Efficient and Scalable Execution of Fine-Grained Dynamic Linear Pipelines

ARISTEIDIS MASTORAS and THOMAS R. GROSS, ETH Zurich

8

We present *Pipelite*, a dynamic scheduler that exploits the properties of dynamic linear pipelines to achieve high performance for fine-grained workloads. The flexibility of *Pipelite* allows the stages and their data dependences to be determined at runtime. *Pipelite* unifies communication, scheduling, and synchronization algorithms with suitable data structures. This unified design introduces the *local suspension* mechanism and a *wait-free enqueue* operation, which allow efficient dynamic scheduling. The evaluation on a 44-core machine, using programs from three widely used benchmark suites, shows that *Pipelite* implies low overhead and significantly outperforms the state of the art in terms of speedup, scalability, and memory usage.

CCS Concepts: • **Software and its engineering** → **Software performance**; **Runtime environments**;

Additional Key Words and Phrases: Dynamic linear pipeline, dynamic partitioning, dynamic communication, dynamic synchronization, multi-threading; dynamic scheduling, parallelization directives

ACM Reference format:

Aristeidis Mastoras and Thomas R. Gross. 2019. Efficient and Scalable Execution of Fine-Grained Dynamic Linear Pipelines. *ACM Trans. Archit. Code Optim.* 16, 2, Article 8 (April 2019), 26 pages.
<https://doi.org/10.1145/3307411>

1 INTRODUCTION

Parallelization of loops with cross-iteration dependences remains one of the main challenges of utilizing multi-core multi-processors. Cross-iteration dependences do not inhibit parallel execution, but they introduce scheduling constraints. Many real programs, such as compression algorithms and audio, video, and image processing applications, include loops with cross-iteration dependences. Previous research efforts [5, 14, 16, 18, 22, 24, 36, 37, 39, 43, 46] show that these programs can benefit from pipeline parallelism as they process a stream of data. Particularly, the loop of such a program can be expressed as a *linear pipeline*, i.e., a sequence of stages, and the overlapped execution of different loop iterations results in pipeline parallelism.

Most prior techniques on linear pipelining [14, 16, 36, 37, 39, 43, 46] map different stages onto different threads, and the communication and synchronization is typically achieved with concurrent queues. Unfortunately, these techniques handle only *static linear pipelines*, i.e., the stages and

Computing resources were provided by SNF grant 206021_133835, and partial support was provided by SNF grant IZKSZ2_162084.

Authors' addresses: A. Mastoras and T. R. Gross, Department of Computer Science, ETH Zurich, 8092 Zurich, Switzerland; emails: {aristeidis.mastoras, thomas.gross}@inf.ethz.ch.

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

© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.

1544-3566/2019/04-ART8

<https://doi.org/10.1145/3307411>

Fig. 1. DAG representation for three successive iterations of the loop from Listing 1. The $(i - 1)$ -th and i -th loop iterations process P-frames, and the $(i + 1)$ -th loop iteration processes an I-frame.

their data dependences must be known at compile time. However, there exist popular programs, e.g., the x264 video encoder, which cannot be handled efficiently as static linear pipelines.

Listing 1 shows pseudo-code for a simplified x264 video encoder that relies only on I-frames and P-frames, i.e., there are no B-frames. The outer loop reads frames from an input stream, segments the frame into macroblocks, encodes all macroblocks, and writes the encoded frame to the output stream. The macroblocks are encoded either as intra or as predicted macroblocks, depending on the type of the frame.

```

while (fr = readFrame(in_stream)) {
    while (mb = readMacroblock(fr))
        if (fr.type == I_FRAME)
            intraEncoding(mb);
        else
            predictedEncoding(mb);
    writeFrame(fr, out_stream);
}

```

Listing 1. Pseudo-code for a simplified x264 video encoder.

A straightforward approach expresses the loop from Listing 1 as a static linear pipeline, where the first stage includes the loop condition, the second one consists of the nested while loop, and the third writes the encoded frame to the output stream. The execution of a linear pipeline can be represented with a directed acyclic graph (DAG). Figure 1(a) illustrates three successive loop iterations, when the loop is expressed as a static linear pipeline. The arrows of the DAG indicate data dependences, i.e., the source of the arrow generates data for the target. Cross-iteration dependences are shown using dashed arrows to differentiate them from data dependences within a loop iteration. This static linear pipeline does not create sufficient parallelization opportunities, since the second stage that encodes the macroblocks is the most time-consuming, and encoding cannot be done simultaneously for macroblocks of different frames due to cross-iteration dependences. Nevertheless, encoding of a single macroblock does not depend on all macroblocks of the previous frame. Therefore, the second stage can be partitioned into a number of stages that allow encoding of different frames to overlap in time. However, these stages cannot be determined at compile time,

as the number of stages depends on the number of macroblocks, which is determined by the size of the frame.

To utilize the inherent parallelism of x264 and achieve high performance, x264 must be expressed as a *dynamic linear pipeline*, i.e., the stages and their data dependences must be determined at run-time. There exist two key observations. First, the inner while loop can be partitioned dynamically into stages such that each stage consists of encoding some macroblocks. Second, encoding of an intra macroblock does not depend on macroblocks of earlier frames. Thus, macroblocks of I-frames can be encoded in parallel with macroblocks of earlier frames, achieving data parallelism.

Figure 1(b) shows the parallelization opportunities that are created by expressing the loop from Listing 1 as a dynamic linear pipeline. First, encoding of macroblocks can overlap in time for different frames. Second, provided that the $(i + 1)$ -th loop iteration processes an I-frame, all macroblocks of this frame can be encoded immediately. Therefore, the data dependences of stages can be determined at runtime, allowing I-frames and P-frames to be handled differently.

Static scheduling of dynamic linear pipelines is straightforward; the challenge and the focus of this work is to achieve load-balancing. Particularly, dedup, ferret, and x264 are the pipelined programs of the PARSEC [3, 4] benchmark suite, and they all include a *load-imbalanced loop*, i.e., loop iterations differ substantially in execution time. Therefore, to efficiently handle all of these benchmarks, we need a scheduler that supports dynamic linear pipelines and achieves load-balancing. In addition, two of these programs, dedup and x264, consist of fine-grained loop iterations, which is the reason that previous research efforts fail to achieve good performance for both programs. Thus, the design of a scheduler that efficiently handles fine-grained workloads and scales with a large number of threads is not straightforward due to the overhead of handling cross-iteration dependences.

In this article, we present *Pipelite*, a dynamic scheduler that is designed specifically for dynamic linear pipelines. To efficiently handle the execution of fine-grained workloads, the design of *Pipelite* unifies suitable data structures with communication, scheduling, and synchronization algorithms. Furthermore, this unified design enables two mechanisms: *local suspension* and a *wait-free enqueue* operation, which are tailored to dynamic linear pipelines to make *Pipelite* efficient and scalable for a large number of threads. To show that *Pipelite* is a promising solution that requires little effort on behalf of the user, we developed a directive-based transformation on a prototype source-to-source compiler for C programs. The evaluation on 44 cores confirms that *Pipelite* is efficient for load-imbalanced fine-grained loops, for a number of popular and important applications selected from three benchmark suites [1–4]. The experimental results show that *Pipelite* significantly outperforms the state of the art in terms of speedup, scalability, and memory usage.

2 BACKGROUND

A dynamic linear pipeline is a loop partitioned into a sequence of stages, and we refer to executions of stages as *stage iterations*; stages form the loop body. Particularly, the k -th iteration of stage S_n is denoted as S_n^k , with $k > 0$ and $n > 0$. The linear property implies that iterations of different stages are executed sequentially within a loop iteration. Thus, arbitrary data dependences are allowed within a loop iteration. However, dynamic linear pipelines satisfy the constraint that no stage iteration S_n^j reads data that are generated by a stage iteration S_q^i with $i < j$ and $n < q$.

Stage iterations are classified into two types, depending on scheduling constraints that are introduced from cross-iteration dependences. These constraints are expressed only between successive iterations of a stage. A stage iteration S_n^j is *dependent* if its execution necessarily occurs after the completion of stage iteration S_n^{j-1} with $j > 1$. However, a stage iteration S_n^j is *independent* if it can be executed in parallel with the previous iteration of the same stage.

Cross-iteration dependences between non-successive iterations or between iterations of different stages are allowed. Yet they must not introduce additional scheduling constraints that are not expressed with the scheduling constraints between successive iterations.

2.1 Properties of Dynamic Linear Pipelines

Dynamic linear pipelines have some properties that must be exploited by the scheduler to efficiently handle fine-grained workloads. Particularly, there exist three properties:

- I. Stage iterations are executed sequentially within a loop iteration. Thus, dynamic linear pipelines avoid any overhead introduced by mechanisms that handle task parallelism in a loop iteration.
- II. Scheduling constraints are expressed only between successive iterations of a stage. Thus, synchronization is simple; it is required only between successive iterations of a stage.
- III. Only dependent stage iterations require data generated by previous iterations. Thus, suspension of a loop iteration may occur only before the execution of a dependent stage iteration.

2.2 Problem Statement

Previous research efforts present schedulers that can handle linear pipelines. Therefore, we explain the reasons that these schedulers are unsuitable for efficient and scalable execution of fine-grained dynamic linear pipelines, which is the contribution of this work.

2.2.1 Scheduling of Linear Pipelines. Kamruzzaman et al. [18] propose Load-Balanced Pipeline Parallelism (LBPP), a simple and efficient technique for fine-grained linear pipelines. However, LBPP is restricted to static linear pipelines and relies on round-robin scheduling. Therefore, LBPP cannot efficiently handle loops that must be expressed as dynamic linear pipelines, e.g., the x264 video encoder. In addition, LBPP assumes load-balanced loops, and round-robin scheduling does not achieve load-balancing for load-imbalanced loops.

