run pipeline parallelism include the following: [1, 11, 17, 25–27, 29–32, 35, 37, 38].

We have extended the Cilk parallel-programming model [15, 20, 24] to augment its native fork-join parallelism with ***on-the-fly*** pipeline parallelism, where the linear pipeline is constructed dynamically as the program executes. The Cilk-P system provides a flexible linguistic model for pipelining that allows the structure of the pipeline to be determined dynamically as a function of data in the input stream. Cilk-P also admits a variable number of stages across iterations, allowing the pipeline to take on shapes other than

[1] Pipeline parallelism should not be confused with instruction pipelining in hardware [34] or software pipelining [22].

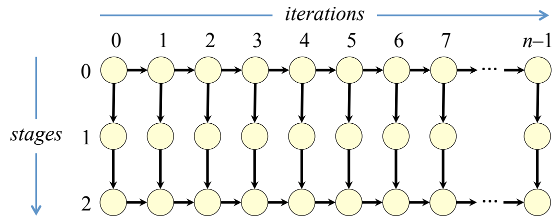

Figure 1: Modeling the execution of *ferret*'s linear pipeline as a pipeline dag. Each column contains nodes for a single iteration, and each row corresponds to a stage of the pipeline. Vertices in the dag correspond to nodes of the linear pipeline, and edges denote dependencies between the nodes. Throttling edges are not shown.

simple rectangular grids. The Cilk-P programming model is flexible, yet restrictive enough to allow provably efficient scheduling, as Sections 5 through 8 will show. In particular, Cilk-P's scheduler provides automatic "throttling" to ensure that the computation uses bounded space. As a testament to the flexibility provided by Cilk-P, we were able to parallelize the *x264* benchmark from PARSEC, an application that cannot be programmed easily using TBB [33].

Cilk-P's support for defining linear pipelines on the fly is more flexible than the `ordered` directive in OpenMP [29], which supports a limited form of on-the-fly pipelining, but it is less expressive than other approaches. Blelloch and Reid-Miller [6] describe a scheme for on-the-fly pipeline parallelism that employs futures [3, 14] to coordinate the stages of the pipeline, allowing even nonlinear pipelines to be defined on the fly. Although futures permit more complex, nonlinear pipelines to be expressed, this generality can lead to unbounded space requirements to attain even modest speedups [7].

To illustrate the ideas behind the Cilk-P model, consider a simple 3-stage linear pipeline such as in the *ferret* benchmark from PARSEC [4, 5]. Figure 1 shows a ***pipeline dag*** (directed acyclic graph) $G = (V,E)$ representing the execution of the pipeline. Each of the 3 horizontal rows corresponds to a stage of the pipeline, and each of the n vertical columns is an iteration. We define a pipeline ***node*** $(i, j) \in V$, where $i = 0,1,\ldots,n-1$ and $j = 0,1,2$, to be the execution of $\mathcal{S}_j(a_i)$, the jth stage in the ith iteration, represented as a vertex in the dag. The edges between nodes denote dependencies. A ***stage edge*** goes between two nodes (i, j) and (i, j'), where $j < j'$, and indicates that (i, j') cannot start until (i, j) completes. A ***cross edge*** between nodes $(i-1, j)$ and (i, j) indicates that (i, j) can start execution only after node $(i-1, j)$ completes. Cilk-P always executes nodes of the same iteration in increasing order by stage number, thereby creating a vertical chain of stage edges. Cross edges between corresponding stages of adjacent iterations are optional.

We can categorize the stages of a Cilk-P pipeline. A stage is a ***serial stage*** if all nodes belonging to the stage are connected by cross edges, it is a ***parallel stage*** if none of the nodes belonging to the stage are connected by cross edges, and it is a ***hybrid stage*** otherwise. The *ferret* pipeline, for example, exhibits a static structure often referred to as an "SPS" pipeline, since Stage 0 and Stage 2 are serial and Stage 1 is parallel. Cilk-P requires that pipelines be linear, since iterations are totally ordered and dependencies go between adjacent iterations, and in fact, Stage 0 of any Cilk-P pipeline is always a serial stage. Later stages may be serial, parallel, or hybrid, as we shall see in Sections 2 and 3.

To execute a linear pipeline, Cilk-P follows the lead of TBB and adopts a ***bind-to-element*** approach [25, 27], where ***workers*** (scheduling threads) execute pipeline iterations either to completion or until an unresolved dependency is encountered. In particular, Cilk-P and TBB both rely on "work-stealing" schedulers (see, for example, [2, 8, 10, 13, 15, 21]) for load balancing. In contrast, many systems that support pipeline parallelism, including typical Pthreaded implementations, execute linear pipelines using a ***bind-to-stage*** approach, where each worker executes a distinct stage and coordination between workers is handled using concurrent queues [17, 35, 38]. Some researchers report that the bind-to-element approach generally outperforms bind-to-stage [28, 33], since a work-stealing scheduler can do a better job of dynamically load-balancing the computation, but our own experiments show mixed results.

A natural theoretical question is, how much parallelism is inherent in the *ferret* pipeline (or in any pipeline)? How much speedup can one hope for? Since the computation is represented as a dag $G = (V,E)$, one can use a simple work/span analysis [12, Ch. 27] to answer this question. In this analytical model, we assume that each vertex $v \in V$ executes in some time $w(v)$. The ***work*** of the computation, denoted T_1, is essentially the serial execution time, that is, $T_1 = \sum_{v \in V} w(v)$. The ***span*** of the computation, denoted T_∞, is the length of a longest weighted path through G, which is essentially the time of an infinite-processor execution. The ***parallelism*** is the ratio T_1/T_∞, which is the maximum possible speedup attainable on any number of processors, using any scheduler.

Unlike in some applications, in the *ferret* pipeline, each node executes serially, that is, its work and span are the same. Let $w(i, j)$ be the execution time of node (i, j). Assume that the serial Stages 0 and 2 execute in unit time, that is, for all i, we have $w(i,0) = w(i,2) = 1$, and that the parallel Stage 1 executes in time $r \gg 1$, that is, for all i, we have $w(i,1) = r$. Because the pipeline dag is grid-like, the span of this SPS pipeline can be realized by some staircase walk through the dag from node $(0,0)$ to node $(n-1,2)$. The work of this pipeline is therefore $T_1 = n(r+2)$, and the span is

$$\begin{aligned} T_\infty &= \max_{0 \le x < n} \left\{ \sum_{i=0}^{x} w(i,0) + w(x,1) + \sum_{i=x}^{n-1} w(i,2) \right\} \\ &= n + r \ . \end{aligned}$$

Consequently, the parallelism of this dag is $T_1/T_\infty = n(r+2)/(n+r)$, which for $1 \ll r \le n$ is at least $r/2+1$. Thus, if Stage 1 contains much more work than the other two stages, the pipeline exhibits good parallelism.

Cilk-P guarantees to execute the *ferret* pipeline efficiently. In particular, on an ideal shared-memory computer with up to $T_1/T_\infty = O(r)$ processors, Cilk-P guarantees linear speedup. Generally, Cilk-P executes a pipeline with linear speedup as long as the parallelism of the pipeline exceeds the number of processors on which the computation is scheduled. Moreover, as Section 3 will describe, Cilk-P allows stages of the pipeline themselves to be parallel using recursive pipelining or fork-join parallelism.

In practice, it is also important to limit the space used during an execution. Unbounded space can cause thrashing of the memory system, leading to slowdowns not predicted by simple execution models. In particular, a bind-to-element scheduler must avoid creating a ***runaway*** pipeline — a situation where the scheduler allows many new iterations to be started before finishing old ones. In Figure 1, a runaway pipeline might correspond to executing many nodes in Stage 0 (the top row) without finishing the other stages of the computation in the earlier iterations. Runaway pipelines can cause space utilization to grow unboundedly, since every started but incomplete iteration requires space to store local variables.

Cilk-P automatically ***throttles*** pipelines to avoid runaway pipelines. On a system with P workers, Cilk-P inhibits the start of iteration $i+K$ until iteration i has completed, where $K = \Theta(P)$ is the ***throttling limit***. Throttling corresponds to putting ***throttling edges*** from the last node in each iteration i to the first node in iter-

ation $i+K$. For the simple pipeline from Figure 1, throttling does not adversely affect asymptotic scalability if stages are uniform, but it can be a concern for more complex pipelines, as Section 11 will discuss. The Cilk-P scheduler guarantees efficient scheduling of pipelines as a function of the parallelism of the dag in which throttling edges are included in the calculation of span.

Contributions

Our prototype Cilk-P system adapts the Cilk-M [23] work-stealing scheduler to support on-the-fly pipeline parallelism using a bind-to-element approach. This paper makes the following contributions:

- We describe linguistics for Cilk-P that allow on-the-fly pipeline parallelism to be incorporated into the Cilk fork-join parallel programming model (Section 2).
- We illustrate how Cilk-P linguistics can be used to express the *x264* benchmark as a pipeline program (Section 3).
- We characterize the execution dag of a Cilk-P pipeline program as an extension of a fork-join program (Section 4).
- We introduce the PIPER scheduling algorithm, a theoretically sound randomized work-stealing scheduler (Section 5).
- We prove that PIPER is asymptotically efficient, executing Cilk-P programs on P processors in $T_P \leq T_1/P + O(T_\infty + \lg P)$ expected time (Sections 6 and 7).
- We bound space usage, proving that PIPER on P processors uses $S_P \leq P(S_1 + fDK)$ stack space for pipeline iterations, where S_1 is the serial stack space, f is the "frame size," D is the depth of nested pipelines, and K is the throttling limit (Section 8).
- We describe our implementation of PIPER in the Cilk-P runtime system, introducing two key optimizations: lazy enabling and dependency folding (Section 9).
- We demonstrate that the *ferret*, *dedup*, and *x264* benchmarks from PARSEC, when hand-compiled for the Cilk-P runtime system (we do not as yet have a compiler for the Cilk-P language), run competitively with existing Pthreaded implementations (Section 10).

