Chunking for Dynamic Linear Pipelines

ARISTEIDIS MASTORAS and THOMAS R. GROSS, ETH Zurich, Switzerland

Dynamic scheduling and dynamic creation of the pipeline structure are crucial for efficient execution of pipelined programs. Nevertheless, dynamic systems imply higher overhead than static systems. Therefore, chunking is the key to decrease the synchronization and scheduling overhead by grouping activities. We present a chunking algorithm for dynamic systems that handles dynamic linear pipelines, which allow the number and duration of stages to be determined at run-time. The evaluation on 44 cores shows that chunking brings the overhead of dynamic scheduling down to that of a static scheduler, and it enables efficient and scalable execution of fine-grained dynamic linear pipelines.

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

Additional Key Words and Phrases: Dynamic linear pipeline, parallelization directives, chunking, load-balancing, multi-threading; static and dynamic scheduling

ACM Reference format:

Aristeidis Mastoras and Thomas R. Gross. 2019. Chunking for Dynamic Linear Pipelines. *ACM Trans. Archit. Code Optim.* 16, 4, Article 44 (November 2019), 25 pages.

<https://doi.org/10.1145/3363815>

1 INTRODUCTION

Pipelining is a widely used technique to map single-threaded programs onto multi-core multi-processors. Pipeline parallelism arises from the execution of data-dependent tasks, when independent parts of the tasks can overlap in time. Loops with cross-iteration dependences are a good target for pipeline parallelism, since they usually consist of a large number of iterations (tasks), which may create opportunities for overlapped execution. Pipelining is possible only when cross-iteration dependences allow overlapped execution of different loop iterations.

A *linear pipeline* is a loop where the body is partitioned into a sequence of stages such that stages do not need data generated by subsequent stages; therefore, data of a linear pipeline flow from earlier to later stages. The linear property implies that all statements are executed sequentially within a loop iteration, but different loop iterations can overlap in time, achieving pipeline parallelism. Linear pipelines are ubiquitous in compression algorithms as well as various audio, video, and image processing applications. In particular, they all process a stream of data, and a sequence of functions that form a linear pipeline operates on these data.

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.

1544-3566/2019/11-ART44

<https://doi.org/10.1145/3363815>

Pipeline parallelism has attracted the interest of many researchers. For example, PS-DSWP [26], LBPP [12], and HELIX [6] are systems that automatically map sequential loops onto the pipeline model. All these systems rely on static scheduling algorithms and perform poorly for *load-imbalanced* loops, i.e., loops with iterations that differ substantially in execution time. URTS [17] is a technique that maintains the efficiency of static scheduling and achieves load-balancing by performing dynamic scheduling. However, none of these systems can efficiently handle *dynamic linear pipelines*, i.e., linear pipelines for which stages and their data dependences are determined at run-time. Thus, they all fail to efficiently handle an important class of linear pipelines found in real applications.

PARSEC [3, 4] is a popular benchmark suite that includes pipelined programs. All these programs consist of load-imbalanced loops, and thus, they cannot be handled efficiently by static schedulers. Moreover, Lee et al. [13] show that it is possible to efficiently execute the x264 video encoder included in PARSEC by expressing the main loop as a dynamic linear pipeline. Therefore, to efficiently handle all the pipelined programs included in PARSEC, we need a dynamic system that deals with load-balancing and the flexible structure of dynamic linear pipelines.

Furthermore, previous systems, e.g., PS-DSWP and URTS, map different stages onto different threads, and the decision for the number of threads allocated per stage is crucial for achieving load-balancing and good performance. To solve this problem, Suleman et al. [30] present an algorithm to automatically determine a proper number of allocated threads at run-time. However, such an algorithm is vulnerable to varied execution times found in load-imbalanced loops, since the execution time of later loop iterations cannot be predicted. Therefore, the number of allocated threads must change dynamically, depending on the execution time of different loop iterations.

Pipelite [18] and Piper [13] are dynamic systems that efficiently handle dynamic linear pipelines, and they achieve load-balancing. In contrast to earlier systems [6, 12, 17, 26], both Pipelite and Piper perform dynamic mapping of stages onto threads to achieve load-balancing. Therefore, they successfully address the problem of thread-to-stage allocation that limits the performance of previous systems. Nevertheless, the overhead of dynamic systems is naturally higher than that of static systems as a result of additional flexibility. Moreover, algorithms that reduce the overhead of static systems are incompatible with the flexibility supported by dynamic systems.

In this article, we investigate the performance of dynamic systems for the execution of dynamic linear pipelines. We propose a novel algorithm for chunking that groups together a number of successive iterations, and then any synchronization and scheduling overhead occurs once per group instead of once per iteration. Simple chunking algorithms that have been proposed for static systems cannot be applied to dynamic systems, since the bounds of stages are unknown at compile-time and may vary for different loop iterations.

To evaluate the performance of chunking, we implemented the described algorithm in two systems, a static scheduler for dynamic linear pipelines and the dynamic scheduler of Pipelite [18]. The experimental results obtained on a 44-core machine, using programs selected from three popular collections, show that the algorithm for chunking significantly reduces the overhead of both the static and the dynamic scheduler for the execution of fine-grained dynamic linear pipelines with an average execution time of less than $100\mu\text{sec}$ per loop iteration. In addition, the algorithm for chunking makes the overhead implied by the dynamic scheduler similar to that of the simple static scheduler. Therefore, the algorithm for chunking enables efficient and scalable execution of fine-grained dynamic linear pipelines with the dynamic scheduler.

2 BACKGROUND

A dynamic linear pipeline is a loop partitioned into a sequence of stages. We refer to executions of stages as *stage iterations*; stages form the loop body. 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. First, a stage iteration S_n^j is *dependent* if its execution necessarily occurs after the completion of stage iteration S_n^{j-1} with $j > 1$. Second, 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 dependencies between non-successive iterations or between iterations of different stages are allowed. However, they must not introduce additional scheduling constraints that are not expressed with the scheduling constraints between successive iterations.

These constraints are related to the scheduling algorithm and have been used by the state-of-the-art [13, 18], but they are orthogonal to the contribution of this article. In particular, the chunking algorithm determines the execution order of stage iterations for a chunk of loop iterations. Stage iterations for different loop iterations of a chunk are executed sequentially; they do not create any opportunities for parallel execution. Parallelism is achieved by overlapping the execution of different chunks, and the execution order is determined by the scheduling algorithm. Therefore, we assume the scheduling constraints of the state-of-the-art, and a more detailed description of the flexible structure supported by dynamic linear pipelines can be found in References [13, 18]. In addition, the components of the state-of-the-art that are related to the scheduling algorithm are unnecessary for the chunking algorithm. Consequently, in the rest of this section, we present only those components that are necessary for the described chunking algorithm.

2.1 A Simple Example

The main loop of the dedup compression algorithm, included in the PARSEC [3, 4] benchmark suite, is a simple example of a dynamic linear pipeline that is shown in Listing 1. A chunk of data is read from the input file in every loop iteration, and the deduplicate function determines whether this chunk has already been compressed or not, by calculating the SHA1 signature of the chunk. Therefore, compression is performed only if the chunk has not been compressed before, and the compressed chunk or its SHA1 signature is written to the output file.

```

1  #pragma proteas pipelite num_threads(N) chunk_size(C) throttling(K)
2  #pragma proteas pipelite shared(args, rabintab, rabinwintab)
3  #pragma proteas pipelite private(chunk)
4  while ((chunk = get_next(args, rabintab, rabinwintab)) != NULL) {
5
6      #pragma proteas pipelite dependent(2)           // sid = 2
7      deduplicate(chunk);
8
9      if (!chunk->isDuplicate) {
10         #pragma proteas pipelite independent(3)      // sid = 3
11         compress(chunk);
12     }
13
14     #pragma proteas pipelite dependent(4)           // sid = 4
15     write_chunk_to_file(args->fd_out, chunk);
16 }

```

Listing 1. The main loop of dedup annotated with directives for the Pipelite transformation.