Lee et al. [22] propose Piper, an algorithm that integrates pipeline parallelism into the work-stealing scheduler of Cilk [7, 13]. However, work-stealing is a generic solution to achieve load-balancing, and it does not exploit the properties of dynamic linear pipelines. Piper is a flexible algorithm that allows arbitrarily nested pipeline and fork-join parallelism, by expressing pipeline parallelism with the fork-join model. Unfortunately, this flexibility relies on design choices that are unsuitable for efficient handling of dynamic linear pipelines. In other words, Piper focuses on applicability and sacrifices performance, even though the authors do not present any application that requires this flexibility of Piper. Moreover, although Piper is a provably asymptotically efficient algorithm, in practice, high scheduling overhead dominates for fine-grained workloads [24, 25].

2.2.2 Work-Stealing Scheduling. Since Piper [22] and Intel Threading Building Blocks (TBB) [17] are two examples of systems that achieve load-balancing by relying on work-stealing, we explain here the drawbacks of work-stealing schedulers. These drawbacks apply to every system that executes a loop iteration either to completion or until it encounters a dependent stage iteration that needs data that are not available. If the data for a dependent stage iteration are not available, the execution of the loop iteration is suspended, and the thread chooses work from its own deque. If the deque is empty, the thread becomes a thief and steals work from another deque.

For example, Figure 2 illustrates a possible parallel execution for the dynamic linear pipeline from Figure 1(b). In this scenario, there exist two threads, T_A and T_B , and each thread starts the execution of a loop iteration. We assume that at t_1 point of time, Thread T_B cannot execute the stage iteration S_n^i due to data required from the stage iteration S_n^{i-1} , which are not available. Thus,

Fig. 2. Possible parallel execution of the dynamic linear pipeline from Figure 1(b).

the i -th loop iteration is suspended, and at t_2 point of time, Thread T_B continues with the execution of the $(i + 1)$ -th loop iteration. Furthermore, at t_3 point of time, Thread T_A completes the execution of the $(i - 1)$ -th loop iteration, and at t_4 point of time, Thread T_A continues with the rest of the i -th loop iteration that has been previously suspended by Thread T_B .

According to property (I), the completion of the stage iteration S_{n-1}^{i-1} enables the execution of the stage iteration S_{n+1}^{i-1} if all required data are available, and thus Thread T_A continues with the execution of the stage iteration S_{n+1}^{i-1} . Provided that the stage iteration S_{n-1}^{i-1} is completed and S_n^i is a dependent stage iteration, the execution of the i -th loop iteration may have been suspended due to the stage iteration S_n^i that requires data generated by the stage iteration S_{n-1}^{i-1} as shown in Figure 2. In this case, the completion of the stage iteration S_{n-1}^{i-1} enables the execution of the stage iteration S_n^i , and the stage iteration S_n^i is stored in the deque of Thread T_A .

Therefore, the deque of a thread contains either zero or one stage iteration, depending on whether the execution of a stage iteration has been enabled or not. The deque may also be empty when a stage iteration has been enabled but has already been stolen by another thread. A deque with more than two enabled stage iterations is impossible, as scheduling constraints are expressed only between successive iterations of a stage according to property (II).

Particularly, the completion of a stage iteration of the $(i - 1)$ -th loop iteration may enable only the execution of a stage iteration of the i -th loop iteration. Furthermore, a thread that executes the stage iteration S_{n-1}^{i-1} stores the enabled stage iteration S_n^i in its own deque only when the execution of the i -th loop iteration has been suspended. Otherwise, either the stage iteration S_n^i cannot be enabled since the stage iteration S_{n-1}^{i-1} is not completed or S_{n-1}^{i-1} is completed but the execution of the i -th loop iteration has not been suspended. In other words, a thread enables the execution of a stage iteration of the next loop iteration and stores it in its own deque only when the next loop iteration is suspended. Therefore, for two threads that execute stage iterations of two successive loop iterations, it is impossible that they both store an enabled stage iteration in their own deque.

In an ideal scenario of a work-stealing scheduler, at a given point of time, all threads execute stage iterations of non-successive loop iterations, and thus all threads enable stage iterations of different loop iterations. In other words, all threads store the enabled stage iteration in their own deque, and no stealing occurs after the completion of the currently executed stage iteration. However, in the worst case, no suspension is performed, and at a given point of time, all threads execute stage iterations of successive loop iterations. In this case, a single stage iteration is stored in some deque and belongs to a loop iteration that is currently executed by no thread. Therefore, after completing or suspending the execution of a loop iteration, every thread tries to steal a stage iteration, excluding the one thread that stores the unique enabled stage iteration in its own deque. The probability of immediately finding a stage iteration depends on the number of non-empty deques, which depends on the total number of deques, i.e., the number of threads. Consequently,

the overhead implied by a work-stealing scheduler increases with the number of threads, and thus work-stealing schedulers may show poor scalability for dynamic linear pipelines.

Moreover, a work-stealing scheduler is a generic solution to achieve load-balancing; it is not aware of the dynamic linear pipeline structure and implies overhead for two main reasons. First, each thread maintains its own deque, and work is necessarily selected from this deque, ignoring the work stored in other deques. This situation inevitably leads to suspension of loop iterations, as stage iterations of earlier loop iterations must be executed first. Second, a thread with an empty deque tries to steal work. The choice of the victim depends on the work-stealing implementation but is unrelated to the properties of dynamic linear pipelines. For example, Piper [22] randomly chooses the victim and ignores the properties of dynamic linear pipelines.

2.2.3 DAG Scheduling. Since the execution of a dynamic linear pipeline can be represented with a DAG, DAG schedulers [15] can handle dynamic linear pipelines. However, DAG schedulers do not exploit the properties of the data dependences of dynamic linear pipelines. Particularly, DAGs allow arbitrary data dependences, and DAG schedulers are more generic than schedulers for dynamic linear pipelines. Thus, DAG schedulers do not avoid unnecessary communication and synchronization overhead, and they do not guarantee optimal scheduling decisions either. DAG schedulers may perform well for coarse-grained dynamic linear pipelines, but they imply unnecessary overhead that may lead to poor performance for fine-grained dynamic linear pipelines.

For example, Piper is an algorithm that does not handle DAGs with arbitrary data dependences, but it allows arbitrarily nested pipeline and fork-join parallelism. Although Piper does not support arbitrary DAGs, programs supported by Piper may lead to DAGs that are more generic than that of dynamic linear pipelines. According to the evaluation for fine-grained workloads presented in Section 4, this additional flexibility may be one of the reasons that Piper performs significantly worse than schedulers that are tailored to dynamic linear pipelines.

2.2.4 Stream Scheduling. Dynamic schedulers for stream programming [28, 39] are generally not optimal for linear pipelines either. Programs that are handled by stream schedulers do not satisfy properties (I) and (II). Thus, stream schedulers introduce mechanisms and data structures to provide additional scheduling flexibility, although at the expense of higher overhead for linear pipelines. The focus of this work is how to avoid unnecessary overhead and efficiently execute programs that satisfy the properties of linear pipelines, which are a large class of popular applications [1–4]. In addition, stream schedulers do not handle dynamic linear pipelines, i.e., the stages and their data dependences cannot be determined at runtime.

3 PIPELITE

Pipelite is a dynamic scheduler that achieves efficient and scalable execution of dynamic linear pipelines by exploiting the properties of their data dependences. To achieve high performance for fine-grained dynamic linear pipelines, *Pipelite* relies on a unified design of data structures, algorithms, and mechanisms that are tailored to dynamic linear pipelines. Furthermore, the user can map a while loop written in C onto a dynamic linear pipeline by using a directive-based transformation, which is implemented in the prototype source-to-source compiler of Proteas [24].

3.1 Transformation

Listing 2 shows the directives that are required by the *Pipelite* transformation for the parallelization of the loop from Listing 1. The user annotates the loop with the directive `#pragma proteas pipelite`, and the compiler automatically generates C code, which uses POSIX threads and a library for the runtime system of *Pipelite*.


```

1  #pragma proteas pipelite num_threads(N) throttling(K)
2  #pragma proteas pipelite private(fr, mb, sid)
3  #pragma proteas pipelite shared(in_stream, out_stream)
4  while (fr = readFrame(in_stream)) {
5      sid = init_sid(fr);
6      while (mb = readMacroblock(fr)) {
7          if (fr.type == I_FRAME) {
8              if (new_stage_iteration(fr, mb)) {
9                  sid = sid + 1;
10                 #pragma proteas pipelite independent(sid)
11             }
12             intraEncoding(mb);
13         } else {
14             if (new_stage_iteration(fr, mb)) {
15                 sid = sid + 1;
16                 #pragma proteas pipelite dependent(sid)
17             }
18             predictedEncoding(mb);
19         }
20     }
21
22     sid = MAX_ULONG;
23     #pragma proteas pipelite dependent(sid)
24     writeFrame(fr, out_stream);
25 }

```

Listing 2. The loop from Listing 1 annotated with directives.

To describe the data dependences of the loop, the user indicates whether the data are private or shared, similar to the OpenMP [32] model. The *private* data are first written and then read within every loop iteration. In contrast, the *shared* data are first read and either are read-only or exhibit cross-iteration dependences. In addition, the user may specify the number of threads and the throttling limit with the directives `num_threads(N)` and `throttling(K)`, respectively. These parameters are further explained in the rest of this section.

To partition a loop iteration into a sequence of stage iterations, the user annotates the loop body with directives. To create a dependent stage iteration, the user inserts the directive `#pragma proteas pipelite dependent (sid)` before the first statement of the corresponding stage iteration. Similarly, `#pragma proteas pipelite independent (sid)` is used for the creation of an independent stage iteration. To create stage iterations at runtime, these directives are inserted within a conditional statement or an iterative structure as shown in lines 10 and 16. A stage iteration S_n^k is identified with the integer n , and we refer to it as *stage identifier* (*sid*).

