Studying the expressiveness and performance of parallelization abstractions for linear pipelines

Aristeidis Mastoras*
Computing Systems Laboratory
Zurich Research Center
Huawei Technologies, Switzerland

Albert-Jan N. Yzelman
Computing Systems Laboratory
Zurich Research Center
Huawei Technologies, Switzerland

Abstract

Semi-automatic parallelization provides abstractions that simplify the programming effort and allow the user to make decisions that cannot be made by tools. However, abstractions for general-purpose systems usually do not carry sufficient knowledge about the structure of the program, and thus parallelization with them may lead to poor performance.

In this paper, we present a popular class of programs, called linear pipelines, that cannot be easily and efficiently parallelized with general-purpose abstractions. We discuss the difficulties and inefficiencies of parallelizing linear pipelines with general-purpose abstractions, and we explain how pattern-specific abstractions overcome these problems. We present the properties of linear pipelines that should be described with pattern-specific abstractions and how these properties are exploited by the state of the art. In addition, we discuss the importance of exposing the performance parameters and how they are combined by pattern-specific knowledge. We claim that designing pattern-specific abstractions for general-purpose programming models is one way to simplify parallel programming and improve performance without sacrificing any expressive power. Consequently, we propose possible pattern-specific extensions to general-purpose parallel programming models, e.g., OpenMP, to support easy and efficient parallelization of linear pipelines.

CCS Concepts • **Software and its engineering → Parallel programming languages**; **Concurrent programming languages**; **Runtime environments**;

Keywords parallel programming models, parallelization abstractions, pattern-specific knowledge, performance parameters, linear pipelines

*Part of this work was done while Aristeidis Mastoras was with ETH Zurich.

PMAM'23, February 25–March 1 2023, Montreal, QC, Canada
© 2023 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *The 14th International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM'23), February 25–March 1 2023, Montreal, QC, Canada*, https://doi.org/10.1145/3582514.3582522.

ACM Reference Format:
Aristeidis Mastoras and Albert-Jan N. Yzelman. 2023. Studying the expressiveness and performance of parallelization abstractions for linear pipelines. In *The 14th International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM'23), February 25–March 1 2023, Montreal, QC, Canada.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3582514.3582522

1 Introduction

Explicit parallel programming using threads [25, 48] has proven to be a tedious task even for experts, and researchers have focused on developing tools for automatic parallelization [1, 6, 8, 14]. However, there exist two main challenges on automatic parallelization: data dependence analysis [38] and runtime system configuration, which may either prohibit parallel execution or may lead to poor performance.

Data dependence analysis [2, 41, 51] identifies opportunities for parallel execution. Once this knowledge is available, a compiler can use it to automatically generate parallel code. Nevertheless, if automatic parallelization fails to identify this knowledge, no runtime system can achieve good performance. Moreover, runtime system configuration [12, 13, 42, 43] determines proper values for all performance parameters. For example, the number of allocated threads, the granularity of tasks, and partitioning a program into a number of tasks are common performance parameters for the execution of a parallel program. Thus, it is necessary to ensure that the provided knowledge about data dependences and runtime system configuration can lead to good performance. Automatic discovery of this knowledge remains one of the main challenges of parallel programming.

One way to overcome this limitation is by introducing an intermediate layer of abstractions that decouples automatic parallelization from the generation of parallel code, and it is often necessary that the user provides all the knowledge for efficient parallel execution. Therefore, we should follow a different approach based on semi-automatic parallelization as used by general-purpose systems such as OpenMP [39], Cilk [7, 17], and Intel Threading Building Blocks (TBB) [22]. While all these systems provide parallelization abstractions, there exist two main issues. First, in the absence of data dependences, writing parallel programs that express both data and task parallelism may be straightforward. However, parallelizing programs with data dependences is non-trivial

even for experts, as it is usually necessary to significantly rewrite or at least annotate the code in various places. Second, parallelization with general-purpose abstractions does not usually lead to good performance due to the lack of sufficient knowledge about the structure of the program.

In automatic parallelization, the knowledge collected from program analysis can be used by the compiler to generate optimized parallel code. However, this is not necessarily the case when parallelization is done using abstractions as they may fail to express the available knowledge. Thus, it is important that parallelization abstractions can express all the available knowledge about the structure of the program.

Linear pipelines [10, 26, 31, 44] are a large class of programs that cannot be easily and efficiently parallelized with general-purpose abstractions. One reason for poor performance may be the implementation of a general-purpose system. We argue instead that the main reason, and the focus of this paper, is the lack of pattern-specific abstractions in general-purpose systems. For example, if OpenMP is extended with pattern-specific directives for linear pipelines, the compiler can decide whether it is necessary a) to generate code that is usually generated based on general-purpose abstractions, or b) to generate optimized code based on pattern-specific abstractions. Thus, the main reason that general-purpose systems perform poorly for some programs is due to design decisions and not due to a poor implementation.