Listing 1 shows the main loop of dedup annotated with directives for Pipelite. The user (programmer) annotates the while loop with the directive `#pragma proteas pipelite`. In addition, the user may configure the performance parameters, e.g., the number of threads with the directive `num_threads(N)`. Similar to the OpenMP [22] model, the data dependences are described by classifying the data used in the loop as either private or shared.

To partition a loop iteration into a sequence of stage iterations, the user annotates the loop body with directives. The first stage iteration of every loop iteration is implicitly declared and includes the loop condition; it may include more statements depending on the beginning of the next stage iteration. Different loop iterations may consist of a different number of stage iterations, and different iterations of the same stage may have different types, i.e., dependent or independent. Therefore, partitioning applies to a single loop iteration. Since the loop condition of a while loop is executed sequentially for different loop iterations, the first stage iteration is always dependent.

A stage iteration S_n^k of the k th loop iteration is identified with the integer n , and we refer to n as *stage identifier (sid)*. The sid of the first stage iteration is 1 for all loop iterations, and the stage identifier must strictly increase for iterations of different stages within the same loop iteration. The sid may be determined explicitly by the user with the argument of the directive, as shown in Listing 1. Otherwise, the default value is automatically determined by the compiler, and it is equal to the sid of the same iteration for the previous stage incremented by one. The directive `#pragma proteas pipelite dependent(sid)` indicates the start of a dependent stage iteration. Similarly, `#pragma proteas pipelite independent(sid)` denotes the start of an independent stage iteration. The directives for partitioning may be inserted within a conditional statement or a loop structure to create stage iterations at run-time.

The loop iterations of dedup are partitioned into three or four stage iterations, depending on whether the chunk of data processed in a given loop iteration must be compressed or not. The number of stage iterations is determined at run-time based on the evaluation of the condition in line 9 of Listing 1. Therefore, the stage iteration with a sid equal to 3 may be missing for some loop iterations, since no compression is performed for the given chunk.

2.2 Parallel Execution of Dynamic Linear Pipelines

The scheduling entity of Pipelite [18] and Piper [13] is a task that represents a single loop iteration. Both systems execute tasks either to completion or until they encounter a dependent stage iteration that needs data that are not available. Therefore, the execution of a task may be suspended and resumed later when the data are available. In particular, a scheduled task is either a new loop iteration or a previously suspended loop iteration that is now ready for execution, i.e., the required data for the execution of the next stage iteration are available.

Figure 1 shows a possible scenario for parallel execution of three loop iterations for the dynamic linear pipeline from Listing 1. The data dependences between stage iterations are indicated with arrows, i.e., the source generates data for the target, and cross-iteration dependences are illustrated with dashed arrows. We assume that the stage iteration S_3^i is missing from the i th loop iteration, since the condition that determines the creation of this stage iteration was evaluated to false in line 9 of Listing 1. There exist two threads; Thread T_A and Thread T_B start with the execution of the $(i - 1)$ -th and i th loop iterations, respectively. At point of time t_1 , Thread T_B cannot execute the dependent stage iteration S_4^i , since the execution of S_4^i requires data generated by the stage iteration S_4^{i-1} , which has not completed. Therefore, the i th loop iteration is suspended, and Thread T_B starts the execution of the $(i + 1)$ -th loop iteration. Once the data for the stage iteration S_4^i are available, i.e., the stage iteration S_4^{i-1} completed, the i th loop iteration is resumed by Thread T_A .

Fig. 1. Possible parallel execution of three loop iterations for the dynamic linear pipeline of dedup, using two threads. The $(i - 1)$ -th and $(i + 1)$ -th loop iterations consist of four stage iterations, and the i th loop iteration consists of three stage iterations, since the processed chunk of data has already been compressed.

2.3 Pipelite Design

Pipelite [18] and Piper [13] are two systems that handle dynamic linear pipelines and achieve load-balancing. They use the same synchronization algorithm to ensure the sequential execution of dependent stage iterations, but they use different mechanisms to handle suspension of tasks. In particular, Pipelite introduces a lightweight mechanism to perform suspension and continuation of tasks. However, the Intel Cilk Plus [29] implementation of Piper uses common setjmp buffers, which imply higher overhead. In the rest of this section, we present the components of Pipelite that are necessary for the chunking algorithm for dynamic linear pipelines. These components rely on the variables `stage` and `stage_sign`, which are included in tasks of Pipelite.

2.3.1 Synchronization Algorithm. Scheduling constraints are expressed only between successive iterations of a stage. Thus, synchronization is required only between pairs of successive loop iterations and is achieved with the stage variable that shows the progress of a loop iteration. There exists a variable `stage` for every loop iteration, and only two threads have access to `stage`.

The thread that executes the task for the i th loop iteration writes to `stage` of the i th loop iteration, and the thread that executes the task for the $(i + 1)$ -th loop iteration reads from the `stage` variable of the i th loop iteration. The value of `stage` is updated based on the `sid` of every completed stage iteration. Therefore, the value of `stage` strictly increases during the execution of a task. In addition, the completion of a stage iteration for the i th loop iteration may enable the execution of a dependent stage iteration for the $(i + 1)$ -th loop iteration. Therefore, if the task for the $(i + 1)$ -th loop iteration is suspended, its execution may be resumed.

For example, in Figure 1, once the two threads start the execution of the $(i - 1)$ -th and the i th loop iteration, the `stage` variable is initialized to 0 for each of them. Then, the `stage` variable of both loop iterations is updated with the `sid` of every completed stage iteration. Hence, the `stage` variable of the $(i - 1)$ -th loop iteration is set to 1 after the completion of the stage iteration S_1^{i-1} . Thread T_B can execute the stage iteration S_1^i , provided that the `stage` variable of the $(i - 1)$ -th loop iteration is greater or equal to 1, since S_1^i is a dependent stage iteration.

2.3.2 Local Suspension. Pipelite achieves efficient suspension of tasks with the local suspension mechanism [18]. In particular, all necessary data for the execution of a task are stored in the underlying data structure of Pipelite. Thus, there is no need to store or load any data when a task is suspended or resumed, respectively. Moreover, suspension may happen only due to a dependent stage iteration, and the beginning of a dependent stage iteration is determined with the directives. Hence, to perform suspension of a task, we need to know only the suspension point, which is

Fig. 2. Parallel execution of four loop iterations for the dynamic linear pipeline of dedup from Listing 1, using two threads and different chunk sizes. All loop iterations consist of four stage iterations.

determined with the directives for dependent stage iterations. To perform suspension and continuation of tasks, the `stage_sign` variable stores a value that indicates the suspension point. Once a task is scheduled, the execution is continued by transferring the control to the proper point.

2.4 Chunking

Chunking groups stage iterations of successive loop iterations into the same task to reduce the frequency of synchronization and the scheduling overhead. In particular, threads execute sequentially a chunk of iterations for each stage, instead of a single iteration, and then they execute another chunk of iterations for the next stage. Therefore, chunking reduces the number of scheduled tasks, and synchronization is performed once per chunk of stage iterations instead of once per single stage iteration. The number of loop iterations that are grouped together is called *chunk size*.

Figure 2 illustrates the parallel execution of four successive loop iterations with chunk size 1 and 2 for the dynamic linear pipeline from Listing 1. For chunk size 1, each thread executes all stage iterations of a single loop iteration. For example, Thread T_A starts the execution of the $(i - 1)$ -th loop iteration, and Thread T_B executes stage iterations of the i th loop iteration. Then, Thread T_A continues with the execution of stage iterations for the $(i + 1)$ -th loop iteration, and so on.

For chunk size 2, each thread executes a chunk of stage iterations, i.e., two successive iterations, as shown in Figure 2. In particular, Thread T_A executes the first stage iterations of the $(i - 1)$ -th and the i th loop iteration, and then it continues with the execution of the next stage iterations for the same loop iterations, and so on. Similarly, Thread T_B executes two successive iterations for every stage of the $(i + 1)$ -th and the $(i + 2)$ -th loop iteration.

