A

On-the-Fly Pipeline Parallelism

I-TING ANGELINA LEE,[1] CHARLES E. LEISERSON, TAO B. SCHARDL, and ZHUNPING ZHANG, MIT CSAIL
JIM SUKHA, Intel Corporation

Pipeline parallelism organizes a parallel program as a linear sequence of 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 $T_P \leq T_1/P + O(T_\infty + \lg P)$ 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.

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.

Additional Key Words and Phrases: Cilk, multicore, multithreading, parallel programming, pipeline parallelism, on-the-fly pipelining, scheduling, work stealing.

1. INTRODUCTION

Pipeline parallelism[2] [Blelloch and Reid-Miller 1997; Giacomoni et al. 2008; Gordon et al. 2006; MacDonald et al. 2004; McCool et al. 2012; Navarro et al. 2009; Pop and Cohen 2011; Reed et al. 2011; Sanchez et al. 2011; Suleman et al. 2010] is a well-known parallel-programming pattern that

[1] I-Ting Angelina Lee is currently affiliated with Washington University in St. Louis.
[2] Pipeline parallelism should not be confused with instruction pipelining in hardware [Rojas 1997] or software pipelining [Lam 1988].

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.
Author's addresses: I-T. A. Lee (angelee@wustl.edu), 1 Brookings Drive, Campus Box 1045, St. Louis, MO 63130; C. E. Leiserson (cel@mit.edu), T. B. Schardl (neboat@mit.edu), and Z. Zhang (gnipnuhz@gmail.com), MIT CSAIL, 32 Vassar Street, Cambridge, MA 02139; J. Sukha (jim.sukha@intel.com), Intel Corporation, 77 Reed Road, Hudson, MA 01749

DOI: http://dx.doi.org/10.1145/0000000.0000000

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 [Bienia et al. 2008; Bienia and Li 2010], exhibit parallelism in the form of a ***linear*** pipeline, where a linear sequence $\mathcal{S} = \langle \mathcal{S}_0, \ldots, \mathcal{S}_{m-1} \rangle$ 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) [McCool et al. 2012], where the pipeline stages and their dependencies are defined *a priori* before execution. Systems that support construct-and-run pipeline parallelism are described in: [Agrawal et al. 2010; Consel et al. 2003; Gordon et al. 2006; MacDonald et al. 2004; Mark et al. 2003; McCool et al. 2012; OpenMP 3.0 2008; Ottoni et al. 2005; Pop and Cohen 2011; Rangan et al. 2004; Sanchez et al. 2011; Suleman et al. 2010; Thies et al. 2007].

We have extended the Cilk parallel-programming model [Frigo et al. 1998; Leiserson 2010; Intel Corporation 2013] to Cilk-P, a system that augments Cilk's 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. For example, Cilk-P allows pipelines to have a variable number of stages across iterations. 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 [Reed et al. 2011].

Although Cilk-P's support for defining linear pipelines on the fly is more flexible than construct-and-run approaches and the `ordered` directive in OpenMP [OpenMP 3.0 2008], which supports a limited form of on-the-fly pipelining, it is less expressive than other approaches. Blelloch and Reid-Miller [Blelloch and Reid-Miller 1997] describe a scheme for on-the-fly pipeline parallelism that employs futures [Friedman and Wise 1978; Baker and Hewitt 1977] 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 [Blumofe and Leiserson 1998].

To illustrate the ideas behind the Cilk-P model, consider a simple 3-stage linear pipeline such as in the *ferret* benchmark from PARSEC [Bienia et al. 2008; Bienia and Li 2010]. 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*** from node (i, j) to node (i, j'), where $j < j'$, indicates that (i, j') cannot start until (i, j) completes. A ***cross edge*** from node $(i-1, j)$ to node (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, meaning that stages within

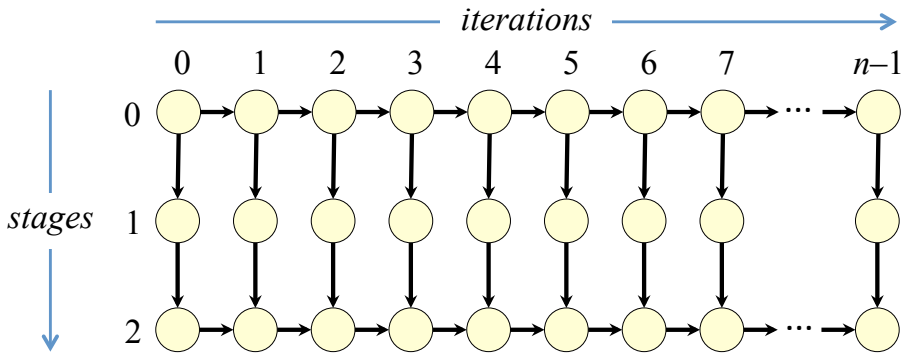

Fig. 1. Modeling the execution of *ferret*'s linear pipeline as a pipeline dag. Each column contains nodes for a single iteration, and each row corresponds to a stage of the pipeline. Vertices in the dag correspond to nodes of the linear pipeline, and edges denote dependencies between the nodes. Throttling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

an iteration must be executed one after another. 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 [McCool et al. 2012; MacDonald et al. 2004], 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, [Arora et al. 2001; Blumofe and Leiserson 1999; Burton and Sleep 1981; Frigo et al. 1998; Finkel and Manber 1987; Kranz et al. 1989]) 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 [Gordon et al. 2006; Sanchez et al. 2011; Thies et al. 2007]. Some researchers report that the bind-to-element approach generally outperforms bind-to-stage [Navarro et al. 2009; Reed et al. 2011], 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 [Cormen et al. 2009, 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.

We can apply the work/span analysis to the *ferret* pipeline shown in Figure 1. This pipeline has the special structure that each node executes serially, that is, without any nested parallelism inside the node. Thus, in the *ferret* pipeline, the work and span of each node is 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 $1 \ll r \leq n$ is at least $r/2+1$. Thus, if stage 1 contains much more work than the other two stages, the pipeline exhibits good parallelism.

On an ideal shared-memory computer, Cilk-P guarantees to execute the *ferret* pipeline efficiently. In particular, Cilk-P guarantees linear speedup on a computer with up to $T_1/T_\infty = O(r)$ processors. 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 iteration $i+K$. 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 [Lee et al. 2010] runtime 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" (roughly, the maximum number of bytes consumed on the stack by any one pipeline iteration), 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 for reducing synchronization overhead when two consecutive pipeline iterations execute in parallel: "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, run competitively with existing Pthreaded implementations (Section 10).

— We prove two theorems regarding the performance impact of throttling (Section 11). First we show that, if each stage has approximately the same cost and dependencies in every iteration, then throttling only reduces the parallelism in a pipeline by a constant factor. We then show that, if the cost of a stage can vary dramatically between iterations, however, then it is impossible for any scheduler to achieve parallel speedup when executing the pipeline without using a large amount of space.

We conclude in Section 12 with a discussion of potential future work.

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++ [Leiserson 2010] and Intel® Cilk™ Plus [Intel Corporation 2013] which augments serial C/C++ code with two principal keywords: `cilk_spawn` and `cilk_sync`.[3] 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.

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_stage` and `pipe_stage_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_stage(j)` or `pipe_stage_wait(j)` statement, where $j > j'$, then node (i,j') ends, and control flow proceeds to node (i,j). A `pipe_stage(j)` statement indicates that node (i,j) can start executing immediately, whereas a `pipe_stage_wait(j)` statement indicates that node (i,j) cannot start until node $(i-1,j)$ completes. The `pipe_stage_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 "dynamic dependency folding" runtime optimizations.

The `pipe_stage` and `pipe_stage_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 $j+1$, meaning that control moves onto the next stage.

Cilk-P's semantics for `pipe_stage` and `pipe_stage_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_stage_wait`, where $j' < j'' < j$, then we implicitly create a ***null node*** (i,j'') in the pipeline dag, which has no associated work and

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

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 [Wiegand et al. 2003].

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,[4] if one exists within the nearby range. If f_i is a ***B-frame***, then macroblock (x, y)'s encoding can also be based on nearby macroblocks in nearby preceding or succeeding frames, specifically, frames in the interval between the most recently preceding I-frame and the next succeeding I- or P-frame.

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 Cilk-P pseudocode for an *x264* linear pipeline. Conceptually, the *x264* pipeline begins with a serial stage (lines 7–16) 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 17–24), 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 26–28), and then the `END` stage updates the output stream with the processed frames (line 30).

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 19–23 of Figure 2 by executing a `pipe_stage_wait` or `pipe_stage` 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 16 by skipping w additional stages relative to the previous iteration. A similar stage-skipping trick is used on line 25 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.

4. COMPUTATION-DAG MODEL

Although the pipeline-dag model provides intuition for programmers to understand the execution of a pipeline program, it is not precise enough to prove theoretical performance guarantees. For exam-

[4] 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;

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

Fig. 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_stage_wait` on line 16, the input-data dependent `pipe_stage_wait` or `pipe_stage` on lines 19–23, and the `pipe_stage` on line 25.

ple, 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." First, we present an example of a simple pipeline program using `pipe_while` loops, and explain how to transform it into an ordinary Cilk program with special function calls to enforce non-fork-join dependencies. Then we describe how to model these transformed programs as computation dags.

We shall model an execution of a pipeline program as a "(pipeline) computation dag," which is based on the notion of a fork-join computation dag for ordinary Cilk programs [Blumofe and Leiserson 1998, 1999] without pipeline parallelism. Let us first review this computation dag model for ordinary Cilk programs. A ***fork-join computation dag*** $G = (V, E)$ represents the execution of a Cilk program, where each vertex in V denotes a unit-cost instruction. For convenience, we shall assume that instructions that call into the runtime system execute in unit time. 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 `cilk_sync` instruction (as well as an ordinary serial edge from the instruction that executed immediately before the `cilk_sync`). We can model a particular execution of an ordinary fork-join Cilk program as conceptually generating the computation dag G dynamically, as it executes. Thus, after the program has finished executing, we have a complete dag G that captures the structure of parallelism in that execution.

Intuitively, we shall model an execution of a pipeline program as a ***(pipeline) computation dag*** by augmenting a traditional fork-join computation dag with cross and throttling dependencies. More formally, to generate a pipeline computation dag for an arbitrary pipeline-program execution, we use the following three-step process:

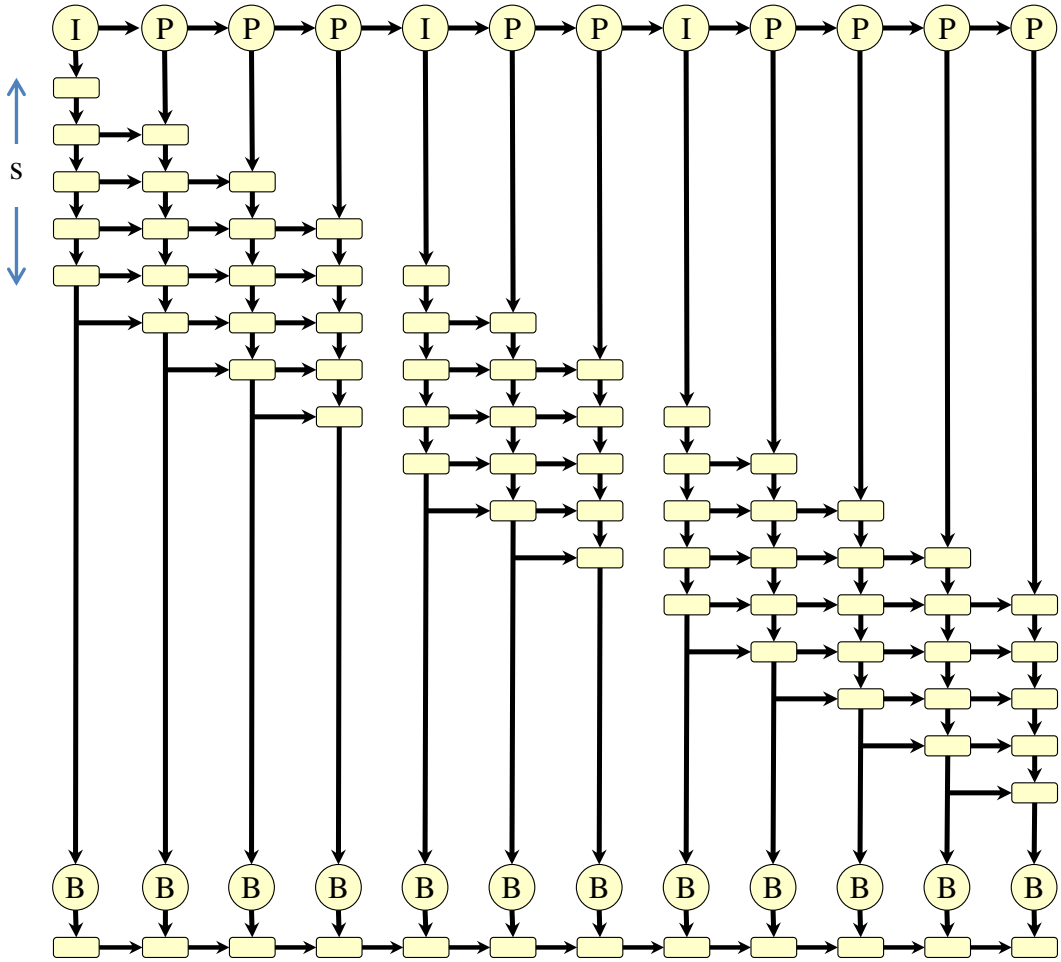

Fig. 3. The pipeline dag generated for *x264*. Each iteration processes either an I- or P-frame, each consisting of s rows. As the iteration index i increases, the number of initial stages skipped in the iteration also increases. This stage skipping produces cross edges into an iteration i from null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

(1) Transform the executed code in each `pipe_while` loop into ordinary Cilk code, augmented with special functions to implement cross and throttling dependencies.
(2) Model the execution of this augmented Cilk program as a fork-join computation dag, ignoring cross and throttling dependencies.
(3) Augment the fork-join computation dag with cross and throttling edges derived from the special functions.