To simplify parallel programming and enable good performance, we should design pattern-specific parallelization abstractions based on the four following principles:

i. *Pattern-specificity.* The semantics is clear as the abstractions are intended to handle a specific class of programs. This restriction allows the compiler to prevent the introduction of both functional and performance bugs.
ii. *Flexibility.* Programs with the same structure have their own performance parameters. Abstractions expose these parameters and allow the user to make decisions, e.g., choosing the scheduling policy and the number of threads.
iii. *Conciseness.* The user inserts as few annotations as possible and does not rewrite the sequential code. Low-level details of parallel programming, e.g., synchronization algorithm, are hidden, since they are always handled in the same way for a specific class of programs.
iv. *PI − Annotations → SI.* If we remove the annotations from the Parallel Implementation (PI), the remaining code is a Sequential Implementation (SI) with the same semantics. This property makes it easier to reason about the parallel code, since the parallel code looks sequential.

This paper presents the state-of-the-art techniques and parallelization abstractions for linear pipelines. The discussion leads to new insights by presenting the key ideas on the design of existing pattern-specific abstractions. To summarize, this paper makes the following contributions:

- it compares pattern-specific with general-purpose abstractions for parallelization of linear pipelines;
- it explains why general-purpose abstractions are error-prone and cannot enable good performance;
- it shows that pattern-specific abstractions do not automatically lead to good performance even though the necessary knowledge is available;
- it discusses how pattern-specific abstractions simplify parallelization and express knowledge about the structure of linear pipelines;
- it proposes pattern-specific extensions for easy and efficient execution of linear pipelines by general-purpose parallel programming models; and
- it emphasizes on the importance of exposing the performance parameters.

2 Motivation and problem statement

Linear pipelines [10, 24, 26, 31, 44] are a large class of loops with cross-iteration dependences that benefit from pipeline parallelism and are ubiquitous in streaming applications, e.g., compression algorithms. A linear pipeline is a loop that can be partitioned into a sequence of stages such that stages do not generate data for earlier stages. Statements of the same loop iteration are executed sequentially due to data dependences, and pipeline parallelism arises by overlapping the execution of different loop iterations.

```
1 void ferret() {
2   int i = -1;
3   while (load(img[++i])) {
4     #pragma oss task in(img[i]) out(seg[i])
5     seg[i] = t_seg(img[i]);
6     #pragma oss task in(seg[i]) out(extr[i])
7     extr[i] = t_extr(seg[i]);
8     #pragma oss task in(extr[i]) out(vec[i])
9     vec[i] = t_vec(extr[i]);
10    #pragma oss task in(vec[i]) out(rank[i])
11    rank[i] = t_rank(vec[i]);
12    #pragma oss task in(rank[i]) out(outstr)
13    t_out(rank[i], outstr);
14  }
15  #pragma oss taskwait
16 }
```

Figure 1. Parallel OmpSs implementation for the ferret benchmark distributed with PARSEC [3, 4]. The implementation is based on that presented by Chasapis et al. [11].

Figure 1 shows a parallel implementation for the linear pipeline of ferret [3, 4]. All variables are global, unless they are declared within the `ferret` function. In line 13, the function `t_out` writes to the shared variable `outstr`. Parallelization of a linear pipeline with OpenMP [39], OpenStream [40], and OmpSs [16] is similar, since all these models rely on similar compiler directives. We discuss parallelization of ferret using OmpSs as an example.

Parallelization with the OmpSs general-purpose directives demands creation of tasks that are synchronized according to data-flow relationships. The user annotates every stage with a directive for the creation of tasks and describes how the data flow from one task to the other. Tasks are created while iterations are executed sequentially, and once the execution is completed, explicit synchronization is enforced with `taskwait` in line 15. However, there exist two main issues with such a parallel implementation for linear pipelines.

First, the user describes data-flow relationships between the stages of a linear pipeline. This step requires understanding how the data flow from one stage to the other. However, statements of the same loop iteration are executed sequentially for linear pipelines, and thus, ideally, data-flow relationships within the same loop iteration should not be described. Moreover, none of OmpSs, OpenMP and OpenStream provides abstractions for parallelization of while loops. Parallelization is performed by following a task-based approach that requires combination of various directives and explicit synchronization. Second, the performance may be poor for fine-grained tasks as the runtime system does not exploit any knowledge about linear pipelines. In particular:

- The runtime system is not aware of the loop structure nor of the user's intention to execute a linear pipeline. Multiple tasks are scheduled for the different stages of an iteration, but the runtime system does not know a priori that these tasks cannot be executed in parallel.
- The sequential execution of tasks is ensured with data-flow relationships. Monitoring data dependences leads to additional overhead.
- By spawning tasks for each stage, it is necessary to move data between the different tasks of a loop iteration. Instead, multiple stages should be handled as a single task that represents a whole loop iteration.
- There is no distinction between stages with and without cross-iteration dependences, thus making it impossible to select an optimized schedule based on the structure of data dependences in linear pipelines.