For chunk size 1, we notice that Thread T_B can start the execution of a stage iteration after the completion of a single stage iteration by Thread T_A . However, this is not the case for chunk size 2. Thread T_B can execute a stage iteration only after the completion of two successive iterations for a stage executed by Thread T_A . Therefore, small chunk sizes create more opportunities for overlapped execution than large chunk sizes. Nevertheless, small chunk sizes also increase the

frequency of synchronization and scheduling. This is a clear tradeoff, and the overall performance of pipelining depends on various parameters, e.g., the execution time of stage iterations, the synchronization and scheduling overhead, the number of loop iterations, and the system configuration.

2.5 Problem Statement

Although there exist various studies that investigate chunking for static linear pipelines [12, 14, 16, 33], the design of a chunking algorithm for dynamic linear pipelines is not straightforward. Stage iterations of dynamic linear pipelines are created at run-time, and the end of a stage iteration cannot be determined statically by the compiler. In particular, there exist three challenges that must be considered in the design of a chunking algorithm for dynamic linear pipelines:

- stage iterations are created at run-time;
- stage iterations with the same stage identifier (sid) may consist of a different sequence of statements in the loop body;
- and some stage iterations may be missing for some loop iterations, i.e., different loop iterations may consist of a different number of stage iterations.

Therefore, an algorithm for chunking must address these challenges to support all the flexibility of dynamic linear pipelines and reduce the overhead for fine-grained workloads.

3 CHUNKING ALGORITHM

We present an algorithm for chunking that reduces the synchronization and scheduling overhead to achieve efficient execution of fine-grained dynamic linear pipelines. We have implemented the algorithm in two systems: a static round-robin scheduler and the Pipelite [18] dynamic scheduler. Moreover, we have developed directive-based transformations for both systems in the Proteas [16] source-to-source compiler. The chunk size for both systems may be indicated by the user with the parameter `chunk_size(C)`, as shown in Listing 1. Otherwise, the default value is $C = 1$.

Pipelite achieves load-balancing by suspending tasks when the data for the execution of a stage iteration are not available. Contrary to Pipelite, the round-robin scheduler does not suspend tasks; threads wait until the required data become available. Nevertheless, chunking enforces the suspension of a loop iteration after the execution of every single stage iteration, since a group of successive iterations for a stage are executed sequentially before the execution of iterations for later stages of the same chunk. Thus, the round-robin static scheduler uses the local suspension mechanism of Pipelite to perform chunking. The static scheduler and Pipelite differ only in the mechanisms and data structures that are necessary for suspension and continuation of tasks required by the dynamic scheduling algorithm. Hence, in the rest of this, we discuss the algorithm for chunking by using the dynamic scheduler of Pipelite and its transformation as a baseline.

The algorithm for chunking relies on the data structures, algorithms, and mechanisms of Pipelite [18]. However, to perform chunking, a task represents a group of successive loop iterations instead of a single loop iteration. In particular, a task maintains an array to store a replica for the private data of every loop iteration for the chunk of size C . Moreover, a task maintains a single reference to the shared data; and a replica of `ticket`, `state`, `stage`, and `stage_sign`. The variables `ticket` and `state` are irrelevant for the chunking algorithm, and they are not discussed further. Since stage iterations with the same `sid` may consist of different sequences of statements in the loop body, the suspension point of a loop iteration may differ for different loop iterations of a chunk. Therefore, there exists a different replica of the `stage_sign` variable for every loop iteration. Furthermore, a task used by the algorithm for chunking requires some additional data, as shown in dark blue on the right of Figure 3, and they are explained later.

ticket	next_sid[C]
state	min_sid
stage	next_min_sid
stage_sign[C]	current
private[C] shared	chunk_size

Fig. 3. Task for the chunking algorithm.

Fig. 4. Sequential execution of stage iterations for three loop iterations of a chunk. The loop iterations consist of different numbers of stage iterations, and stage iterations have different sid.

3.1 Execution of Stage Iterations for a Chunk of Loop Iterations

The basic idea of the chunking algorithm is that a thread executes a stage iteration for the first loop iteration of the chunk, and then it executes iterations with the same sid for all later loop iterations of the chunk. This is always possible for the first stage iteration, since the first stage iteration cannot be missing and the sid of the first stage iteration is 1 for every loop iteration. For example, Figure 4 illustrates the execution of the stage iterations for three successive loop iterations of a chunk. In the beginning, the stage iteration with sid equal to 1 is executed for every loop iteration.

However, dynamic linear pipelines allow iterations of later stages to be missing, and both the number and the stage identifier (sid) of later stage iterations are determined at run-time for every loop iteration of the chunk. To make a decision about the execution order of the next stage iterations, the chunking algorithm needs to know the sid of the next stage iteration for all loop iterations of the chunk. Hence, tasks maintain an array `next_sid` that stores the sid of the next stage iteration that will be executed for every loop iteration of the chunk, as shown in Figure 3. The `next_sid` variable is initialized to 1, since the sid of the first stage iteration is always 1. After the completion of every stage iteration, `next_sid` is updated with the sid of the next stage iteration.

After the execution of all iterations for the first stage in Figure 4, the chunking algorithm executes stage iterations with a sid equal to 6. However, the stage iteration with sid equal to 6 is missing from the i th loop iteration. In this case, no stage iteration is executed for the i th loop iteration, and the execution continues with the stage iteration of the $(i + 1)$ -th loop iteration.

The sid of the next stage iteration that will be executed for the first loop iteration of the chunk may be greater than the sid of the next stage iteration that will be executed for a later loop iteration of the chunk. Therefore, the chunking algorithm calculates the minimum value for the sid of the next stage iteration for all loop iterations of the chunk; and then for every loop iteration of the chunk, it executes stage iterations with a sid that is equal to the minimum sid. Consequently, tasks maintain the `min_sid` variable that indicates the stage identifier based on which chunking is currently performed. The `min_sid` variable is initialized to 1 for every task, since the sid of the

first stage iteration is 1 for all loop iterations. During the execution of iterations for a stage based on the value of `min_sid`, the chunking algorithm calculates the next value of the `min_sid` variable, i.e., the second minimum `sid`, based on which iterations will be executed after the completion of all iterations with a `sid` equal to the current `min_sid`. Therefore, tasks maintain the `next_min_sid` variable that stores the next minimum `sid`, and it is initialized to the maximum possible value.

For example, in Figure 4, after the completion of all stage iterations with a `sid` equal to 6, the `sid` of the next stage iteration that will be executed for the first loop iteration of the chunk is 10. However, the `sid` of the next stage iteration that will be executed for the $(i + 1)$ loop iteration is 7. Therefore, the value of `min_sid` is 7 and chunking is performed based on 7. Since there is no stage iteration with a `sid` equal to 7 for the first two loop iterations of the chunk, no stage iteration is executed. Nevertheless, while the algorithm traverses all loop iterations of the chunk, it calculates the value of the `next_min_sid` variable. Once the single stage iteration with a `sid` equal to 7 is completed, the execution continues based on the next minimum `sid`, which is 10. Therefore, all stage iterations with a `sid` equal to 10 are executed and the whole task is completed.

Finally, tasks maintain the variable `current` that indicates the loop iteration of the chunk for which a stage iteration is currently executed. The `current` variable is initialized to 0, since the execution starts with the first loop iteration of the chunk. In addition, since the total number of loop iterations may not be divided evenly by the chunk size, the last task, i.e., chunk of loop iterations, may consist of less than C loop iterations. Therefore, `chunk_size` stores the chunk size; it is initialized to C , and this value may change later only for the last task.

3.2 Design and Implementation of the Chunking Algorithm

The Proteas [16] source-to-source compiler generates code that is executed by both the static scheduler and the Pipelite dynamic scheduler. Since the code for the two systems differs only in the statements for suspension of tasks that is required for dynamic scheduling, we discuss only the chunking algorithm for the Pipelite dynamic scheduler.