The remainder of this section examines each of these steps in detail.

Code transformation for a `pipe_while` *loop*

Let us first consider the process of translating a `pipe_while` loop into ordinary Cilk code. Conceptually, a `pipe_while` loop is transformed into an augmented ordinary Cilk program in which an ordinary `while` loop sequentially spawns off each iteration of the `pipe_while` loop. In this `while` loop, first, each iteration executes stage 0. Upon executing the first `pipe_stage` or `pipe_stage_wait` in-

```
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_stage_wait(1);
9      bool isDuplicate = deduplicate(chunk);
10     pipe_stage(2);
11     if (!isDuplicate)
12       compress(chunk);
13     pipe_stage_wait(3);
14     write_to_file(fd_out, chunk, isDuplicate);
15   }
16 }
```

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

struction in an iteration i, the remainder of iteration i is spawned off, allowing the remaining stages of iteration i to execute in parallel with iteration $i+1$. By executing stage 0 of a `pipe_while` iteration before spawning the remaining stages, stage 0 is ensured to execute sequentially across all iterations of the `while` loop. Each iteration may execute additional runtime functions to enforce cross and throttling dependencies between iterations.

This conceptual transformation of a `pipe_while` loop is complicated by specific semantic features of `pipe_while` iterations. For example, although stage 0 of each iteration executes before the remaining stages of each iteration are spawned, the runtime executes each iteration within its own ***(function) frame*** — activation record — to ensure that all stages of an iteration operate on the same set of iteration-local variables. Furthermore, to ensure that an iteration executes pipeline stages sequentially, the runtime executes an implicit `cilk_sync` at the end of each stage, which syncs all child functions spawned within the stage before allowing the next stage to begin.

To more precisely illustrate the semantic features of `pipe_while` iterations, including how the Cilk-P runtime manages frames and iterations of a `pipe_while` loop, let us consider a Cilk-P implementation of a specific pipeline program, namely, the *dedup* compression program from PARSEC [Bienia et al. 2008; Bienia and Li 2010]. 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_stage_wait` on line 8. Stage 2, which the `pipe_stage` 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).

Figure 5 illustrates how the Cilk-P runtime system manages frames and pipeline iterations for the `pipe_while` loop for *dedup* presented in Figure 4. This code transformation has six key components, which illustrate the general structure of parallelism in pipeline programs.

(1) As shown in lines 3–50, a `pipe_while` loop is "lifted" using a C++ lambda function [Stroustrup 2013, Sec.11.4] and converted to an ordinary `while` loop whose iterations correspond to iterations of the pipeline. This lambda function declares a ***control frame*** object `pcf` (on line 4) to keep track of runtime state needed for the `pipe_while` loop, including a variable `pcf.i` to index iterations, which line 4 initializes to 0.
(2) Each iteration of the `while` loop allocates an ***iteration frame*** to store local data for each pipeline iteration. Before starting a pipeline iteration `pcf.i`, the loop allocates a new iteration frame `next_iter_f` for iteration `pcf.i`, as shown in line 6. The iteration frame stores local vari-

```
1  int fd_out = open_output_file();
2  bool done = false;
3  [&]() {
4    _Cilk_pipe_control_frame pcf(0);

5    while (true) {
6      _Cilk_pipe_iter_frame* next_iter_f = pcf.get_new_iter_frame(pcf.i);
7      // Stage 0 of an iteration.
8      [&]() {
9          next_iter_f->continue_after_stage0 = false;
10         if (!done) {
11           next_iter_f->chunk = get_next_chunk();
12           if (next_iter_f->chunk == NULL)
13             done = true;
14           else
15             next_iter_f->continue_after_stage0 = true;
16         }
17         cilk_sync;
18     }();
19     // Spawn the remaining stages of iteration pcf.i, if they exist.
20     if (next_iter_f->continue_after_stage0) {
21       cilk_spawn [&](_Cilk_pipe_iter_frame* iter_f) {

22         // assert(iter_f->stage_counter < 1);
23         iter_f->stage_counter = 1;

24         // node (i,1) begins
25         iter_f->stage_wait(1);
26         iter_f->isDuplicate = deduplicate(iter_f->chunk);
27         cilk_sync;

28         // assert(iter_f->stage_counter < 2);
29         iter_f->stage_counter = 2;

30         // node (i,2) begins
31         if (!iter_f->isDuplicate)
32           compress(iter_f->chunk);
33         cilk_sync;

34         // assert(iter_f->stage_counter < 3);
35         iter_f->stage_counter = 3;

36         // node (i,3) begins
37         iter_f->stage_wait(3);
38         write_to_file(fd_out, iter_f->chunk, iter_f->isDuplicate);
39         cilk_sync;

40         iter_f->stage_counter = INT64_MAX;
41       }(next_iter_f);
42     } else {
43       break;
44     }
45     // Advance to next iteration and check for throttling.
46     pcf.i++;
47     pcf.throttle(pcf.i - pcf.K);
48   }
49   cilk_sync;
50 }();
```

Fig. 5. Pseudocode resulting from translating the execution of the Cilk-P *dedup* implementation from Figure 4 into Cilk Plus code augmented by cross and throttling dependencies, implemented by `iter_f->stage_wait` and `pcf.throttle`, respectively. The unbound variable `pcf.K` is the throttling limit.

ables declared in the body of an iteration that persist across pipeline stages. For *dedup*, for example, Figure 4 shows that the local variable `chunk` is used through all stages. The iteration frame also stores a ***stage-counter*** variable, `iter_f->stage_counter`, to track the currently executing stage for the iteration. Although Figure 5 shows iteration frames of a generic type `Cilk_pipe_iter_frame`, in practice, a compiler would generate a unique iteration frame type for each `pipe_while` loop body, since the local variables stored in the frame are specific to the loop body.

(3) The body of this `while` loop is split into two nested lambda functions, the first for stage 0 of the iteration (lines 8–18), and the second for the remaining stages in the iteration (lines 21–

41), if they exist. This transformation guarantees that stage 0 is always a serial stage, since the first lambda function is directly called in the body of the `while` loop. The test condition of the `pipe_while` loop is evaluated as part of stage 0, as demonstrated in line 10. In contrast, the `cilk_spawn` in line 21 allows the remaining stages of an iteration to execute in parallel with the next iteration of the loop. The `cilk_sync` immediately after the end of the `while` loop (line 49) ensures that all spawned iterations complete before the `pipe_while` loop finishes.

(4) The last statement in the `while` loop (line 47) is a call to a special function `throttle`, defined by the control frame `pcf`, which enforces the throttling dependency that iteration `pcf.i` can not start until iteration `pcf.i - pcf.K` has completed.

(5) A `pipe_stage` statement in the original `pipe_while` loop is transformed into an update to `iter_f->stage_counter`, while a `pipe_stage_wait` statement is transformed into an update followed by a call to `iter_f->stage_wait`, which ensures that the cross dependency on the previous iteration is satisfied. In *dedup*, stages 1, 2, and 3 are thus delineated by updates to `iter_f->stage_counter` in lines 23, 29, and 35, respectively. The end of the iteration is delineated by setting `iter_f->stage_counter` to its maximum value, such as in line 40.

(6) At the end of each stage, a `cilk_sync` (lines 17, 27, 33, and 39), guarantees that any nested fork-join parallelism is enclosed within the stage, that is, any functions spawned in `cilk_spawn` statements within the stage return before the next stage begins.

For *dedup*, Figure 5 is able to use lambda functions to capture the parallel control structure of Figure 4 directly in Cilk, without changing the semantics of the `cilk_spawn` or `cilk_sync` keywords. This transformation introduces an additional variable in the iteration frame, `continue_after_stage0`, so that execution can resume correctly at the continuation of stage 0 in the second lambda function in each iteration. The lambda functions in line 3 and line 8 exist only to create nested scopes for parallelism and ensure the desired behavior for a `cilk_sync` statement. Without the lambda function in line 3, the last `cilk_sync` in line 49 would also synchronize with any functions that were spawned in the enclosing function before calling the `pipe_while` loop. Similarly, the lambda for stage 0 in line 8 exists only to guarantee that the `cilk_sync` in line 17 joins only the parallelism within stage 0, and not with any of the lambda functions spawned in line 21. All the lambda functions in Figure 5 capture the environment of the enclosing function by reference because the body of the `pipe_while` loop is allowed to access variables declared in the enclosing function, such as `fd_out` and `done`.

While Figure 5 illustrates the semantics that Cilk-P requires for compiling the code in Figure 4, it is not intended to be a complete compiler transformation for `pipe_while` loops. In practice, we expect it to be simpler and more efficient for compiler transforms for `pipe_while` loops to operate and produce output at a lower level than ordinary Cilk Plus code. For example, a more complicated pipeline iteration in which stage 0 may end in the middle of a loop is tricky to express using lambdas and ordinary Cilk Plus code, because of the scoping of local variables and the semantics of `cilk_sync` statements. At a lower level, however, a compiler could handle the end of stage 0 by directly generating code that saves the program state analogously to an ordinary `cilk_spawn`. The compiler might also be eliminate one or more of the nested lambda functions from Figure 5, and instead generate code directly for modified versions of `cilk_sync` and `cilk_spawn` statements specifically for `pipe_while` loop transformations. The generated code would not be directly expressible in Cilk Plus, but would likely be simpler and more efficient.

Similarly, although Figure 5 describes an iteration as being split into two lambda functions — one for stage 0 and one for the subsequent stages of the iteration — in practice, it may be simpler to merge those lambda functions. Instead of using an ordinary `cilk_spawn` to spawn the rest of the stages of an iteration separately from stage 0, for example, a system might instead try to spawn a single lambda function for the entire iteration. Then, the system might allow other workers to steal the continuation of the spawn of the iteration only after the iteration finishes its stage 0, not immediately after the spawn occurs.

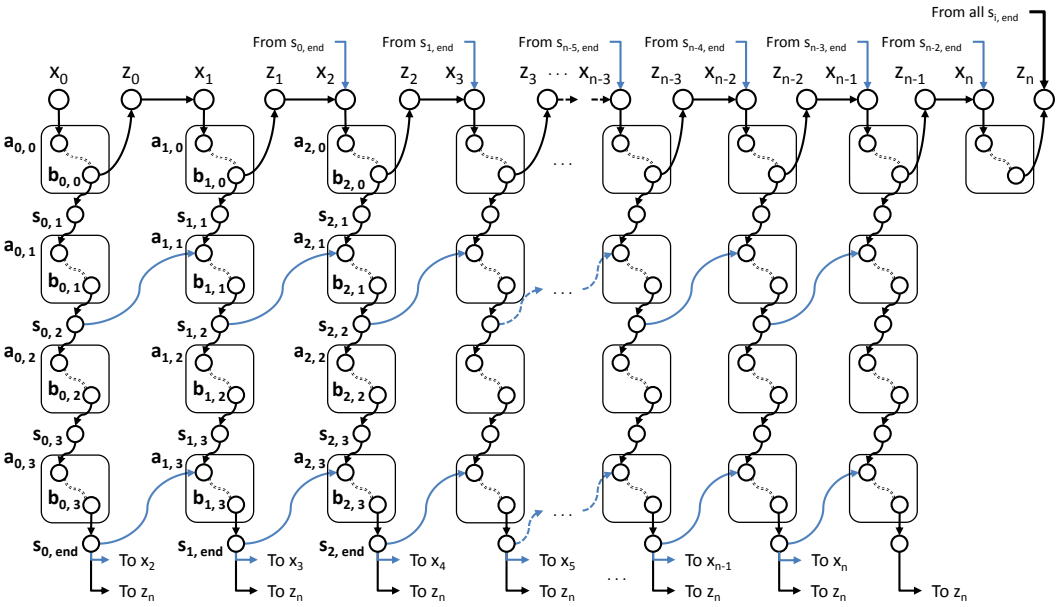

Fig. 6. An example pipeline computation dag for a `pipe_while` loop with n iterations, $m = 4$ stages, and throttling limit $K = 2$, corresponding to the transformation shown in Figure 5. The vertices are organized to reflect their organization in a pipeline dag. Each rounded box contains the vertices corresponding to the execution of a node. A double-dashed line indicates a computation subdag whose structure is not shown. A column of these boxes corresponds to an iteration of the `pipe_while`, while a row of these boxes corresponds to a stage. Additional vertices and edges appear in this dag to denote instructions executed by the runtime to handle iterations of a `pipe_while`, as well as their parallel control dependencies. Cross and throttling edges are colored blue, while edges in typical Cilk programs are colored black.

Pipeline computation dag for `dedup`

Given the transformed code for a `pipe_while` loop, the second and third steps generate a pipeline computation dag that models the execution of this transformed loop. The second step models the execution of the transformed code when ignoring all calls to `stage_wait` and `throttle`, and then the third step augments the resulting fork-join computation dag with cross and throttling edges derived from those calls. Figure 6 illustrates the salient features of the final pipeline computation dag that corresponds to executing the code in Figure 5. Let us examine the structure of the dag in Figure 6 by first considering the vertices and edges that model the execution of Figure 5, ignoring calls to `stage_wait` and `throttle`, and then examining the cross and throttling edges added by these calls.

Let us first see how the vertices in Figure 6 correspond to the lines of code in Figure 5. Let i be an integer where $0 \leq i \leq n$, and let j be an integer greater than 0.

— The vertices labeled x_i and z_i correspond to the execution of instructions inserted by the runtime. Vertices x_0 and z_n correspond to the first and final instruction, respectively, of the lambda for the `pipe_while` loop. Vertex x_0, which is called the ***pipeline root***, corresponds to executing line 4 to create the control frame for the `pipe_while` loop, while vertex z_n, which is called the ***pipeline terminal***, corresponds to executing line 49. The remaining vertices z_i and x_i are called ***iteration-increment*** and ***iteration-throttle*** vertices, respectively. Between each iteration of the `while` loop, the iteration-increment vertices z_i correspond to executing line 46, and the iteration-throttle vertices x_i correspond to executing line 47.