We conclude in Section 11 with a discussion of the performance implications of throttling.

2. ON-THE-FLY PIPELINE PROGRAMS

Cilk-P's linguistic model supports both fork-join and pipeline parallelism, which can be nested arbitrarily. For convenience, we shall refer to programs containing nested fork-join and pipeline parallelism simply as ***pipeline programs***. Cilk-P's on-the-fly pipelining model allows the programmer to specify a pipeline whose structure is determined during the pipeline's execution. This section reviews the basic Cilk model and shows how on-the-fly parallelism is supported in Cilk-P using a "`pipe_while`" construct.

We first outline the basic semantics of Cilk without the pipelining features of Cilk-P. We use the syntax of Cilk++ [24] and Cilk Plus [20] which augments serial C/C++ code with two principal keywords: `cilk_spawn` and `cilk_sync`.[2] When a function invocation is preceded by the keyword `cilk_spawn`, the function is ***spawned*** as a ***child*** subcomputation, but the runtime system may continue to execute the statement after the `cilk_spawn`, called the ***continuation***, in parallel with the spawned subroutine without waiting for the child to return. The complementary keyword to `cilk_spawn` is `cilk_sync`, which acts as a local barrier and joins together all the parallelism forked by `cilk_spawn` within a function. Every function contains an implicit `cilk_sync` before the function returns.

[2]Cilk++ and Cilk Plus also include other features that are not relevant to the discussion here.

To support on-the-fly pipeline parallelism, Cilk-P provides a `pipe_while` keyword. A `pipe_while` loop is similar to a serial `while` loop, except that loop iterations can execute in parallel in a pipelined fashion. The body of the `pipe_while` can be subdivided into stages, with stages named by user-specified integer values that strictly increase as the iteration executes. Each stage can contain nested fork-join and pipeline parallelism.

The boundaries of stages are denoted in the body of a `pipe_while` using the special functions `pipe_continue` and `pipe_wait`. These functions accept an integer ***stage argument***, which is the number of the next stage to execute and which must strictly increase during the execution of an iteration. Every iteration i begins executing Stage 0, represented by node $(i,0)$. While executing a node (i,j'), if control flow encounters a `pipe_wait(j)` or `pipe_continue(j)` statement, where $j > j'$, then node (i,j') ends, and control flow proceeds to node (i,j). A `pipe_continue(j)` statement indicates that node (i,j) can start executing immediately, whereas a `pipe_wait(j)` statement indicates that node (i,j) cannot start until node $(i-1,j)$ completes. The `pipe_wait(j)` in iteration i creates a cross edge from node $(i-1,j)$ to node (i,j) in the pipeline dag. Thus, by design choice, Cilk-P imposes the restriction that pipeline dependencies only go between adjacent iterations. As we shall see in Section 9, this design choice facilitates the "lazy enabling" and "dependency folding" runtime optimizations.

The `pipe_continue` and `pipe_wait` functions can be used without an explicit stage argument. Omitting the stage argument while executing stage j corresponds to an implicit stage argument of $j+1$, i.e., control moves onto the next stage.

Cilk-P's semantics for `pipe_continue` and `pipe_wait` statements allow for ***stage skipping***, where execution in an iteration i can jump stages from node (i,j') to node (i,j), even if $j > j'+1$. If control flow in iteration $i+1$ enters node $(i+1,j'')$ after a `pipe_wait`, where $j' < j'' < j$, then we implicitly create a ***null node*** (i,j'') in the pipeline dag, which has no associated work and incurs no scheduling overhead, and insert stage edges from (i,j') to (i,j'') and from (i,j'') to (i,j), as well as a cross edge from (i,j'') to $(i+1,j'')$.

3. ON-THE-FLY PIPELINING OF *x264*

To illustrate the use of Cilk-P's `pipe_while` loop, this section describes how to parallelize the *x264* video encoder [39].

We begin with a simplified description of *x264*. Given a stream $\langle f_0, f_1, \ldots \rangle$ of video frames to encode, *x264* partitions the frame into a two dimensional array of "macroblocks" and encodes each macroblock. A macroblock in frame f_i is encoded as a function of the encodings of similar macroblocks within f_i and similar macroblocks in frames "near" f_i. A frame f_j is ***near*** a frame f_i if $i-b \leq j \leq i+b$ for some constant b. In addition, we define a macroblock (x',y') to be ***near*** a macroblock (x,y) if $x-w \leq x' \leq x+w$ and $y-w \leq y' \leq y+w$ for some constant w.

The type of a frame f_i determines how a macroblock (x,y) in f_i is encoded. If f_i is an ***I-frame***, then macroblock (x,y) can be encoded using only ***previous*** macroblocks within f_i — macroblocks at positions (x',y') where $y' < y$ or $y' = y$ and $x' < x$. If f_i is a ***P-frame***, then macroblock (x,y)'s encoding can also be based on nearby macroblocks in nearby preceding frames, up to the most recent preceding I-frame,[3] if one exists within the nearby range. If f_i is a ***B-frame***, then macroblock (x,y)'s encoding can be based also on nearby macroblocks in nearby frames, likewise, up to the most recently preceding I-frame and up to the next succeeding I- or P-frame.

[3]To be precise, up to a particular type of I-frame called an ***IDR-frame***.

```
1   // Symbolic names for important stages
2   const uint64_t PROCESS_IPFRAME = 1;
3   const uint64_t PROCESS_BFRAMES = 1 << 40;
4   const uint64_t END = PROCESS_BFRAMES + 1;
5   int i = 0;
6   int w = mv_range/pixel_per_row;

8   pipe_while(frame_t *f = next_frame()) {
9     vector<frame_t *> bframes;
10    f->type = decide_frame_type(f);
11    while(f->type == TYPE_B) {
12      bframes.push_back(f);
13      f = next_frame();
14      f->type = decide_frame_type(f);
15    }
16    int skip = w * i++;
17    pipe_wait(PROCESS_IPFRAME + skip);
18    while(mb_t *macroblocks = next_row(frame)) {
19      process_row(macroblocks);
20      if(f->type == TYPE_I) {
21        pipe_continue;
22      } else {
23        pipe_wait;
24      }
25    }
26    pipe_continue(PROCESS_BFRAMES);
27    cilk_for(int j=0; j<bframes.size(); ++j) {
28      process_bframe(bframes[j]);
29    }
30    pipe_wait(END);
31    write_out_frames(frame, bframes);
32  }
```

Figure 2: Example C++-like pseudocode for the *x264* linear pipeline. This pseudocode uses Cilk-P's linguistics to define hybrid pipeline stages on the fly, specifically with the `pipe_wait` on line 17, the input-data dependent `pipe_wait` or `pipe_continue` on lines 20–24, and the `pipe_continue` on line 26.

Based on these frame types, an *x264* encoder must ensure that frames are processed in a valid order such that dependencies between encoded macroblocks are satisfied. A parallel *x264* encoder can pipeline the encoding of I- and P-frames in the input stream, processing each set of intervening B-frames after encoding the latest I- or P-frame on which the B-frame may depend.

Figure 2 shows pseudocode for an *x264* linear pipeline. Conceptually, the *x264* pipeline begins with a serial stage (lines 8–17) that reads frames from the input stream and determines the type of each frame. This stage buffers all B-frames at the head of the input stream until it encounters an I- or P-frame. After this initial stage, s hybrid stages process this I- or P-frame row by row (lines 18–25), where s is the number of rows in the video frame. After all rows of this I- or P-frame have been processed, the `PROCESS_BFRAMES` stage processes all B-frames in parallel (lines 27–29), and then the `END` stage updates the output stream with the processed frames (line 31).

Two issues arise with this general pipelining strategy, both of which can be handled using on-the-fly pipeline parallelism. First, the encoding of a P-frame must wait for the encoding of rows in the previous frame to be completed, whereas the encoding of an I-frame need not. These conditional dependencies are implemented in lines 20–24 of Figure 2 by executing a `pipe_wait` or `pipe_continue` statement conditionally based on the frame's type. In contrast, many construct-and-run pipeline mechanisms assume that the dependencies on a stage are fixed for the entirety of a pipeline's execution, making such dynamic dependencies more difficult to handle. Second, the encoding of a macroblock in row x of P-frame f_i may depend on the encoding of a macroblock in a later row $x+w$ in the preceding I- or P-frame f_{i-1}. The code in Figure 2 handles such offset dependencies on line 17 by skipping w additional stages relative to the previous iteration. A similar stage-skipping trick is used on line 26 to ensure that the processing of a P-frame in iteration i depends only on the processing of the previous I- or P-frame, and not on the processing of preceding B-frames. Figure 3 illustrates the pipeline dag corresponding to the execution of the code in Figure 2, assuming that $w=1$. Skipping stages shifts the nodes of an iteration down, adding null nodes to the pipeline, which do not increase the work or span.

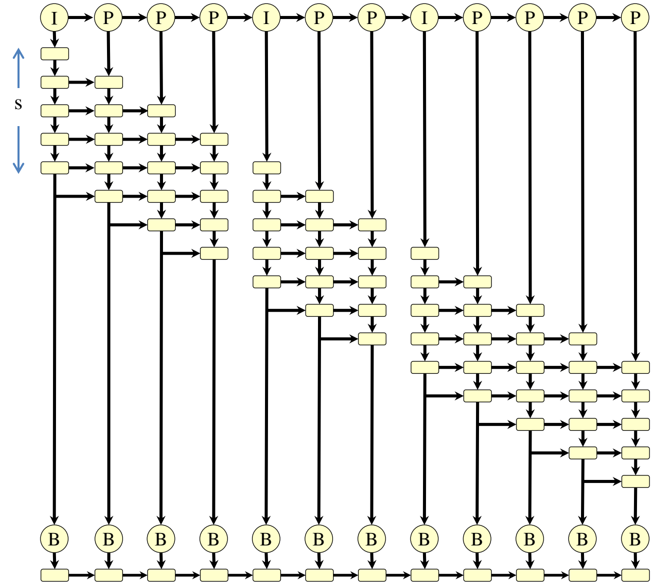

Figure 3: The pipeline dag generated for *x264*. Each iteration processes either an I- or P-frame, each consisting of s rows. As the iteration index i increases, the number of initial stages skipped in the iteration also increases. This stage skipping produces cross edges into an iteration i from null nodes in iteration $i-1$. Null nodes are represented as the intersection between two edges.