The compiler replaces the directives for the creation of stage iterations with code that is similar to that presented in Listing 2. The chunking algorithm is essentially performed with the interaction between the code of Listing 3 and the code of Listing 2. In particular, the code in Listing 3 transfers the control to the next stage iteration that will be executed, and once this stage iteration is completed, the code in Listing 2 transfers the control back to the code in Listing 3 to choose

```

1      // code of completed stage iteration with sid = 1 goes here
2
3      t.stage_sign[t.current] = 2;
4      t.next_sid[t.current] = 2;      // t.next_sid[t.current]++; if sid is omitted
5
6      if (t.next_sid[t.current] < t.next_min_sid)
7          t.next_min_sid = t.next_sid[t.current];
8
9      if (++t.current < t.chunk_size)
10         goto stage_handler;
11
12     goto update_stage_handler;
13
14     // the label and the if statement are required
15     // only for dependent stage iterations
16 p2: if (proceed(r, t, t.next_sid[t.current]) != 0)
17     goto suspend;
18
19 s2: // code of next stage iteration with sid = 2 goes here

```

Listing 2. Pseudo-code executed by the thread pool after the completion of a stage iteration.

```

1  stage_handler:
2      switch (t.stage_sign[t.current]) {
3          case 1:
4              if (t.next_sid[t.current] == t.min_sid)
5                  if (t.current == 0) goto p1;
6                  else goto s1;
7              break;
8          case 2:
9              if (t.next_sid[t.current] == t.min_sid)
10                 if (t.current == 0) goto p2;
11                 else goto s2;
12              break;
13          case 3:
14              if (t.next_sid[t.current] == t.min_sid)
15                 goto s3;
16              break;
17          case 4:
18              if (t.next_sid[t.current] == t.min_sid)
19                 if (t.current == 0) goto p4;
20                 else goto s4;
21      }
22
23      if (t.next_sid[t.current] < t.next_min_sid)
24          t.next_min_sid = t.next_sid[t.current];
25
26      while (++t.current < t.chunk_size) {
27          if (t.next_sid[t.current] == t.min_sid)
28              goto stage_handler;
29
30          if (t.next_sid[t.current] < t.next_min_sid)
31              t.next_min_sid = t.next_sid[t.current];
32      }
33
34  update_stage_handler:
35      update_stage(r, t, t.next_min_sid - 1);
36
37      t.min_sid = t.next_min_sid;
38      t.next_min_sid = MAX_ULONG;
39      t.current = 0;
40      goto stage_handler;

```

Listing 3. Pseudo-code that transfers the control to the proper label of the scheduling loop.

the next stage iteration. The code of both Listing 2 and Listing 3 is part of the function shown in Listing 4. This function includes the scheduling loop and is executed by the thread pool of the scheduler.

3.2.1 Completion of Stage Iteration. Listing 2 presents the code that replaces the directive `#pragma proteas pipelite dependent(2)` of Listing 1. In fact, this code separates the execution of two successive stage iterations for the same loop iteration. Lines 16 and 19 show that two different labels are introduced for a replaced directive that creates a dependent stage iteration. In particular, the label `p` points to the test that determines whether the next stage iteration can be executed or not, depending on whether there are required data that are not available. However, the label `p` is not introduced for independent stage iterations, since there are no scheduling constraints that may prevent the execution of independent stage iterations. In addition, the label `s` determines the beginning of a stage iteration and is used to execute a chunk of iterations for a stage.

Once a stage iteration is completed, the execution of the current loop iteration is suspended, and the control is transferred either to a later iteration for the same stage or to an earlier iteration for a later stage, depending on whether the execution of all iterations for the stage are completed

```

1  void func(pipelite_rts_t r) {
2      shared_t s = r.shared_data;
3
4      do {
5          task_t t = get_task(r);
6          private_t p = t.private_data;
7
8          // control transfer goes here
9
10 pl:    if (proceed(r, t, 1) != 0)
11         goto suspend;
12
13 sl:    if (r.condition) {
14         // the while is replaced with an if
15         if ((loop_condition)) {
16
17             // loop body goes here; directives are suitably replaced by Listing 2
18
19             t.next_sid[t.current] = MAX_ULONG;
20             if (++t.current < t.chunk_size)
21                 goto stage_handler;
22
23             complete_task(r, t);
24             goto complete;
25
26 suspend:    suspend_task(r, t);
27 complete:
28         } else {
29             t.chunk_size = t.current;
30             terminate_rts(r, t);
31
32             if (t.current == 0)
33                 complete_task(r, t);
34             else {
35                 t.current = 0;
36                 goto stage_handler;
37             }
38         }
39     } else {
40         complete_task(r, t);
41     }
42 } while (r.condition || exist_ready_tasks(r));
43 }

```

Listing 4. Pseudo-code for the function that is executed by the thread pool for the loop from Listing 1.

or not. The suspension of the loop iteration is performed using the local suspension mechanism of Pipelite. Therefore, the `stage_sign` variable is set to a value that indicates the suspension point such that the execution can be resumed later from the same point with the execution of the same iteration for the next stage. This value is independent of the `sid` for the next stage iteration. The value used for the update of `stage_sign` can be any arbitrary integer and is statically determined by the compiler, e.g., 2, as shown in line 3 in Listing 2. The only requirement is that the same value is used by the code shown in Listing 3, which is explained later, to resume the execution of the suspended loop iteration and to continue the execution with the same iteration for the next stage. Although it is unnecessary, the same value is used for the names of the labels `p` and `s`. In fact, if a directive for a stage iteration is included in a loop structure, a number of stage iterations may be created at run-time. However, the suspension point is the same for all these stage iterations that are determined at run-time. Therefore, the `stage_sign` variable is updated with the same value.

Furthermore, the `next_sid` variable is updated with the `sid` of the next stage iteration that will be executed for the current loop iteration of the chunk. The `sid` is defined in the directive that creates the next stage iteration, i.e., 2, for the example shown in Listing 2. If the `sid` is omitted from the directive, then the compiler generates code that will determine the value for the `next_sid` variable at run-time. In particular, the value of `next_sid` is incremented by one. Since the `sid` of the next stage iteration is available, the condition in line 6 checks if this `sid` is less than the minimum `sid` that has been found so far for the next stage iteration that will be executed for all loop iterations of the chunk. Therefore, the calculation of the minimum value for the `sid` of the next stage iteration that will be executed is completed with the completion of all iterations for the stage. Once all necessary data have been updated, it is necessary to check whether there is another iteration of this stage that must be executed for this chunk or not in line 9. If any exists, then the control is transferred to `stage_handler`. Otherwise, the last iteration of the stage has been completed, and the control is transferred to the label `update_stage_handler` that increases the value of the stage variable, which shows the progress of the executed task.

The execution of the stage iteration that follows the label `s2` is performed after the completion of all iterations for the previous stage. For dependent stage iterations, the control is transferred to the label `p2` for the execution of the first iteration for a stage. It is necessary to invoke `proceed`, as shown in line 16, to determine whether the execution of the stage iteration is possible or not due to scheduling constraints. However, the control is transferred directly to the label `s2` for later iterations of the stage, since the completion of the first iteration guarantees that all later iterations for the same stage can be executed. In addition, the control is transferred directly to the label `s2` for independent stage iterations, as their execution is not restricted by any scheduling constraints.

3.2.2 Control Transfer. Listing 3 presents the code that corresponds to the control transfer algorithm required for the dedup benchmark from Listing 1. After the completion of a stage iteration, the control is transferred to the `stage_handler` label in line 1. For each starting point for the execution of a stage iteration, one case is introduced in the `switch` statement shown in line 2. This `switch` statement determines the next stage iteration that must be executed according to the chunking algorithm, and it transfers the control to the proper stage iteration.

The execution of a task continues based on the value of the `stage_sign` variable that is used for the local suspension mechanism. The `stage_sign` variable is initialized to 1 during the creation of a new task, since the execution starts with the stage iteration of the first stage. If the value of `stage_sign` is different than 1, then the task has been suspended before, and the value of `stage_sign` shows the suspension point. Therefore, the execution must be resumed from the same point to continue with the execution of the next stage iteration.