3 Pattern-specific parallelization abstractions for linear pipelines

Various pipelining techniques [10, 24, 26, 31, 44] have been proposed and evaluated using more than 20 programs collected from different popular benchmark suites [24, 32, 34, 44, 47]. Hence, earlier work shows that linear pipelines are sufficiently important to have their own pattern-specific abstractions. Existing general-purpose programming models, e.g., OpenMP, can be extended by adopting pattern-specific abstractions for parallelization of linear pipelines.

3.1 Structure of linear pipelines

All stages of a loop iteration are executed sequentially due to data dependences, and overlapped execution of different iterations results in pipeline parallelism. In the absence of data dependences, it is possible to execute a stage for different iterations in parallel. However, cross-iteration dependences exist between every pair of successive loop iterations.

Every data dependence of a stage implies a scheduling constraint, since the scheduling algorithm must ensure that all required data are available before the execution of it. Thus, the performance depends on the number of data dependences. Reducing the number of data dependences is impossible, unless we rewrite the program. The key idea is to improve the performance of the parallel execution by reducing the number of scheduling constraints.

Linear pipelines are modeled as loops with cross-iteration dependences that fulfill two conditions:

- statements are executed sequentially within an iteration due to data dependences, and
- data dependences exist between successive iterations.

Most of the state-of-the-art techniques [10, 24, 26, 31, 44] are optimized for overlapped execution of loop iterations that fulfill these two conditions. This is a rational design decision as most pipelined programs lead to a linear sequence of stages. For example, linear pipelines are common in streaming applications, where every iteration reads data from an input stream, processes the data, and then writes the processed data to an output stream. Raman et al. [44] present PS-DSWP that allows the execution of statements of the same loop iteration to overlap in time, but such a non-linear communication scheme may degrade the overall performance [45].

3.2 Scheduling constraints

Provided that the two above conditions are fulfilled, data dependences become unimportant. Although data dependences imply scheduling constraints, some of them can be lifted as they are satisfied by other scheduling constraints. In contrast to general-purpose abstractions of programming models that express data-flow relationships, pattern-specific abstractions for linear pipelines express scheduling constraints. This situation simplifies the programming effort and improves the performance as the important knowledge is passed directly to the runtime system. The performance does not depend on whether the compiler can automatically discover the minimum number of scheduling constraints or not. The user describes all the required knowledge, and the compiler simply generates efficient parallel code.

The execution of every stage is determined by either one or two scheduling constraints. One constraint always enforces the sequential execution of all stages within a loop iteration, and the other constraint determines whether a stage can be executed in parallel with the same stage of the previous loop iteration or not. Figure 2 shows a Directed Acyclic Graph (DAG) representation of a linear pipeline that is partitioned into three stages. The arrows indicate scheduling constraints introduced by data dependences. We notice

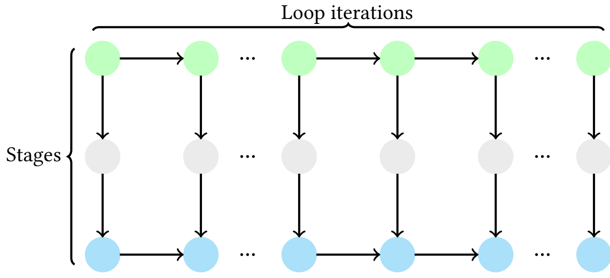


Figure 2. DAG representation for a linear pipeline of three stages. The figure is based on a figure presented in [27].

that stages with different colors are executed sequentially within a loop iteration, but the execution of stages with different colors may overlap in time for different loop iterations. Parallel execution of stages with gray color is possible, but parallel execution of stages with green and blue color is not possible due to scheduling constraints. Therefore, an optimized scheduling algorithm ignores the data dependences and makes decisions based on scheduling constraints.

Figure 3 presents three possible scenarios to show that the number of scheduling constraints cannot be more than two for the execution of each stage. Figure 3a shows that a data dependence from the green to the blue stage does not introduce any scheduling constraint, since the two constraints shown with magenta ellipses already satisfy the scheduling constraint of this additional data dependence. Figure 3b presents a cross-iteration dependence between the green stages of non-successive iterations, and this data dependence does not imply any additional constraint either. Figure 3c shows that a cross-iteration dependence between the green and the gray stage does not introduce any scheduling constraint, since the constraint implied by this data dependence is already satisfied as shown with the magenta ellipses.

This simple model bounds the number of scheduling constraints, but it does not enable all possible parallelization opportunities that may be found in loops with cross-iteration dependences. For example, if a loop does not include data dependences between all successive iterations or includes statements of the same iteration that can be executed in parallel, the loop can be handled as a linear pipeline but perhaps not as efficiently as possible. The simplicity of the model does not reduce applicability; it may degrade the performance.