4. COMPUTATION-DAG MODEL

Although the pipeline-dag model provides intuition for programmers to understand the execution of a pipeline program, it is not as precise as we shall require. For example, a pipeline dag has no real way of representing nested fork-join or pipeline parallelism within a node. This section describes how to represent the execution of a pipeline program as a more refined "computation dag."

Let us first review the notion of a computation dag for ordinary fork-join Cilk programs [7, 8] without pipeline parallelism. A ***fork-join computation dag*** $G=(V,E)$ represents the execution of a Cilk program, where the vertices belonging to V are unit-cost instructions. Edges in E indicate ordering dependencies between instructions. The normal serial execution of one instruction after another creates a ***serial edge*** from the first instruction to the next. A `cilk_spawn` of a function creates two dependency edges emanating from the instruction immediately before the `cilk_spawn`: the ***spawn edge*** goes to the first instruction of the spawned function, and the ***continue edge*** goes to the first instruction after the spawned function. A `cilk_sync` creates a ***return edge*** from the final instruction of each spawned function to the instruction immediately after the `cilk_sync` (as well as an ordinary serial edge from the instruction that executed immediately before the `cilk_sync`).

To model an arbitrary pipeline-program execution as a ***(pipeline) computation dag***, we follow a three-step process. First, we translate the code executed in each `pipe_while` loop into ordinary Cilk code augmented with special functions to handle cross and throttling dependencies. Second, we model the execution of this augmented Cilk program as a fork-join computation dag. Third, we show how to augment the fork-join computation dag with cross and throttling edges using the special functions.

The first step of this process does not reflect how a Cilk-P com-

```
1   int fd_out = open_output_file();
2   bool done = false;
3   pipe_while (!done) {
4     chunk_t *chunk = get_next_chunk();
5     if (chunk == NULL) {
6       done = true;
7     } else {
8       pipe_wait(1);
9       bool isDuplicate = deduplicate(chunk);
10      pipe_continue(2);
11      if (!isDuplicate)
12        compress(chunk);
13      pipe_wait(3);
14      write_to_file(fd_out, chunk);
15    }
16  }
```

Figure 4: Cilk-P pseudocode for the parallelization of the *dedup* compression program as an SSPS pipeline.

piler would actually compile a `pipe_while` loop. Indeed, such a code transformation is impossible for a compiler, because the boundaries of nodes are determined on the fly. Instead, this code-transformation step is simply a theoretical construct for the purpose of describing how the PIPER algorithm works in a way that can be analyzed.

We shall illustrate this three-step process on a Cilk-P implementation of the *dedup* compression program from PARSEC [4, 5]. The benchmark can be parallelized by using a `pipe_while` to implement an SSPS pipeline. Figure 4 shows Cilk-P pseudocode for *dedup*, which compresses the provided input file by removing duplicated "chunks," as follows. Stage 0 (lines 4–6) of the program reads data from the input file and breaks the data into chunks (line 4). As part of Stage 0, it also checks the loop-termination condition and sets the `done` flag to `true` (line 6) if the end of the input file is reached. If there is more input to be processed, the program begins Stage 1, which calculates the SHA1 signature of a given chunk and queries a hash table whether this chunk has been seen using the SHA1 signature as key (line 9). Stage 1 is a serial stage as dictated by the `pipe_wait` on line 8. Stage 2, which the `pipe_continue` on line 10 indicates is a parallel stage, compresses the chunk if it has not been seen before (line 12). The final stage, a serial stage, writes either the compressed chunk or its SHA1 signature to the output file depending on whether it is the first time the chunk has been seen (line 14).

As the first step in building the computation dag for an execution of this Cilk-P program, we transform the code executed from running the code in Figure 4 into the ordinary Cilk program shown in Figure 5. As shown in lines 3–32, a `pipe_while` is "lifted" using a C++ lambda function [36, Sec.11.4] and converted to an ordinary `while` loop using the variable `i` to index iterations. The loop body executes Stage 0 and spawns off a C++ lambda function that executes the remainder of the iteration (line 12). As one can see from this transformation, Stage 0 of a `pipe_while` loop is always a serial stage and the test condition of the `pipe_while` loop is considered part of Stage 0. These constraints guarantee that the repeated tests of the `pipe_while` loop-termination condition execute serially. Each stage ends with a `cilk_sync` (lines 10, 16, 21, and 25.) The last statement in the loop (line 29) is a call to a special function `throttle`, which implements the throttling dependency. The `cilk_sync` immediately after the end of the `while` loop (line 31) waits for completion of the spawned iterations.

The second step models the execution of this Cilk program as a fork-join computation dag $G = (V, E)$ as in [7, 8].

The third step is to produce the final pipeline computation dag by augmenting the fork-join computation dag with cross and throttling edges based on the special functions `pipe_wait`, `pipe_continue`, and `throttle`. For example, when iteration i executes the `pipe_wait` call in line 22, it specifies the start of

```
1   int fd_out = open_output_file();
2   bool done = false;
3   [&]() {
4     int i = 0;  // iteration index
5     while (!done) { // pipe_while
6       chunk_t *chunk = get_next_chunk();
7       if (chunk == NULL) {
8         done = true;
9       } else {
10        cilk_sync;
11        // Additional stages of iteration i
12        cilk_spawn [i, chunk, fd_out]() {
13          pipe_wait(1);
14          // node (i,1) begins
15          bool isDuplicate = deduplicate(chunk);
16          cilk_sync;
17          pipe_continue(2);
18          // node (i,2) begins
19          if (!isDuplicate)
20            compress(chunk);
21          cilk_sync;
22          pipe_wait(3);
23          // node (i,3) begins
24          write_to_file(fd_out, chunk);
25          cilk_sync;
26        }();
27      }
28      i++;
29      throttle(i - K);
30    }
31    cilk_sync;
32  }();
```

Figure 5: The Cilk Plus pseudocode that results from transforming the execution of the Cilk-P *dedup* implementation from Figure 4 into fork-join code augmented by dependencies indicated by the `pipe_wait`, `pipe_continue`, and `throttle` special functions. The unbound variable `K` is the throttling limit.

node $(i, 3)$ and adds a cross edge from the last instruction of node $(i-1, 3)$ to the first instruction of node $(i, 3)$. If node $(i-1, 3)$ is a null node, then the cross edge goes from the last instruction of the last real node in iteration $i-1$ before $(i-1, 3)$. This "collapsing" of null nodes may cause multiple cross edges to be generated from a single vertex in iteration $i-1$ to different vertices in iteration i. The `pipe_continue` call in line 17 simply indicates the start of node $(i, 2)$. The `throttle` call in line 29 changes the normal return edge from the last instruction in iteration $i-K$ (the `return` represented by the closing brace in line 26) into a throttling edge. Rather than going to the `cilk_sync` in line 31 as the `return` would, the edge is redirected to the invocation of `throttle` in iteration i.

5. THE PIPER SCHEDULER

PIPER executes a pipeline program on a set of P workers using work-stealing. For the most part, PIPER's execution model can be viewed as modification of the scheduler described by Arora, Blumofe, and Plaxton [2] (henceforth referred to as the ABP model) for computation dags arising from pipeline programs. PIPER deviates from the ABP model in one significant way, however, in that it performs a "tail-swap" operation.

We describe the operation of PIPER in terms of the pipeline computation dag $G = (V, E)$. Each worker p in PIPER maintains an ***assigned vertex*** corresponding to the instruction that p executes on the current time step. We say that a vertex x is ***ready*** if all its predecessors have been executed. Executing an assigned vertex v may ***enable*** a vertex x that is a direct successor of v in G by making x ready. Each worker maintains a ***deque*** of ready vertices. Normally, a worker pushes and pops vertices from the tail of its deque. A "thief," however, may try to steal a vertex from the head of another worker's deque. It is convenient to define the ***extended deque*** $\langle v_0, v_1, \ldots, v_r \rangle$ of a worker p, where $v_0 \in V$ is p's assigned vertex and $v_1, v_2, \ldots, v_r \in V$ are the vertices in p's deque in order from tail to head.

On each time step, each PIPER worker p follows a few sim-

ple rules for execution based on the type of p's assigned vertex v and how many direct successors are enabled by the execution of v, which is at most 2. (Although v may have multiple immediate successors in the next iteration due to the collapsing of null nodes, executing v can enable at most one such vertex, since the stages in the next iteration execute serially.) We assume that the rules are executed atomically.

If the assigned vertex v is not the last vertex of an iteration and its execution enables only one direct successor x, then p simply changes its assigned vertex from v to x. Executing a vertex v can enable two successors if v spawns a child x with continuation y or if v is the last vertex in a node in iteration i, which enables both the first vertex x of the next node in i and the first vertex y in a node in iteration $i+1$. In either case, p pushes y onto the tail of its deque and changes its assigned vertex from v to x. If executing v enables no successors, then p tries to pop an element z from the tail of its deque, changing its assigned vertex from v to z. If p's deque is empty, p becomes a ***thief***. As a thief, p randomly picks another worker to be its ***victim***, tries to steal the vertex z at the head of the victim's deque, and sets the assigned vertex of p to z if successful. These cases are consistent with the normal ABP model.

PIPER handles the end of an iteration differently, however, due to throttling edges. Suppose that a worker p has an assigned vertex v representing the last vertex in a given iteration in a given `pipe_while` loop, and suppose that the edge leaving v is a throttling edge to a vertex z. When p executes v, two cases are possible. In the first case, executing v does not enable z, in which case no new vertices are enabled, and p acts accordingly. In the second case, however, executing v does enable z, in which case p performs two actions. First, p changes its assigned vertex from v to z. Second, if p has a nonempty deque, then p performs a ***tail swap***: it exchanges its assigned vertex z with the vertex at the tail of its deque.