However, the execution of the next stage iteration is possible only if the value of `next_sid` for the current loop iteration is equal to the minimum `sid` of the next stage iteration that will be executed for all loop iterations of the chunk. Hence, `next_sid` is compared with `min_sid`, and if these two values are equal, then the control is transferred either to the `p` or to the `s` label. For dependent stage iterations, the control is transferred to the label `p` when the currently executed iteration is the first of the chunk, since it is necessary to check whether there exist any scheduling constraints or not. There are no scheduling constraints for later loop iterations of the chunk, since scheduling constraints are expressed only between successive iterations of a stage (as explained in Section 2), and the chunking algorithm guarantees that the previous iteration of the stage is necessarily completed as it belongs to the same task, i.e., chunk of loop iterations. If the next executed stage iteration of the current loop iteration is independent, then the control is transferred directly to the label `s`, as shown in line 15 of Listing 3. Hence, the conditional statement that checks the value of the current variable is missing for independent stage iterations.

The control is transferred to neither an `s` nor a `p` label when the `sid` of the next stage iteration is not equal to `min_sid`. In this case, the execution of the next stage iteration is skipped for the moment. The conditional statement in line 23 of Listing 3 calculates the next minimum value for the `sid` of the next stage iterations, which will be used for the execution of the next stage iterations. In addition, there may exist a later loop iteration of the chunk, where the `sid` of the next executed stage iteration is equal to the `min_sid` variable. The while loop in line 26 searches for such a loop iteration, and in the case of success, the control is transferred back to `stage_handler`.

Otherwise, the completion of the while loop guarantees that all iterations of the stage with a `sid` equal to `min_sid` have been completed. Therefore, the variable `stage` that shows the progress of the task is updated by invoking `update_stage`, as shown in line 35, such that a dependent stage iteration of the next task can be executed. Moreover, the control is transferred to the `update_stage_handler` with the code shown in Listing 2 after the completion of the last iteration for a stage. Since all iterations of a stage are completed, the execution continues with iterations for the next stage, which is determined based on the `min_sid` value. Therefore, `min_sid` is set to `next_min_sid` in line 37, and `next_min_sid` is initialized to the maximum possible value in line 38. Finally, the execution of the task continues with a stage iteration for the first loop iteration of the chunk, and thus, `current` is set to 0, and the control is transferred to `stage_handler` to determine the next stage iteration that will be executed.

3.2.3 Scheduling Loop. The function for the chunking algorithm is presented with pseudo-code in Listing 4. This function is executed by the thread pool of Pipelite [18] for any dynamic linear pipeline, e.g., `dedup` from Listing 1. The condition of the annotated loop is inserted in line 15, and the loop body is inserted in line 17.

Each thread executes the scheduling loop, i.e., the `do` loop, while the loop condition is true or there exist previously suspended tasks that are now ready for execution. In particular, a thread receives a task in every loop iteration, and the control is transferred to the proper stage iteration depending on whether the task has been suspended before or not. Control transfer is achieved with the code shown in Listing 3, which is missing from line 8 of Listing 4.

Once the last stage iteration of a loop iteration is completed, the `next_sid` variable is set to the maximum possible value, as shown in line 19, such that `next_sid` of the completed iteration has no impact on the calculation of the minimum `sid`. Moreover, the control is transferred to the label `stage_handler`, as shown in line 21 in Listing 4, and the execution continues with a stage iteration for a later loop iteration of the chunk. After the completion of the last stage iteration for all loop iterations of the chunk, the function `complete_task` is invoked in line 23. Then the control is transferred to the label `complete`, and the execution of the task is completed.

Provided that the loop condition of the dynamic linear pipeline has already been evaluated to false in line 15, it will not be evaluated again as a result of the conditional statement in line 13. The condition in line 13 is evaluated to false if there are no new tasks to be scheduled, and threads invoke `complete_task` to immediately complete the execution of the current task in line 40.

Furthermore, if the condition in line 15 is evaluated to false, the size of the last chunk becomes known, and the variable `chunk_size` is updated in line 29. In particular, the chunk size is set to the identifier of the currently executed loop iteration, i.e., the variable `current`. The size of the last chunk may be less than the parameter `C` if the total number of loop iterations for the dynamic linear pipeline is not divided evenly by `C`. Therefore, the thread invokes the function `terminate_rts` to inform the scheduling algorithm that this is the last scheduled task.

The next step checks the size of the last chunk in line 32. If the variable `current` is equal to 0, then the loop condition has been evaluated to false for the first loop iteration of the last task. Therefore, no further stage iteration must be executed, and the task is completed by invoking `complete_task`.

in line 33. Otherwise, the first stage iteration for all loop iterations of the last chunk are completed, and the execution of the task continues with the next stage iterations. Consequently, `current` is initialized to 0, as shown in line 35, and the control is transferred to the label `stage_handler`, which will transfer the control to the proper stage iteration.

3.3 Design and Performance Tradeoffs

The described algorithm performs chunking based on the minimum `sid` of the next stage iteration for every loop iteration of the chunk. This design essentially implies an exhaustive search of all stage iterations with a `sid` that is equal to every different `sid` defined for some stage iteration of the task. Hence, for large values of the chunk size, the chunking algorithm may not perform well for dynamic linear pipelines where there exist only few iterations for each stage. In particular, the algorithm traverses all the loop iterations of the chunk, and in the worst case, it may execute a single stage iteration with a `sid` that is equal to the minimum. It is possible to avoid this situation by using additional data structures that store the first and the last loop iteration of the chunk that includes a stage iteration with a given `sid`. However, these additional computing steps imply overhead that may degrade the performance for common situations found in real applications.

Another design option is to execute the next stage iteration for the first loop iteration of the chunk, and then to continue with the execution of all stage iterations with a `sid` that is less or equal to that of the stage iteration that was executed for the first loop iteration. This is a simple approach that essentially merges iterations of successive stages for some loop iterations, and thus this approach leads to poor performance for common situations with a missing stage iteration.

To avoid the drawbacks of both approaches, i.e., exhaustive search and merging of stage iterations, we can combine the two approaches and execute a single stage iteration for every loop iteration of the chunk. In particular, chunking is performed based on the `sid` of the next stage iteration for the first loop iteration of the chunk. However, the `sid` of the next stage iteration for a later loop iteration may be less than that of the next stage iteration for the first loop iteration. In this case, the chunking algorithm skips the execution of the next stage iteration for the first loop iteration of the chunk until it is equal to the minimum `sid` of the next stage iteration for all loop iterations of the chunk. A single stage iteration is executed for all later loop iterations of the chunk when their `sid` is less than that of the skipped stage iteration. In this way, the chunking algorithm does not merge stage iterations, since only a single stage iteration can be executed for every loop iteration of the chunk, and it avoids exhaustive searches that lead to the execution of a stage iteration for a single loop iteration. However, this approach may group together the execution of stage iterations with a different `sid`, which implies that the stage variable that shows the progress of the task is updated based on the minimum `sid` of all executed stage iterations. In other words, the chunking algorithm may execute stage iterations with a greater `sid`, but the scheduling algorithm is not aware of this additional progress of the execution. Therefore, this situation may lead to performance degradation. We have implemented this approach, and the evaluation shows that it performs similarly to the chunking algorithm described in this article for all benchmarks considered in Section 4.

4 EVALUATION

We conducted an extensive evaluation of the described algorithm for chunking based on programs selected from three widely used collections [1–4]. All programs include a main pipelined loop that is load-imbalanced, it is executed once, and it contributes about 99% of the total sequential execution time. Table 1 presents a description of the programs and the data set used for the evaluation. The last column reports the pipeline pattern for each program, where the letters D and I stand for dependent and independent stage iterations, respectively.