Although there exist loops with data dependences that cannot be captured precisely by the simple model for linear pipelines, it is unclear whether a more complex and precise model would lead to better performance. The overhead for expression of additional parallelization opportunities may negate any benefit from parallel execution. For example, statements of a loop iteration are always executed sequentially for linear pipelines, and the parallel execution of these statements would not necessarily lead to better performance, e.g., as shown in [45]. Therefore, a simple model that is designed for efficient execution of linear pipelines may lead to good performance even for different classes of loops.

3.3 Parallelization with Cilk

Lee et al. [26] propose an innovative pattern-specific structure to simplify parallelization of linear pipelines with Cilk. Figure 4 shows how to use the `pipe_while` structure, which allows overlapped execution of different loop iterations for the parallelization of ferret. Data flow within a loop iteration is not described since all statements are executed sequentially. Data flow across different iterations is not described either, since Cilk expresses scheduling constraints instead of data-flow relationships expressed by task-based systems. In line 10, `pipe_stage_wait` indicates the beginning of a stage with a scheduling constraint, i.e., this stage is executed after the completion of the same stage for the previous iteration. In line 5, `pipe_stage` indicates the beginning of a stage without a scheduling constraint.

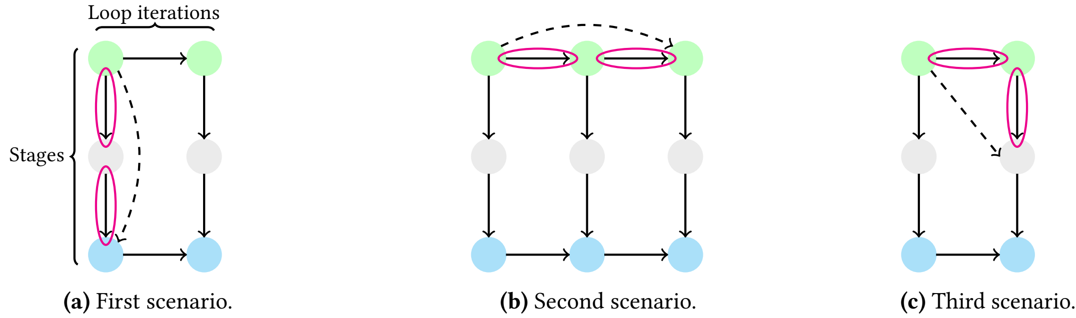
Although the pattern-specific annotations of Cilk carry knowledge that enables the design of an optimized runtime system, this knowledge is not fully exploited by the runtime system. In particular, Mastoras and Gross [34] show that the work-stealing scheduling algorithm is not suitable for the execution of linear pipelines, since it does not exploit the structure of data dependences in linear pipelines, and the pattern-specific `pipe_while` structure achieves poor performance for fine-grained linear pipelines. This situation confirms that the design of suitable parallelization abstractions does not necessarily imply efficient execution; an efficient scheduling algorithm is necessary to exploit the pattern-specific knowledge described in the abstractions.

3.4 Parallelization with Proteas

Proteas [27, 31] aims at simplifying parallelization of linear pipelines by providing pattern-specific compiler directives. The syntax of the directives is similar to that of the OpenMP [39] directives as shown in Figure 5.

Proteas directives are designed to simplify parallelization and avoid some common functional bugs by performing static analysis, but it is still the user's responsibility to annotate the code with suitable directives. Proteas directives are similar to the annotations proposed by Lee et al. [26], since both abstractions are pattern-specific and express scheduling constraints. Proteas is designed to avoid rewriting code, protect the user from functional and performance bugs, and provide additional flexibility [31], as follows.

Figure 5 presents the linear pipeline of ferret annotated with directives. Proteas supports six transformations for parallelization of linear pipelines according to different pipelining techniques. The selected technique is a parameter indicated by `transf-name`, and the existence of scheduling constraints is determined by `stage-type`. Figure 5 shows that the user has the flexibility to select proper values



(a) First scenario. (b) Second scenario. (c) Third scenario.

Figure 3. Possible scenarios of data dependences for linear pipelines that do not imply additional scheduling constraints.

```
1 void ferret() {
2   int ii = -1;
3   pipe_while (load(img[++ii])) {
4     int i = ii;
5     pipe_stage;
6     seg[i] = t_seg(img[i]);
7     extr[i] = t_extr(seg[i]);
8     vec[i] = t_vec(extr[i]);
9     rank[i] = t_rank(vec[i]);
10    pipe_stage_wait;
11    t_out(rank[i], outstr);
12  }
13 }
```

Figure 4. Parallel Cilk implementation for the ferret benchmark distributed with PARSEC [3, 4].

```
1 void ferret() {
2   #pragma proteas transf-name [perf-param-list] \
3       shared(outstr) private(img, seg, extr, vec, rank)
4   while (load(img)) {
5     // no scheduling constraint
6     #pragma proteas transf-name stage-type
7     seg = t_seg(img);
8     extr = t_extr(seg);
9     vec = t_vec(extr);
10    rank = t_rank(vec);
11    // scheduling constraint
12    #pragma proteas transf-name stage-type
13    t_out(rank, outstr);
14  }
15 }
```

Figure 5. Parallel Proteas implementation for the ferret benchmark distributed with PARSEC [3, 4].

for all performance parameters, e.g., number of threads and chunk size, with the optional `perf-param-list`. Thus, the design of Proteas shows that it is possible to provide directives that are the same for different pipelining techniques [27, 30, 32–35], since all low-level details, such as discussed around Figure 1, are hidden from the user.