This tail-swap operation is designed to reduce PIPER's space usage. Without the tail swap, in a normal ABP-style execution, when a worker p finishes an iteration i that enables a vertex via a throttling edge, p would conceptually choose to start a new iteration $i+K$, even if iteration $i+1$ were already suspended and on its deque. With the tail swap, p resumes iteration $i+1$, leaving $i+K$ available for stealing. The tail swap also enhances cache locality by encouraging p to execute consecutive iterations.

It may seem, at first glance, that a tail-swap operation might significantly reduce the parallelism, since the vertex z enabled by the throttling edge is pushed onto the bottom of the deque. Intuitively, if there were additional work above z in the deque, then a tail swap could significantly delay the start of iteration $i+K$. Lemma 4 will show, however, that a tail-swap operation only occurs on deques with exactly 1 element. Thus, whenever a tail swap occurs, z is at the top of the deque and is immediately available to be stolen.

6. STRUCTURAL INVARIANTS

During the execution of a pipeline program by PIPER, the worker deques satisfy two structural invariants, called the "contour" property and the "depth" property. This section states and proves these invariants.

Intuitively, we would like to describe the structure of the worker deques in terms of ***frames*** — activation records — of functions' local variables, since the deques implement a "cactus stack" [18, 23]. A `pipe_while` loop would correspond to a parent frame with a spawned child for each iteration. Although the actual Cilk-P implementation manages frames in this fashion, the control of a `pipe_while` really does follow the schema illustrated in Figure 5, where Stage 0 of an iteration i executes in the same lambda function as the parent, rather than in the child lambda function which contains the rest of i. Consequently, we introduce "contours" to represent this structure.

Consider a computation dag $G=(V,E)$ that arises from the execution of a pipeline program. A ***contour*** is a path in G composed only of serial and continue edges. A contour must be a path, because there can be at most one serial or continue edge entering or leaving any vertex. We call the first vertex of a contour the ***root*** of the contour, which (except for the initial instruction of the entire computation) is the only vertex in the contour that has an incoming spawn edge. Consequently, contours can be organized into a tree hierarchy, where one contour is a parent of a second if the first contour contains a vertex that spawns the root of the second. Given a vertex $v \in V$, let $c(v)$ denote the contour to which v belongs.

The following two lemmas describe two important properties exhibited in the execution of a pipeline program.

LEMMA 1. *Only one vertex in a contour can belong to any extended deque at any time.*

PROOF. The vertices in a contour form a chain and are, therefore, enabled serially. □

The structure of a `pipe_while` guarantees that the "top-level" vertices of each iteration correspond to a contour, and that all iterations of the `pipe_while` share a common parent in the contour tree. These properties lead to the following lemma.

LEMMA 2. *If an edge (x,y) is a cross edge, then $c(x)$ and $c(y)$ are siblings in the contour tree and correspond to adjacent iterations in a `pipe_while` loop. If an edge (x,y) is a throttling edge, then $c(y)$ is the parent of $c(x)$ in contour tree.* □

As PIPER executes a pipeline program, the deques of workers are highly structured with respect to contours.

DEFINITION 3. *At any time during an execution of a pipeline program which produces a computation dag $G=(V,E)$, consider the extended deque $\langle v_0, v_1, \ldots, v_r\rangle$ of a worker p. This deque satisfies the **contour property** if for all $k=0,1,\ldots r-1$, one of the following two conditions holds:*

1. *$c(v_{k+1})$ is the parent of $c(v_k)$.*
2. *The root of $c(v_k)$ is the start of some iteration i, the root of $c(v_{k+1})$ is the start of the next iteration $i+1$, and if $k+2 \le r$, then $c(v_{k+2})$ is the common parent of both $c(v_k)$ and $c(v_{k+1})$.*

Contours allow us to prove an important property of the tail-swap operation.

LEMMA 4. *At any time during an execution of a pipeline program which produces a computation dag $G=(V,E)$, suppose that worker p enables a vertex x via a throttling edge as a result of executing its assigned vertex v_0. If p's deque satisfies the contour property (Definition 3), then either*

1. *p's deque is empty and x becomes p's new assigned vertex, or*
2. *p's deque contains a single vertex v_1 which becomes p's new assigned vertex and x is pushed onto p's deque.*

PROOF. Because x is enabled by a throttling edge, v_0 must be the last node of some iteration i, which by Lemma 2 means that $c(x)$ is the parent of $c(v_0)$. Because x is just being enabled, Lemma 1 implies that no other vertex in $c(x)$ can belong to p's deque. Suppose that p's extended deque $\langle v_0, v_1, \ldots, v_r\rangle$ contains $r \ge 2$ vertices. By Lemma 1, either v_1 or v_2 belongs to contour $c(x)$, neither of which is possible, and hence $r=0$ or $r=1$. If $r=0$, then x is p's assigned vertex. If $r=1$, then the root of $c(v_1)$ is the start of iteration $i+1$. Since x is enabled by a throttling edge, a tail swap occurs, making v_1 the assigned vertex of p and putting x onto p's deque. □

To analyze the time required for PIPER to execute a computation dag $G = (V, E)$, define the ***enabling tree*** $G_T = (V, E_T)$ as the tree containing an edge $(x, y) \in E_T$ if x is the last predecessor of y to execute. The ***enabling depth*** $d(x)$ of $x \in V$ is the depth of x in the enabling tree G_T.

DEFINITION 5. *At any time during an execution of a pipeline program which produces a computation dag $G = (V, E)$, consider the extended deque $\langle v_0, v_1, \ldots, v_r \rangle$ of a worker p. The deque satisfies the **depth property** if the following conditions hold:*

1. *For $k = 1, 2, \ldots, r-1$, we have $d(v_{k-1}) \geq d(v_k)$.*
2. *For $k = r$, we have $d(v_{k-1}) \geq d(v_k)$ or v_k has an incoming throttling edge.*
3. *The inequalities are strict for $k > 1$.*

THEOREM 6. *At all times during an execution of a pipeline program by* PIPER*, all deques satisfy the contour and depth properties (Definitions 3 and 5).*

PROOF. The proof is similar to the inductive proof of Lemma 3 from [2]. Intuitively, we replace the "designated parents" discussed in [2] with contours, which exhibit similar parent-child relationships. Although most of the proof follows from this substitution, we address the two most salient differences. The other minor cases are either straightforward or similar to these two cases.

First, we consider the consequences of the tail-swap operation, which may occur if the assigned vertex v_0 is the end of an iteration and executing v_0 enables a vertex z via a throttling edge. Lemma 4 describes the structure of a worker p's extended deque in this case, and in particular, states that the deque contains at most 1 vertex. If $r = 0$, the deque is empty and the properties hold vacuously. Otherwise, $r = 1$ and the deque contains one element v_1, in which case the tail-swap operation assigns v_1 to p and puts z into p's deque. The contour property holds, because $c(z)$ is the parent of $c(v_1)$. The depth property holds, because z is enabled by a throttling edge.

Second, we must show that the contour and depth properties hold when a worker p's assigned vertex v_0 belongs to some iteration i of a `pipe_while`, and executing v_0 enables a vertex y belonging to iteration $i+1$ via a cross edge (v_0, y). Assume that executing v_0 also enables x, where x also belongs to iteration i. (The case where x is not enabled is similar.) Since both v_0 and x belong to the same iteration, $c(v_0) = c(x)$, and by Lemma 2, $c(x)$ is a sibling of $c(y)$ in the contour tree. Suppose that before v_0 executes, p's extended deque is $\langle v_0, v_1, \ldots, v_r \rangle$, and thus after v_0 executes, p's extended deque is $\langle x, y, v_1, \ldots, v_r \rangle$. For vertices $v_2, v_3, \ldots, v_r$, if they exist, the conditions of the contour property continue to hold by induction. Since $c(x)$ and $c(y)$ are adjacent siblings in the contour tree, we need only show that $c(v_1)$, if it exists, is their parent. But if $c(v_1)$ is not the parent of $c(v_0) = c(x)$, then by induction it must be that $c(x)$ and $c(v_1)$ are adjacent siblings. In this case $c(v_1) = c(y)$, which is impossible by Lemma 1. The depth property holds because $d(x) = d(y) = d(v_0) + 1 \geq d(v_1) + 1 > d(v_1)$. □

7. TIME ANALYSIS OF PIPER

This section bounds the completion time for PIPER, showing that PIPER executes pipeline program asymptotically efficiently. Specifically, suppose that a pipeline program produces a computation dag $G = (V, E)$ with work T_1 and span T_∞ when executed by PIPER on P processors. We show that for any $\varepsilon > 0$, the running time is $T_P \leq T_1/P + O(T_\infty + \lg P + \lg(1/\varepsilon))$ with probability at least $1 - \varepsilon$, which implies that the expected running time is $\mathrm{E}[T_P] \leq T_1/P + O(T_\infty + \lg P)$. This bound is comparable to the work-stealing bound for fork-join dags originally proved in [8].

We adapt the potential-function argument of Arora, Blumofe, and Plaxton [2]. PIPER executes computation dags in a style similar to their work-stealing scheduler, except for tail swapping. Although Arora et al. ignore the issue of memory contention, we handle it using the "recycling game" analysis from [8], which contributes the additive $O(\lg P)$ term to the bounds.