— The computation subdag rooted at $a_{i,0}$ and terminated at vertex $b_{i,0}$ correspond to executing stage 0 and associated runtime instructions for managing the `while` loop in iteration i. Vertex $a_{i,0}$ corresponds to executing line 5. Vertex $b_{i,0}$ corresponds to executing the `cilk_spawn` statement

on line 21, except when $i = n$, in which case $b_{n,0}$ corresponds to executing line 43. The vertices in Figure 6 on paths from $a_{i,0}$ to $b_{i,0}$ correspond to executing the intervening instructions in lines 5–21. The `cilk_sync` statement in the lambda for stage 0 ensures that vertex $b_{i,0}$ is the single leaf vertex for this computation subdag.

— For $i < n$, the computation subdag rooted at $a_{i,j}$ and terminated at $b_{i,j}$ corresponds to the execution of node (i, j) in the pipeline dag. For example, in iteration i, vertex $a_{i,1}$ corresponds to executing line 25 — the first instruction in node $(i,1)$ — and vertex $b_{i,1}$ corresponds to executing line 27 — the final instruction in node $(i,1)$. The vertices on paths from $a_{i,1}$ to $b_{i,1}$, in Figure 6, correspond to executing the intervening instructions in lines 25–27. Notice that, if node (i, j) is the destination of a cross edge, then $a_{i,j}$ corresponds to executing `stage_wait`. The `cilk_sync` statement at the end of each stage — lines 27, 33, and 39 for stages 1, 2, and 3, respectively — ensure that $b_{i,j}$ is the single leaf in the computation subdag corresponding to the execution of node (i, j).

— The ***stage-counter*** vertices $s_{i,end}$ and $s_{i,j}$ for integers $j > 0$ correspond to updates in iteration i to the iteration frame's `stage_counter` variable. For example, $s_{i,2}$ corresponds to executing line 29 in iteration i. Vertex $s_{i,end}$ corresponds to executing line 40 in iteration i, which terminates the iteration. We call $s_{i,end}$ the ***terminal*** vertex for iteration i.

For convenience, in the computation subdag that models the execution of node (i, j), we call vertex $a_{i,j}$ the ***node root***, and we call vertex $b_{i,j}$ the ***node terminal***.

The correspondence between instructions in Figure 5 and the vertices of Figure 6 describes most of the edges in Figure 6, based on the structure of fork-join computation dags. For example, the code in Figure 5 shows that, for each i where $0 \le i \le n$, edge $(x_i, a_{i,0})$ is a serial edge, edge $(b_{i,0}, s_{i,1})$ is a spawn edge, and edge $(b_{i,0}, z_i)$ is a continue edge. Meanwhile, for each iteration i where $0 \le i < n$, edge (z_i, x_{i+1}) is a serial edge, reflecting the fact that stage 0 is a serial stage. Similarly, for $j > 1$, the edges $(b_{i,j-1}, s_{i,j})$ and $(s_{i,j}, a_{i,j})$ that connect the node terminal of $(i, j-1)$ to the node root of (i, j) are serial edges, reflecting the fact that each iteration of the pipeline executes the pipeline stages sequentially. Finally, for each iteration i where $0 \le i < n$, edge $(b_{i,3}, s_{i,end})$ is a serial edge, and edge $(s_{i,end}, z_n)$ is a return edge. These vertex and edge definitions are established by modeling an execution of the transformed code as an ordinary Cilk Plus program, when `stage_wait` and `throttle` instructions are ignored.

Finally, we consider the cross and throttling edges in Figure 6 enforced by `stage_wait` and `throttle` instructions.

For each iteration i where $0 < i < n$, a call to `stage_wait` implements a cross edge, which connects a stage-counter vertex in iteration $i-1$ to a node root in iteration i. For example, in each iteration i of the loop in Figure 5, the `stage_wait` call on line 25 implements the cross edge $(s_{i-1,2}, a_{i,1})$, and the `stage_wait` call on line 37 implements the cross edge $(s_{i-1,end}, a_{i,3})$. Conceptually, because a stage-counter vertex $s_{i,j}$ occurs after the node terminal for stage $j-1$ and before the node root for stage j, a cross edge $(s_{i-1,j}, a_{i,j-1})$ ensures that node $(i, j-1)$ in iteration i executes after node $(i-1, j-1)$. When j is the final stage in an iteration $i-1$, the iteration terminal $s_{i-1,end}$ fills the role of the stage-counter vertex $s_{i-1,j+1}$.

A throttling edge connects the terminal of iteration $i \le n-K$ to the iteration-throttle vertex x_{i+K} in iteration $i+K$, where K is the throttling limit. Figure 6 illustrates throttling edges when $K = 2$ and shows that a throttling edge exists from $s_{i,end}$ to x_{i+2} for each iteration i where $0 \le i < n-2$. These throttling edges thus prevent node $(i,0)$ from executing before all nodes in iteration $i-K$ complete, thereby limiting the number of iterations that may execute simultaneously.

General pipeline computation dags

To generalize the structure of the pipeline computation dag in Figure 6 for arbitrary Cilk-P pipelines, we must specify how null nodes are handled. In some iteration i, for stage $j > 0$, suppose that node (i, j) is a null node. In this case, none of the stage-counter vertices $s_{i,j}$, node roots $a_{i,j}$, node terminals $b_{i,j}$, nor any of the vertices on paths between these, map to executed instructions, and

therefore these vertices do not exist in the computation dag. To demonstrate what happens to the edges that would normally enter and exit these vertices, we may suppose that the computation dag is originally constructed with dummy vertices $s_{i,j}$, $a_{i,j}$, and $b_{i,j}$ connected in a path, and then all three of these vertices are contracted into the stage-counter vertex following $b_{i,j}$. Because $a_{i,j}$ is a dummy vertex, it does not correspond to a call to `stage_wait`, and thus it has no incoming cross edge. Furthermore, this method for handling null nodes may cause multiple cross edges to exit the same stage-counter vertex. We shall see that this is does not pose a problem for the PIPER scheduler.

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 a modification of the scheduler described by Arora, Blumofe, and Plaxton [Arora et al. 2001] (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, a special operation introduced to handle throttling of pipeline iterations.

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 u is ***ready*** if all its predecessors have been executed. Executing an assigned vertex v may ***enable*** a vertex u that is a direct successor of v in G by making u 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 simple 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 cross-edge dependencies from null nodes, executing v can enable at most one such vertex, because the nodes in the next iteration execute serially.) To simplify the mathematical analysis, we assume that all rules are executed atomically.[5]

First, we consider the cases where the assigned vertex v of a worker p is not the terminal of an iteration.

— If executing v enables only one direct successor u, then p simply changes its assigned vertex from v to u.
— If executing v enables two successors u and w, then p changes its assigned vertex from v to one successor u, and pushes the other successor w onto its deque. The decision of which successor to push onto the deque depends on the type of v. If v is a vertex corresponding to a normal spawn, then u follows the spawn edge (u is the child), and w follows the continue edge (w is the continuation).[6] If v is a stage-counter vertex in iteration i that is not the terminal of iteration i, then u is the node root of the next node in iteration i, and w is the node root of a node in iteration $i+1$.
— If executing v enables no successors and the deque of p is not empty, then p pops the bottom element u from its deque and changes its assigned vertex from v to u.
— If executing v enables no successors and the deque of p is empty, then p becomes a ***thief***. As a thief, p randomly picks another worker to be its ***victim*** and tries to steal the vertex at the head of

[5]This assumption, which is also implicit in the ABP model, need not correspond to an actual implementation. Work-stealing schedulers can typically optimize away many atomic operations by using more sophisticated synchronization.

[6]This choice to have a worker follow the spawn edge instead a continue edge is consistent with traditional "parent-stealing" execution model of Cilk, which on a spawn begins executing the "child" spawned function and leaves the continuation in the parent frame to be stolen. Unlike child stealing, parent stealing allows us to prove a bound on space in Section 8.

the victim's deque. If such a vertex u exists, the thief p sets its assigned vertex to u. Otherwise, the victim's deque is empty, p's assigned node becomes NULL, and p remains a thief.

These cases are consistent with the normal ABP model.

PIPER handles the terminal of an iteration a little differently, because of throttling edges. Suppose that a worker p has an assigned vertex v which is the terminal of an iteration. When p executes v, it can enable some combination of a pipeline terminal, an iteration-throttle vertex, or the destination a cross edge, if there is a cross edge leaving v. We can distinguish these cases, however, based on whether or not v enables an iteration-throttle vertex.

— Suppose that executing v does not enable an iteration-throttle vertex. Then, p acts as it would in the normal ABP model. In particular, p can only enable either the destination a of a cross edge or a pipeline terminal z, if it enables any vertex. In this situation, p executes as it normally would when enabling zero or one vertices.
— Suppose that executing v does enable an iteration-throttle vertex x. If executing v also enables the destination a of a cross edge, then p behaves as it would in the normal ABP model, pushing x onto its deque and setting its assigned vertex to a. If executing v does not enable the destination of a cross edge, then p performs two actions. First, p changes its assigned vertex from v to x. Second, if p has a nonempty deque, then p performs a ***tail swap***: it exchanges its assigned vertex x with the vertex at the tail of its deque.

This tail-swap operation is designed to empirically reduce PIPER's space usage and cause PIPER to favor retiring old iterations over starting new ones. 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 6.4 in Section 6 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, because these frames implement a cactus stack [Hauck and Dent 1968; Lee et al. 2010] that reflects the parallel control structure of the program. In variants of Cilk, every spawned function creates a new frame, and every iteration of a parallel loop executes in its own frame. The cactus stack ensures that every function is allowed to access the variables in its frame and its parent frames, in spite of their parallel execution. For non-pipelined computation dags, the creation of new frames matches the parallel control structure, and pushing a vertex onto a worker's deque corresponds to creating a new frame.

We would like to have an analogous structure for `pipe_while` loops. As Section 4 describes, PIPER creates a new frame for each iteration and executes all stages of the iteration using that frame. The creation of these per-iteration frames does not match how Piper manipulates the worker deques, however. In particular, stage 0 of an iteration executes using that iteration's frame before the remaining stages of the iteration are spawned. Consequently, the iteration's frame is created before stage 0 executes, but no vertex from this iteration is pushed onto a worker's deque until after stage 0 completes.

To get around this problem with frames, we introduce "contours" to model how deques are modified during the execution of a `pipe_while` loop. Consider a computation dag $G = (V, E)$ that arises from executing 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 is the only vertex in the contour that has an incoming spawn edge (except for the initial instruction of the entire computation, which has no incoming edges). Consequently, contours can be organized into a tree hierarchy, where one contour is a parent of another 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. For convenience, we shall assume that all contours are maximal, meaning that no two vertices in distinct contours are connected by a serial or continue edge.

Figure 7 illustrates contours for a simple function `F` with both nested fork-join and pipeline parallelism. For the `pipe_while` loop in `G`, stage 0 of pipeline iteration 0 (a_7 and a_8) is considered part of the same contour that starts the `pipe_while` loop, not part of contour f which represents the rest of the stages of iteration 0. In terms of function frames, however, it is natural to consider stage 0 as sharing a function frame with the rest of the stages in the same iteration. Although contour boundaries happen to align with function boundaries when we consider only fork-join parallelism in Cilk, contours and function frames are actually distinct orthogonal concepts, as highlighted by `pipe_while` loops.

One important property of contours, which can be shown by structural induction, is that, for any function invocation `F`, the vertices p and q corresponding to the first and last instructions in `F` belong to the same contour, that is, $c(p) = c(q)$. Using this property and the identities of its edges, one can show the following facts about contours in a pipeline computation dag.

FACT 1. For a given `pipe_while` loop on n iterations, the pipeline root x_0, pipeline terminal z_n, iteration-increment vertices $z_1, z_2, \ldots, z_{n-1}$, iteration-throttle vertices $x_1, x_2, \ldots, x_n$, stage-0 node roots $a_{0,0}, a_{1,0}, \ldots, a_{n,0}$, and stage-0 node terminals $b_{0,0}, b_{1,0}, \ldots, b_{n,0}$ all lie in the same contour. In other words, for $i = 0, 1, \ldots n$, we have $c(x_i) = c(a_{i,0}) = c(b_{i,0}) = c(z_i)$.

FACT 2. For an iteration i of a `pipe_while` loop and $j = 1, 2, \ldots, m-1$, if node (i, j) is not a null node, then the stage-counter vertices $s_{i,j}$ and $s_{i,end}$ and the node root $a_{i,j}$ and node terminal $b_{i,j}$ for node (i, j) all lie in the same contour, that is, $c(s_{i,end}) = c(s_{i,j}) = c(a_{i,j}) = c(b_{i,j})$.

For convenience, we say that the ***root*** of a `pipe_while` iteration i is the first stage-counter vertex in iteration i, specifically, the stage-counter vertex in iteration i that is the destination of a spawn edge from the stage-0 node terminal for iteration i. Consequently, the root of the contour containing the stage-counter vertices for an iteration i is the root of iteration i.

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

LEMMA 6.1. *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 each iteration creates a separate contour for all its stages after stage 0, and that all contours for iterations of the `pipe_while` share a common parent in the contour tree, namely the contour containing stage 0 of all loop iterations. These properties naturally lead to the following lemma, which specifies the relationship between the contours that contain the endpoints of a cross edge or a throttling edge.