Table 1 reports the transformations of Proteas: `psdswp`, `urts`, `lbpp`, `pipelight`, `static`, and `pipelite`. The first four transformations handle static linear pipelines, and the type of stages is either parallel or sequential, depending on whether it can be executed in parallel or not for different loop iterations. For dynamic linear pipelines, stages and their scheduling constraints are determined at run-time. The type independent implies no scheduling constraint, and dependent is used when execution is possible only after the completion of the same stage for the previous iteration. The types sequential/parallel determine the type of a stage for all iterations at compile-time, and dependent/independent determine the type of a stage for a single iteration at run-time. The types dependent/independent may have an argument that defines an identifier for the created stage and enables dynamic linear pipelines that allow synchronization of stages that are seemingly different [34].

All transformations are accompanied by performance parameters. The first parameter determines the number of threads used for parallel execution of a linear pipeline, but the semantics of it may differ for different transformations as the total number of created threads depends on the way of mapping stages onto threads. The chunk size has the same semantics for all transformations and determines the number of successive loop iterations that are grouped together into a single task to increase the granularity and reduce the overhead. The third parameter bounds memory usage by limiting the maximum number of active tasks. Its semantics may differ for different transformations, since the queue size for the `psdswp` transformation refers to local data structures per thread, whereas both the buffer size and the throttling limit refer to the size of a shared data structure. A detailed description of these performance parameters and their default values can be found in [31–35].

4 Exploitation of pattern-specific knowledge for linear pipelines

Pipelining techniques differ in the way stages are mapped onto threads. The two main mapping schemes [5] are the *fixed data* mapping that assigns a generic role to threads and the *fixed code* mapping that assigns a specific role to threads. A combination of these two schemes is the *hybrid* mapping [30]. The way stages are mapped onto threads is often determined at compile-time, and there exist systems

Transformation	Supported types of stages	Threads	Chunk size	Active tasks
`psdswp`	`sequential` / `parallel`	`num_threads(T)`	`chunk_size(C)`	`queue_size(Q)`
`urts`	`sequential` / `parallel`	`num_threads(T)`	`chunk_size(C)`	`buffer_size(K)`
`lbpp`	`sequential` / `parallel`	`num_threads(T)`	`chunk_size(C)`	n/a
`pipelight`	`sequential` / `parallel`	`num_threads(T)`	`chunk_size(C)`	n/a
`static`	`dependent[(s)]` / `independent[(s)]`	`num_threads(T)`	`chunk_size(C)`	n/a
`pipelite`	`dependent[(s)]` / `independent[(s)]`	`num_threads(T)`	`chunk_size(C)`	`throttling(K)`

Table 1. Pattern-specific compiler directives for the six transformations of PROTEAS and their performance parameters.

that perform dynamic mapping. Most pipelining techniques handle a static pipeline structure, i.e., stages are known at compile-time, and there exist systems that handle linear pipelines with a dynamic structure.

Technique	Mapping	Structure	Scheduling policy
LBPP [24]	fixed data	static	round robin
HELIX [10]	fixed data	static	round robin
Static [33]	fixed data	dynamic	round robin
PS-DSWP [44]	fixed code	static	round robin
URTS [32]	fixed code	static	dynamic
Pipelight [35]	hybrid	static	dynamic
Piper [26]	dynamic	dynamic	work stealing
Pipelite [33, 34]	dynamic	dynamic	dynamic

Table 2. Pipelining technique specifications.

Table 2 presents an overview of the state of the art on linear pipelining. Figure 6 illustrates the sequential execution of a linear pipeline partitioned into four stages and its parallel execution according to the different mapping schemes. The arrows between stages indicate scheduling constraints, i.e., the source stage is executed before the target stage.

PROTEAS [27, 31] provides directive-based transformations for parallelization according to the state of the art shown in Table 2, except for HELIX [10] and Piper [26]. In the rest, we discuss the design of these techniques and explain how their design decisions exploit the pattern-specific knowledge of the PROTEAS directives. We provide new insights that are the key ideas on the design of both pattern-specific parallelization abstractions and efficient runtime systems.

LBPP [24] automatically parallelizes linear pipelines and exploits the knowledge about scheduling constraints between all stages of a loop iteration by assigning their execution to the same thread as shown with the fixed data mapping in Figure 6. Data of iterations are stored locally, and inter-thread communication of shared data is achieved via shared memory. Since scheduling constraints are expressed between successive iterations, earlier iterations are executed first, by scheduling iterations in round robin order. Synchronization for the sequential execution of a stage with scheduling constraints is achieved with a token, which is passed from one thread directly to the thread that executes the next iteration.