As in [2], the crux of the proof is to bound the number of steal attempts performed during the execution of a computation dag G in terms of its span T_∞. We measure progress through the computation by defining a potential function for a vertex in the computation dag based on its depth in the enabling tree. Consider a particular execution of a computation dag $G = (V, E)$ by PIPER. For that execution, we define the ***weight*** of a vertex v as $w(v) = T_\infty - d(v)$, and we define the ***potential*** of vertex v at a given time as

$$\phi(v) = \begin{cases} 3^{2w(v)-1} & \text{if } v \text{ is assigned ,} \\ 3^{2w(v)} & \text{otherwise .} \end{cases}$$

We define the potential of a worker p's extended deque $\langle v_0, v_1, \ldots, v_r \rangle$ as $\phi(p) = \sum_{k=0}^{r} \phi(v_k)$.

Given this potential function, the proof of the time bound follows the same overall structure as the proof in [2]. We sketch the proof.

First, we prove two properties of worker deques involving the potential function.

LEMMA 7. *At any time during an execution of a pipeline program which produces a computation dag $G = (V, E)$, the extended deque $\langle v_0, v_1, \ldots, v_r \rangle$ of every worker p satisfies the following:*

1. $\phi(v_r) + \phi(v_{r-1}) \geq 3\phi(p)/4$.
2. *Let ϕ' denote the potential after p executes v_0. Then we have $\phi(p) - \phi'(p) = 2(\phi(v_0) + \phi(v_1))/3$, if p performs a tail swap, and $\phi(p) - \phi'(p) \geq 5\phi(v_0)/9$ otherwise.*

PROOF. Property 1 follows from the depth property of Theorem 6. Property 2 follows from Lemma 4, if p performs a tail swap, and the analysis in [2] otherwise. □

As in [2], we analyze the behavior of workers randomly stealing from each other using a balls-and-weighted-bins analog. We want to analyze the case where the top 2 elements are stolen out of any deque, however, not just the top element. To address this case, we modify Lemma 7 of [2] to consider the probability that 2 out of $2P$ balls land in the same bin.

LEMMA 8. *Consider P bins, where for $p = 1, 2, \ldots, P$, bin p has weight W_p. Suppose that $2P$ balls are thrown independently and uniformly at random into the P bins. For bin p, define the random variable X_p as*

$$X_p = \begin{cases} W_p & \textit{if at least 2 balls land in bin } p \textit{ ,} \\ 0 & \textit{otherwise .} \end{cases}$$

Let $W = \sum_{p=1}^{P} W_p$ and $X = \sum_{p=1}^{P} X_p$. For any β in the range $0 < \beta < 1$, we have $\Pr\{X \geq \beta W\} > 1 - 3/(1-\beta)e^2$.

PROOF. For each bin p, consider the random variable $W_p - X_p$. It takes on the value W_p when 0 or 1 ball lands in bin p, and otherwise it is 0. Thus, we have

$$\begin{aligned} \mathrm{E}\left[W_p - X_p\right] &= W_p\left((1-1/P)^{2P} + 2P(1-1/P)^{2P-1}(1/P)\right) \\ &= W_p(1-1/P)^{2P}(3P-1)/(P-1) \ . \end{aligned}$$

Since $(1-1/P)^P$ approaches $1/e$ and $(3P-1)/(P-1)$ approaches 3, we have $\lim_{P\to\infty} \mathrm{E}\left[W_p - X_p\right] = 3W_p/e^2$. In fact, one can show that $\mathrm{E}\left[W_p - X_p\right]$ is monotonically increasing, approaching the limit

from below, and thus $\mathrm{E}[W-X] \leq 3W/e^2$. By Markov's inequality, we have that $\Pr\{(W-X) > (1-\beta)W\} < \mathrm{E}[W-X]/(1-\beta)W$, from which we conclude that $\Pr\{X < \beta W\} \leq 3/(1-\beta)e^2$. □

To use Lemma 8 to analyze PIPER, we divide the time steps of the execution of G into a sequence of ***rounds***, where each round (except the first, which starts at time 0) starts at the time step after the previous round ends and continues until the first time step such that at least $2P$ steal attempts — and hence less than $3P$ steal attempts — occur within the round. The following lemma shows that a constant fraction of the total potential in all deques is lost in each round, thereby demonstrating progress.

LEMMA 9. *Consider a pipeline program executed by* PIPER *on P processors. Suppose that a round starts at time step t and finishes at time step t'. Let ϕ denote the potential at time t, let ϕ' denote the potential at time t', let $\Phi = \sum_{p=1}^{P} \phi(p)$, and let $\Phi' = \sum_{p=1}^{P} \phi'(p)$. Then we have* $\Pr\{\Phi - \Phi' \geq \Phi/4\} > 1 - 6/e^2$.

PROOF. We first show that stealing twice from a worker p's deque contributes a potential drop of at least $\phi(p)/2$. The proof follows a similar case analysis to that in the proof of Lemma 8 in [2] with two main differences. First, we use the two properties of ϕ in Lemma 7. Second, we must consider the case unique to PIPER, where p performs a tail swap after executing its assigned vertex v_0. In this case, p's deque contains a single ready vertex v_1 and p may perform a tail swap if executing v_0 enables a vertex via an outgoing throttling edge. If so, however, then by Lemma 7, the potential drops by at least $2(\phi(v_0) + \phi(v_1))/3 > \phi(p)/2$, since $\phi(p) = \phi(v_0) + \phi(v_1)$.

Now, suppose that we assign each worker p a weight of $W_p = \phi(p)/2$. These weights W_p sum to $W = \Phi/2$. If we think of steal attempts as ball tosses, then the random variable X from Lemma 8 bounds from below the potential decrease due to actions on p's deque. Specifically, if at least 2 steal attempts target p's deque in a round (which corresponds conceptually to at least 2 balls landing in bin p), then the potential drops by at least W_p. Moreover, X is a lower bound on the potential decrease within the round, i.e., $X \leq \Phi - \Phi'$. By Lemma 8, we have $\Pr\{X \geq W/2\} > 1 - 6/e^2$. Substituting for X and W, we conclude that $\Pr\{(\Phi - \Phi') \geq \Phi/4\} > 1 - 6/e^2$. □

We are now ready to prove the completion-time bound.

THEOREM 10. *Consider an execution of a pipeline program by* PIPER *on P processors which produces a computation dag with work T_1 and span T_∞. For any $\varepsilon > 0$, the running time is $T_P \leq T_1/P + O(T_\infty + \lg P + \lg(1/\varepsilon))$ with probability at least $1 - \varepsilon$.*

PROOF. On every time step, consider each worker as placing a token in a bucket depending on its action. If a worker p executes an assigned vertex, p places a token in the ***work bucket***. Otherwise, p is a thief and places a token in the ***steal bucket***. There are exactly T_1 tokens in the work bucket at the end of the computation. The interesting part is bounding the size of the steal bucket.

Divide the execution of G into rounds. Recall that each round contains at least $2P$ and less than $3P$ steal attempts. Call a round ***successful*** if after that round finishes, the potential drops by at least a $1/4$ fraction. From Lemma 9, a round is successful with probability at least $1 - 6/e^2 \geq 1/6$. Since the potential starts at $\Phi_0 = 3^{2T_\infty - 1}$, ends at 0, and is always an integer, the number of successful rounds is at most $(2T_\infty - 1)\log_{4/3}(3) < 8T_\infty$. Consequently, the expected number of rounds needed to obtain $8T_\infty$ successful rounds is at most $48T_\infty$, and the expected number of tokens in the steal bucket is therefore at most $3P \cdot 48T_\infty = 144PT_\infty$.

For the high-probability bound, suppose that the execution takes $n = 48T_\infty + m$ rounds. Because each round succeeds with probability at least $p = 1/6$, the expected number of successes is at least $np = 8T_\infty + m/6$. We now compute the probability that the number X of successes is less than $8T_\infty$. As in [2], we use the Chernoff bound $\Pr\{X < np - a\} < e^{-a^2/2np}$, with $a = m/6$. Choosing $m = 48T_\infty + 21\ln(1/\varepsilon)$, we have

$$\Pr\{X < 8T_\infty\} < e^{\frac{-(m/6)^2}{16T_\infty + m/3}} < e^{\frac{-(m/6)^2}{m/4 + m/3}} = e^{-m/21} \leq \varepsilon .$$

Hence, the probability that the execution takes $n = 96T_\infty + 21\ln(1/\varepsilon)$ rounds or more is less than ε, and the number of tokens in the steal bucket is at most $288T_\infty + 63\ln(1/\varepsilon)$.

The additional $\lg P$ term comes from the "recycling game" analysis described in [8], which bounds any delay that might be incurred when multiple processors try to access the same deque in the same time step in randomized work-stealing. □

8. SPACE ANALYSIS OF PIPER

This section derives bounds on the stack space required by PIPER by extending the bounds in [8] for fully strict fork-join parallelism to include pipeline parallelism. We show that PIPER on P processors uses $S_P \leq P(S_1 + fDK)$ stack space for pipeline iterations, where S_1 is the serial stack space, f is the "frame size," D is the depth of nested linear pipelines, and K is the throttling limit.

To model PIPER's usage of stack space, we partition the vertices of the computation dag G of the pipeline program into a tree of contours, as described in Section 6. Each contour in this partition is rooted at the start of a spawned subcomputation. The control for each `pipe_while` loop, which corresponds to a `while` loop as in line 5 of Figure 5, belongs to some contour in the contour tree with its iterations as children. Define the ***pipe nesting depth*** D of G as the maximum number of `pipe_while` contours on any path from leaf to root in the contour tree.

We assume that every contour c of G has an associated ***frame size*** representing the stack space consumed by c while it or any of its descendant contours are executing. The space used by PIPER on any time step is the sum of frame sizes of all contours c which are either (1) associated with a vertex in some worker's extended deque, or (2) ***suspended***, meaning that the earliest unexecuted vertex in the contour is not ready. Let S_P denote the maximum over all time steps of the stack space used by PIPER during a P-worker execution of G. Thus, S_1 is the stack space used by PIPER for a serial execution. We now generalize the space bound $S_P \leq PS_1$ from [8], which deals only with fork-join parallelism, to pipeline programs.