LEMMA 6.2. *If an edge (u, v) is a cross edge, then $c(u)$ and $c(v)$ are siblings in the contour tree and correspond to adjacent iterations in a* `pipe_while` *loop. If an edge (u, v) is a throttling edge, then $c(v)$ is the parent of $c(u)$ in the contour tree.*

PROOF. This lemma follows naturally from the structure of pipeline computation dags. For some $i > 0$, every cross edge (u, v) connects a stage-counter vertex in iteration $i-1$ (that is, u equals either

```
void F(int n) {
  if (n < 2)
    G(n);
  else {
    cilk_spawn F(n-1);
    F(n-2);
    cilk_sync;
  }
}
```

```
void G(int n) {
  if (n == 0) {
    int i = 0;
    pipe_while(i < 2) {
      ++i; // Stage 0
      pipe_stage_wait(1);
      H(); // Stage 1.
    }
  }
}
```

```
void H() {
  cilk_spawn foo();
  bar();
  cilk_sync;
}
```

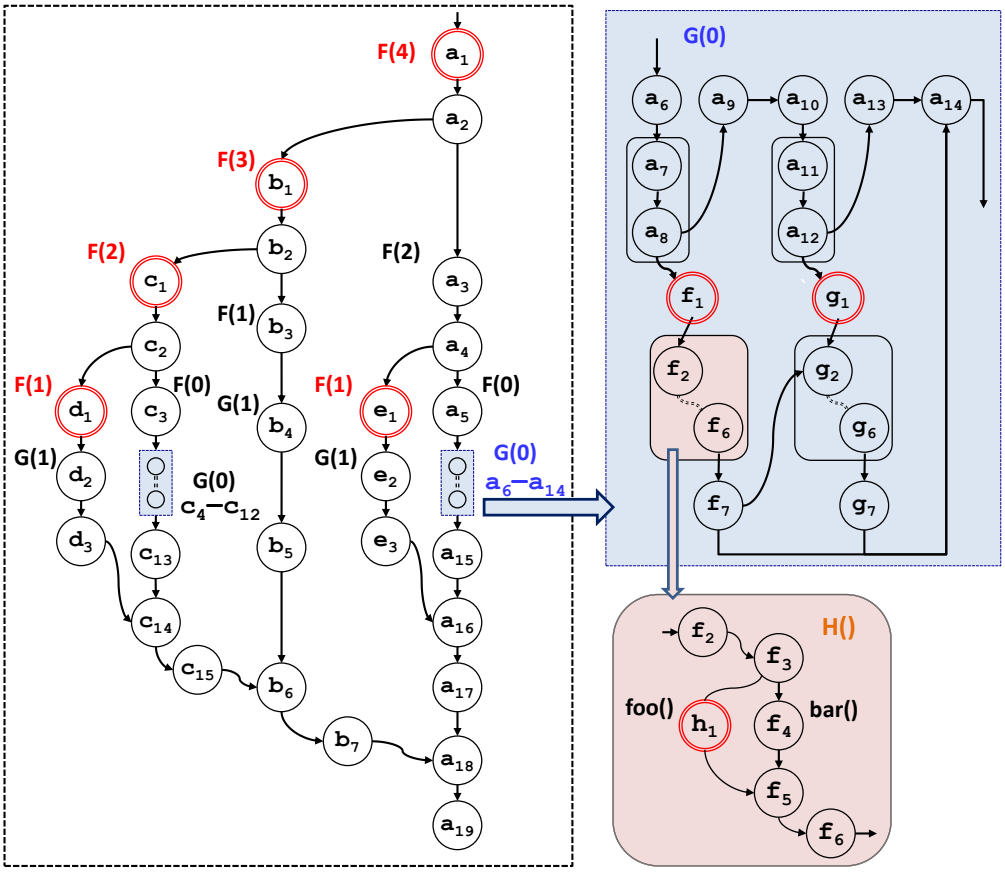

Fig. 7. Contours for a computation with fork-join and pipeline parallelism. The fork-join function `F` contains nested calls to a function `G` that contains a `pipe_while` loop with two iterations. The function `G` itself calls a fork-join function `H` in stage 1 of each iteration. Each letter a through h labels a contour in the dag for `F(4)`. The vertices $a_1, b_1, \ldots, h_1$ are contour roots. For example, the root of $c(a_k)$ is a_1 for all k. Similar to Figure 6, the rounded rectangles in the subdag for `G(0)` represent pipeline stages. A double-dashed line represents an additional subdag whose structure is not shown.

$s_{i-1,j}$ for some j or $s_{i-1,end}$) to a node root $v = a_{i,k}$ in iteration i in the same `pipe_while` loop. By Fact 2, the root of the contour $c(u)$ is the root of iteration $i-1$. Thus the contour $c(u)$ is a child of the contour $c(b_{i-1,0})$ in the contour tree. Similarly, the root of the contour $c(v) = c(a_{i,k})$ is a child of the contour $c(b_{i,0})$ containing the node terminal $b_{i,0}$. Because $b_{i-1,j}$ and $a_{i,k}$ belong to iterations of the same `pipe_while` loop, Fact 1 implies that $c(b_{i-1,0}) = c(b_{i,0})$. Because $c(u)$ and $c(v)$ are both children of this contour, $c(u)$ and $c(v)$ are siblings in the contour tree, showing the first part of the lemma.

For a `pipe_while` loop of n iterations with throttling limit K, a throttling edge (u,v) connects u, the terminal of an iteration $i < n-K$, to a vertex $v = x_{i+K}$ in the computation dag. By the reasoning above, we know that u is in a child contour of $c(b_{i,0})$. By Fact 1, we know $c(b_{i,0}) = c(x_{i+K}) = c(v)$. Thus, u is in a child contour of $c(v)$, showing the second part of the lemma statement. □

During PIPER's execution of a pipeline program, the workers' deques are highly structured with respect to contours. As the following definition details, with one exception, the contours for two adjacent vertices in a worker's extended deque obey a strict parent-child relationship. The sole exception to this property is that, for any particular `pipe_while` loop, an extended deque can contain at most two vertices that belong to contours for iterations of that loop. These two vertices will be adjacent in the extended deque, and the contours that contain them will correspond to sibling `pipe_while` loop iterations.

Definition 6.3. 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) Contour $c(v_{k+1})$ is the parent of $c(v_k)$.
(2) The root of $c(v_k)$ is the root for some iteration i, the root of $c(v_{k+1})$ is the root for the next iteration $i+1$, and if $k+2 \leq r$, then $c(v_{k+2})$ is the common parent of both $c(v_k)$ and $c(v_{k+1})$.

The following lemma shows that, when a worker p performs a tail-swap operation, if p's deque satisfies the contour property, then p's deque must have a specific structure.

LEMMA 6.4. *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, which is the terminal of iteration i of some* `pipe_while` *loop. If p's deque satisfies the contour property (Definition 6.3), then one of the following conditions holds:*

(1) Worker p's deque is empty and x becomes p's new assigned vertex.
(2) Worker p's deque contains a single vertex v_1, where the root of $c(v_1)$ is the root of iteration $i+1$ of the same `pipe_while` *loop, and v_1 becomes p's new assigned vertex while x is pushed onto p's deque.*

PROOF. Let $\langle v_0, v_1, \ldots, v_r \rangle$ denote the vertices in p's extended deque at the time v_0 is executed. If $r = 0$, then x becomes p's assigned vertex, satisfying Case 1 of the lemma. Otherwise, we have $r \geq 1$, and we shall show that, in fact, $r = 1$ and the root of $c(v_1)$ is the root of iteration $i+1$. In this situation, Case 2 of the lemma is satisfied as follows: because x is enabled by a throttling edge, a tail swap occurs, making v_1 the assigned vertex of p and pushing x onto p's deque.

Now we will show that, if $r \geq 1$, then we must have $r = 1$ and the root of $c(v_1)$ is the root of iteration $i+1$. Because x is enabled by a throttling edge, v_0 must be the terminal of some iteration i, and Lemma 6.2 implies that $c(x)$ is the parent of $c(v_0)$. The contour property (Definition 6.3) applied to v_0 states that either $c(v_1) = c(x)$ or $c(v_1)$ is the root of iteration $i+1$. The first case, $c(v_1) = c(x)$, is impossible, because Lemma 6.1 implies that vertices v_1 and x cannot simultaneously inhabit p's deque. We therefore have that $c(v_1)$ is the root of iteration $i+1$. In this case, the contour property and Lemma 6.1 imply that $r = 1$ as follows: if $r \geq 2$, then the contour property implies that $c(v_2) = c(x)$, and Lemma 6.1 implies that v_2 and x cannot simultaneously inhabit p's deque. □

Sections 7 and 8 use the structure of the workers' deques to bound the time and space, respectively, that PIPER uses to execute a pipeline program. While Section 8 uses contours directly, Section 7 uses a property that PIPER upholds while upholding the contour property. Intuitively, the analysis in Section 7 uses a measurement of the "distance" of each vertex in a worker's deque from the final vertex in the computation dag. To formalize this intuition, for a computation dag $G = (V,E)$, we define the ***enabling tree*** $G_T = (V,E_T)$ as the tree containing an edge $(u,v) \in E_T$ if u is the last predecessor of v to execute. The ***enabling depth*** $d(u)$ of $u \in V$ is the depth of u in the enabling tree G_T. Section 7 performs its analysis using the following "depth property," which roughly states

that, except perhaps for the topmost vertex, the vertices in a worker's deque are sorted from bottom to top in order of decreasing depth:

Definition 6.5. 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. This 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 $k > 1$.

The depth property handles the topmost vertex in a worker's deque as a special case because the tail swap operation impedes our ability to relate the depth of this vertex to the depths of the other vertices in the deque. Although this special case prevents us from analyzing PIPER by simply applying the analysis of Arora et al. [Arora et al. 2001], Section 7 extends the analysis of Arora et al. to overcome this hurdle.

The following theorem shows that, during the execution of a pipeline program by PIPER, all workers' extended deques satisfy both the contour and the depth properties.

THEOREM 6.6. *At all times during an execution of a pipeline program by* PIPER*, all extended deques satisfy the contour and depth properties (Definitions 6.3 and 6.5).*

PROOF. The proof follows a similar induction to the proof of Lemma 3 from [Arora et al. 2001]. Intuitively, we replace the "designated parents" discussed in [Arora et al. 2001] with contours, which exhibit similar parent-child relationships.

The claim holds vacuously in the base case, that is, for any empty deque.

Assuming inductively that the statement is true, consider the possible actions of PIPER that modify the contents of the deque. For $r \geq 1$, let $v_0, v_1, \ldots, v_r$ denote the vertices on p's extended deque before p executes v_0, and let $v'_0, v'_1, \ldots, v'_{r'}$ denote the vertices on p's extended deque afterwards. A worker p may execute its assigned vertex v_0, thereby enabling 0, 1, or 2 vertices, or another worker q may steal a vertex from the top of the deque.

Worker q steals a vertex from p's deque. The statement holds because the identities of the remaining vertices in p's deque are unchanged. Similarly, the claim holds vacuously for q because q's extended deque has only the stolen vertex.

Executing v_0 enables 0 vertices. Worker p pops v_1 from the bottom of its deque to become its new assigned vertex v'_0. This action shifts all vertices in the deque down, that is, $r' = r - 1$ and for all k we have $v'_k = v_{k+1}$. The statement holds because the identities of the remaining vertices in p's deque are unchanged.

Executing v_0 enables 1 vertex u. Worker p changes its assigned vertex from v_0 to $v'_0 = u$ and leaves all other vertices in the deque unchanged, that is, $r' = r$ and $v'_k = v_k$ for all $k > 1$. For vertices $v_2, v_3, \ldots, v_r$, if they exist, both Definitions 6.3 and 6.5 hold by induction. We therefore only need to consider the relationship between u and v_1.

The contour property holds by induction if $c(u) = c(v_0)$, that is, if the edge (v_0, u) is a serial or continue edge. The depth property also holds by induction because we are replacing v_0 on the extended deque with a successor node u, and thus $d(u) > d(v_0)$. Consequently, we need only consider the cases where (v_0, u) is either a spawn edge, a return edge, a cross edge, or a throttling edge.

— Edge (v_0, u) cannot be a spawn edge because executing a spawn node always enables 2 children.
— If (v_0, u) is a return edge, then $c(u)$ is the parent of $c(v_0)$. In this case, we can show that p's deque is empty, in which case the properties hold vacuously. For the sake of contradiction, suppose that p's deque contains a vertex v_1. By the inductive hypothesis, we have two cases:
  (1) Contour $c(v_1)$ is the parent of contour $c(v_0)$. In this case, we have $c(v_1) = c(u)$. Lemma 6.1 tells us, however, that u and v_1 cannot simultaneously inhabit p's deque. Therefore, v_1 cannot exist in p's deque.

(2) The root of $c(v_0)$ is the root of some iteration i of a `pipe_while` loop and the root of $c(v_0)$ is the root of some iteration $i+1$ of that loop. In this case, u terminates the `pipe_while` loop. One of u's predecessors, however, is the terminal vertex $s_{i+1,end}$ of iteration $i+1$, which vertex v_1 precedes in the dag. Vertex v_1 therefore cannot exist in p's deque.

— If (v_0,u) is a throttling edge, then Lemma 6.4 specifies the structure of worker p's extended deque. In particular, Lemma 6.4 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 u into p's deque. The contour property holds, because $c(u)$ is the parent of $c(v_1)$. The depth property holds, because z is enabled by a throttling edge.

— Suppose that (v_0,u) is a cross edge. Lemma 6.2 shows that a cross edge (v_0,u) can only exist between vertices in sibling iteration contours. By the inductive hypothesis, $c(v_1)$ must be either the parent of $c(v_0)$ or equal to $c(u)$. In the latter case, however, enabling u would place two vertices from $c(u)$ on the same deque, which Lemma 6.1 implies is impossible. Contour $c(v_1)$ is therefore the common parent of $c(v_0)$ and $c(u)$, and thus setting $v_0' = u$ maintains the contour property. The depth property holds because u is a successor of v_0, and $d(u) > d(v_0)$.

Executing v_0 enables 2 vertices, u and w. Without loss of generality, assume that PIPER pushes the vertex w onto the bottom of its deque and assigns itself vertex u. Hence, we have $r' = r+1$, $v_0' = u$, $v_1' = w$, and $v_k' = v_{k-1}$ for all $1 < k \leq r'$. Definition 6.5 holds by induction, because the enabling edges (v_0,u) and (v_0,w) imply that $d(v_0) < d(u) = d(w)$. For vertices $v_2, v_3, \ldots, v_r$, if they exist, Definition 6.3 holds by induction. We therefore need only verify Definition 6.3 for vertices u and w.