The directives required by *Pipelite* are similar to those used by the OpenMP model. Thus, it is the user's responsibility to insert correct directives to describe the data dependences of the loop. Otherwise, the compiler will generate incorrect code. In addition, the user must partition the loop iterations into suitable stage iterations and must configure the runtime system properly with the parameters for the number of threads and the throttling limit. A sub-optimal partitioning or configuration of the runtime system may lead to poor performance. Moreover, the annotations required by the *Pipelite* transformation are similar to those required by Piper [22]. Therefore, the main contribution of this work is the design of an efficient dynamic scheduler for dynamic linear pipelines, and the implementation of the directive-based transformation is used as a means to confirm that *Pipelite* is a promising solution that requires little effort on behalf of the user.

3.2 Dynamic Linear Pipeline Structure

The first stage iteration includes the loop condition, and it is implicitly declared for every loop iteration, i.e., no directive is required. The first stage iteration may include other statements,

depending on the beginning of the second stage iteration. The type of the first stage iteration is dependent for every loop iteration, as the loop condition is evaluated sequentially for while loops. Subsequent stage iterations may be either dependent or independent. In other words, different stage iterations of the same loop iteration may have different types, and different iterations of a stage may also have different types. Thus, there is no restriction about the type of stage iterations, excluding iterations of the first stage, which are always dependent. Moreover, two successive independent stage iterations can always be merged into a single independent stage iteration.

The number of stage iterations may differ for different loop iterations, and some iterations may not exist for some stages. For example, the stage iteration S_n^i may exist, but the stage iteration S_n^{i-1} may not exist. This applies to both dependent and independent stage iterations. In other words, the existence of a dependent stage iteration S_n^i does not imply that there exists a stage iteration S_n^{i-1} in the previous loop iteration. Thus, *Pipelite* allows a flexible structure with missing stage iterations.

The sid of all iterations of the first stage is 1, and that of later stage iterations may be determined in the directives by the user. Therefore, the sid argument is optional, and it can be determined automatically for static linear pipelines. The default value is equal to the sid of the same iteration of the previous stage incremented by one. The sid must strictly increase for iterations of different stages during the execution of a loop iteration, and it applies to a single stage iteration. This implies that the sid of a stage iteration may be different from the sid of another stage iteration that consists of the same sequence of statements for another loop iteration. Hence, *Pipelite* allows the expression of scheduling constraints between statements that seemingly belong to different stages.

In particular, a stage iteration S_n^j may depend on data generated by a stage iteration S_n^i with $i < j$, but the sequence of statements that corresponds to the stage iteration S_n^i follows the sequence of statements of the stage iteration S_n^j . Such a data dependence does not violate the definition of dynamic linear pipelines that no stage iteration S_n^j depends on a stage iteration S_q^i with $i < j$ and $n < q$. These stage iterations have the same sid, irrespective of whether they consist of the same sequence of statements or not. This flexible structure allows *Pipelite* to efficiently handle x264, and stage iterations are created at runtime with the auxiliary functions `new_stage_iteration` and `init_sid` as shown in Listing 2. These functions are required only for this example, and they are not discussed further. For example, `dedup` and `ferret` do not require any auxiliary functions.

3.3 Runtime System

The design of *Pipelite* unifies data structures, algorithms, and mechanisms that allow efficient execution of dynamic linear pipelines. Figure 3 illustrates the design of the runtime system. *Pipelite* creates a thread pool of size N , and this size may be determined with the directive `num_threads(N)`. The scheduling entity of *Pipelite* is a *tasklet*, a lightweight task that represents a loop iteration. There exist three types of data for a tasklet: *private*; *shared*; and four scalar variables, i.e., `ticket`, `state`, `stage`, and `stage_sign`. Every tasklet maintains its own replica of the private data and a reference to the shared data, which are available to all tasklets throughout the execution of a loop. A tasklet is implicitly created with the execution of a new loop iteration. The overhead for the creation of a tasklet lies in the initialization of the four scalar variables.

Pipelite relies on two data structures, a buffer and a ready queue, which are shared by all threads. Shared data structures are a key feature, as they allow the scheduler to consider the properties of dynamic linear pipelines and to try to execute stage iterations of early loop iterations first. As shared data structures typically cause contention, the design of *Pipelite* efficiently addresses this challenge. First, *Pipelite* introduces a ticket mechanism that coordinates access to the buffer. Second, the enqueue operation respects the wait-free property because its design is tailored to the scheduling algorithm of *Pipelite*. Therefore, both data structures avoid contention and provide fast and concurrent access to threads.

Fig. 3. The *Pipelite* runtime system.

The design of *Pipelite* exploits the properties of dynamic linear pipelines. In particular, since stage iterations are executed sequentially within a loop iteration according to property (I), *Pipelite* uses the buffer to achieve implicit communication through shared memory. According to property (II), scheduling constraints are expressed only between successive iterations of a stage. Therefore, threads know which stage iteration generates data that are necessary for another stage iteration, and threads must be able to detect immediately the data of a given stage iteration. The design of the buffer exploits property (II) and provides index-based access to the data of all stage iterations. Hence, inter-thread communication is achieved through shared memory. In addition, *Pipelite* introduces the *local suspension* mechanism, which considers property (III) to enable efficient suspension of loop iterations. This unified approach implies low overhead and makes *Pipelite* efficient and scalable for dynamic scheduling of fine-grained dynamic linear pipelines.

3.3.1 Communication. The communication of threads is implicitly achieved with a *buffer*, a concurrent data structure that is shared by all threads. Each element contains all the necessary data for the execution of a tasklet, and we refer to a buffer element as a *memory location*.

Since every tasklet maintains its own replica for private data, a large amount of memory may be required for the parallel execution when a large number of tasklets is active simultaneously. To avoid unlimited memory, *Pipelite* allows the user to control memory by setting a *throttling* limit K . Thus, *Pipelite* allows a limited number of K active tasklets at any given point of time.

The buffer has a predefined size of K memory locations, which may be determined by the user with the directive `throttling(K)`. Otherwise, the default value is $K = 4N$, where N is the size of the thread pool. The i -th tasklet stores its data in the $(i \bmod K)$ -th memory location, and the $(i + K)$ -th loop iteration cannot be executed unless the i -th is completed. Hence, the buffer provides fast index-based access, and threads access the memory location of any given tasklet in $\Theta(1)$ time.

The memory of the buffer is allocated once in the beginning, and it is released in the end. This setup implies that *Pipelite* avoids the overhead of dynamic memory allocation. Moreover, the buffer avoids any overhead from operations that coordinate different data structures, e.g., enqueue operations provided by a queue, as the data of the buffer do not move; threads have only read and write access to the data. This property makes the buffer similar to the data structure presented by Miranda et al. [29] that achieves point-to-point communication for dataflow tasks, and to that used by Mastoras and Gross [25] for static linear pipelines.

3.3.2 Scheduling. The thread pool of *Pipelite* executes tasklets either to completion or until it encounters a dependent stage iteration that needs data that are not available. Then, a thread that completes or suspends a tasklet receives another tasklet from the scheduler. Therefore, the scheduling algorithm of *Pipelite* must decide whether the next scheduled tasklet is one that has been suspended previously due to data dependences but is now ready for execution, or one that corresponds to a new loop iteration. To distinguish between tasklets with different states, every tasklet maintains the state variable that has one of the following values:

- EXECUTED, to indicate that a thread is currently executing this tasklet;
- READY, for a tasklet that is ready for execution, but it is currently scheduled to no thread; and
- SUSPENDED, for a tasklet that has been suspended due to data dependences, and the data are not yet available.

Since the data dependences of dynamic linear pipelines have some properties that imply scheduling constraints, a scheduling algorithm must make decisions by exploiting all the information about the data dependences. Particularly, the dependent iterations of a stage are necessarily executed sequentially, and all iterations of the first stage are always dependent. Therefore, the scheduling algorithm of *Pipelite* must always try to schedule tasklets of early loop iterations to avoid suspension of tasklets for later loop iterations. This implies that the scheduler must first try to schedule a ready tasklet, and if none exists, it must schedule a tasklet for a new loop iteration.

Since *Pipelite* aims at achieving high performance for fine-grained workloads, the design must rely on efficient mechanisms that immediately detect tasklets for early loop iterations. *Pipelite* meets this demand by using a single *ticket counter* and a single *ready queue* that provides enqueue and dequeue access to all threads. Particularly, the ticket counter is incremented by one to issue a ticket every time that a tasklet is created. The issued ticket is stored in the *ticket* variable of the tasklet. Similarly, *Pipelite* maintains a ready queue to store ready tasklets in $\Theta(1)$ time. Therefore, the ready queue allows threads to immediately detect ready tasklets, avoiding the overhead of searching for ready tasklets in the whole buffer.

Listing 3 shows pseudo-code for the function that is executed by the thread pool for the loop from Listing 2. This function is automatically generated by the directive-based transformation of *Pipelite*. The scheduler assigns tasklets to threads while the loop condition is true or there exist ready tasklets. Threads call the function `get_tasklet` that returns a scheduled tasklet.

The function `get_tasklet` tries first to return a ready tasklet and resume its execution. If it fails, it tries to return a tasklet for a new loop iteration. However, this can be done only if the memory location of this new loop iteration is available, as the number of active tasklets is limited to K . If a tasklet for a new loop iteration cannot be scheduled, the scheduler tries again to return a ready tasklet from the ready queue. Thus, the scheduler iteratively tries to return either a ready tasklet from the ready queue or a tasklet for a new loop iteration.

Local suspension. Suspension of tasklets requires that the context is stored such that the execution can be resumed later from the same point. Thus, suspension typically implies some overhead. However, this overhead is not inherent and can be avoided with the *local suspension* mechanism.