THEOREM 11. *Consider a pipeline program with pipe nesting depth D executed on P processors by* PIPER *with throttling limit K. The execution requires $S_P \leq P(S_1 + fDK)$ stack space, where f is the maximum frame size of any contour of any* `pipe_while` *iteration and S_1 is the serial stack space.*

PROOF. We show that except for suspended contours that are `pipe_while` iterations, PIPER still satisfies the "busy-leaves property" [8]. More precisely, at any point during the execution, in the tree of active and suspended contours, each leaf contour either (1) is currently executing on some worker, or (2) is a suspended `pipe_while` iteration with a sibling iteration that is currently executing on some worker. In fact, one can show that for any `pipe_while` loop, the contour for the leftmost (smallest) iteration that has not completed is either active or has an active descendant in the contour tree. The bound of PS_1 covers the space used by all contours that fall into Case (1).

To bound the space used by contours from Case (2), observe that any `pipe_while` loop uses at most fK space for iteration contours, since the throttling edge from the leftmost active iteration precludes having more than K active or suspended iterations in any one `pipe_while` loop. Thus, each worker p has at most fDK iteration contours for any `pipe_while` loop that is an ancestor of the contour p's assigned vertex. Summing the space used over all workers gives $PfDK$ additional stack-space usage. □

9. CILK-P RUNTIME DESIGN

This section describes the Cilk-P implementation of the PIPER scheduler. We first introduce the data structures Cilk-P uses to implement a `pipe_while` loop. Then we describe the two main optimizations that the Cilk-P runtime exploits: lazy enabling and dependency folding.

Data structures

Like the Cilk-M runtime [23] on which it is based, Cilk-P organizes runtime data into frames. Cilk-P executes a `pipe_while` loop in its own function, whose frame, called a ***control frame***, handles the spawning and throttling of iterations. Furthermore, each iteration of a `pipe_while` loop executes as an independent child function, with its own ***iteration frame***. This frame structure is similar to that of an ordinary `while` loop in Cilk-M, where each iteration spawns a function to execute the loop body. Cross and throttling edges, however, may cause the iteration and control frames to suspend.

Cilk-P's runtime employs a simple mechanism to track progress of an iteration i. The frame of iteration i maintains a ***stage counter***, which stores the stage number of the current node in i, and a ***status*** field, which indicates whether i is suspended due to an unsatisfied cross edge. Because executed nodes in an iteration i have strictly increasing stage numbers, checking whether a cross edge into iteration i is satisfied amounts to comparing the stage counters of iterations i and $i-1$. Any iteration frame that is not suspended corresponds to either a currently executing or a completed iteration.

Cilk-P implements throttling using a ***join counter*** in the control frame. Normally in Cilk-M, a frame's join counter simply stores the number of active child frames. Cilk-P also uses the join counter to limit the number of active iteration frames in a `pipe_while` loop to the throttling limit K. Starting an iteration increments the join counter, while returning from an iteration decrements it. If a worker tries to start a new iteration when the control frame's join counter is K, the control frame suspends until a child iteration returns.

Using these data structures, one could implement PIPER directly, by pushing and popping the appropriate frames onto deques as specified by PIPER's execution model. In particular, the normal THE protocol [15] could be used for pushing and popping frames from a deque, and frame locks could be used to update fields in the frames atomically. Although this approach directly matches the model analyzed in Sections 7 and 8, it incurs unnecessary overhead for every node in an iteration. Cilk-P implements lazy enabling and dependency folding to reduce this overhead.

Lazy enabling

In the PIPER algorithm, when a worker p finishes executing a node in iteration i, it may enable an instruction in iteration $i+1$, in which case p pushes this instruction onto its deque. To implement this behavior, intuitively, p must ***check right*** — read the stage counter and status of iteration $i+1$ — whenever it finishes executing a node. The work to check right at the end of every node could amount to substantial overhead in a pipeline with fine-grained stages.

Lazy enabling allows p's execution of an iteration i to defer the check-right operation, as well as avoid any operations on its deque involving iteration $i+1$. Conceptually, when p enables work in iteration $i+1$, this work is kept on p's deque implicitly. When a thief p' tries to steal iteration i's frame from p's deque, p' first checks right on behalf of p to see whether any work from iteration $i+1$ is implicitly on the deque. If so, p' resumes iteration $i+1$ as if it had found it on p's deque. In a similar vein, the Cilk-P runtime system also uses lazy enabling to optimize the ***check-parent*** operation — the enabling of a control frame suspended due to throttling.

Lazy enabling requires p to behave differently when p completes an iteration. When p finishes iteration i, it first checks right, and if that fails (i.e., iteration $i+1$ need not be resumed), it checks its parent. It turns out that these checks find work only if p's deque is empty. Therefore, p can avoid performing these checks at the end of an iteration if its deque is not empty.

Lazy enabling is an application of the ***work-first principle*** [15]: minimize the scheduling overheads borne by the work of a computation, and amortize them against the span. Requiring a worker to check right every time it completes a node adds overhead proportional to the work of the `pipe_while` in the worst case. With lazy enabling, the overhead can be amortized against the span of the computation. For programs with sufficient parallelism, the work dominates the span, and the overhead becomes negligible.

Dependency folding

In ***dependency folding***, the frame for iteration i stores a cached value of the stage counter of iteration $i-1$, hoping to avoid the checking of already satisfied cross edges. In a straightforward implementation of PIPER, before a worker p executes each node in iteration i with an incoming cross edge, it reads the stage counter of iteration $i-1$ to see if the cross edge is satisfied. Reading the stage counter of iteration $i-1$, however, can be expensive. Besides the work involved, the access may contend with whatever worker p' is executing iteration $i-1$, because p' may be constantly updating the stage counter of iteration $i-1$.

Dependency folding mitigates this overhead by exploiting the fact that an iteration's stage counter must strictly increase. By caching the most recently read stage-counter value from iteration $i-1$, worker p can sometimes avoid reading this stage counter before each node with an incoming cross edge. For instance, if p' finishes executing a node $(i-1,j)$, then all cross edges from nodes $(i-1,0)$ through $(i-1,j)$ are necessarily satisfied. Thus, if p reads j from iteration $i-1$'s stage counter, p need not reread the stage counter of $i-1$ until it tries to execute a node with an incoming cross edge (i,j') where $j' > j$. This optimization is particularly useful for fine-grained stages that execute quickly.

10. EVALUATION

This section presents empirical studies of the Cilk-P prototype system. We investigated the performance and scalability of Cilk-P using the three PARSEC [4, 5] benchmarks that we ported, namely *ferret*, *dedup*, and *x264*. The results show that Cilk-P's implementation of pipeline parallelism has negligible overhead compared to its serial counterpart. We compared the Cilk-P implementations to TBB and Pthreaded implementations of these benchmarks. We found that the Cilk-P and TBB implementations perform comparably, as do the Cilk-P and Pthreaded implementations for *ferret* and *x264*. The Pthreaded version of *dedup* outperforms both Cilk-P and TBB, because the bind-to-element approaches of Cilk-P and TBB produce less parallelism than the Pthreaded bind-to-stage approach. Moreover, the Pthreading approach benefits more from "oversubscription." We study the effectiveness of dependency folding on a

P	Processing Time (T_P) Cilk-P	Pthreads	TBB	Speedup (T_S/T_P) Cilk-P	Pthreads	TBB	Scalability (T_1/T_P) Cilk-P	Pthreads	TBB
1	691.2	692.1	690.3	1.00	1.00	1.00	1.00	1.00	1.00
2	356.7	343.8	351.5	1.94	2.01	1.97	1.94	2.01	1.96
4	176.5	170.4	175.8	3.92	4.06	3.93	3.92	4.06	3.93
8	89.1	86.8	89.2	7.76	7.96	7.75	7.76	7.97	7.74
12	60.3	59.1	60.8	11.46	11.70	11.36	11.46	11.71	11.35
16	46.2	46.3	46.9	14.98	14.93	14.75	14.98	14.95	14.74

Figure 6: Performance comparison of the three *ferret* implementations. The experiments were conducted using *native*, the largest input data set that comes with the PARSEC benchmark suite.[4] The left-most column shows the number of cores used (P). Subsequent columns show the running time (T_P), speedup over serial running time (T_S/T_P), and scalability (T_1/T_P) for each system. The throttling limit was $K = 10P$.

P	Processing Time (T_P) Cilk-P	Pthreads	TBB	Speedup (T_S/T_P) Cilk-P	Pthreads	TBB	Scalability (T_1/T_P) Cilk-P	Pthreads	TBB
1	58.0	51.1	57.3	1.01	1.05	1.00	1.00	1.00	1.00
2	29.8	23.3	29.4	1.96	2.51	1.99	1.94	2.19	1.95
4	16.0	12.2	16.2	3.66	4.78	3.61	3.63	4.18	3.54
8	10.4	8.2	10.3	5.62	7.08	5.70	5.57	6.20	5.58
12	9.0	6.6	9.0	6.53	8.83	6.57	6.47	7.72	6.44
16	8.6	6.0	8.6	6.77	9.72	6.78	6.71	8.50	6.65

Figure 7: Performance comparison of the three *dedup* implementations. The experiments were conducted using *native*, the largest input data set that comes with the PARSEC benchmark suite. The column headers are the same as in Figure 6. The throttling limit was $K = 4P$.

P	Encoding Time (T_P) Cilk-P	Pthreads	Speedup (T_S/T_P) Cilk-P	Pthreads	Scalability (T_1/T_P) Cilk-P	Pthreads
1	217.1	223.0	1.02	0.99	1.00	1.00
2	97.0	105.3	2.27	2.09	2.24	2.12
4	47.7	53.3	4.63	4.14	4.55	4.19
8	25.9	26.7	8.53	8.27	8.40	8.36
12	18.6	19.3	11.84	11.44	11.66	11.57
16	16.0	16.2	13.87	13.63	13.66	13.76

Figure 8: Performance comparison between the Cilk-P implementation and the Pthreaded implementation of *x264* (encoding only). The experiments were conducted using *native*, the largest input data set that comes with the PARSEC benchmark suite. The column headers are the same as in Figure 6. The throttling limit was $K = 4P$.

synthetic benchmark called *pipe-fib*, demonstrating that this optimization can be effective for applications with fine-grained stages.