To enable 2 vertices, v_0 must have at least 2 outgoing edges. We therefore have only three cases for v_0: either v_0 executes a `cilk_spawn`, or v_0 is the terminal of some iteration i in a `pipe_while` loop, or v_0 is a stage-counter vertex.

— If v_0 executes a `cilk_spawn`, then $c(w) = c(v_0)$ and $c(u)$ is a child contour of $c(v_0)$, maintaining Definition 6.3.

— If v_0 is the terminal of an iteration i, then it might have up to 3 outgoing edges: a cross edge (v_0,a), a throttling edge (v_0,x), and a return edge (v_0,z). We first observe that at most one of x or z is enabled, because both x and z belong to the same contour — the parent contour of $c(v_0)$ — and Lemma 6.1 therefore precludes enabling both x and z simultaneously. Consequently, u is the destination a of the cross edge and w is one of x or z.
We now justify that the new contents of the deque satisfy Definition 6.3. By Lemma 6.2, we have that $c(v_0)$ and $c(u)$ are sibling contours corresponding to the adjacent iterations i and $i+1$, respectively, of a `pipe_while` loop, and $c(w)$ is the parent of both $c(v_0)$ and $c(u)$. Furthermore, we can show as follows that p's deque is otherwise empty. Lemma 6.4 specifies that the deque contains at most one vertex v_1, where the root of $c(v_1)$ is the root of iteration $i+1$. We therefore have $c(u) = c(v_1)$, and therefore, Lemma 6.1 states that u and v_1 cannot simultaneously appear on p's deque. Because executing v_0 enabled u, vertex v_1 cannot exist on p's deque, implying that p's deque is empty.

— If v_0 is a stage-counter vertex that is not the terminal of an iteration, then w must be the destination of a cross edge. In this case, vertex u is a node root in the same contour $c(u) = c(v_0)$, and by Lemma 6.2, $c(u)$ and $c(w)$ are adjacent siblings in the contour tree. As such, we need only show that $c(v_1)$, if it exists, is their parent. Suppose that $c(v_1)$ is not the parent of $c(v_0) = c(u)$, for the sake of contradiction. Then, by induction, it must be that $c(u)$ and $c(v_1)$ are adjacent siblings. We therefore have that $c(v_1) = c(w)$, which Lemma 6.1 shows is impossible.

□

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 $T_P \le T_1/P + O(T_\infty + \lg P + \lg(1/\varepsilon))$ with probability at least $1-\varepsilon$, which implies that the expected running time is $\mathrm{E}[T_P] \le T_1/P + O(T_\infty + \lg P)$. This bound is comparable to the work-stealing bound for fork-join dags originally proved in [Blumofe and Leiserson 1999].

We adapt the potential-function argument of Arora et al. [Arora et al. 2001], because PIPER executes computation dags in a style similar to the 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 of Blumofe et al. [Blumofe and Leiserson 1999], which contributes the additive $O(\lg P)$ term to the bounds.

The crux of the analysis is to bound the number of steal attempts performed during the execution of a computation dag in terms of its span. Following the analysis of Arora et al., 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 $G = (V,E)$ with span T_∞ 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 $\phi(p) = \sum_{k=0}^{r} \phi(v_k)$. The total potential of a computation at a given time is simply the sum of the potentials of each worker's deque. Arora et al. show that every action of their scheduler decreases the total potential over the course of the program's execution. We likewise use decreases in the total potential to measure the progress PIPER makes in executing a computation.

Given this potential function, the proof of the time bound follows the same overall structure as the analysis in [Arora et al. 2001]. Conceptually, their analysis relies on the fact that the topmost node in each worker's deque has the smallest depth of all of the nodes in the deque, and therefore it accounts for a constant fraction of the total potential of the deque. Arora et al. use this property to argue that, after P steal attempts, with high probability, the topmost vertex in a particular deque is being executed, which implies that the deque's potential has dropped by a constant fraction and, therefore, that progress has been made.

Our potential-function analysis differs from that of Arora et al. because of the addition of the tail-swap operation in PIPER. First, we show that the potential of a worker's deque decreases when a tail-swap occurs. Second, as the depth property (Definition 6.5) shows, the tail-swap causes the topmost vertex in each worker's deque to not necessarily have the smallest depth. Consequently, this topmost vertex does not necessarily account for a constant fraction of the deque's total potential, meaning that the analysis of Arora et al. does not directly apply to PIPER. To overcome this hurdle, conceptually, our analysis argues that the top two vertices in each deque account for a constant fraction of the deque's potential, and after $2P$ steal attempts, with high probability, a given deque's potential has probably decreased by a constant fraction, implying that progress has been made. The following lemmas and theorem formalize this conceptual approach.

First, we prove that a constant fraction of the potential of a worker's deque lies in its top two vertices and that every action of PIPER on a deque decreases that deque's potential.

LEMMA 7.1. *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 properties:*

(1) $\phi(v_r) + \phi(v_{r-1}) \ge 3\phi(p)/4$.

(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. The analysis to show Property 1 is analogous to the analysis of Arora et al. [Arora et al. 2001, Lem. 6]. Because the result holds trivially for $r \leq 1$, we focus on the case where $r \geq 2$. Because Theorem 6.6 shows that p's extended deque satisfies the depth property, we have

$$d(v_0) \geq d(v_1) > d(v_2) > \cdots > d(v_{r-2}) > d(v_{r-1}) .$$

The definition of the weight of a vertex, $w(v) = T_\infty - d(v)$, therefore gives us

$$w(v_0) \leq w(v_1) < w(v_2) < \cdots < w(v_{r-2}) < w(v_{r-1}) .$$

Because all weights are integers, we have that $w(v_{k-1}) \leq w(v_k) - 1$ for $k = 2, 3, \ldots, r-1$. Equivalently, we can bound all w_{v_k} in terms of $w(v_{r-1})$ as

$$w(v_k) \leq \begin{cases} w(v_{r-1}) - (r-1-k) & \text{if } 1 \leq k \leq (r-1) , \\ w(v_{r-1}) - (r-2) & \text{if } k = 0 . \end{cases}$$

For $r \geq 2$, we therefore have

$$\begin{aligned}
\phi(p) &= \sum_{k=1}^{r} 3^{2w(v_k)} + 3^{2w(v_0)-1} \\
&= 3^{2w(v_r)} + 3^{2w(v_{r-1})} + \sum_{k=1}^{r-2} 3^{2w(v_k)} + \frac{1}{3} \cdot 3^{2w(v_0)} \\
&\leq 3^{2w(v_r)} + 3^{2w(v_{r-1})} + \left(\sum_{k=1}^{r-2} \frac{1}{3^{2(r-k-1)}} \right) \cdot 3^{2w(v_{r-1})} + \frac{1}{3} \cdot \frac{1}{3^{2(r-2)}} \cdot 3^{2w(v_{r-1})} \\
&= 3^{2w(v_r)} + 3^{2w(v_{r-1})} + \left(\frac{1}{8} + \frac{5}{24 \cdot 3^{2(r-2)}} \right) \cdot 3^{2w(v_{r-1})} \\
&\leq 3^{2w(v_r)} + 3^{2w(v_{r-1})} + \frac{1}{3} \cdot 3^{2w(v_{r-1})} \\
&= 3^{2w(v_r)} + \frac{4}{3} \cdot 3^{2w(v_{r-1})} \\
&\leq \frac{4}{3} \cdot \left(3^{2w(v_r)} + 3^{2w(v_{r-1})} \right) ,
\end{aligned}$$

and thus $\phi(v_r) + \phi(v_{r-1}) \geq 3\phi(p)/4$.

Now we argue that, in any time step t during which worker p executes its assigned vertex v_0, the potential of p's extended deque decreases. Let ϕ' denote the potential after the time step. If v_0 is the terminal of an iteration i and PIPER performs a tail swap after executing v_0, then Lemma 6.4 dictates the state of the deque before and after p executes v_0, from which we deduce that $\phi(p) - \phi'(p) = 2(\phi(v_0) + \phi(v_1))/3$. The remaining cases follow from the analysis of Arora et al. [Arora et al. 2001], which shows that $\phi(p) - \phi'(p) \geq 5\phi(v_0)/9$. □

As in [Arora et al. 2001], 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 [Arora et al. 2001] to consider the probability that 2 out of $2P$ balls land in the same bin.

LEMMA 7.2. *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

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

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

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

Since $(1-1/P)^P$ approaches $1/e$ and $(3P-1)/(P-1)$ approaches 3, we have $\lim_{P\to\infty} \mathrm{E}[W_p - X_p] = 3W_p/e^2$. In fact, one can show that $\mathrm{E}[W_p - X_p]$ 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 7.2 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 7.3. *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 [Arora et al. 2001] with two main differences. First, we use the two properties of ϕ in Lemma 7.1. Second, we must consider the case unique to PIPER, where p performs a tail swap after executing its assigned vertex v_0.

We first observe that, if p is the target of at least 2 steal attempts, then PIPER's actions on p's extended deque between time steps t and t' contribute a potential drop of at least $\phi(p)/2$. Let $\langle v_0, v_1, \ldots, v_r \rangle$ denote the vertices on p's extended deque at time t, and suppose that at least 2 steal attempts target p between time step t and time step t'.

— If p's extended deque is empty, then $\phi(p) = 0$, and the statement holds trivially.
— If $r = 0$, then p's extended deque consists solely of a vertex v_0 assigned to p, and $\phi(p) = \phi(v_0)$. By time t', worker p has executed vertex v_0, and Property 2 in Lemma 7.1 shows that the potential decreases by at least $5\phi(p)/9 \geq \phi(p)/2$.
— Suppose that $r > 1$. By time t', both v_r and v_{r-1} have been removed from p's deque, either by being stolen or by being assigned to p. In either case, the overall potential decreases by more than $2\phi(v_r)/3 + 2\phi(v_{r-1})/3$ — the decrease in potential from simply assigning v_r and v_{r-1} — which is $2\left(\phi(v_r) + \phi(v_{r-1})\right)/3 \geq 2\left(3\phi(p)/4\right)/3 = \phi(p)/2$ by Lemma 7.1.

— Suppose that $r = 1$. If executing v_0 causes p to perform a tail swap, then by Lemma 7.1, the potential drops by at least $2(\phi(v_0)+\phi(v_1))/3 \geq \phi(p)/2$, since $\phi(p) = \phi(v_0)+\phi(v_1)$. Otherwise, by Lemma 7.1, the execution of v_0 results in an overall decrease in potential of $5\phi(v_0)/9$, and because p is the target of at least 2 steal attempts, v_1 is assigned by time t', decreasing the potential by $2\phi(v_1)/3$.

We now consider all P workers and $2P$ steal attempts between time steps t and t'. We model these steal attempts as ball tosses in the experiment described in Lemma 7.2. 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 7.2 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 7.2, 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 7.4. *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 time steps of 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 7.3, 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 [Arora et al. 2001], we use the Chernoff bound $\Pr\{X < np - a\} < e^{-a^2/2np}$, with $a = m/6$. Choosing $m = 48T_\infty + 24\ln(1/\varepsilon)$, we have

$$\Pr\{X < 8T_\infty\} < e^{\frac{-(m/6)^2}{16T_\infty + m/3}} < e^{\frac{-(m/6)^2}{m/3 + m/3}} = e^{-m/24} \leq \varepsilon\ .$$

Hence, the probability that the execution takes $n = 96T_\infty + 24\ln(1/\varepsilon)$ rounds or more is less than ε, and the number of tokens in the steal bucket is at most $288T_\infty + 72\ln(1/\varepsilon)$.

The additional $\lg P$ term comes from the "recycling game" analysis described in [Blumofe and Leiserson 1999], 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 [Blumofe and Leiserson 1999] 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 a pipeline computation dag G into a tree of contours, in a similar manner to that described in Section 6. 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 ***active*** — c is associated with a vertex in some worker's extended deque — or ***suspended*** — the earliest unexecuted vertex in the contour is not ready.

As Section 4 describes, the contours of a `pipe_while` loop do not directly correspond to the control and iteration frames allocated for the loop. In particular, as demonstrated in the code transformation in Figure 5, stage 0 allocates an iteration frame for all stages of the iteration and executes using that iteration frame. To account for the space used when executing an iteration i of a `pipe_while` loop, consider an active or suspended contour c, and let v be the vertex in c on a worker's deque, if c is active, or the earliest unexecuted vertex in c, if c is suspended. If v lies on a path in c between a stage 0 node root $a_{i,0}$ and its corresponding node terminal $b_{i,0}$ for some iteration i of a `pipe_while` loop, then v incurs an additional space cost equal to the size of i's iteration frame.

The following theorem bounds the stack space used by PIPER. Let S_P denote the maximum over all time steps of the stack space PIPER uses during a P-worker execution of G. Thus, S_1 is the stack space used by PIPER for a serial execution. 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. The following theorem generalizes the space bound $S_P \leq PS_1$ from [Blumofe and Leiserson 1999], which deals only with fork-join parallelism, to pipeline programs.

THEOREM 8.1. *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, at each time step during PIPER's execution of a pipeline program, PIPER satisfies a variant of the "busy-leaves property" [Blumofe and Leiserson 1999] with respect to the tree of active and suspended contours during that time step. This proof follows a similar induction to that in [Blumofe and Leiserson 1999, Thm. 3], with one change, namely, that a leaf contour c may stall if the earliest unexecuted vertex in c is the destination of a cross edge.