The local suspension mechanism exploits the design of the buffer and property (III) of dynamic linear pipelines in the following way. First, all necessary data for the execution of a tasklet are stored in a memory location of the buffer. Hence, there is no need to store or load any data when a tasklet is suspended or resumed, respectively, excluding register spilling that may still happen. Second, a tasklet may be suspended only due to a dependent stage iteration. Therefore, the suspension point is not an arbitrary point of a tasklet, but it is the start point of a dependent stage iteration

```

1  void func(pipelite_rts_t r) {
2      shared_t s = r.shared_data;
3
4      do {
5          tasklet_t t = get_tasklet(r);
6          private_t p = t.private_data;
7
8          switch (t.stage_sign) {
9              case 1: goto s1;
10             case 2: goto s2;
11             case 3: goto s3;
12         }
13
14     s1:  if (!proceed(r, t, 1)) goto suspend;
15
16         if (r.condition) {
17             // the while is replaced with an if
18             if (p.fr = readFrame(s.in_stream)) {
19                 p.sid = init_sid(p.fr);
20                 while (p.mb = readMacroblock(p.fr)) {
21                     if (p.fr.type == I_FRAME) {
22                         if (new_stage_iteration(p.fr, p.mb)) {
23                             p.sid = p.sid + 1;
24
25                             update_stage(r, t, p.sid - 1);
26                         }
27                         intraEncoding(p.mb);
28                     } else {
29                         if (new_stage_iteration(p.fr, p.mb)) {
30                             p.sid = p.sid + 1;
31
32                             update_stage(r, t, p.sid - 1);
33                             t.stage_sign = 2;
34                         }
35                     }
36                     if (!proceed(r, t, p.sid)) goto suspend;
37                     predictedEncoding(p.mb);
38                 }
39                 p.sid = MAX_ULONG;
40
41                 update_stage(r, t, p.sid - 1);
42                 t.stage_sign = 3;
43             s3:  if (!proceed(r, t, p.sid)) goto suspend;
44
45                 writeFrame(p.fr, s.out_stream);
46                 complete_tasklet(r, t);
47                 goto complete;
48
49             suspend:  suspend_tasklet(r, t);
50             complete:
51                 } else {
52                     terminate_rts(r, t); complete_tasklet(r, t);
53                 }
54             } else {
55                 complete_tasklet(r, t);
56             }
57         } while (r.condition || exist_ready_tasklets(r));
58     }

```

Listing 3. Pseudo-code for the function that is executed by the thread pool for the dynamic linear pipeline from Listing 2. The function is automatically generated by the Proteas source-to-source compiler.

and can be determined from the directives used for code partitioning. Therefore, the scheduling algorithm needs to know only the dependent stage iteration that causes suspension.

Consequently, the design of local suspension relies on labels, the variable `stage_sign` of a tasklet, and a switch statement. A label is introduced before every directive that creates a dependent stage iteration. Hence, three labels, `s1`, `s2`, and `s3`, are introduced for the loop from Listing 2 as shown in Listing 3. The suspension point from which the execution must be resumed is indicated with `stage_sign`. Thus, `stage_sign` is updated before the labels as shown in lines 33 and 42 for the labels `s2` and `s3`, respectively. The update before the label `s1` is omitted, as `stage_sign` is initialized to 1 during the creation of a tasklet such that the execution starts with the iteration of the first stage. Finally, the switch statement uses the `stage_sign` variable to resume the execution from the proper point, by jumping with the `goto` statement to the proper label.

The local suspension mechanism is similar to that proposed by Karrenberg and Hack [19] to handle barrier synchronization for OpenCL. Although the main idea is the same, OpenCL semantics require that all iterations reach the same barrier, i.e., they jump to the same label.

Wait-free enqueue operation. The design of an efficient wait-free queue with multiple enqueueers and multiple dequeuers is challenging for general-purpose situations. For example, Kogan and Petrank [20] present a wait-free queue, i.e., an implementation that guarantees progress for all threads. However, the authors state that this wait-free queue is typically slower than the lock-free queue presented by Michael and Scott [27]. Moreover, Morrison and Afek [30] present one of the most efficient queue implementations, but it does not provide wait-free guarantees.

Pipelite allows the design of an efficient queue that provides a wait-free enqueue operation, as the queue is tailored to the scheduling algorithm. In particular, since the number of active loop iterations is limited to K , the number of ready tasklets that may be enqueued at any given point of time is bounded by K . Therefore, the maximum size of the ready queue is K , and the scheduling algorithm guarantees that the enqueue operation can always be completed.

The ready queue is represented with an array of references to tasklets. The enqueue operation is implemented using an index `tail`, which indicates the location of the array where the next tasklet will be inserted, as shown in Listing 4. The index is incremented by one with the fetch-and-add atomic operation (line 5), as there exist multiple enqueueers. Similarly, the dequeue operation uses an index `head`, which indicates the location of the next tasklet that will be dequeued.

A thread reads the value of `head` (line 11) and the tasklet in the specified location of the array (line 12). Then, the thread confirms that the value of `head` remains the same (line 14). This implies that the tasklet corresponds to the initial value that has been read from `head` and not to a greater value that targets the same location of the array. In addition, the thread confirms that the tasklet is not `NULL`, which implies that an enqueue operation has inserted a tasklet in this location. The condition of the while loop does not guarantee that there exists a tasklet, as the statement in line 5 may have been executed but the statement in line 6 has not been completed yet.

To ensure that a single thread dequeues the tasklet, a thread uses compare-and-swap (line 15) to obtain exclusive access to the tasklet and to set this location of the array to `NULL`. After a successful compare-and-swap, the thread ensures that the value of `head` remains the same (line 16), and `head` is incremented by one to complete the dequeue operation. Otherwise, the thread has succeeded due to the ABA problem [10]. Therefore, the dequeue operation cannot be completed, and the tasklet is written back to the array (line 20).

The ABA problem may occur when a thread has executed the statement in line 14 but the statement in line 15 has not been executed yet. In the meantime, another thread may have dequeued the tasklet, and a new tasklet may have been enqueued in the same location. If both the new and the old tasklets correspond to the same loop iteration, then the references of both tasklets are the

```

1  tasklet_t array[K] = {NULL};
2  unsigned long head = 0, tail = 0;
3
4  void enqueue(tasklet_t d) {
5      unsigned long t = fetch-and-add(tail, 1);
6      array[t % K] = d;
7  }
8
9  tasklet_t dequeue() {
10     while (tail > head) {
11         unsigned long h = head;
12         tasklet_t d = array[h % K];
13
14         if (d != NULL && h == head) {
15             if (compare-and-swap(array[h % K], d, NULL)) {
16                 if (h == head) {
17                     head++;
18                     return d;
19                 }
20                 array[h % K] = d;
21             }
22         }
23     }
24     return NULL;
25 }

```

Listing 4. Pseudo-code for the ready queue implementation.

same. However, the new tasklet corresponds to a greater value of head, which targets the same location of the array. This may happen when the same tasklet has been suspended twice due to different stage iterations. Thus, these tasklets represent different data of the ready queue.

3.3.3 Synchronization. The synchronization algorithm relies on three out of four variables that are stored for every tasklet: ticket, state, and stage. The variable ticket indicates the tasklet that stores its data in a memory location, i.e., ticket stores the value issued by the ticket counter of the scheduling algorithm. Furthermore, the variable stage shows the progress of a tasklet and provides efficient synchronization between successive iterations of a stage.

In particular, the function `update_stage` is invoked when a stage iteration S_n^i is completed as shown in lines 25, 32, and 41 in Listing 3. Hence, stage is updated such that another thread can execute a dependent stage iteration S_n^{i+1} . The update can be performed by setting stage to the stage identifier (sid) of the completed stage iteration. However, this solution cannot handle the flexible structure with missing stage iterations supported by *Pipelite*. Thus, the update is performed based on the sid of the next stage iteration instead of that of the completed stage iteration, i.e., stage is updated with the sid of the next stage iteration decremented by one.

Moreover, the function `update_stage` uses the state variable to check whether the tasklet of the stage iteration S_n^{i+1} is suspended or not. If the tasklet is suspended, the suspension has occurred due to data dependences on the stage iteration S_n^i . Therefore, the execution of the tasklet can be resumed now; the state variable of the tasklet of the $(i + 1)$ -th loop iteration is changed from SUSPENDED to READY, and the tasklet is enqueued in the ready queue.

Independent stage iterations do not require any synchronization mechanism. However, a thread ensures that the stage iteration S_n^i is completed before the execution of a dependent stage iteration S_n^{i+1} . This is achieved by invoking the function `proceed` as shown in lines 14, 34, and 43 in Listing 3. The function `proceed` allows a thread to execute a dependent stage iteration S_n^{i+1} when one of the two following conditions holds:

- stage $\geq n$ for the tasklet of the i -th loop iteration;
- the memory location of the i -th loop iteration, $(i \bmod K)$, stores data of the $(i + K)$ -th loop iteration, i.e., the i -th loop iteration is completed and ticket stores the ticket for the $(i + K)$ -th loop iteration.

Otherwise, the tasklet is suspended by transferring the control to the label `suspend`, i.e., the function `suspend_tasklet` is invoked, which sets state to the value `SUSPENDED`.

3.4 Performance Tradeoffs

Pipelite allows the user to control memory usage by setting a throttling limit K . This restriction enables the design of two suitable data structures that allow efficient dynamic scheduling, and it avoids the overhead of dynamic memory allocation. Nevertheless, this restriction may limit the overall performance [22, 24, 25], as the $(i + K)$ -th loop iteration cannot be executed unless the i -th is completed. Therefore, the user should set the size of the buffer to a sufficiently large value. However, this value depends on the dynamic linear pipeline, the dataset, and the system configuration. Hence, choosing a value that leads to high performance is far from obvious.