We ran all experiments on an AMD Opteron system with 4 2 GHz quad-core CPU's having a total of 8 GBytes of memory. Each processor core has a 64-KByte private L1-data-cache and a 512-KByte private L2-cache. The 4 cores on each chip share the same 2-MByte L3-cache. The benchmarks were compiled with GCC (or G++ for TBB) 4.4.5 using `-O3` optimization, except for *x264*, which by default comes with `-O4`.

Performance evaluation on PARSEC benchmarks

We implemented the Cilk-P versions of the three PARSEC benchmarks by hand-compiling the relevant `pipe_while` loops using techniques similar to those described in [23]. We then compiled the hand-compiled benchmarks with GCC. The *ferret* and *dedup* applications can be parallelized as simple pipelines with a fixed number of stages and a static dependency structure. In particular, *ferret* uses the 3-stage SPS pipeline shown in Figure 1, while *dedup* uses a 4-stage SSPS pipeline as described in Figure 4.

For the Pthreaded versions, we used the code distributed with PARSEC. The PARSEC Pthreaded implementations of *ferret* and *dedup* employ the ***oversubscription method*** [33], a bind-to-stage approach that creates more than one thread per pipeline stage and utilizes the operating system for load balancing. For the Pthreaded implementations, when the user specifies an input parameter of Q, the code creates Q threads per stage, except for the first (input) and last (output) stages which are serial and use only one thread each. To ensure a fair comparison, for all applications, we ran the Pthreaded implementation using `taskset` to limit the process to P cores (which corresponds to the number of workers used in Cilk-P and TBB), but experimented to find the best setting for Q.

We used the TBB version of *ferret* that came with the PARSEC benchmark, and implemented the TBB version of *dedup*, both using the same strategies as for Cilk-P. TBB's construct-and-run approach proved inadequate for the on-the-fly nature of *x264*, however, and indeed, in their study of these three applications, Reed, Chen, and Johnson [33] say, "Implementing *x264* in TBB is not impossible, but the TBB pipeline structure is not suitable." Thus, we had no TBB benchmark for *x264* to include in our comparisons.

For each benchmark, we throttled all versions similarly. For Cilk-P, a throttling limit of $4P$, where P is the number of cores, seems to work well in general, although since *ferret* scales slightly better with less throttling, we used a throttling limit of $10P$ for our experiments. TBB supports a settable parameter that serves the same purpose as Cilk-P's throttling limit. For the Pthreaded implementations, we throttled the computation by setting a size limit on the queues between stages, although we did not impose a queue size limit on the last stage of *dedup* (the default limit is 2^{20}), since the program deadlocks otherwise.

Figures 6–8 show the performance results for the different implementations of the three benchmarks. Each data point in the study was computed by averaging the results of 10 runs. The standard deviation of the numbers was typically just a few percent, indicating that the numbers should be accurate to within better than 10 percent with high confidence (2 or 3 standard deviations). We suspect that the superlinear scalability obtained for some measurements is due to the fact that more L1- and L2-cache is available when running on multiple cores.

The three tables from Figures 6–8 show that the Cilk-P and TBB implementations of *ferret* and *dedup* are comparable, indicating that there is no performance penalty incurred by these applications for using the more general on-the-fly pipeline instead of a construct-and-run pipeline. Both Cilk-P and TBB execute using a bind-to-element approach.

The *dedup* performance results for Cilk-P and TBB are inferior to those for Pthreads, however. The Pthreaded implementation scales to about 8.5 on 16 cores, whereas Cilk-P and TBB seem to plateau at around 6.7. There appear to be two reasons for this discrepancy.

First, the *dedup* benchmark on the test input has limited parallelism. We modified the Cilkview scalability analyzer [19] to measure the work and span of our hand-compiled Cilk-P *dedup* programs, observing a parallelism of merely 7.4. The bind-to-stage Pthreaded implementation creates a pipeline with a different structure from the bind-to-element Cilk-P and TBB versions, which enjoys slightly more parallelism.

Second, since file I/O is the main performance bottleneck for *dedup*, the Pthreaded implementation effectively benefits from ***oversubscription*** — using more threads than processing cores — and its strategic allocation of threads to stages. Specifically, since the first and last stages perform file I/O, which is inherently serial, the Pthreaded implementation dedicates one thread to each of these stages, but dedicates multiple threads to the other compute-intensive stages. While the writing thread is performing file I/O (i.e., writing data out to the disk), the OS may deschedule it, allowing the compute-intensive threads to be scheduled. This behavior explains how the Pthreaded implementation scales by more than a

[4] We dropped four out of the 3500 input images from the original *native* data set, because those images are black-and-white, which trigger an array index out of bound error in the image library provided.

Program	Dependency Folding	T_S	T_1	T_{16}	Serial Overhead	Speedup T_S/T_{16}	Scalability T_1/T_{16}
pipe-fib	no	20.8	22.3	3.8	1.07	5.15	5.82
pipe-fib-256	no	20.8	20.9	1.7	1.01	12.26	12.31
pipe-fib	yes	20.8	21.7	1.8	1.05	11.85	12.40
pipe-fib-256	yes	20.8	20.9	1.7	1.01	12.56	12.62

Figure 9: Performance evaluation using the *pipe-fib* benchmark. We tested the system with two different programs, the ordinary *pipe-fib*, and the *pipe-fib-256*, which is coarsened. Each program is tested with and without the dependency folding optimization. For each program for a given setting, we show the running time of its serial counter part (T_S), running time executing on a single worker (T_1), on 16 workers (T_{16}), its serial overhead, scalability, and speedup obtained running on 16 workers.

factor of P for $P = 2$ and 4, even though the computation is restricted to only P cores using `taskset`. Moreover, when we ran the Pthreaded implementation without throttling on a single core, the computation ran about 20% faster than the serial implementation, which makes sense if computation and file I/O are effectively overlapped. With multiple threads per stage, throttling appears to inhibit threads working on stages that are further ahead, allowing threads working on heavier stages to obtain more processing resources, and thereby balancing the load.

In summary, Cilk-P performs comparably to TBB while admitting more expressive semantics for pipelines. Cilk-P also performs comparably to the Pthreaded implementations of *ferret* and *x264*, although its bind-to-element strategy seems to suffer on *dedup* compared to the bind-to-stage strategy of the Pthreaded implementation. Despite losing the *dedup* "bake-off," Cilk-P's strategy has the significant advantage that it allows pipelines to be expressed as deterministic programs. Determinism greatly reduces the effort for debugging, release engineering, and maintenance (see, for example, [9]) compared with the inherently nondeterministic code required to set up Pthreaded pipelines.

Evaluation of dependency folding

We also studied the effectiveness of dependency folding. Since the PARSEC benchmarks are too coarse grained to permit such a study, we implemented a synthetic benchmark, called *pipe-fib*, to study this optimization technique. The *pipe-fib* benchmark computes the nth Fibonacci number F_n in binary. It uses a pipeline algorithm that operates in $\Theta(n^2)$ work and $\Theta(n)$ span. To construct the base case, *pipe-fib* allocates three arrays of size $\Theta(n)$ and initializes the first two arrays with the binary representations of F_1 and F_2, both of which are 1. To compute F_3, *pipe-fib* performs a ripple-carry addition on the two input arrays and stores the sum into the third output array. To compute F_n, *pipe-fib* repeats the addition by rotating through the arrays for inputs and output until it reaches F_n. In the pipeline for this computation, each iteration i computes F_{i+2}, and a stage j within the iteration computes the jth bit of F_{i+2}. Since the benchmark stops propagating the carry bit as soon as possible, it generates a triangular pipeline dag in which the number of stages increases with iteration number.

Figure 9 shows the performance results[5] obtained by running the ordinary *pipe-fib* with fine-grained stages, as well as *pipe-fib-256*, a coarsened version of *pipe-fib* in which each stage computes 256 bits instead of 1. As the data in the first row show, even though the serial overhead for *pipe-fib* without coarsening is merely 7%, it fails to scale and exhibits poor speedup. The reason is that checking for dependencies due to cross edges has a relatively high overhead compared to the little work in each fine-grained stage. As the data for *pipe-fib-256* in the second row show, coarsening the stages improves both serial overhead and scalability. Ideally, one would like the system to coarsen automatically, which is what dependency folding effectively does.

[5] Figure 9 shows the results from a single run, but these data are representative of other runs with different input sizes.

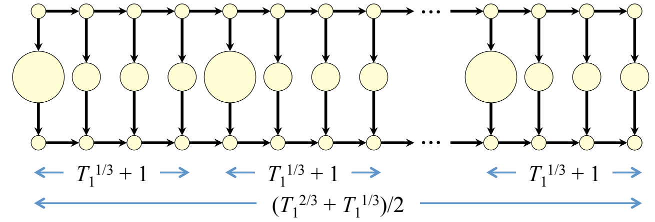

Figure 10: Sketch of the pathological unthrottled linear pipeline dag, which can be used to prove Theorem 13. Small circles represent nodes with unit work, medium circles represent nodes with $T_1^{1/3} - 2$ work, and large circles represent nodes with $T_1^{2/3} - 2$ work. The number of iterations per cluster is $T_1^{1/3} + 1$, and the total number of iterations is $(T_1^{2/3} + T_1^{1/3})/2$.

Further investigation revealed that the time spent checking for cross edges increases noticeably when the number of workers increases from 1 to 2. It turns out that when iterations are run in parallel, each check for a cross-edge dependency necessarily incurs a true-sharing conflict between the two adjacent active iterations, an overhead that occurs only during parallel execution. Dependency folding eliminated much of this overhead for *pipe-fib*, as shown in the third row of Figure 9, leading to scalability that exceeds the coarsened version without the optimization, although a slight price is still paid in speedup. Employing both optimizations, as shown in the last row of the table, produces the best numbers for both speedup and scalability.