If v is the destination of a cross edge, then v must be associated with some `pipe_while` loop, specifically, as a node root $a_{i,j}$ for some iteration i in the loop. The source of this cross edge must be in a previous iteration of the same `pipe_while` loop. Consider the ***leftmost*** iteration $i' < i$ of this `pipe_while` loop, that is, the iteration with the smallest iteration index that has not completed. By definition of the leftmost iteration, all previous iterations in this `pipe_while` have completed, and thus no node root $a_{i',j}$ in iteration i' may be the destination of a cross edge whose source has not executed. In other words, the contour associated with this leftmost iteration must be active or have an active descendant. Consequently, each suspended contour c is associated with a `pipe_while` loop, and for each such contour c, there exists an active sibling contour associated with the same `pipe_while` loop.

We account for the stack space PIPER uses by separately considering the active and suspended leaf contours. Because there are P processors executing the computation, at each time step, PIPER has at most P active leaf contours, each of which may use at most S_1 stack space. Each suspended leaf contour, meanwhile, is associated with a `pipe_while` loop, whose leftmost iteration during this time step either is active or has an active descendant. Any `pipe_while` loop uses at most fK stack space to execute its iterations, because the throttling edge from the terminal of the leftmost iteration precludes having more than K active or suspended iterations in any one `pipe_while` loop. Thus, for each vertex in its deque that is associated with an iteration of a `pipe_while` loop, each worker p accounts for at most fK stack space in suspended sibling contours for iterations of the same `pipe_while` loop, or fDK stack space overall. Summing the stack 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 dynamic dependency folding.

Data structures

Like the Cilk-M runtime [Lee et al. 2010] on which it is based, Cilk-P organizes runtime data into frames. The transformed code in Figure 5 reflects the frames Cilk-P allocates to execute a `pipe_while` loop. In particular, Cilk-P executes a `pipe_while` loop in its own control frame, which 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. As seen in Figure 5, the frame of iteration i maintains a stage counter, which stores the stage number of the current node in i. In addition, the iteration i's frame maintains 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 the earliest active iteration decrements it. (For implementation simplicity, Cilk-P additionally ensures that iterations return in order.) 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 [Frigo et al. 1998] 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 dynamic 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 q tries to steal iteration i's frame from p's deque, q first checks right on behalf of p to see whether any work from iteration $i+1$ is implicitly on the deque. If so, q 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 (because 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, that is, if all other work on p's deque has been stolen. 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*** [Frigo et al. 1998]: 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.

Dynamic dependency folding

In ***dynamic dependency folding***, the frame for iteration i stores a cached value of the stage counter of iteration $i-1$, hoping to avoid checking 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 q is executing iteration $i-1$, because q may be constantly updating the stage counter of iteration $i-1$.

Dynamic 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 q 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,k) where $k > j$. 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 [Bienia et al. 2008; Bienia and Li 2010] 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, that is, using more threads than the number of available hardware cores. We study the effectiveness of dynamic dependency folding on a synthetic benchmark called *pipe-fib*, demonstrating that this optimization can be effective for applications with fine-grained stages.

We ran two sets of experiments on two different machines. The first set was collected on an AMD Opteron 8354 system with 4 2 GHz quad-core CPU's and 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`. The machine was running a custom version of Debian 6.08 (squeeze) modified by MIT CSAIL, using a custom Linux kernel 3.4.0, patched with support for thread-local memory mapping for Cilk-M [Lee et al. 2010]. The second set was collected on an Intel Xeon E5-2665 system with 2 2.4 GHz 8-core CPU's having a total of 32 GBytes of memory. Each processor core has a 32-KByte private L1-data-cache and a 256-KByte private L2-cache. The 8 cores on each chip share the same 20-MByte L3-cache. The benchmarks were compiled with GCC (or G++ for TBB) 4.6.3 using `-O3` optimization, except for *x264*, which by default comes with `-O4`. The machine was running

Fedora 16, using a custom Linux kernel 3.6.11, also patched with support for thread-local memory mapping for Cilk-M. This Intel Xeon machine is several years newer than the AMD machine, which precludes a direct comparison between these systems.

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 [Lee et al. 2010]. 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*** [Reed et al. 2011], 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 [Reed et al. 2011] 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, unless specified otherwise in the figure captions. For Cilk-P on the AMD Opteron system, we used the default throttling limit of $4P$, where P is the number of cores. This default value seems to work well in general, although since *ferret* scales slightly better with less throttling, we used a throttling limit of $10P$ for *ferret* in our experiments. Similarly, for Cilk-P on the Intel Xeon system, a throttling of $10P$ seems to work well, and thus we used a throttling of $10P$ for experiments on the Intel Xeon system. 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 doing so causes the program to deadlock.

Figures 8–10 show the performance results for the different implementations of the three benchmarks running on the AMD Opteron system. Each data point in the study was computed by averaging the results of 10 runs. The standard deviation of the numbers was less than 5% for a majority of the data points, except for a couple outliers for which the standard deviation was about 10%. 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 8–10 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. Recall that 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.

[7] 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.

	Processing Time (T_P)			Speedup (T_S/T_P)			Scalability (T_1/T_P)		
P	Cilk-P	Pthreads	TBB	Cilk-P	Pthreads	TBB	Cilk-P	Pthreads	TBB
1	432.4	430.0	432.7	1.01	1.01	1.00	1.00	1.00	1.00
2	220.4	212.2	223.8	1.97	2.05	1.94	1.96	2.03	1.93
3	146.9	140.8	147.0	2.96	3.09	2.96	2.94	3.05	2.94
4	111.5	106.0	111.8	3.90	4.10	3.89	3.88	4.06	3.87
5	89.2	89.9	90.8	4.87	4.83	4.79	4.85	4.78	4.77
6	74.8	73.8	76.1	5.81	5.88	5.71	5.78	5.82	5.67
7	64.7	64.2	65.9	6.71	6.76	6.59	6.68	6.70	6.57
8	57.3	57.0	57.7	7.58	7.62	7.53	7.54	7.54	7.50
9	51.1	49.8	52.9	8.50	8.72	8.34	8.46	8.64	8.31
10	46.4	45.5	47.3	9.36	9.55	9.19	9.32	9.46	9.16
11	42.5	41.7	43.2	10.22	10.41	10.05	10.18	10.30	10.01
12	39.4	38.6	40.0	11.03	11.26	10.85	10.98	11.15	10.81
13	36.6	37.2	37.6	11.87	11.67	11.54	11.82	11.55	11.49
14	34.4	35.0	35.3	12.64	12.41	12.29	12.58	12.28	12.25
15	32.2	32.9	33.5	13.48	13.19	12.96	13.42	13.06	12.91
16	30.7	31.3	31.8	14.17	13.89	13.67	14.07	13.75	13.61

Fig. 8. Performance comparison of the three *ferret* implementations running on the AMD Opteron system. The experiments were conducted using *native*, the largest input data set that comes with the PARSEC benchmark suite.[7] The left-most column shows the number of cores used (P). Subsequent columns show the running time (T_P), speedup (T_S/T_P) over serial running time $T_S = 434.4$ seconds, and scalability (T_1/T_P) for each system. The throttling limit was $K = 10P$.

	Processing Time (T_P)			Speedup (T_S/T_P)			Scalability (T_1/T_P)		
P	Cilk-P	Pthreads	TBB	Cilk-P	Pthreads	TBB	Cilk-P	Pthreads	TBB
1	56.4	51.7	55.9	1.04	1.13	1.05	1.00	1.00	1.00
2	29.5	23.5	29.5	1.99	2.49	1.98	1.91	2.20	1.89
3	20.1	15.8	20.4	2.91	3.71	2.88	2.80	3.28	2.75
4	15.8	12.4	16.1	3.71	4.73	3.64	3.57	4.17	3.47
5	13.5	11.3	13.7	4.33	5.19	4.28	4.16	4.58	4.09
6	11.9	10.5	12.1	4.92	5.56	4.85	4.73	4.90	4.63
7	10.8	9.5	11.0	5.42	6.18	5.33	5.22	5.45	5.09
8	10.1	8.6	10.2	5.82	6.81	5.73	5.61	6.01	5.48
9	9.5	7.6	9.6	6.15	7.69	6.09	5.92	6.79	5.81
10	9.1	7.1	9.2	6.42	8.28	6.35	6.18	7.31	6.07
11	8.8	6.8	9.0	6.65	8.61	6.54	6.40	7.59	6.24
12	8.6	6.7	8.8	6.80	8.75	6.67	6.54	7.72	6.37
13	8.5	6.7	8.7	6.93	8.80	6.76	6.67	7.76	6.46
14	8.3	6.5	8.6	7.04	9.04	6.79	6.78	7.98	6.49
15	8.3	6.2	8.6	7.08	9.48	6.84	6.82	8.37	6.54
16	8.2	6.0	8.5	7.12	9.76	6.88	6.85	8.61	6.57

Fig. 9. Performance comparison of the three *dedup* implementations running on the AMD Opteron system. 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 8. The serial running time T_S was 58.6 seconds. The throttling limit was $K = 4P$.

First, the *dedup* benchmark on the test input has limited parallelism. We modified the Cilkview scalability analyzer [He et al. 2010] to measure the work and span of our hand-compiled Cilk-P *dedup* programs, and we measured the parallelism of *dedup* to be 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 to write 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 factor of P for $P = 1$–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 original serial implementation of *dedup*. This performance boost may be explained if the computation and file I/O operations are effectively

	Encoding Time (T_P)		Speedup (T_S/T_P)		Scalability (T_1/T_P)	
P	Cilk-P	Pthreads	Cilk-P	Pthreads	Cilk-P	Pthreads
1	211.1	219.8	1.04	0.99	1.00	1.00
2	99.1	103.5	2.21	2.11	2.13	2.12
3	65.3	67.8	3.35	3.22	3.23	3.24
4	49.6	51.6	4.40	4.23	4.25	4.26
5	40.9	42.0	5.34	5.21	5.16	5.24
6	34.4	35.8	6.35	6.11	6.13	6.14
7	29.9	31.5	7.31	6.94	7.06	6.98
8	26.7	28.5	8.20	7.66	7.92	7.70
9	23.9	25.8	9.13	8.49	8.82	8.53
10	22.0	23.0	9.92	9.50	9.58	9.55
11	20.5	21.0	10.68	10.41	10.31	10.47
12	19.1	19.5	11.47	11.18	11.07	11.25
13	18.0	18.7	12.12	11.69	11.70	11.76
14	17.1	17.4	12.82	12.56	12.38	12.63
15	16.4	16.5	13.34	13.21	12.88	13.28
16	15.8	16.0	13.81	13.67	13.34	13.75

Fig. 10. Performance comparison between the Cilk-P implementation and the Pthreaded implementation of *x264* (encoding only) running on the AMD Opteron system. 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 8. The serial running time T_S was 218.6 seconds. The throttling limit was $K = 4P$.

	Processing Time (T_P)			Speedup (T_S/T_P)			Scalability (T_1/T_P)		
P	Cilk-P	Pthreads	TBB	Cilk-P	Pthreads	TBB	Cilk-P	Pthreads	TBB
1	153.2	152.5	151.5	1.04	1.04	1.05	1.00	1.00	1.00
2	77.5	89.7	77.0	2.05	1.77	2.06	1.98	1.70	1.97
3	51.8	56.9	53.5	3.07	2.79	2.97	2.96	2.68	2.83
4	39.9	42.8	40.0	3.99	3.72	3.98	3.84	3.57	3.79
5	32.3	34.4	32.5	4.93	4.63	4.89	4.75	4.44	4.66
6	27.4	29.7	27.6	5.81	5.36	5.75	5.59	5.14	5.48
7	23.5	25.7	24.0	6.76	6.18	6.61	6.51	5.93	6.30
8	21.0	22.7	21.3	7.57	7.00	7.45	7.29	6.72	7.10
9	19.0	21.8	19.4	8.37	7.31	8.20	8.07	7.01	7.81
10	17.3	18.8	17.8	9.19	8.46	8.94	8.86	8.11	8.52
11	15.8	17.3	16.4	10.05	9.20	9.68	9.68	8.82	9.23
12	14.7	15.8	15.3	10.81	10.08	10.37	10.42	9.67	9.88
13	13.8	14.6	14.5	11.52	10.88	10.98	11.10	10.43	10.47
14	13.0	13.7	13.8	12.20	11.64	11.56	11.76	11.17	11.02
15	12.2	12.9	13.1	12.98	12.30	12.15	12.51	11.80	11.58
16	11.6	12.2	12.5	13.77	13.02	12.70	13.26	12.48	12.10

Fig. 11. Performance comparison of the three *ferret* implementations running on the Intel Xeon system. The experiments were conducted using *native*, and the column headers are the same as in Figure 8. The serial running time T_S was 159.0 seconds. The throttling limit was $K = 10P$.

overlapped. With multiple threads per stage, the Pthreaded implementation performs better, however, since throttling appears to inhibit threads working on stages that are further ahead, allowing threads working on heavier stages to obtain more processing resources, thereby balancing the load.

Figures 11–13 show the performance results for the different implementations of the three benchmarks running on the Intel Xeon system. The relative performance between the three implementations for each benchmark follows similar trend as in the results from running on the AMD Opteron system. The two sets of results differ in the following ways. First, the serial running time across implementations for each benchmark is about 2–3 times faster on the Intel Xeon system than on the AMD Opteron system. This discrepancy of serial running times can be explained by the fact that the Intel Xeon processors have higher clock frequency and that the system overall has more memory. In addition, the memory bandwidth on the Intel system is about 2–3 times higher than on the AMD system[8], depending on the data size accessed by the computation. Second, we observed less speedup from *dedup* across the three implementations on the Intel system than on the AMD system. The reason for this smaller speedup is because the number of last-level cache misses is increased

[8]We measured the memory latencies using the latest software release of lmbench [McVoy and Staelin 1996] at the time of printing, that is, version 3.0-a9.