Table 1. Benchmark Specifications

Benchmark	Benchmark Suite	Description	Data Set	Pipeline Pattern
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	$D \rightarrow I \rightarrow D$
bzip2	Parallel BZIP2	Compression	53.8 MB	$D \rightarrow I \rightarrow D$
x264	PARSEC	Video encoding	native	$D \rightarrow I \rightarrow (D I)^+ \rightarrow D$
ferret	PARSEC	Content-based similarity search	native	$D \rightarrow I \rightarrow D$
dedup	PARSEC	Compression	native	$D \rightarrow D(\rightarrow I) \rightarrow D$

Table 2. Loop Details

Benchmark	Sequential Execution Time	Loop Iterations (#)	Average Execution Time of Loop Iterations	Granularity
hmmcalibrate	37 sec	85,000	435 μ sec	fine-grained
hmmsearch	116 sec	122,564	949 μ sec	coarse-grained
bzip2	6 sec	60	96,000 μ sec	coarse-grained
x264	515 sec	512	1,007,000 μ sec	coarse-grained
ferret	381 sec	3,500	110,700 μ sec	coarse-grained
dedup	23 sec	369,312	60 μ sec	fine-grained

The source code for both the static scheduler and Pipelite 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 programs 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 3% of the average execution times, excluding very few experiments of the dedup benchmark and the very fine-grained hmmcalibrate, which reach up to 4% and 15%, respectively.

The experiments were conducted on an Intel Xeon E5-2699 v4 with two processors and a total of 44 cores. The clock frequency is 2.20 GHz, and the turbo frequency reaches 3.60 GHz. To show scalability 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 to 2.80 GHz. The main memory is 512 GB, and the last-level cache is 55 MB. The memory allocation becomes the bottleneck for dedup. To remove this performance inhibitor, we used *libjemalloc.so.1* instead of the default memory allocator. Moreover, 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.

4.1 Chunk Size

Figures 5, 6, 7, 8, 9, and 10 show the performance of the static scheduler and Pipelite for $N = 44$ threads when we vary the chunk size. To show that the overhead implied by the chunking algorithm is negligible, we use two additional lines, Static and Pipelite, that correspond to the speedups obtained from both schedulers when the algorithm for chunking is not used. Hence, these two lines are horizontal, since they show the performance of the two systems without chunking, and for all benchmarks, we notice that the speedups obtained from Chunked Static and Chunked Pipelite for chunk size $C = 1$ are similar to that obtained from Static and Pipelite, respectively.

Since the performance of Pipelite depends on the throttling limit (K) that determines the maximum number of active tasks, we used a value that reaches the highest performance for $C = 1$. Although a smaller value may be used for the throttling limit of Chunked Pipelite for some benchmarks, we used the same value, since it does not affect the performance, unless stated otherwise.

For the hmmcalibrate benchmark, both the static scheduler and the Pipelite dynamic scheduler achieve similar performance when chunking is not used as shown in Figure 5. Furthermore, they both benefit from chunking and achieve the highest performance for the same chunk size. The

Fig. 5. Speedups of `hmmcalibrate` using $N = 44$, $K = 176$ for Pipelite, and varying the chunk size.

Fig. 6. Speedups of `hmmsearch` using $N = 44$, $K = 880$ for Pipelite, and varying the chunk size.

Fig. 7. Speedups of `bzip2` using $N = 44$, $K = 88$ for Pipelite, and varying the chunk size.

Fig. 8. Speedups of x264 using $N = 44$, $K = 880$ for Pipelite, and varying the chunk size.

Fig. 9. Speedups of ferret using $N = 44$, $K = 440$ for Pipelite, and varying the chunk size.

Fig. 10. Speedups of dedup using $N = 44$, $K = 88$ K for Pipelite, $K = 1,760$ for Chunked Pipelite, and varying the chunk size.

static scheduler achieves slightly better performance than Pipelite for chunk size $C = 20$, and the performance is degraded for both schedulers for a large chunk size, i.e., 1 K. This situation is expected, as large chunk sizes reduce the parallelization opportunities.

The experiments for *hmmsearch* show more interesting results, as depicted in Figure 6. Pipelite achieves significantly better performance than Static, as a result of better load-balancing achieved by the dynamic scheduling algorithm. The performance of Pipelite is not improved further with chunking, since Pipelite achieves almost linear speedups without chunking. In other words, chunking does not improve the performance of Pipelite by reducing the already low overhead. Since the overhead implied by the dynamic scheduler is low, it is expected that the overhead implied by the static scheduler is even lower. Thus, chunking cannot reduce further the overhead of Static. However, chunking improves the performance of Static by 34% when the chunk size is set to $C = 1$ K. Furthermore, we notice that Static achieves the same speedups for *hmmcalibrate* and *hmmsearch* when chunking is not used. Although the loop of *hmmcalibrate* is finer-grained than that of *hmmsearch* (as shown in Table 2), chunking improves the performance for *hmmcalibrate* only up to 12% but that for *hmmsearch* up to 34%. These two observations imply that the reason for the significant performance improvement for *hmmsearch* is not the reduced overhead. The static scheduler does not suspend tasks if the data for the execution of a stage iteration are not available; threads wait until the data become available. Hence, by grouping together a number of successive iterations for a stage, the workload may evenly be distributed to tasks. This situation reduces the waiting time of threads and improves the performance of the static scheduler.

The speedups obtained from *bzip2* and *x264* are presented in Figures 7 and 8, respectively. Both benchmarks include a load-imbalanced loop, and thus Pipelite outperforms Static as a result of better load-balancing achieved by dynamic scheduling. However, the performance for neither *bzip2* nor *x264* is improved with chunking. This is expected, since both benchmarks include very coarse-grained loops that consist of a small number of loop iterations (as shown in Table 2).

Figure 9 shows the experimental results for *ferret*. The Pipelite dynamic scheduler outperforms the static scheduler by a large margin due to the load-imbalanced loop when chunking is not performed. In addition, the performance of Pipelite is degraded for chunk sizes that are greater than $C = 2$, since the loop is coarse-grained. In contrast to Pipelite, chunking improves the performance of the static scheduler up to 164% for $C = 100$. Similar to *hmmsearch*, the significant performance improvement for *ferret* is a result of even distribution of the workload to tasks. Experimental results for larger chunk sizes are not included, since larger values lead to a small number of tasks, e.g., fewer than the number of threads, and both the static scheduler and Pipelite perform poorly.

For *dedup*, the performance of Pipelite is similar to that of Static when chunking is not used, as shown in Figure 10. Both schedulers benefit from chunking and reach the highest performance for $C = 1$ K. However, chunking amortizes the high overhead of dynamic scheduling and makes Pipelite more efficient than the static scheduler, thanks to better load-balancing. For Chunked Pipelite, the throttling limit $K = 1,760$ leads to slightly worse performance than the highest when $C = 1$, but we used this configuration for the sake of simplicity. Moreover, although Static achieves the highest performance for $N = 44$ threads, a smaller number of threads leads to better performance for Pipelite due to overhead. Since Static and Pipelite achieve the highest performance for different configurations, we used $N = 44$ threads for both techniques. For Pipelite, this configuration leads to the largest improvement as a result of chunking, but this is not the case for Static. Hence, additional experimental results for different numbers of threads are presented in the next section.

Fig. 11. Speedups of hmmcalibrate using $C = 20$, $K = 176$ for Pipelite, and varying the number of threads.

Fig. 12. Speedups of hmmsearch using $C = 1$ K for Chunked Static, $C = 1$ for Chunked Pipelite, $K = 880$ for Pipelite, and varying the number of threads.

4.2 Scalability

For those benchmarks that the chunking algorithm improves the performance of the evaluated schedulers, we include additional experimental results to show the scalability when we vary the number of threads. We configured the runtime system of both schedulers properly, by using the chunk size (C) and the throttling limit (K) that reach the highest performance.

For hmmcalibrate, Pipelite achieves load-balancing as a result of dynamic scheduling, and it slightly benefits from chunking, as shown in Figure 11. The static scheduler does not achieve load-balancing and performs poorly for a small number of threads. However, the performance of the static scheduler is significantly improved with chunking. The main reason for this improvement is the even distribution of the workload to tasks rather than the reduced overhead.