11. CONCLUSION

What impact does throttling have on theoretical performance? PIPER relies on throttling to achieve its provable space bound and avoid runaway pipelines. Ideally, the user should not worry about throttling, and the system should perform well automatically, and indeed, PIPER's throttling of a pipeline computation is encapsulated in Cilk-P's runtime system. But what price is paid?

We can pose this question theoretically in terms of a pipeline computation G's ***unthrottled dag***: the dag $\widehat{G} = (V, \widehat{E})$ with the same vertices and edges as G, except without throttling edges. How does adding throttling edges to an unthrottled dag affect span and parallelism?

The following two theorems provide two partial answers to this question. First, for ***uniform*** pipelines, where the cost of a node (i, j) is identical across all iterations i — all stages have the same cost — throttling does not affect the asymptotic performance of PIPER executing $\widehat{G}$.

THEOREM 12. *Consider a uniform unthrottled linear pipeline $\widehat{G} = (V, \widehat{E})$ having n iterations and s stages. Suppose that* PIPER *throttles the execution of $\widehat{G}$ on P processors using a window size of $K = aP$, for some constant $a > 1$. Then* PIPER *executes $\widehat{G}$ in time $T_P \le (1 + c/a)T_1/P + cT_\infty$ for some sufficiently large constant c, where T_1 is the total work in $\widehat{G}$ and T_∞ is the span of $\widehat{G}$.* □

Second, we consider ***nonuniform*** pipelines, where the cost of a node (i, j) may vary across iterations. It turns out that nonuniform pipelines can pose performance problems, not only for PIPER, but for any scheduler that throttles the computation. Figure 10 illustrates a pathological nonuniform pipeline for any scheduler that uses throttling. In this dag, T_1 work is distributed across $(T_1^{1/3} + T_1^{2/3})/2$ iterations such that any $T_1^{1/3} + 1$ consecutive iterations consist of 1 ***heavy*** iteration with $T_1^{2/3}$ work and $T_1^{1/3}$ ***light*** iterations of $T_1^{1/3}$ work each. Intuitively, achieving a speedup of 3 on this dag requires having at least 1 heavy iteration and $\Theta(T_1^{1/3})$

light iterations active simultaneously, which is impossible for any scheduler that uses a throttling limit of $K = o(T_1^{1/3})$. The following theorem formalizes this intuition.

THEOREM 13. *Let $\widehat{G} = (V, \widehat{E})$ denote the nonuniform unthrottled linear pipeline shown in Figure 10, with work T_1 and span $T_\infty \le 2T_1^{2/3}$. Let S_1 denote the optimal stack-space usage when $\widehat{G}$ is executed on* 1 *processor. Any P-processor execution of $\widehat{G}$ that achieves $T_P \le T_1/\rho$, where ρ satisfies $3 \le \rho \le O(T_1/T_\infty)$, uses space $S_P \ge S_1 + (\rho - 3)T_1^{1/3}/2 - 1$.* □

Intuitively, these two theorems present two extremes of the effect of throttling on pipeline dags. One interesting avenue for research is to determine what are the minimum restrictions on the structure of an unthrottled linear pipeline G that would allow a scheduler to achieve parallel speedup on P processors using a throttling limit of only $\Theta(P)$.

12. ACKNOWLEDGMENTS

Thanks to Loren Merritt of x264 LLC and Hank Hoffman of University of Chicago (formerly of MIT CSAIL) for answering questions about *x264*. Thanks to Yungang Bao of Institute of Computing Technology, Chinese Academy of Sciences (formerly of Princeton) for answering questions about the PARSEC benchmark suite. Thanks to Bradley Kuszmaul of MIT CSAIL for tips and insights on file I/O related performance issues. Thanks to Arch Robison of Intel for providing constructive feedback on an early draft of this paper. Thanks to Will Hasenplaugh of MIT CSAIL and Nasro Min-Allah of COMSATS Institute of Information Technology in Pakistan for helpful discussions. We especially thank the reviewers for their thoughtful comments.

13. REFERENCES

[1] K. Agrawal, C. E. Leiserson, and J. Sukha. Executing task graphs using work-stealing. In *IPDPS*, pp. 1–12. IEEE, 2010.

[2] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory of Computing Systems*, pp. 115–144, 2001.

[3] H. C. Baker, Jr. and C. Hewitt. The incremental garbage collection of processes. *SIGPLAN Notices*, 12(8):55–59, 1977.

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

[5] C. Bienia and K. Li. Characteristics of workloads using the pipeline programming model. In *ISCA*, pp. 161–171. Springer-Verlag, 2010.

[6] G. E. Blelloch and M. Reid-Miller. Pipelining with futures. In *SPAA*, pp. 249–259. ACM, 1997.

[7] R. D. Blumofe and C. E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM Journal on Computing*, 27(1):202–229, Feb. 1998.

[8] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *JACM*, 46(5):720–748, 1999.

[9] R. L. Bocchino, Jr., V. S. Adve, S. V. Adve, and M. Snir. Parallel programming must be deterministic by default. In *First USENIX Conference on Hot Topics in Parallelism*, 2009.

[10] F. W. Burton and M. R. Sleep. Executing functional programs on a virtual tree of processors. In *FPCA*, pp. 187–194. ACM, 1981.

[11] C. Consel, H. Hamdi, L. Réveillère, L. Singaravelu, H. Yu, and C. Pu. Spidle: a DSL approach to specifying streaming applications. In *GPCE*, pp. 1–17. Springer-Verlag, 2003.

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, third edition, 2009.

[13] R. Finkel and U. Manber. DIB — A distributed implementation of backtracking. *ACM TOPLAS*, 9(2):235–256, 1987.

[14] D. Friedman and D. Wise. Aspects of applicative programming for parallel processing. *IEEE Transactions on Computers*, C-27(4):289–296, 1978.

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

[16] J. Giacomoni, T. Moseley, and M. Vachharajani. FastForward for efficient pipeline parallelism: A cache-optimized concurrent lock-free queue. In *PPoPP*, pp. 43–52. ACM, 2008.

[17] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *ASPLOS*, pp. 151–162. ACM, 2006.

[18] E. A. Hauck and B. A. Dent. Burroughs' B6500/B7500 stack mechanism. *Proceedings of the AFIPS Spring Joint Computer Conference*, pp. 245–251, 1968.

[19] Y. He, C. E. Leiserson, and W. M. Leiserson. The Cilkview scalability analyzer. In *SPAA*, pp. 145–156, 2010.

[20] Intel Corporation. *Intel® Cilk™ Plus Language Extension Specification, Version 1.1*, 2013. Document 324396-002US. Available from http://cilkplus.org/sites/default/files/open_specifications/Intel_Cilk_plus_lang_spec_2.htm.

[21] D. A. Kranz, R. H. Halstead, Jr., and E. Mohr. Mul-T: A high-performance parallel Lisp. In *PLDI*, pp. 81–90. ACM, 1989.

[22] M. Lam. Software pipelining: an effective scheduling technique for VLIW machines. In *PLDI*, pp. 318–328. ACM, 1988.

[23] I.-T. A. Lee, S. Boyd-Wickizer, Z. Huang, and C. E. Leiserson. Using memory mapping to support cactus stacks in work-stealing runtime systems. In *PACT*, pp. 411–420. ACM, 2010.

[24] C. E. Leiserson. The Cilk++ concurrency platform. *J. Supercomputing*, 51(3):244–257, 2010.

[25] S. MacDonald, D. Szafron, and J. Schaeffer. Rethinking the pipeline as object-oriented states with transformations. In *HIPS*, pp. 12 – 21. IEEE, 2004.

[26] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard. Cg: a system for programming graphics hardware in a C-like language. In *SIGGRAPH*, pp. 896–907. ACM, 2003.

[27] M. McCool, A. D. Robison, and J. Reinders. *Structured Parallel Programming: Patterns for Efficient Computation*. Elsevier, 2012.

[28] A. Navarro, R. Asenjo, S. Tabik, and C. Caşcaval. Analytical modeling of pipeline parallelism. In *PACT*, pp. 281–290. IEEE, 2009.

[29] *OpenMP Application Program Interface, Version 3.0*, 2008. Available from http://www.openmp.org/mp-documents/spec30.pdf.

[30] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic thread extraction with decoupled software pipelining. In *MICRO*, pp. 105–118. IEEE, 2005.

[31] A. Pop and A. Cohen. A stream-computing extension to OpenMP. In *HiPEAC*, pp. 5–14. ACM, 2011.

[32] R. Rangan, N. Vachharajani, M. Vachharajani, and D. I. August. Decoupled software pipelining with the synchronization array. In *PACT*, pp. 177–188. ACM, 2004.

[33] E. C. Reed, N. Chen, and R. E. Johnson. Expressing pipeline parallelism using TBB constructs: a case study on what works and what doesn't. In *SPLASH*, pp. 133–138. ACM, 2011.

[34] R. Rojas. Konrad Zuse's legacy: The architecture of the Z1 and Z3. *IEEE Annals of the History of Computing*, 19(2):5–16, Apr. 1997.

[35] D. Sanchez, D. Lo, R. M. Yoo, J. Sugerman, and C. Kozyrakis. Dynamic fine-grain scheduling of pipeline parallelism. In *PACT*, pp. 22–32. IEEE, 2011.

[36] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, fourth edition, 2013.

[37] M. A. Suleman, M. K. Qureshi, Khubaib, and Y. N. Patt. Feedback-directed pipeline parallelism. In *PACT*, pp. 147–156. ACM, 2010.

[38] W. Thies, V. Chandrasekhar, and S. Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in C programs. In *MICRO*, pp. 356–369. IEEE, 2007.

[39] T. Wiegand, G. J. Sullivan, G. Bjϕntegaard, and A. Luthra. Overview of the H.264/AVC video coding standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):560–576, 2003.