	Processing Time (T_P)			Speedup (T_S/T_P)			Scalability (T_1/T_P)		
P	Cilk-P	Pthreads	TBB	Cilk-P	Pthreads	TBB	Cilk-P	Pthreads	TBB
1	29.6	29.2	30.1	1.01	1.02	0.99	1.00	1.00	1.00
2	18.0	14.6	21.1	1.65	2.03	1.41	1.64	2.00	1.43
3	12.8	10.0	18.6	2.32	2.96	1.60	2.30	2.92	1.62
4	10.4	8.2	15.7	2.85	3.62	1.89	2.83	3.56	1.92
5	9.3	7.5	15.4	3.20	3.95	1.94	3.19	3.89	1.96
6	8.5	8.0	14.5	3.48	3.71	2.05	3.46	3.65	2.08
7	8.0	7.2	13.7	3.70	4.12	2.17	3.68	4.05	2.20
8	7.6	6.5	13.9	3.89	4.55	2.13	3.86	4.48	2.16
9	7.5	7.2	13.3	3.94	4.13	2.24	3.92	4.06	2.27
10	7.5	6.7	12.5	3.95	4.42	2.38	3.93	4.35	2.41
11	7.6	7.1	12.1	3.91	4.17	2.45	3.89	4.10	2.49
12	7.8	7.3	11.2	3.81	4.05	2.65	3.79	3.98	2.69
13	8.1	7.5	11.1	3.67	3.97	2.68	3.65	3.90	2.72
14	8.5	7.8	11.0	3.50	3.84	2.70	3.48	3.77	2.74
15	8.8	8.8	10.4	3.38	3.36	2.85	3.36	3.30	2.89
16	9.3	9.1	10.4	3.21	3.26	2.86	3.19	3.20	2.90

Fig. 12. Performance comparison of the three *dedup* implementations running on the Intel Xeon system. The experiments were conducted using *native*, and the column headers are the same as in Figure 8. The serial running time T_S was 29.7 seconds. The throttling limit was $K = 10P$ for Cilk-P and TBB, and $4P$ for Pthreads, because the Cilk-P and TBB implementations performed better with throttling limit of $10P$ than $4P$, whereas the Pthreaded implementation was the other way around.

	Encoding Time (T_P)		Speedup (T_S/T_P)		Scalability (T_1/T_P)	
P	Cilk-P	Pthreads	Cilk-P	Pthreads	Cilk-P	Pthreads
1	95.1	95.2	1.02	1.02	1.00	1.00
2	49.4	51.9	1.97	1.87	1.93	1.83
3	33.4	35.5	2.91	2.73	2.85	2.68
4	25.5	27.9	3.82	3.48	3.74	3.41
5	20.8	22.6	4.66	4.30	4.56	4.21
6	17.5	19.6	5.53	4.97	5.42	4.86
7	15.3	16.9	6.36	5.74	6.22	5.62
8	13.4	14.9	7.24	6.50	7.08	6.37
9	12.2	13.4	7.97	7.26	7.80	7.11
10	11.1	12.1	8.73	8.01	8.55	7.84
11	10.4	11.0	9.37	8.83	9.17	8.65
12	9.6	10.2	10.07	9.54	9.85	9.34
13	9.2	9.5	10.55	10.21	10.32	10.00
14	8.8	9.0	11.04	10.80	10.80	10.58
15	8.3	8.6	11.68	11.34	11.43	11.12
16	8.0	8.3	12.08	11.77	11.82	11.53

Fig. 13. Performance comparison between the Cilk-P implementation and the Pthreaded implementation of *x264* (encoding only) running on the Intel Xeon system. The experiments were conducted using *native*, and the column headers are the same as in Figure 8. The serial running time T_S was 97.1 seconds. The throttling limit was $K = 10P$ for Cilk-P and $4P$ for Pthreads, because the Cilk-P implementation performed better with throttling limit of $10P$ than $4P$, whereas the Pthreaded implementation was the other way around.

substantially on the Intel system between a serial execution and a parallel execution. This increase in number of cache misses causes the time spent in the user code to double when running on 16 processors compared with running serially. The AMD system, on the other hand, does not appear to exhibit the same cache behavior. We are unsure of what exactly creates this discrepancy in cache behavior between the two systems, but our measurements suggest that these additional cache misses come mostly from the compress library used by *dedup*, for which we lack the source code.

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, [Bocchino, Jr. et al. 2009]) compared with the inherently nondeterministic code required to set up Pthreaded pipelines.

Program	*Dependency Folding*	T_S	T_1	T_{16}	*Serial Overhead*	*Speedup* T_S/T_{16}	*Scalability* T_1/T_{16}
pipe-fib	no	20.7	23.5	4.7	1.13	4.40	4.98
pipe-fib-256	no	20.7	21.7	1.7	1.05	12.32	12.90
pipe-fib	yes	20.7	21.7	1.8	1.04	11.65	12.17
pipe-fib-256	yes	20.7	21.7	1.7	1.05	12.43	13.02

Fig. 14. Performance evaluation using the *pipe-fib* benchmark on the AMD Opteron system. We tested the Cilk-P system with two different programs, the ordinary *pipe-fib*, and *pipe-fib-256*, which is coarsened. Each program is tested with and without the dynamic 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.

Evaluation of dynamic dependency folding

We also studied the effectiveness of dynamic 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. Given that each stage in *pipe-fib* starts with a `pipe_stage_wait`, and each stage contains little work, it serves as an excellent microbenchmark to study the overhead of `pipe_stage_wait`.

Figure 14 shows the performance results[9] on the AMD Opteron system, 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 13%, 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 dynamic dependency folding effectively does.

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, in which parallel workers simultaneously access the same shared memory location. Dynamic dependency folding eliminated much of this overhead for *pipe-fib*, as shown in the third row of Figure 14, leading to scalability that is much closer to 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.

We have done a similar performance study on the Intel Xeon system, and the relative performance of *pipe-fib* and *pipe-fib-256*, with and without dependency folding, show similar trends as in the results on the AMD system. The parallel running times on the Intel system are affected more by the cache misses due to ***false sharing*** on the three arrays, in which parallel workers access different locations in these arrays that happen to lie in the same cache line. This false sharing causes dependency folding to produce less speedup for *pipe-fib* on the Intel system, specifically, a speedup

[9]Figure 14 shows the results from runs with a single input size, but these data are representative of other runs with different input sizes.

of 8.8. This cache effect goes away if we simply use larger data type for the array and coarsen it slightly to avoid false sharing.

In this section, we have empirically evaluated the performance and scalability of the Cilk-P prototype system. Provided that the application has ample parallelism, such as in the case of *ferret* and *x264*, Cilk-P demonstrates good scalability on machines that we tested, which each contain 16 cores. One might wonder, can one expect Cilk-P to continue to scale as future multicore systems contain increasingly more cores? Since Cilk-P implements PIPER, its provably good time bound predicts near-linear speedup assuming the application contains ample parallelism. Due to "scheduling overhead," however, one would not expect an application to scale linearly up to T_1/T_∞ processors, even though theory predicts that the application could use that many processors. The question then becomes how much parallelism is considered ample for the application to scale linearly with respect to the number of processors. The answer is unfortunately architecture dependent, since it is dictated by, for example, how much additional bookkeeping is necessary to enable parallel execution and how many cache misses are incurred due to data migration on a successful steal. Nevertheless, we believe that applications written in Cilk-P should generally scale as well as applications with comparable parallelism written using the baseline Cilk, since the additional work that the runtime performs for pipeline parallelism is within a small constant factor of the original scheduling overhead. Applications in Cilk-P also incur overhead in the form of cache misses to check cross edges. This overhead is akin to the spawn overhead in that one can amortize the overhead against the work done within the stage corresponding to the cross edge. Thus, as long as each stage contains substantial amount of work, this overhead should not impede scalability.

11. PIPELINE THROTTLING

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. We first consider ***uniform*** pipelines, which contain no hybrid stages and in which, for each stage j, node (i, j) is nearly identical across all iterations i. For uniform pipelines, we show that the cost of executing a stage in an iteration is within a constant factor of executing that stage in any other iteration — throttling does not affect the asymptotic performance of PIPER executing $\widehat{G}$.

THEOREM 11.1. *Consider a uniform unthrottled linear pipeline $\widehat{G} = (V, \widehat{E})$. Suppose that* PIPER *throttles the execution of $\widehat{G}$ on P processors using a throttling limit of $K = aP$, for some constant $a > 1$. Then* PIPER *executes $\widehat{G}$ in time $T_P \le (1 + c/a)\widehat{T}_1/P + c(\widehat{T_\infty} + \lg P + \lg(1/\varepsilon))$ for some sufficiently large constant c with probability at least $1 - \varepsilon$, where $\widehat{T}_1$ is the total work in $\widehat{G}$ and $\widehat{T_\infty}$ is the span of $\widehat{G}$.*

PROOF. For convenience, let us suppose that each stage has exactly the same cost in every iteration. It is straightforward to generalize the result such that the cost of a stage may vary by a constant amount between iterations.

Let $\widehat{G}$ denote the computation dag of a uniform pipeline with n iterations and m stages in each iteration. Let W be the total work in each iteration. We assume that W includes the work performed within each stage in an iteration, as well as the cost incurred to checking any cross-edge dependencies from the previous iteration. Then for the unthrottled dag $\widehat{G}$, we know the work is $\widehat{T}_1 = nW$. Similarly, let S be the work of the most expensive serial stage in an iteration.

For the unthrottled uniform dag $\widehat{G}$, any longest path from the beginning of any node $(i, 0)$ to the end of any node $(i + x, m - 1)$ has cost $W + xS$. Any path connecting these nodes is represented by

some interleaving of m vertical steps and x horizontal steps. For any path, the work of all vertical steps is exactly W, since the vertical steps must execute each stage 0 through $m-1$. Each horizontal step executes exactly one stage, thus incurring cost at most xS. In particular, this result implies that $\widehat{T_\infty} = W + nS$, for the path from node $(0,0)$ to $(n-1,m-1)$.

Now consider the work T_1 and the span T_∞ of the throttled dag G. Conceptually, since G adds $n-K$ zero-cost throttling edges to $\widehat{G}$, we have $T_1 = \widehat{T}_1$; the work remains the same.

The throttling edges may increase the span, however. Consider a critical path p through G, which has cost T_∞. Suppose this path p contains $q \geq 0$ throttling edges. Label the throttling edges along p in order of increasing iteration number, with throttling edge k connecting the node terminal of (the subdag corresponding to the execution of) $(i_k, m-1)$ to the node root of $(i_k+K, 0)$, for $1 \leq k \leq q$. Removing all q throttling edges from p splits p into $q+1$ segments $p_0, p_1, \ldots, p_q$. More precisely, let p_0 denote the path from the beginning of node $(0,0)$ to the end of $(i_1, m-1)$, and let p_k denote the path from the beginning of node $(i_k+K, 0)$ to the end of $(i_{k+1}, m-1)$, where $i_{q+1} = n-1$. By our previous result, the cost of p_0 is $W + (i_1+1)S$, and the cost of each segment p_k is $W + (i_{k+1} - i_k - K)S$. We thus have that T_∞, which is the total cost of p, satisfies

$$\begin{aligned}
T_\infty &= W + (i_1+1)S + \sum_{k=1}^{q} (W + (i_{k+1} - i_k - K)S) \\
&= W + qW + (i_{q+1}+1)S - qKS \\
&= W + qW + (n - qK)S \\
&= W + nS + q(W - KS) \\
&= \widehat{T_\infty} + q(W - KS) \ .
\end{aligned}$$

A cut-and-paste argument shows that throttling edges in G can only increase the span if we have $W > KS$; otherwise, one can create an equivalent or longer path by going horizontally through K copies of the most expensive stage, rather than down an iteration i, and across a throttling edge that skips to the beginning of iteration $i+K$.

Applying Theorem 7.4 to the throttled dag G, we know PIPER executes G in expected time T_P which satisfies

$$\begin{aligned}
T_P &\leq \frac{T_1}{P} + c\left(T_\infty + \lg P + \lg(1/\varepsilon)\right) \\
&= \frac{\widehat{T}_1}{P} + c(\widehat{T_\infty} + \lg P + \lg(1/\varepsilon)) + cq(W - KS) \ .
\end{aligned}$$

If $q = 0$, then the extra $cq(W-KS)$ term is 0, giving the desired bound. Otherwise, assume $q > 0$. Since every throttling edge skips ahead K iterations, we know that the critical path uses at most $q < n/K$ throttling edges. Using this bound for q and letting $K = aP$ for some constant $a > 1$, we can rewrite the expression for T_P as

$$\begin{aligned}
T_P &\leq \frac{\widehat{T}_1}{P} + c(\widehat{T_\infty} + \lg P + \lg(1/\varepsilon)) + \frac{cn}{aP}(W - aPS) \\
&= \frac{\widehat{T}_1}{P} + c(\widehat{T_\infty} + \lg P + \lg(1/\varepsilon)) + \left(\frac{c}{a}\right)\left(\frac{nW}{P}\right)\left(1 - \frac{aPS}{W}\right) \\
&= \frac{\widehat{T}_1}{P} + c(\widehat{T_\infty} + \lg P + \lg(1/\varepsilon)) + \left(\frac{c}{a}\right)\left(\frac{\widehat{T}_1}{P}\right)\left(1 - \frac{aPS}{W}\right) \\
&= \left(1 + \frac{c}{a}\left(1 - \frac{aPS}{W}\right)\right)\frac{\widehat{T}_1}{P} + c(\widehat{T_\infty} + \lg P + \lg(1/\varepsilon)) \ .
\end{aligned}$$

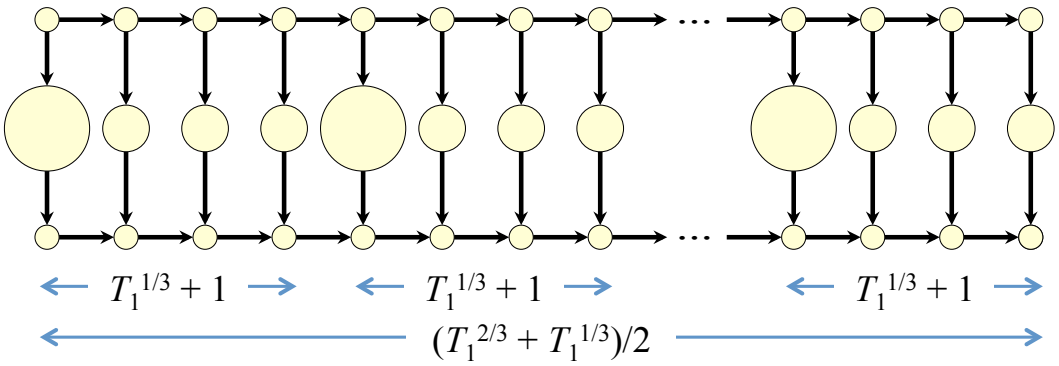

Fig. 15. Sketch of the pathological unthrottled linear pipeline dag, which can be used to prove Theorem 11.2. Small circles represent nodes with unit work, medium circles represent nodes with $T_1^{1/3}-2$ work, and large circles represent nodes with $T_1^{2/3}-2$ work. The number of iterations per cluster is $T_1^{1/3}+1$, and the total number of iterations is $(T_1^{2/3}+T_1^{1/3})/2$.