The experimental results for hmmsearch and ferret are similar, as shown in Figures 12 and 13, respectively. Both benchmarks include a coarse-grained loop, and thus any performance improvement for the static scheduler is not due to overhead reduction. The reason is that the workload is distributed evenly to tasks as a result of grouping together successive iterations of a stage. The data set of ferret consists of 3,500 loop iterations, which implies a total of 35 tasks for chunk size $C = 100$. Therefore, the performance of Static is the same for 36 or more threads. The performance

Fig. 13. Speedups of ferret using $C = 100$ for Chunked Static, $C = 2$ for Chunked Pipelite, $K = 440$ for Pipelite, and varying the number of threads.

Fig. 14. Speedups of dedup using $C = 1$ K, $K = 88$ K for Pipelite, $K = 1, 760$ for Chunked Pipelite, and varying the number of threads.

of Pipelite is essentially not improved further with chunking neither for hmmsearch nor for ferret, since Pipelite achieves linear speedups without chunking. In particular, chunk size $C = 1$ leads to the highest speedups for hmmsearch, and chunk size $C = 2$ leads to the highest speedups for ferret.

Figure 14 shows that Pipelite significantly outperforms the static scheduler for dedup for almost all numbers of threads. However, Pipelite implies higher overhead than the static scheduler for such a fine-grained loop, and thus the two schedulers achieve similar performance for $N = 44$ threads. Chunking significantly improves the performance of both schedulers, but the dynamic scheduler still outperforms the static scheduler for all numbers of threads.

4.3 Overhead Measurement

Chunking improves the performance of both static and dynamic schedulers for fine-grained linear pipelines. However, it is unclear whether chunking makes dynamic schedulers efficient for fine-grained linear pipelines that are efficiently handled by static schedulers. To provide an answer to this question, we use *hmmcalibrate*, which allows the user to choose the length of the sequence that is randomly produced in every loop iteration. We use sequence length 10, which results in an average execution time of only $9 \mu\text{sec}$ per loop iteration. Such a fine-grained loop is suitable for our

Fig. 15. Speedups of `hmmcalibrate` for sequences' length 10 using $N = 44$, $K = 176$ for Pipelite, and varying the chunk size.

evaluation, since it is sufficiently fine-grained to reveal the overhead of dynamic scheduling, but it does not inhibit linear speedups for a small number of threads. Moreover, the loop is load-balanced to allow the static scheduler to perform as well as the dynamic scheduler.

The speedups of `hmmcalibrate` for different chunk sizes are presented in Figure 15. We notice that both schedulers significantly benefit from chunking, and Pipelite reaches about $4\times$ higher speedups when chunking is used. Although the data set of `hmmcalibrate` includes a load-balanced loop, Pipelite slightly outperforms the static scheduler, and there exist two possible explanations.

First, chunking makes the static scheduler more dynamic by adding suspension of loop iterations, since the execution of a loop iteration is suspended after the completion of every single stage iteration. In particular, the chunking algorithm requires more data to be available at the same time, and this amount of data depends on the chunk size. Therefore, chunking may lead to worse data locality for the static scheduler, which may have significant impact on the overall performance.

Second, although the loop is load-balanced, there is no guarantee that threads of the static scheduler do not wait for data that are not available. A thread that executes an early task may make progress more slowly than a thread that executes a later task. Thus, the second thread may have to wait until the first thread completes the execution of a stage iteration. This delay on the progress of a thread is possible when the operating system scheduler assigns another task to the core used by the static scheduler. Moreover, hyper-threading technology, turbo frequency, and NUMA design may affect the progress of threads that execute tasks with equivalent workload. Hence, the dynamic scheduler may perform slightly better than a static scheduler even for load-balanced loops, since dynamic scheduling can tolerate the dynamic workload variation better than static scheduling.

Figure 16 shows that both schedulers achieve the same performance for up to $N = 8$ threads when chunking is not used, and Static outperforms Pipelite for a larger number of threads as a result of higher overhead. The performance of both schedulers is improved with chunking, and Pipelite achieves similar performance with the static scheduler for up to 20 threads. The difference on the performance of the two schedulers for more than 20 threads is an indication that the NUMA design leads to dynamic workload variation for the load-balanced loop.

5 RELATED WORK

We present various techniques that perform chunking for static linear pipelines. Moreover, we present the state-of-the-art on load-balancing for static linear pipelines and systems that support

Fig. 16. Speedups of `hmmcalibrate` for sequences' length 10 using $C = 500$, $K = 176$ for Pipelite, and varying the number of threads.

dynamic linear pipelines but lack support for chunking. Since parallelization and configuration of the runtime system using directives is a non-trivial task for the user, we present systems that perform automatic parallelization of linear pipelines, and they automatically configure the runtime system to achieve efficient execution for different data sets and machine configurations.

Otoni et al. [23] present DSWP, a technique that automatically partitions a loop into stages to achieve pipeline parallelism. 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 [27, 28]. DSWP transfers data between every pair of threads that execute iterations of data-dependent stages, and it uses different queues for different data. Although this design allows overlapped execution of stage iterations included in the same loop iteration, Rangan et al. [27] state that such a complex communication scheme may limit the overall performance.

Raman et al. [26] describe PS-DSWP, which is an extension of DSWP that allocates multiple threads for the execution of independent stage iterations, and it schedules stage iterations to threads in round-robin order. The evaluation shows that chunking may improve the performance of PS-DSWP for some benchmarks, by eliminating false-sharing caused by round-robin scheduling. Vachharajani et al. [32] present another extension of DSWP, called SpecDSWP, which handles loops that are not handled by DSWP. In particular, SpecDSWP speculates highly predictable data dependences by using control and silent store speculation. Zhang et al. [33] present an inter-core communication mechanism for low overhead and a mechanism to perform chunking that makes techniques that are similar to DSWP efficient for commodity multi-core processors.

In addition, Kamruzzaman et al. [12] present LBPP, a technique with compiler support that assigns whole loop iterations to threads and uses token-based synchronization to ensure the sequential execution of dependent stage iterations. The evaluation shows that chunking is the key to reduce the synchronization overhead and make LBPP efficient on commodity systems. Lucas and Araujo [14] present a characterization and quantitative analysis of the source for the communication overhead. To reduce this overhead, they present BDX, a technique that is similar to LBPP and uses chunking to improve the overall pipeline performance. Similar to LBPP, Campanoni et al. [6] describe HELIX, a system that automatically chooses sequential loops that are suitable for pipelining. HELIX allows arbitrary control flow and performs optimizations to reduce the communication overhead. Nevertheless, none of all these systems handles dynamic linear pipelines.

Navarro et al. [21] present an analytical model for performance estimation of pipeline parallelism based on queueing theory. Mastoras and Gross [16] study the parallelization tradeoffs for

linear pipelines, and they propose directive-based transformations for PS-DSWP and LBPP, which allow a fair evaluation. The directives enable the configuration of the chunk size by the user, and an in-depth evaluation of chunking for the PS-DSWP and LBPP transformations is presented.

Lee et al. [13] describe Piper, a flexible algorithm that supports arbitrarily nested pipeline and fork-join parallelism for dynamic linear pipelines. Piper is a provably asymptotically efficient algorithm that relies on the work-stealing scheduler of Cilk [5, 10] to achieve load-balancing. Mastoras and Gross [17] present URTS, a technique that sacrifices the flexibility of Piper to achieve efficient and scalable execution of static linear pipelines for fine-grained workloads. Nevertheless, URTS maps different stages onto different threads, and a partitioning that leads to load-balancing is not straightforward. Mastoras and Gross [18] describe Pipelite that overcomes this limitation of URTS. Pipelite supports dynamic linear pipelines and achieves load-balancing with an efficient dynamic scheduler that is tailored to dynamic linear pipelines. Although Pipelite outperforms Piper for fine-grained dynamic linear pipelines, it does not support nested parallelism.

All these systems that achieve load-balancing allow a maximum number of K active loop iterations. This feature allows control of memory usage but may limit the pipeline performance. Furthermore, the design of these systems relies on the assumption that the i th loop iteration is always completed before the $(i + 1)$ -th during parallel execution. Therefore, the $(i + K)$ -th loop iteration cannot be executed unless the i th is completed, even if the i th is the only active loop iteration. To overcome this limitation, Mastoras and Gross [19] present Pipelight, which uses a concurrent unbounded data structure that avoids the need for proper configuration. Pipelight relies on the hybrid mapping [15] to simplify the design of an efficient algorithm for dynamic scheduling. The price paid for this flexibility of Pipelight is additional overhead for fine-grained linear pipelines, but the evaluations show that this overhead is successfully amortized with chunking.