PS-DSWP [44] automatically parallelizes linear pipelines and assigns a specific role to threads according to the fixed code mapping shown in Figure 6. Threads execute only a

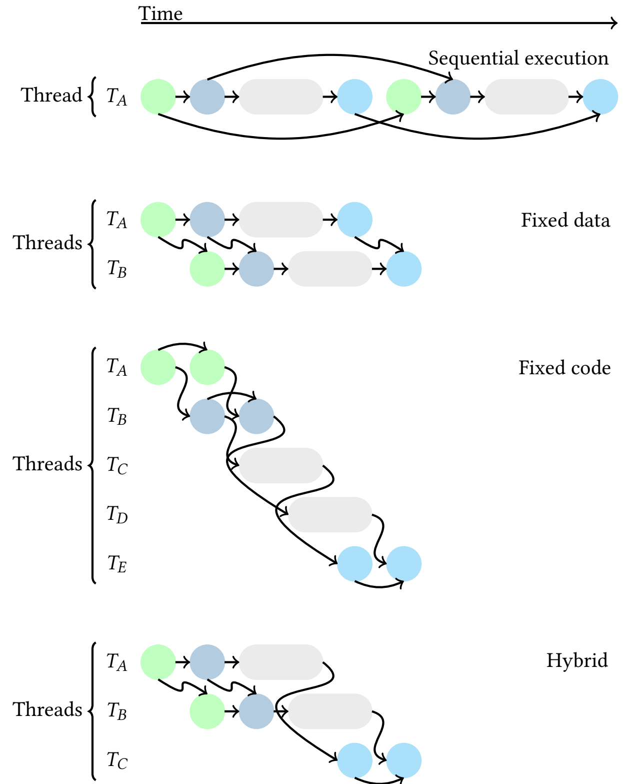


Figure 6. Sequential and parallel execution of the first two iterations for a four-stage linear pipeline using different mapping schemes. The third stage does not imply any scheduling constraint. The figure is based on a figure presented in [27].

single stage but for all iterations. Communication and synchronization between threads that execute different stages is achieved via queues. Stages with scheduling constraints are executed by a single thread, and stages without a scheduling constraint may be executed by multiple threads. To avoid contention on shared queues, PS-DSWP maintains a private input/output queue for each thread. All queues have a single enqueuer and a single dequeuer, which enables the design of an efficient wait-free queue. Earlier iterations are executed first, and the data of iterations are enqueued to the input queues of multiple threads in round robin order.

URTS [32] is an extension of PS-DSWP that combines the fixed code mapping with a suitable concurrent data structure to enable efficient dynamic scheduling. URTS avoids any synchronization overhead for the execution of stages with a scheduling constraint, since all iterations are executed by the same thread. A single bounded data structure is shared by all threads and consists of memory slots that store data required for the execution of an iteration. Threads have read and write access to these slots, and communication is implicitly achieved through shared memory; data do not move between different slots. Access to the data structure is given based on tickets issued when a new iteration is scheduled. Thus, contention on the shared data structure is avoided since different threads have access to different memory slots. Dynamic scheduling is performed based on the tickets, which determine the order in which iterations are executed, such that earlier iterations are executed first. Synchronization between threads that execute different stages of the same iteration is achieved with a token passed from one thread directly to the thread that executes the next stage.

Pipelight [35] relies on the hybrid mapping, which decouples a stage without a scheduling constraint from a stage with a scheduling constraint as shown in Figure 6. These two stages are executed by different threads to enable efficient dynamic scheduling. Most stages of an iteration are executed by the same thread, and most of the time communication is achieved via shared memory. Dynamic scheduling is performed by issuing tickets that determine the execution order of iterations. For stages with a scheduling constraint, ticket locks ensure synchronization of iterations executed by different threads. Communication and synchronization between decoupled stages is achieved with a data structure that exists only between successive stages, since stages of a loop iteration are always executed sequentially.

Pipelite threads execute loop iterations either to completion or until they encounter a stage with a scheduling constraint that is not satisfied. Thus, *Pipelite* avoids any communication overhead from transferring data between different threads. There exist two types of scheduled tasks. These are new tasks that represent loop iterations that have not been scheduled before and ready tasks for previously suspended iterations that are now ready for execution. Communication is achieved with a data structure that is shared by all threads and stores data required for the execution of each task, i.e., a group of successive loop iterations. Threads have read and write access to the data of each task, and if a task is executed by a single thread, then the data of the task are exclusively accessed by this thread. Moreover, synchronization occurs only between pairs of tasks for successive loop iterations, and each task maintains a single variable that shows its progress.