To allow all threads to work on different tasklets, the buffer size must not be less than the number of threads. In addition, since some threads may not be able to make progress due to data dependences, the size of the buffer must be greater than the number of threads. In other words, threads should be able to switch to the execution of a different tasklet instead of waiting until these data become available. Therefore, a sufficiently large value for the throttling limit allows all threads to work on different tasklets and make progress at different rates.

By setting the throttling limit to an unnecessarily large value, *Pipelite* provides additional flexibility to the scheduling algorithm. Therefore, this flexibility may lead to good performance. However, there exist two main drawbacks of this approach. First, a large value for the throttling limit may lead to worse data locality, which may have significant impact on the overall performance. Second, a large value implies unnecessary allocation of memory for the buffer.

4 EVALUATION

We conducted an in-depth evaluation based on applications that were selected from three widely used benchmark suites [1–4]. We compare *Pipelite* with static scheduling and the Piper [22] algorithm of Cilk. A comparison to Intel TBB [17] is not included, as Intel TBB does not readily support dynamic linear pipelines [38]. In addition, both Intel TBB and Piper rely on work-stealing to achieve load-balancing, and Lee et al. [22] show that Intel TBB performs similarly to Piper. To provide a clear picture about the comparison of *Pipelite* with Piper, we used all the real applications used by Lee et al. [22] and experimented with the same datasets. We include three additional applications to investigate the overhead, scalability, and memory usage.

4.1 Implementation

The code of *Pipelite* was generated using the directive-based transformation of Proteas [24]. Since the compiler for LBPP [18] is not publicly available, we also used the directive-based transformation of Proteas [24] for LBPP. LBPP does not support dynamic linear pipelines, and thus we implemented a parallel version of x264, which relies on round-robin scheduling, to show the effect of load-imbalance. Both the obtained speedups from the LBPP transformation and the round-robin scheduling for x264 are indicated as *Static*. Chunking is a mechanism that reduces the overhead of LBPP [18, 24], but the evaluation does not consider chunking because neither *Pipelite* nor Piper currently support chunking. Finally, Piper [22] does not provide compiler support, and the code was parallelized using macros for the Intel Cilk Plus [41, 42] implementation.

Table 1. Benchmark Specifications

Benchmark	Benchmark Suite	Description	Dataset	Pipeline Pattern
bzip2	Parallel BZIP2	Compression	53.8MB	$D \rightarrow I \rightarrow D$
hmmcalibrate	SPEC CINT2006	Determines statistical significance parameters	leng100.hmm	$D \rightarrow I \rightarrow D$
hmmsearch	SPEC CINT2006	Searches a biological sequence database	nph3.hmm, swiss41	$D \rightarrow I \rightarrow D$
ferret	PARSEC	Content-based similarity search	native	$D \rightarrow I \rightarrow D$
dedup	PARSEC	Compression	native	$D \rightarrow D \rightarrow I \rightarrow D$
x264	PARSEC	Video encoding	native	$D \rightarrow I \rightarrow (D I)^+ \rightarrow D$

Table 2. Loop Details

Benchmark	Sequential Execution Time	Loop Iterations (#)	Average Execution Time of Loop Iterations
bzip2	6 sec	60	96,000 μsec
hmmcalibrate	37 sec	85,000	435 μsec
hmmsearch	115 sec	122,564	949 μsec
ferret	386 sec	3,500	110,700 μsec
dedup	22 sec	369,312	60 μsec
x264	516 sec	512	1,007,000 μsec

4.2 User Intervention

The evaluation considers the programs presented in Table 1. For all techniques, the user inserts one directive that corresponds to the whole loop, and a number of directives for the creation of the stage iterations, as explained in Section 3.1. The number of required directives depends on the pipeline pattern, which is presented in Table 1. Although Listing 2 shows that there exist three directives before the loop, all of them can be merged into a single directive.

For example, the pipeline pattern of dedup is $D \rightarrow D \rightarrow I \rightarrow D$, which implies that the first two stage iterations are dependent (D), the third stage iteration is independent (I), and the last stage iteration is dependent. Therefore, the user inserts four directives in total. One directive is inserted before the loop, and three directives are required for the creation of the stage iterations. No directive is required for the first stage iteration. For x264, the user inserts five directives: one before the loop, one directive for the second stage iteration that is always independent, one directive for the last stage iteration that is always dependent, and two directives for the rest of the stage iterations, which are created at runtime and are either dependent or independent.

4.3 Experimental Methodology

We conducted the experiments on an Intel Xeon E5-2699 v4 with two processors and a total of 44 cores. The clock frequency is 2.20GHz, and the turbo frequency reaches 3.60GHz. To show the scalability of *Pipelite* over the sequential execution and to fully utilize the underlying hardware, we disabled the hyper-threading technology and set the turbo frequency for all numbers of cores at 2.80GHz. The main memory is 512GB, and the last-level cache is 55MB. The source code was compiled with gcc and g++ 4.8.1 (-O3) on Ubuntu 16.04.1 LTS. The execution times of the sequential code for all benchmarks are shown in Table 2. The presented speedups are normalized to the best sequential execution times and are based on an average of 30 measurements for each experiment. The standard deviations are less than 5% of the average execution times, excluding very few experiments of dedup and the very fine-grained hmmcalibrate, which reach up to 10% and

Table 3. Speedups for $N = 44$ Threads and the Throttling Limit K That Reaches the Highest Performance

Benchmark	Static	Piper	K	<i>Pipelite</i>	K	<i>Pipelite</i> vs Static	<i>Pipelite</i> vs Piper
bzip2	8.88×	8.67×	176	11.56×	176	+30%	+33%
hmmcalibrate	18.47×	15.65×	176	18.85×	176	+2%	+20%
hmmsearch	18.27×	36.68×	880	40.24×	880	+220%	+10%
ferret	11.10×	39.61×	440	40.57×	440	+365%	+2%
dedup	5.42×	4.41×	440	5.89×	88,000	+9%	+34%
x264	15.80×	17.78×	176	21.43×	176	+36%	+21%

12%, respectively. The memory allocation becomes the bottleneck for dedup, and thus we used *libjemalloc.so.1* instead of the default memory allocator. In addition, to avoid any NUMA effects when we used a small number of threads that can run on the same node, we set thread-to-core affinities for *Pipelite* and the static schedulers. Similarly, we used taskset for Piper when it leads to higher performance.

4.4 Case Studies

Table 3 presents the speedups obtained from different benchmarks for $N = 44$ threads and a value for the throttling limit that reaches the highest speedup. In other words, we experimented with different values for K and report the best. Lee et al. [22] state that $K = 4N$ is sufficiently large and usually performs quite well, which is confirmed from the evaluation. For those benchmarks that require a larger value, we further experimented and discuss this situation.

The experimental results presented in Table 3 show that *Pipelite* clearly outperforms both the static scheduler and the work-stealing scheduler of Piper [22] for all benchmarks. Although all benchmarks include load-imbalanced loops, we notice that *Pipelite* outperforms the static scheduler by up to 36% for those benchmarks that do not include sufficient parallelism, i.e., bzip2, hmmcalibrate, dedup, and x264. Particularly, *Pipelite* does not achieve linear speedups, and the static scheduler performs well for 44 threads. However, *Pipelite* significantly outperforms the static scheduler for hmmsearch and ferret, as both benchmarks include sufficient parallelism that allows *Pipelite* to achieve linear speedups and expand the gap between dynamic and static scheduling as a result of better load-balancing.

The comparison of *Pipelite* with Piper [22] shows more interesting results. We notice that the difference between the two schedulers heavily depends on the granularity of the loop, which is presented in Table 2. In particular, *Pipelite* slightly outperforms Piper for ferret, as this benchmark consists of a coarse-grained loop. However, the finer-grained is the loop, the larger is the performance gap between *Pipelite* and Piper, according to the speedups for hmmsearch, hmmcalibrate, and dedup. Nevertheless, the granularity is not the only interesting aspect, as we can see from the poor performance of Piper for bzip2 and x264. Both benchmarks consist of coarse-grained loops, but Piper does not achieve high performance, as is explained in the rest of this section.

Figure 4 shows the scalability of the different schedulers for bzip2. The static scheduler scales with the number of threads, but its performance is limited due to load-imbalance. Piper [22] performs well for a small number of threads and reaches the highest speedup for 16 threads, but it does not scale for a larger number of threads and its performance is significantly worse for 44 threads. In fact, the simple static scheduler outperforms Piper for more than 28 threads. Piper performs poorly because this loop consists of 60 iterations, and a large number of threads implies high overhead. However, *Pipelite* reaches the highest speedup for 24 threads, and the trend of the line

Fig. 4. Speedups of bzip2 using $K = 176$ and varying the number of threads.

Fig. 5. Speedups of x264 using $K = 176$ and varying the number of threads.

shows that *Pipelite* maintains this speedup for a larger number of threads. Hence, in contrast to Piper, which requires proper configuration, *Pipelite* is robust with regard to the number of threads.

Figure 5 presents the speedups for the x264 benchmark, varying the number of threads. *Pipelite* scales well with the number of threads, achieving the maximum speedup for 44 threads. However, Piper [22] reaches the highest performance for 36 threads and then the performance decreases. Although the loop of x264 is coarse-grained, almost all loop iterations are partitioned into 71 stage iterations. This situation leads to some fine-grained stage iterations that cannot be handled efficiently by Piper. Static scheduling scales with the number of threads, and the trend shows that the simple static scheduler may outperform Piper for more than 44 threads.

Figure 6 shows the scalability of the schedulers for dedup. The static scheduler scales with the number of threads, but it does not achieve high performance due to load-imbalance. Piper performs better than the static scheduler for a small number of threads, but the high overhead implies poor performance for a larger number of threads. In particular, the simple static scheduler outperforms Piper for more than 24 threads, and the trends of the lines show that this gap expands for a larger number of threads. *Pipelite* clearly outperforms both the static scheduler and Piper for all numbers of threads, and it reaches the highest speedup for 16 threads. The performance decreases for a larger number of threads, as the average execution time of a loop iteration is only 60 μsec .