This article presents a chunking algorithm for linear pipelines that are determined at run-time. Therefore, the contribution focuses on the design of a runtime system for efficient execution of fine-grained linear pipelines, and the user annotates a linear pipeline with compiler directives to describe the data dependences and to configure the runtime system. Therefore, it is important to automate this process by developing tools that generate the required directives.

Previous research efforts present systems that perform data dependence analysis to automatically parallelize loops that are expressed as linear pipelines and to configure the runtime system. For example, PS-DSWP [26], LBPP [12], and HELIX [6] are systems that automatically choose sequential loops that can be expressed as linear pipelines, and they generate parallel code. Moreover, Huda et al. [11] present a template matching technique that identifies parts of sequential code that can be parallelized, and the evaluation shows that this technique can successfully identify pipeline parallelism. Similarly, del Rio Astorga et al. [9] present a tool that automatically identifies parallel patterns, including the pipeline pattern. In addition, Tang and Gedik [31] present a technique that automatically identifies pipeline parallelism in stream processing applications, and it performs dynamic profiling to find the configuration of the runtime system that leads to the highest performance. Therefore, provided that the data dependence analysis used by these systems can detect the non-trivial dependences of programs used in the evaluation of this article, these systems can be used to automatically generate the required directives for Pipelite and the static scheduler.

Furthermore, Moreno et al. [20] describe DPM that aims at improving the pipeline performance at run-time. DPM maps multiple stages that consist of iterations with relatively short execution time onto a single thread, and it allocates multiple threads for the most time-consuming stages. Similarly, Suleman et al. [30] present FDP, which automatically determines the number of threads allocated per stage at run-time. Both DPM and FDP focus on assigning balanced workload to threads, since the performance is limited by the most time-consuming stage. Nevertheless, they are

vulnerable to varied execution times found in load-imbalanced linear pipelines, since the execution time may differ substantially for different loop iterations.

Finally, Cho et al. [8] present a simple analytical model that determines a proper number of allocated threads for a set of simultaneously running applications, and Cho et al. [7] consider the utilization of both CPU cores and memory controllers to determine a proper number of threads for the configuration of co-located applications. Moreover, DoPE [24] and Parcae [25] are systems that perform automatic configuration of the runtime system for various techniques. Thus, they may be used to automatically configure the runtime system for the transformations considered in this article, e.g., by choosing a proper value for the number of allocated threads or the chunk size.

6 CONCLUSION

Dynamic systems achieve load-balancing and handle a flexible pipeline structure to efficiently execute a class of real applications expressed as dynamic linear pipelines. However, the overhead of dynamic systems is naturally higher than that implied by static systems. In this article, we present an algorithm for chunking that reduces the synchronization and scheduling overhead for dynamic linear pipelines. We have implemented the algorithm in a static scheduler and a dynamic scheduler, and the evaluation shows that chunking makes the overhead of the dynamic scheduler comparable to that of an efficient static scheduler for fine-grained dynamic linear pipelines.

ACKNOWLEDGMENTS

We thank 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 [13], which were also used in the evaluation for this article.

REFERENCES

- [1] 2006. Standard Performance Evaluation Corporation (SPEC). Retrieved from: <https://www.spec.org>.
- [2] 2015. Parallel BZIP2 (PBZIP2) Data Compression Software. Retrieved from: <http://compression.ca/pbzip2>.
- [3] Christian Bienia. 2011. *Benchmarking Modern Multiprocessors*. Ph.D. Dissertation. Princeton University.
- [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] 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.
- [6] 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.
- [7] Younghyun Cho, Camilo A. Celis Guzman, and Bernhard Egger. 2018. Maximizing system utilization via parallelism management for co-located parallel applications. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques (PACT'18)*. Article 14, 14 pages.
- [8] Younghyun Cho, Surim Oh, and Bernhard Egger. 2017. Adaptive space-shared scheduling for shared-memory parallel programs. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*. 158–177.
- [9] David del Rio Astorga, Manuel F. Dolz, Luis Miguel Sánchez, J. Daniel García, Marco Danelutto, and Massimo Torquati. 2018. Finding parallel patterns through static analysis in C++ applications. *Int. J. High Perf. Comput. Appl.* 32, 6 (2018), 779–788.
- [10] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'98)*. 212–223.
- [11] Zia Ul Huda, Ali Jannesari, and Felix Wolf. 2015. Using template matching to infer parallel design patterns. *ACM Trans. Archit. Code Optim.* 11, 4, Article 64 (Jan. 2015), 21 pages.
- [12] 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)*. 1–12.

- [13] I-Ting Angelina Lee, Charles E. Leiserson, Tao B. Schardl, Zhunping Zhang, and Jim Sukha. 2015. On-the-fly pipeline parallelism. *ACM Trans. Parallel Comput.* 2, 3 (2015), 17:1–17:42.
- [14] D. C. S. Lucas and G. Araujo. 2015. The batched DOACROSS loop parallelization algorithm. In *Proceedings of the International Conference on High Performance Computing Simulation*. 476–483.
- [15] 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, 53:1–53:2 pages.
- [16] 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.
- [17] Aristeidis Mastoras and Thomas R. Gross. 2018. Unifying fixed code mapping, communication, synchronization and scheduling algorithms for efficient and scalable loop pipelining. *IEEE Trans. Parallel Distrib. Syst.* 29, 9 (2018), 2136–2149.
- [18] Aristeidis Mastoras and Thomas R. Gross. 2019. Efficient and scalable execution of fine-grained dynamic linear pipelines. *ACM Trans. Archit. Code Optim.* 16, 2 (Apr. 2019), 8:1–8:26.
- [19] Aristeidis Mastoras and Thomas R. Gross. 2019. Load-balancing for load-imbalanced fine-grained linear pipelines. *Parallel Comput.* 85, C (July 2019), 178–189.
- [20] A. Moreno, E. Cesar, A. Guevara, J. Sorribes, and T. Margalef. 2012. Load balancing in homogeneous pipeline based applications. *Parallel Comput.* 38, 3 (Mar. 2012), 125–139.
- [21] 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.
- [22] OpenMP Architecture Review Board. 2015. OpenMP Application Program Interface. Version 4.5. Retrieved October 28, 2019 from <https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf>.
- [23] 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'05)*. 105–118.
- [24] Arun Raman, Hanjun Kim, Taewook Oh, Jae W. Lee, and David I. August. 2011. Parallelism orchestration using DoPE: The degree of parallelism executive. In *Proceedings of the 32nd Conference on Programming Language Design and Implementation (PLDI'11)*. 26–37.
- [25] Arun Raman, Ayal Zaks, Jae W. Lee, and David I. August. 2012. Parcae: A system for flexible parallel execution. In *Proceedings of the 33rd Conference on Programming Language Design and Implementation (PLDI'12)*. 133–144.
- [26] 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.
- [27] Ram Rangan, Neil Vachharajani, Guilherme Ottoni, and David I. August. 2008. Performance scalability of decoupled software pipelining. *ACM Trans. Archit. Code Optim.* 5, 2, Article 8 (Sept. 2008), 25 pages.
- [28] 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.
- [29] 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.
- [30] 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.
- [31] Yuzhe Tang and Bugra Gedik. 2013. Autopipelining for data stream processing. *IEEE Trans. Parallel Distrib. Syst.* 24, 12 (Dec. 2013), 2344–2354.
- [32] 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.
- [33] Yuanming Zhang, Gang Xiao, and Takanobu Baba. 2014. Accelerating sequential programs on commodity multi-core processors. *J. Parallel Distrib. Comput.* 74, 4 (Apr. 2014), 2257–2265.

Received March 2019; revised September 2019; accepted September 2019