We know $W \geq KS = aPS$ for the throttling edges to be included as part of the critical path. In general, W could be much larger than aPS, giving us the final result:

$$T_P \leq \left(1+\frac{c}{a}\right)\frac{\widehat{T}_1}{P} + c(\widehat{T_\infty} + \lg P + \lg(1/\varepsilon)) \ .$$

□

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 15 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 11.2. *Let $\widehat{G} = (V,\widehat{E})$ denote the nonuniform unthrottled linear pipeline shown in Figure 15, with work T_1 and span $T_\infty \leq 2T_1^{2/3}$. 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 $T_P \leq T_1/\rho$, where ρ satisfies $3 \leq \rho \leq O(T_1/T_\infty)$, uses space $S_P \geq S_1 + (\rho-2)T_1^{1/3}/2$.*

PROOF. Consider the pipeline computation shown in Figure 15. Suppose that a scheduler executing the pipeline dag requires $S_1 + x - 1$ space to execute x iterations of the pipeline simultaneously, that is, $S_1 - 1$ stack space to execute the function containing the pipeline, plus unit space per pipeline iteration the scheduler executes in parallel. Consequently, the scheduler executes the pipeline serially using S_1 space, and incurs an additional unit space overhead per pipeline iteration it executes in parallel. Furthermore, suppose that each node is a serial computation, that is, no nested parallelism exists in the nodes of Figure 15.

Consider a time step during which the scheduler is executing instructions from k heavy iterations in parallel. Because stage 0 of the pipeline in Figure 15 is serial, to execute instructions from k heavy iterations in parallel requires executing the node for stage 0 in at least $(k-1)T_1^{1/3}+1$ consecutive iterations. Because stage 2 is serial, the scheduler must have executed stage 0, but not stage 2, for at least $(k-1)T_1^{1/3}+1$ iterations. Consequently, the scheduler requires at least $S_P \geq S_1 + (k-1)T_1^{1/3}$ stack space to execute instructions from k heavy iterations in parallel.

We now bound the number k of heavy iterations the scheduler must execute in parallel to achieve a speedup of ρ. The total time T_P the scheduler takes to execute the instructions of the pipeline in

Figure 15 is at least the total time it takes to execute all $T_1^{2/3} \cdot T_1^{1/3}/2 = T_1/2$ instructions in heavy iterations. Assuming the scheduler executes instructions from k heavy iterations simultaneously on each time step it executes an instruction from a heavy iteration, we have $T_P \geq T_1/(2k)$. Rearranging terms gives us that $k \geq T_1/(2T_P) = \rho/2$.

Combining these bounds, we find that the scheduler requires at least $S_P \geq S_1 + (\rho - 2)T_1^{1/3}/2$ space to achieve a speedup of ρ when executing the pipeline in Figure 15. □

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. CONCLUSION

This paper introduces Cilk-P, a system that extends the Cilk parallel-programming model to support on-the-fly pipeline parallelism. In this paper, we describe new language constructs — the `pipe_while` loop and the `pipe_stage_wait` and `pipe_stage` statements — that extend fork-join parallel languages such as Cilk to support on-the-fly pipeline parallelism. We present PIPER, a work-stealing scheduler that executes computations containing `pipe_while` loops in a provably time- and space-efficient manner. We have incorporated these linguistics and the scheduler into a prototype Cilk-P implementation. We demonstrate effectiveness of Cilk-P in practice by parallelizing three benchmarks from the PARSEC suite using our Cilk-P prototype and showing that these implementations are competitive with alternative versions coded using TBB or Pthreads. In particular, we show that `pipe_while` loops are expressive enough for parallelizing *x264*, a benchmark with a complicated pipelining structure which is difficult to express using the pipeline model supported by TBB.

Our investigation also highlights several limitations of our language constructs, however, which represent potential areas for future study. First, because our constructs only allow programmers to specify dependencies between consecutive `pipe_while` loop iterations, it is natural to ask whether one might extend the loop construct to support more generic pipelines. For example, one could imagine extending the `pipe_stage_wait` construct to allow for dependencies on the same stage in multiple preceding iterations, or perhaps even different stages from preceding iterations. In our study, we chose to limit dependencies to only adjacent iterations because this limitation simplifies the semantics of the language construct for pipelining and allows for an efficient implementation. The design of PIPER and the lazy enabling optimization both depend on the fact that dependencies are only between consecutive iterations. In a `pipe_while` loop, when computation of a node in iteration i finishes, the only node in a future iteration that might be enabled is node in iteration $i+1$. Both the runtime implementation and the mathematical analysis become more complicated if nodes in multiple iterations might be enabled by the completion of node in iteration i. On the other hand, extending the loop construct to allow dependencies within a constant-sized sliding window of iterations might be useful for some applications. An interesting open question is whether one can support a more expressive pipeline loop construct without a significant increase in complexity in the linguistic interface or runtime implementation.

It might also be interesting to study whether `pipe_while` loops might be useful for simplifying programs written using other parallel programming models or runtimes. In this work, we focus on integrating `pipe_while` loops with Cilk-like languages, which typically focus on fork-join parallelism and utilize a work-stealing scheduler. The `pipe_while` construct itself is agnostic, however, to the scheduling technique used by the underlying runtime. For example, the semantics of a `pipe_while` loop should be equally useful for programs written using OpenMP, which more often than not assume a work-sharing environment. Depending on the nature of the workload and the available parallelism in the application, alternative scheduler implementations might improve performance.

Finally, an open question about pipeline parallelism is whether one might be able to improve the interaction of `pipe_while` loops with file I/O. For instance, in *dedup*, having a single designated thread that handles the file I/O seems to improve performance. In a dynamic multithreaded language such as Cilk, the file I/O stage for different iterations likely ends up being processed by different workers, because the runtime does not distinguish between I/O stages and computation stages when scheduling a `pipe_while` loop. In principle, one might see improved performance if the scheduler and runtime are aware of the I/O characteristics of pipelines and schedule accordingly.

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.

REFERENCES

Kunal Agrawal, Charles E. Leiserson, and Jim Sukha. 2010. Executing Task Graphs Using Work-Stealing. In *IPDPS*. IEEE, 1–12.

Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. 2001. Thread Scheduling for Multiprogrammed Multiprocessors. *Theory of Computing Systems* (2001), 115–144.

Henry C. Baker, Jr. and Carl Hewitt. 1977. The incremental garbage collection of processes. *SIGPLAN Notices* 12, 8 (1977), 55–59.

Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *PACT*. ACM, 72–81.

Christian Bienia and Kai Li. 2010. Characteristics of Workloads Using the Pipeline Programming Model. In *ISCA*. Springer-Verlag, 161–171.

Guy E. Blelloch and Margaret Reid-Miller. 1997. Pipelining with futures. In *SPAA*. ACM, 249–259.

Robert D. Blumofe and Charles E. Leiserson. 1998. Space-Efficient Scheduling of Multithreaded Computations. *SIAM J. Comput.* 27, 1 (Feb. 1998), 202–229.

Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling Multithreaded Computations by Work Stealing. *JACM* 46, 5 (1999), 720–748.

Robert L Bocchino, Jr., Vikram S. Adve, Sarita V. Adve, and Marc Snir. 2009. Parallel programming must be deterministic by default. In *First USENIX Conference on Hot Topics in Parallelism*.

F. Warren Burton and M. Ronan Sleep. 1981. Executing Functional Programs on a Virtual Tree of Processors. In *FPCA*. ACM, 187–194.

Charles Consel, Hedi Hamdi, Laurent Réveillère, Lenin Singaravelu, Haiyan Yu, and Calton Pu. 2003. Spidle: a DSL approach to specifying streaming applications. In *GPCE*. Springer-Verlag, 1–17.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms* (third ed.). The MIT Press.

Raphael Finkel and Udi Manber. 1987. DIB — A Distributed Implementation of Backtracking. *ACM TOPLAS* 9, 2 (1987), 235–256.

Daniel P. Friedman and David S. Wise. 1978. Aspects of Applicative Programming for Parallel Processing. *IEEE Trans. Comput.* C-27, 4 (1978), 289–296.

Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The Implementation of the Cilk-5 Multithreaded Language. In *PLDI*. ACM, 212–223.

John Giacomoni, Tipp Moseley, and Manish Vachharajani. 2008. FastForward for Efficient Pipeline Parallelism: A Cache-Optimized Concurrent Lock-Free Queue. In *PPoPP*. ACM, 43–52.

Michael I. Gordon, William Thies, and Saman Amarasinghe. 2006. Exploiting Coarse-Grained

Task, Data, and Pipeline Parallelism in Stream Programs. In *ASPLOS*. ACM, 151–162.
Erwin A. Hauck and Benjamin A. Dent. 1968. Burroughs' B6500/B7500 stack mechanism. *Proceedings of the AFIPS Spring Joint Computer Conference* (1968), 245–251.
Yuxiong He, Charles E. Leiserson, and William M. Leiserson. 2010. The Cilkview Scalability Analyzer. In *SPAA*. 145–156.
Intel Corporation 2013. *Intel® Cilk™ Plus Language Extension Specification, Version 1.1*. Intel Corporation. Document 324396-002US. Available from http://cilkplus.org/sites/default/files/open_specifications/Intel_Cilk_plus_lang_spec_2.htm.
David A. Kranz, Robert H. Halstead, Jr., and Eric Mohr. 1989. Mul-T: A High-Performance Parallel Lisp. In *PLDI*. ACM, 81–90.
Monica Lam. 1988. Software pipelining: an effective scheduling technique for VLIW machines. In *PLDI*. ACM, 318–328.
I-Ting Angelina Lee, Silas Boyd-Wickizer, Zhiyi Huang, and Charles E. Leiserson. 2010. Using Memory Mapping to Support Cactus Stacks in Work-Stealing Runtime Systems. In *PACT*. ACM, 411–420.
Charles E. Leiserson. 2010. The Cilk++ Concurrency Platform. *J. Supercomputing* 51, 3 (2010), 244–257.
Steve MacDonald, Duane Szafron, and Jonathan Schaeffer. 2004. Rethinking the Pipeline as Object-Oriented States with Transformations. In *HIPS*. IEEE, 12 – 21.
William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. 2003. Cg: a system for programming graphics hardware in a C-like language. In *SIGGRAPH*. ACM, 896–907.
Michael McCool, Arch D. Robison, and James Reinders. 2012. *Structured Parallel Programming: Patterns for Efficient Computation*. Elsevier.
Larry McVoy and Carl Staelin. 1996. lmbench: portable tools for performance analysis. In *Proceedings of the USENIX 1996 Annual Technical Conference*. USENIX Association, San Diego, CA, 279–294.
Angeles Navarro, Rafael Asenjo, Siham Tabik, and Călin Caşcaval. 2009. Analytical Modeling of Pipeline Parallelism. In *PACT*. IEEE, 281–290.
OpenMP 3.0 2008. *OpenMP Application Program Interface, Version 3.0*. Available from http://www.openmp.org/mp-documents/spec30.pdf.
Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August. 2005. Automatic Thread Extraction with Decoupled Software Pipelining. In *MICRO*. IEEE, 105–118.
Antoniu Pop and Albert Cohen. 2011. A Stream-Computing Extension to OpenMP. In *HiPEAC*. ACM, 5–14.
Ram Rangan, Neil Vachharajani, Manish Vachharajani, and David I. August. 2004. Decoupled Software Pipelining with the Synchronization Array. In *PACT*. ACM, 177–188.
Eric C. Reed, Nicholas Chen, and Ralph E. Johnson. 2011. Expressing pipeline parallelism using TBB constructs: a case study on what works and what doesn't. In *SPLASH*. ACM, 133–138.
Raúl Rojas. 1997. Konrad Zuse's Legacy: The Architecture of the Z1 and Z3. *IEEE Annals of the History of Computing* 19, 2 (April 1997), 5–16.
Daniel Sanchez, David Lo, Richard M. Yoo, Jeremy Sugerman, and Christos Kozyrakis. 2011. Dynamic Fine-Grain Scheduling of Pipeline Parallelism. In *PACT*. IEEE, 22–32.
Bjarne Stroustrup. 2013. *The C++ Programming Language* (fourth ed.). Addison-Wesley.
M. Aater Suleman, Moinuddin K. Qureshi, Khubaib, and Yale N. Patt. 2010. Feedback-Directed Pipeline Parallelism. In *PACT*. ACM, 147–156.
William Thies, Vikram Chandrasekhar, and Saman Amarasinghe. 2007. A Practical Approach to Exploiting Coarse-Grained Pipeline Parallelism in C Programs. In *MICRO*. IEEE, 356–369.
Thomas Wiegand, Gary J. Sullivan, Gisle Bjøntegaard, and Ajay Luthra. 2003. Overview of the H.264/AVC video coding standard. *IEEE Transactions on Circuits and Systems for Video Technology* 13, 7 (2003), 560–576.

Received October 2013; revised May 2015; accepted June 2015