Since the speedups in Table 3 are obtained for different values of K , we provide an in-depth evaluation for those benchmarks that use a $K > 4N$, varying the throttling limit of Piper [22] and *Pipelite*. The static scheduler inherently limits the number of active loop iterations to the number of threads [18, 24]; thus, no results are presented. Table 4 shows that *Pipelite* always reaches better

Fig. 6. Speedups of dedup using $K = 440$ for Piper and $K = 88,000$ for *Pipelite*, and varying the number of threads.

Table 4. Speedups for $N = 44$ Threads and Varying the Throttling Limit K

	hmmsearch		ferret		dedup	
K	Piper	<i>Pipelite</i>	Piper	<i>Pipelite</i>	Piper	<i>Pipelite</i>
88	15.20×	25.99×	14.52×	18.09×	4.19×	5.41×
132	19.34×	30.46×	19.91×	24.31×	4.32×	5.41×
176	23.45×	33.22×	25.34×	30.17×	4.32×	5.36×
220	27.30×	35.06×	30.44×	33.83×	4.33×	5.48×
264	29.90×	36.29×	35.06×	37.37×	4.36×	5.51×
308	32.30×	37.32×	37.75×	39.58×	4.40×	5.46×
352	33.63×	38.03×	39.19×	40.50×	4.28×	5.53×
396	34.61×	38.55×	39.37×	40.59×	4.28×	5.52×
440	35.10×	38.94×	39.61×	40.57×	4.41×	5.56×

performance than Piper for these three benchmarks when we use the same value for K . This implies that *Pipelite* requires less memory than Piper to achieve the same performance. For example, *Pipelite* outperforms Piper by 71% for hmmsearch with $K = 88$, and this gap narrows while the value of K is increased. The difference for the other benchmarks is smaller yet substantial, i.e., *Pipelite* achieves up to 25% and 29% higher speedups than Piper for ferret and dedup, respectively. Note that although Table 3 shows that *Pipelite* requires $K = 88,000$ to achieve the highest speedup for dedup, *Pipelite* performs better than Piper even for smaller values of the throttling limit as shown in Table 4. Therefore, *Pipelite* clearly outperforms Piper by 24% to 29% for all presented values of K . Moreover, both *Pipelite* and Piper reach the highest performance for hmmsearch for $K = 880$ as shown in Table 3.

4.5 Overhead Measurement

To provide a clear picture about the performance of *Pipelite*, we need a very fine-grained loop such that we can measure the overhead of *Pipelite* and compare it to that of the simple static scheduler. Moreover, this loop must be load-balanced such that the static scheduler performs as well as the dynamic scheduler. We use *hmmcalibrate*, which allows the user to choose the length of the sequence that is randomly produced in every loop iteration. Therefore, the loop can become

Fig. 7. Speedups of `hmmcalibrate` for sequences' length 10 using $K = 176$ and varying the number of threads.

Table 5. Statistics for the Work-Stealing Scheduler of Piper, Using the Same Configuration That Was Used for the Experiments of Table 3, i.e., $N = 44$ Threads and the Throttling Limit That Reaches the Highest Performance

Benchmark	Maximum Deques Size	Maximum Number of Non-Empty Deques	Average Number of Successful Steals (%)		Average Number of Wrong Victim Choices (%)	
bzip2	1	1	62	103%	952	1,542%
hmmcalibrate	1	2	109,768	129%	2,816,202	2,566%
hmmsearch	1	2	142,162	116%	744,196	523%
ferret	1	1	3,505	100%	1,114	32%
dedup	1	2	425,809	115%	4,417,319	1,037%
x264	1	1	4,447	868%	1,468	33%

finer-grained or coarser-grained, depending on this length. We use sequence length 10 and present the obtained speedups in Figure 7. The average execution time of a loop iteration is only $9 \mu\text{sec}$, making the scheduling overhead the main bottleneck.

The static scheduler scales up to 12 threads, and then we notice performance degradation due to high overhead. Note that the significant difference between the speedups of 20 and 24 threads is a result of NUMA effects. *Pipelite* implies low overhead and performs similarly to the static scheduler for up to 8 threads. However, as expected, this overhead is higher than that of a simple static scheduler, and *Pipelite* performs worse than the static scheduler for a larger number of threads. Finally, the high overhead of Piper [22] implies poor performance for all numbers of threads.

4.6 Work-Stealing

Section 2 discusses the drawbacks of work-stealing for the execution of dynamic linear pipelines, and the evaluation confirms the poor performance of Piper. However, the reported results do not necessarily imply that work-stealing is the reason for the poor performance of Piper. Therefore, we conducted additional experiments to investigate the performance of work-stealing for dynamic linear pipelines, and we present the obtained results in Table 5.

Table 5 reports the maximum size of deques and the maximum number of non-empty deques observed during the execution of each program when threads try to steal work. It also reports statistics for two metrics that evaluate the overhead of work-stealing scheduling: *Successful Steals* and *Wrong Victim Choices*. The results are collected from the function `random_steal` of the Intel Cilk Plus [41, 42] implementation, and whether a deque is empty or not is determined with the

function `can_steal_from`. We counted only the wrong choices of a victim, when the identifier of the chosen victim was less than 44, as we used 44 threads. Otherwise, the reported numbers would be significantly larger. In addition, Table 5 presents the percentage of successful steals in comparison to the number of loop iterations and the percentage of wrong victim choices in comparison to the number of successful steals. The considered metrics are the following:

- *Successful Steals*, i.e., the total number of successful steals;
- *Wrong Victim Choices*, i.e., the total number of attempts that threads try to steal work but failed to steal, as they randomly chose a victim that had an empty deque while there was another non-empty deque available.

The maximum number of steals that may occur per loop iteration is equal to the number of dependent stage iterations. For all considered benchmarks, except x264, we notice that there is slightly more than one steal per loop iteration. For x264, we notice almost nine steals per loop iteration, as a loop iteration consists of 2 to 70 dependent stage iterations. In addition, every deque contains either zero or one stage iteration at any given point of time. Although in theory there might exist multiple non-empty deques when a thread tries to steal work, in practice there exists almost always a single non-empty deque for all considered benchmarks. For dedup, hmmcalibrate, and hmmsearch, there are few exceptions with two non-empty deques. Nevertheless, the number of these exceptions is negligible in comparison to the large number of steals, i.e., about 0%.

The large number of steals shows that threads do not execute work found in their own deque, and thus threads must perform stealing operations all the time. However, there exists almost always a single non-empty deque, and the reported results show that finding this single deque may require a lot of trials since the victim is randomly chosen. For example, for hmmcalibrate, threads make around 26 wrong choices of a victim before they manage to find a non-empty deque.

The number of wrong choices depends on the number of deques, which is equal to the number of threads. Therefore, the randomized work-stealing scheduler of Piper finds the non-empty deque, if any exists, without any failed attempts when there exist only two threads, as there is only one option for the victim. However, in practice, the number of wrong victim choices is quite large when we use a large number of threads, e.g., 44 threads as in the evaluation of this work.

5 DISCUSSION

The evaluation shows that static scheduling scales with a large number of threads, but its performance is limited due to load-imbalance. Piper [22] achieves load-balancing as a result of work-stealing, but the high overhead of Piper implies poor performance for fine-grained loops. However, *Pipelite* implies low overhead and clearly outperforms Piper for all considered benchmarks, in terms of speedup, scalability, and memory usage.

There exist three main reasons that may explain the poor performance of Piper for fine-grained linear pipelines. First, the overhead may come from the Intel Cilk Plus [41, 42] implementation, as it supports a lot of features that may introduce unnecessary overhead. Second, the overhead may come from the flexibility of Piper to support arbitrarily nested pipeline and fork-join parallelism. Third, the overhead of Piper may be inherent in the work-stealing scheduling algorithm.

Unfortunately, Lee et al. [22] neither report nor discuss the overhead of Piper, and thus the main reason for the high overhead of Piper is unclear. In particular, the evaluation of Lee et al. shows the poor performance of Piper for the dedup benchmark, which is precisely confirmed by the evaluation in Section 4. Therefore, the reported results about the poor performance of Piper are not surprising according to the evaluation of Lee et al. In fact, the explanation of Lee et al.

about the poor performance of Piper is unrelated to any overhead implied due to an unsuitable implementation used for the evaluation or due to the additional flexibility for nested parallelism.

Section 2 explains the reasons that work-stealing is ineffective when it is combined with the Piper algorithm, and the experimental results precisely confirm the drawbacks of work-stealing in practice. The idea of work-stealing schedulers is to avoid lock-contention on a single shared data structure, by using multiple local dequeues such that threads execute work found in their own deque. Hence, threads steal work when they have an empty deque, and the overhead of a work-stealing scheduler is introduced from stealing operations. This implies that work-stealing performs well as long as the deque of every thread contains sufficient work and stealing rarely occurs. However, this is not the case for Piper [22] due to design choices.

In particular, the work-stealing scheduler of Piper does not avoid lock-contention on a single data structure when threads try to steal work, as there is almost always a single non-empty deque available. Therefore, there exist two possible scenarios. First, all threads immediately find the non-empty deque and lock-contention is not avoided. Second, some threads fail to find the non-empty deque due to random choices of a victim. Thus, although lock-contention is avoided for the moment, every failed attempt introduces overhead from additional scheduling steps. Furthermore, the work-stealing scheduler of Piper introduces overhead since the size of the deque cannot be more than one. In other words, after a successful steal, Piper guarantees that all threads that try to steal from the same deque will necessarily fail. However, this is not the case when there is a single data structure. Hence, the work-stealing scheduler does not avoid lock-contention when threads try to steal work; it introduces more overhead than schedulers that rely on a single shared data structure. This overhead is not necessarily the reason for the poor performance of Piper, but it is inherent in work-stealing scheduling and independent of the Intel Cilk Plus [41, 42] implementation.