For the execution of each stage, there may exist only one scheduling constraint introduced by the previous loop iteration. The design of *Pipelite* exploits this knowledge about the structure of data dependences, and each thread ensures that the scheduling constraint is satisfied by checking the progress of a specific task. The first stage of a linear pipeline is always executed sequentially for all loop iterations due to data dependences in the loop condition. A ticket counter dynamically issues tickets that determine the order in which tasks are executed, and earlier iterations are scheduled first to avoid suspension of later iterations. The scheduler maintains a ready queue that stores ready tasks and always tries to schedule a task from the ready queue before it schedules a new task using the ticket counter. In this way, *Pipelite* exploits the pattern-specific knowledge about data dependences in linear pipelines and always tries to execute earlier loop iterations to satisfy the scheduling constraints.

The completion of a stage may enable the execution of another task that is currently suspended. Threads check whether they enable the execution of another task, and if this is the case, they enqueue this task in the ready queue. The design of *Pipelite* exploits the knowledge about scheduling constraints that are expressed only between successive loop iterations, and thus threads check only if they enable the task for the next loop iteration.

Suspension of a task requires storing the data and the suspension point such that the execution can be continued later from the same point. A trivial solution to suspend and continue tasks is by using setjmp and longjmp buffers [46]. However, *Pipelite* introduces the local suspension mechanism that exploits knowledge about the structure of linear pipelines to perform efficient suspension and continuation of tasks. Data of all active tasks are always stored in a shared data structure, and thus threads do not need to store and load data to suspend and continue a task, respectively. Furthermore, the suspension point of a task is not any arbitrary point; suspension may occur only before the execution of a stage with a scheduling constraint, and the beginning of such a stage is indicated with the directives of Proteas. Consequently, each task maintains a scalar variable that shows the suspension point and is updated before a thread checks whether a scheduling constraint is satisfied or not.

5 Related work

HELIX [10] automatically parallelizes linear pipelines and assigns whole loop iterations to threads in round robin. Sequential execution of a stage is achieved with signal-based synchronization, where a thread receives signals from the thread that executes the previous iteration. Each thread has read access to its own memory buffer and write access to the buffer of the thread that executes the next iteration. HELIX reduces the overhead by removing redundant signals and introduces helper threads to prefetch signals.

Intel Threading Building Blocks (TBB) [22] supports parallelization of linear pipelines with the pipeline class, and

Vandierendonck et al. [50] present a parallelization abstraction of queues, called hyperqueues, that allows the expression of deterministic and scale-free pipeline parallelism. However, both Intel TBB and hyperqueues demand significant rewriting of the code to express a linear pipeline.

Griebler et al. [19] study the expressiveness and performance of parallelization abstractions for stream processing, by using programs that can be expressed as linear pipelines, e.g., ferret from the PARSEC [3, 4] benchmark suite. However, abstractions for stream processing express data-flow relationships as discussed in Section 2, and thus they do not benefit from pattern-specific knowledge for linear pipelines. Hoffmann et al. [20] use SPar [18], a domain-specific language for stream processing, to automatically generate parallel code for OpenMP [39] by using a source-to-source compiler. In this way, the parallelization process is simplified, but any pattern-specific knowledge about linear pipelines that is not expressed by the OpenMP directives remains unexploited.

Huda et al. [21] present a template matching technique that identifies pipeline parallelism, and Astorga et al. [15] present a tool that performs static analysis to automatically identify parallel patterns, e.g., pipeline parallelism. Tang and Gedik [49] propose a technique that automatically identifies pipeline parallelism in streaming applications. DPM [37] and FDP [47] aim at improving the pipeline performance at run-time by allocating a number of threads that improves the performance. DoPE [42] and Parcae [43] perform configuration of the runtime system for various techniques. Thus, earlier work may be used to automatically identify pipeline parallelism and perform runtime system configuration, and then to generate the directives required by Proteas.

6 Evaluation

The focus of this paper is the design of parallelization abstractions for linear pipelines. The evaluation provides a qualitative comparison of the design decisions for pattern-specific and general-purpose abstractions. Moreover, it provides an overview for the performance of the considered pipelining techniques according to the evaluation of earlier work.

6.1 Abstractions

Proteas directives are in line with the four principles presented in Section 1. They are pattern-specific as they can be used only for linear pipelines. They offer flexibility since the same directives can be used for one of the six techniques presented in Table 1, and they expose the performance parameters. Proteas directives are concise as the pipelining technique is a parameter, and all the low-level details of the selected pipelining technique, e.g., communication, synchronization, and scheduling algorithm, are hidden. Proteas makes the parallel code look sequential and makes it easier for the user to reason about the correctness of the parallel code. The parallel implementation shown in Figure 5 is essentially the sequential code annotated with directives.

Comparing Figure 1 and Figure 5, the Proteas implementation is much simpler than that of OmpSs, which requires expression of data-flow relationships and explicit synchronization. The OmpSs implementation demands code rewrites, by introducing an array for all private data. Comparing with the Cilk implementation in Figure 4, Cilk does not require an array of private data as they can be declared locally within the loop body. However, in this case, private data cannot be used in the loop condition, i.e., the invocation of the function `load` has to move from the condition to the loop body.