Since there are conceptual reasons that work-stealing introduces unnecessary overhead for the execution of dynamic linear pipelines, this work follows a different approach. We show how to achieve high performance by designing a system that exploits the properties of dynamic linear pipelines, and the evaluation confirms our expectations for fine-grained workloads. *Pipelite* enables the design of suitable data structures that are tailored to dynamic linear pipelines, and they both provide fast and concurrent access to all threads. In addition, *Pipelite* always tries to schedule earlier stage iterations first, and it exploits the local suspension mechanism.

The design of *Pipelite* could be adopted by Piper, and thus the performance of Piper may be improved for fine-grained dynamic linear pipelines. However, the main design issue of Piper is that work-stealing scheduling is ineffective for Piper according to the evaluation. The evaluation in Section 4 confirms all the drawbacks of work-stealing scheduling discussed in Section 2. Nevertheless, the reported results do not necessarily imply that work-stealing is the reason for the poor performance of Piper. Experimental results may provide only weak or strong indications.

Finally, this work shows that work-stealing is ineffective for systems that allow scheduling constraints to be expressed only between successive iterations of a stage, and threads execute loop iterations either to completion or until they encounter a dependent stage iteration that needs data that are not available. Otherwise, work-stealing may be effective for the execution of dynamic linear pipelines. For example, work-stealing may be effective for the execution of dynamic linear pipelines that include nested parallelism, which is an interesting contribution of Piper. However, this work does not cover such an evaluation, as *Pipelite* does not support nested parallelism, and Lee et al. [22] do not present any real application that requires this additional flexibility of Piper.

6 RELATED WORK

Previous research efforts are mainly restricted to static linear pipelines and rely on static scheduling. For example, HELIX [8] is a pipelining technique that performs optimizations to reduce the

communication overhead and proposes an efficient mechanism that uses helper threads to prefetch synchronization signals. However, HELIX does not achieve load-balancing for load-imbalanced loops. Moreover, DSWP [33, 37] achieves pipeline parallelism by mapping different stages onto different threads. DSWP was proposed as a mechanism to effectively tolerate variable latency stalls imposed by memory loads, and it was evaluated in a simulation environment using hardware queues. DSWP transfers data between every pair of threads that execute stage iterations of data-dependent stages. Therefore, DSWP may allow stage iterations of the same loop iteration to overlap in time. However, the performance of DSWP is limited by the number of stages. PS-DSWP [36] is an extension of DSWP that allows independent iterations of a stage to execute in parallel, achieving both pipeline and data parallelism. However, stage iterations are assigned to threads in round-robin order, and thus PS-DSWP does not guarantee load-balancing for load-imbalanced loops.

All of these techniques provide compiler support and automatically perform pipelining by detecting loops that can be expressed as linear pipelines. Particularly, they automatically (a) perform data dependence analysis, (b) partition the linear pipeline into a number of static stages, and (c) configure the runtime system. Although *Pipelite* currently requires directives by the user, it may become a fully automatic system by using existing functionality of these systems. For example, *Pipelite* may use the data dependence analysis and the heuristics for partitioning used by the state of the art to automatically generate the required directives. However, it is unclear whether these tools can successfully detect the non-trivial data dependences of the real applications considered in the evaluation of *Pipelite* or not. Furthermore, partitioning of a dynamic linear pipeline into a number of suitable stages and proper configuration of the runtime system are far from obvious.

SpecDSWP [44] is another extension of DSWP that uses control and silent store speculation. SpecDSWP aims at handling loops that cannot be handled with DSWP due to cross-iteration dependences. In particular, SpecDSWP speculates data dependences that are highly predictable, and the overall performance depends on the misspeculation rate. DSWP+ [16] is a technique that partitions the loop body into stages that can subsequently be parallelized by other techniques, and it was evaluated in commodity hardware. Commutative user-defined annotations may be used with DSWP+ to break cross-iteration dependences and achieve better scalability.

Navarro et al. [31] present an analytical model for performance estimation of pipeline parallelism based on queueing theory. Moreover, they compare the performance of the Intel TBB [17] work-stealing scheduler to that of a static scheduler that benefits from over-subscription [3]. Mastoras and Gross [24] study the parallelization tradeoffs for linear pipelines by evaluating LBPP [18], PS-DSWP [36], and Piper [22]. They show that load-balancing and low scheduling overhead are the main challenges for efficient handling of linear pipelines. Mastoras and Gross [25] also describe URTS, a technique for efficient and scalable execution of fine-grained linear pipelines. In addition, Mastoras and Gross [26] present Pipelight, a technique that simplifies partitioning and the configuration of the runtime system, and it achieves better load-balancing than URTS by relying on a hybrid mapping [23] of stages onto threads. Although both URTS and Pipelight are efficient systems, they are restricted to static linear pipelines and do not efficiently handle all the programs that are handled by *Pipelite*.

Vandierendonck et al. [45] present hyperqueues, a programming abstraction of queues that allows the expression of deterministic and scale-free pipeline parallelism. However, hyperqueues cannot express the flexible structure of dynamic linear pipelines supported by *Pipelite*. Furthermore, OpenMP [32] can handle some limited form of dynamic linear pipelines using the ordered directive. One of the main limitations is that OpenMP cannot create ordered regions that are synchronized together if they correspond to different statements of the loop. The OpenMP specification says: “ordered regions that bind to different regions execute independently of each other.”

However, *Pipelite* allows stage iterations that consist of different sequences of statements to be synchronized together, based on the stage identifier (sid), which is determined at runtime.

OmpSs [12] is a programming model for heterogeneous multi-core architectures. The user annotates the code with compiler directives that are similar to that of OpenMP to express task-based parallelism. Pipeline parallelism can be expressed with dataflow relations between tasks, and Chasapis et al. [9] present an evaluation of OmpSs, which includes dedup and ferret from PARSEC [3, 4]. Although the study of Chasapis et al. includes a task-based implementation of x264 with OmpSs, they do not express x264 as a dynamic linear pipeline. The generic-purpose directives of OmpSs are not suitable for expressing dynamic linear pipelines. Similarly, Pop and Cohen [34, 35] propose OpenStream, an expressive extension of a dynamic feed-forward dataflow programming model for stream programming to the OpenMP [32] model. Drebes et al. [11] present an algorithm that utilizes the data dependences between tasks of stream programs to automatically perform a NUMA-aware dynamic task and data placement for an optimized work-stealing scheduler [21].

OpenMP, OmpSs, and OpenStream are generic-purpose programming models than can handle more programs than *Pipelite*. Hence, the scheduling algorithms of these systems are not aware of the properties of dynamic linear pipelines. For example, they allow the expression of non-linear pipelines, i.e., two stage iterations of the same loop iteration may be executed in parallel. Yet the price paid for this flexibility is unnecessary overhead for dynamic linear pipelines, which may negate the overall performance for fine-grained workloads. However, *Pipelite* focuses on a specific class of programs that can be expressed as dynamic linear pipelines, and it aims at performing efficient dynamic scheduling for fine-grained workloads.

Moreover, pipelining with futures [6] and stream programming can handle even non-linear pipelines, although at the expense of higher overhead for linear pipelines. For example, Sanchez et al. [39] present a work-stealing scheduler with a back pressure mechanism. Min and Eom [28] present DANBI, which supports dynamic scheduling based on probabilistic techniques. Furthermore, Schneider and Wu [40] propose a dynamic scheduler for stream programs that scales up to hundreds of threads. However, none of these systems exploits the properties of data dependences of dynamic linear pipelines, and to the best of our knowledge, they do not allow stages and their data dependences to be determined at runtime either. Consequently, efficient systems for static linear pipelines are available, e.g., URTS [25], and systems that support dynamic linear pipelines are also available, e.g., Piper [22]. The challenge and the contribution of *Pipelite* is how to combine the flexibility of systems that handle dynamic linear pipelines with the efficiency of static systems.

7 CONCLUSION

Dynamic scheduling of dynamic linear pipelines is challenging when the goal is to achieve both high performance and scalability. We successfully address both challenges with *Pipelite*, a dynamic scheduler that is tailored to dynamic linear pipelines. *Pipelite* exploits the specific data dependences of dynamic linear pipelines to achieve high performance for a class of important applications that include fine-grained loops. The experimental results show that the design of *Pipelite* implies low overhead, and *Pipelite* significantly outperforms the state of the art in terms of speedup, scalability, and memory usage. Therefore, *Pipelite* achieves high performance for fine-grained loops, for which the state of the art performs poorly due to high overhead.

Pipelite provides compiler support for a directive-based transformation that automatically generates parallel code, making the dynamic scheduler a promising solution that leaves management of the details to tools. The reported results precisely confirm the expectations about the unsuitable design of the state-of-the-art techniques that rely on work-stealing to achieve load-balancing. Moreover, the results confirm that new hardware requires more efficient schedulers. As the number of cores increases, loops become finer-grained, and any overhead will impact scalability. Since

Pipelite is tailored to support dynamic linear pipelines, it provides a path to support this class of applications on future systems that include larger and larger numbers of cores.

ACKNOWLEDGMENTS

We thank Martin Bättig, Michael Faes, Remigius Meier, and the anonymous reviewers for their comments and contributions. We thank I-Ting Angelina Lee for sharing the PARSEC benchmarks used for the evaluation of Piper, and Jim Sukha for answering questions regarding the Intel Cilk Plus implementation of Piper.

REFERENCES

- [1] SPEC. 2006. Standard Performance Evaluation Corporation (SPEC). Retrieved March 22, 2019 from <https://www.spec.org>.
- [2] 2015. Parallel BZIP2 (PBZIP2) Data Compression Software. Retrieved March 22, 2019 from <http://compression.ca/pbzip2>.
- [3] Christian Bienia. 2011. *Benchmarking Modern Multiprocessors*. Ph.D. Dissertation. Princeton University, Princeton, NJ.
- [4] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT'08)*. 72–81.
- [5] Christian Bienia and Kai Li. 2012. Characteristics of workloads using the pipeline programming model. In *Revised Selected Papers from the 3rd Workshop on Emerging Applications and Many-Core Architecture, Held in Conjunction with the 37th International Symposium on Computer Architecture (ISCA'10)*. 161–171.
- [6] Guy E. Blelloch and Margaret Reid-Miller. 1997. Pipelining with futures. In *Proceedings of the 9th Symposium on Parallel Algorithms and Architectures (SPAA'97)*. 249–259.
- [7] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1995. Cilk: An efficient multithreaded runtime system. In *Proceedings of the 5th Symposium on Principles and Practice of Parallel Programming (PPoPP'95)*. 207–216.
- [8] Simone Campanoni, Timothy Jones, Glenn Holloway, Vijay Janapa Reddi, Gu-Yeon Wei, and David Brooks. 2012. HELIX: Automatic parallelization of irregular programs for chip multiprocessing. In *Proceedings of the 10th International Symposium on Code Generation and Optimization (CGO'12)*. 84–93.
- [9] Dimitrios Chasapis, Marc Casas, Miquel Moretó, Raul Vidal, Eduard Ayguadé, Jesús Labarta, and Mateo Valero. 2015. PARSECs: Evaluating the impact of task parallelism in the PARSEC benchmark suite. *ACM Transactions on Architecture and Code Optimization* 12, 4 (Dec. 2015), Article 41, 22 pages.
- [10] Damian Dechev, Peter Pirkelbauer, and Bjarne Stroustrup. 2010. Understanding and effectively preventing the ABA problem in descriptor-based lock-free designs. In *Proceedings of the 13th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC'10)*. IEEE, Los Alamitos, CA, 185–192.
- [11] Andi Drebes, Antoniu Pop, Karine Heydemann, Albert Cohen, and Nathalie Drach. 2016. Scalable task parallelism for NUMA: A uniform abstraction for coordinated scheduling and memory management. In *Proceedings of the 25th International Conference on Parallel Architectures and Compilation (PACT'16)*. 125–137.
- [12] Alejandro Duran, Eduard Ayguadé, Rosa M. Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. 2011. OmpSs: A proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters* 21, 2 (June 2011), 173–193.
- [13] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The implementation of the cilk-5 multithreaded language. In *Proceedings of the 1998 Conference on Programming Language Design and Implementation (PLDI'98)*. 212–223.
- [14] Michael I. Gordon, William Thies, and Saman Amarasinghe. 2006. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*. 151–162.
- [15] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan. 1979. Optimization and approximation in deterministic sequencing and scheduling: A survey. *Annals of Discrete Mathematics* 4 (1979), 287–326.
- [16] Jialu Huang, Arun Raman, Thomas B. Jablin, Yun Zhang, Tzu-Han Hung, and David I. August. 2010. Decoupled software pipelining creates parallelization opportunities. In *Proceedings of the 8th International Symposium on Code Generation and Optimization (CGO'10)*. 121–130.
- [17] Intel. 2012. *Threading Building Blocks Reference Manual*.
- [18] Md Kamruzzaman, Steven Swanson, and Dean M. Tullsen. 2013. Load-balanced pipeline parallelism. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'13)*. Article 14, 12 pages.

- [19] Ralf Karrenberg and Sebastian Hack. 2012. Improving performance of OpenCL on CPUs. In *Proceedings of the 21st International Conference on Compiler Construction (CC'12)*. 1–20.
- [20] Alex Kogan and Erez Petrank. 2011. Wait-free queues with multiple enqueueers and dequeuers. In *Proceedings of the 16th Symposium on Principles and Practice of Parallel Programming (PPoPP'11)*. 223–234.
- [21] Nhat Minh Lê, Antoniu Pop, Albert Cohen, and Francesco Zappa Nardelli. 2013. Correct and efficient work-stealing for weak memory models. In *Proceedings of the 18th Symposium on Principles and Practice of Parallel Programming (PPoPP'13)*. 69–80.
- [22] I-Ting Angelina Lee, Charles E. Leiserson, Tao B. Schardl, Zhunping Zhang, and Jim Sukha. 2015. On-the-fly pipeline parallelism. *ACM Transactions on Parallel Computing* 2, 3 (2015), Article 17, 42 pages.
- [23] Aristeidis Mastoras and Thomas R. Gross. 2016. Unifying fixed code and fixed data mapping of load-imbalanced pipelined loops. (Poster Abstract) In *Proceedings of the 21st Symposium on Principles and Practice of Parallel Programming (PPoPP'16)*. Article 53, 2 pages.
- [24] Aristeidis Mastoras and Thomas R. Gross. 2018. Understanding parallelization tradeoffs for linear pipelines. In *Proceedings of the 9th International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM'18)*. 1–10.
- [25] Aristeidis Mastoras and Thomas R. Gross. 2018. Unifying fixed code mapping, communication, synchronization and scheduling algorithms for efficient and scalable loop pipelining. *IEEE Transactions on Parallel and Distributed Systems* 29, 9 (2018), 2136–2149.
- [26] Aristeidis Mastoras and Thomas R. Gross. 2019. Load-balancing for load-imbalanced fine-grained linear pipelines. *Parallel Computing*. In press.
- [27] Maged M. Michael and Michael L. Scott. 1996. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the 15th Symposium on Principles of Distributed Computing (PODC'96)*. 267–275.
- [28] Changwoo Min and Young Ik Eom. 2013. DANBI: Dynamic scheduling of irregular stream programs for many-core systems. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT'13)*. 189–200.
- [29] Cupertino Miranda, Antoniu Pop, Philippe Dumont, Albert Cohen, and Marc Duranton. 2010. Erbium: A deterministic, concurrent intermediate representation to map data-flow tasks to scalable, persistent streaming processes. In *Proceedings of the 2010 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'10)*. 11–20.
- [30] Adam Morrison and Yehuda Afek. 2013. Fast concurrent queues for x86 processors. In *Proceedings of the 18th Symposium on Principles and Practice of Parallel Programming (PPoPP'13)*. 103–112.
- [31] Angeles Navarro, Rafael Asenjo, Siham Tabik, and Calin Cascaval. 2009. Analytical modeling of pipeline parallelism. In *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques (PACT'09)*. 281–290.
- [32] OpenMP Architecture Review Board. 2015. OpenMP Application Program Interface. Version 4.5. Retrieved March 22, 2019 from <https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf>.
- [33] Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August. 2005. Automatic thread extraction with decoupled software pipelining. In *Proceedings of the 38th International Symposium on Microarchitecture (MICRO-38)*. 105–118.
- [34] Antoniu Pop and Albert Cohen. 2011. A stream-computing extension to OpenMP. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers (HiPEAC'11)*. 5–14.
- [35] Antoniu Pop and Albert Cohen. 2013. OpenStream: Expressiveness and data-flow compilation of OpenMP streaming programs. *ACM Transactions on Architecture and Code Optimization* 9, 4 (Jan. 2013), Article 53, 25 pages.
- [36] Easwaran Raman, Guilherme Ottoni, Arun Raman, Matthew J. Bridges, and David I. August. 2008. Parallel-stage decoupled software pipelining. In *Proceedings of the 6th International Symposium on Code Generation and Optimization (CGO'08)*. 114–123.
- [37] Ram Rangan, Neil Vachharajani, Manish Vachharajani, and David I. August. 2004. Decoupled software pipelining with the synchronization array. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT'04)*. 177–188.
- [38] 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 *Proceedings of the SPLASH'11 Workshop on Transitioning to MultiCore (TMC'11)*. 133–138.
- [39] Daniel Sanchez, David Lo, Richard M. Yoo, Jeremy Sugerman, and Christos Kozyrakis. 2011. Dynamic fine-grained scheduling of pipeline parallelism. In *Proceedings of the 20th International Conference on Parallel Architectures and Compilation Techniques (PACT'11)*. 22–32.
- [40] Scott Schneider and Kun-Lung Wu. 2017. Low-synchronization, mostly lock-free, elastic scheduling for streaming runtimes. In *Proceedings of the 38th Conference on Programming Language Design and Implementation (PLDI'17)*. 648–661.

- [41] Jim Sukha. 2013. Piper: Experimental Support for Parallel Pipelines in Intel Cilk Plus. Retrieved March 22, 2019 from https://www.cilkplus.org/sites/default/files/experimental-software/PiperReferenceGuideV1.0_0.pdf.
- [42] Jim Sukha. 2015. Brief announcement: A compiler-runtime application binary interface for pipe-while loops. In *Proceedings of the 27th Symposium on Parallelism in Algorithms and Architectures (SPAA'15)*. 83–85.
- [43] M. Aater Suleman, Moinuddin K. Qureshi, Khubaib, and Yale N. Patt. 2010. Feedback-directed pipeline parallelism. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT'10)*. 147–156.
- [44] Neil Vachharajani, Ram Rangan, Easwaran Raman, Matthew J. Bridges, Guilherme Ottoni, and David I. August. 2007. Speculative decoupled software pipelining. In *Proceedings of the 16th International Conference on Parallel Architectures and Compilation Techniques (PACT'07)*. 49–59.
- [45] Hans Vandierendonck, Kallia Chronaki, and Dimitrios S. Nikolopoulos. 2013. Deterministic scale-free pipeline parallelism with hyperqueues. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'13)*. Article 32, 12 pages.
- [46] Yuanming Zhang, Gang Xiao, and Takanobu Baba. 2014. Accelerating sequential programs on commodity multi-core processors. *Journal of Parallel and Distributed Computing* 74, 4 (April 2014), 2257–2265.

Received August 2018; revised December 2018; accepted January 2019