Pattern-specific abstractions can improve performance as they enable the design of an efficient runtime system and provide flexibility by exposing the performance parameters. Although performance parameters may be considered as low-level details, we claim that exposing the performance parameters is better than hiding them, making performance tuning possible without significant code rewrites.

To illustrate, we consider chunking, a mechanism that groups together successive iterations of a loop into the same task to increase the granularity and reduce the overhead. The `for` directive of OpenMP [39] and the `cilk_for` structure of Intel Cilk Plus [23] allow the user to select a proper chunk size, i.e., the number of loop iterations that are grouped together. Nevertheless, choosing the chunk size for a linear pipeline is impossible when parallelization is performed with task-based systems, e.g., OpenMP, OpenStream, and OmpSs. None of these systems are aware of the loop; to implement chunking, the code has to be rewritten by the user. Cilk provides the `pipe_while` structure for parallelization of linear pipelines based on Piper [26], but Cilk does not allow the user to select the chunk size. Although the design of a chunking algorithm for static linear pipelines is trivial, Piper supports dynamic linear pipelines, for which stages and their scheduling constraints are determined at run-time. The lack of an explicit performance parameter that determines the chunk size is not just an implementation issue; it is a design limitation of Piper. Proteas [31] supports *Pipelite* and *Static* that handle dynamic linear pipelines and performs chunking based on a novel algorithm [33], thus exposing the chunk size parameter for both static and dynamic linear pipelines.

6.2 Performance

The state of the art includes techniques that explore the design space, by relying on different mapping schemes, data structures, mechanisms, and algorithms, and they support linear pipelines with static and dynamic structure. The design decisions are very different, and different techniques perform well for different situations. Exhaustive evaluation of Proteas and Piper is presented in [26, 27, 31–35], and it shows that the performance depends mainly on the scheduling policy and the way stages are mapped onto threads.

The round-robin scheduler of LBPP, PS-DSWP, and *Static* implies low overhead and performs well for fine-grained linear pipelines. However, it does not achieve load balancing for load-imbalanced linear pipelines, i.e., when iterations differ substantially in execution time. The work-stealing scheduler of Piper implies higher overhead than the dynamic schedulers of Proteas, and Piper performs worse than URTS, *Pipelight*, and *Pipelite* for fine-grained linear pipelines.

Pipelining techniques that rely on the fixed data mapping, e.g., LBPP, simplify partitioning and inherently achieve load balancing for load-balanced linear pipelines, since all threads execute the same code. However, LBPP does not achieve load balancing for load-imbalanced linear pipelines. Threads that cannot execute a stage due to a scheduling constraint have to wait until the constraint is satisfied. Thus, the number of active iterations cannot exceed the number of threads. PS-DSWP and URTS rely on the fixed code mapping that decouples stages, and threads buffer data of suspended iterations into a data structure. The performance depends on the size of the data structure that determines the maximum number of active iterations. The drawback of the fixed code mapping is that load balancing is not inherently achieved as threads have a specific role and execute different stages. The performance is limited by the most time-consuming stage, and thus partitioning should lead to load-balanced stages. The hybrid mapping aims at simplifying partitioning and load balancing by decoupling only those stages that are necessary to perform dynamic scheduling.

Finally, the evaluation of earlier work [27, 31, 32, 34] shows that Piper [26], which is designed especially for linear pipelines, does not lead to good performance for fine-grained linear pipelines, since Piper does not exploit as much knowledge as the runtime system of Proteas. Thus, it is expected that a more general-purpose runtime system, e.g., OpenMP [39] implies even higher overhead than Piper.

7 Conclusion

Linear pipelines are an important class of programs that cannot be easily and efficiently parallelized with general-purpose abstractions. This is certainly not the only one; recursive functions are another class of programs that benefit from pattern-specific abstractions [36]. We should identify more classes of programs, design pattern-specific abstractions, and integrate them with general-purpose programming models. A comparison of OpenMP directives with those of Proteas shows that Proteas simplifies programming and improves performance, without sacrificing the expressive power of OpenMP. That is, it is possible to adopt pattern-specific directives within general-purpose programming models, thus benefiting of both specificity as well as genericity.

This paper focuses on the design of suitable pattern-specific abstractions that are sufficiently expressive to enable good performance. The design of suitable abstractions can be seen as the first step towards automatic parallelization, as abstractions define an interface between the user and the runtime system. Future work may design algorithms for more precise data dependence analysis, and then some or all of the necessary knowledge required in the directives can automatically be provided by tools. Moreover, robust tools for proper runtime system configuration may be developed by building cost models that estimate the performance of the program. For example, automatic selection for the number of threads and the chunk size for nonblocking execution [28, 29] in GraphBLAS [9, 52] leads to good performance that is very close to that achieved by the best manual configuration.

References

[1] Saman P. Amarasinghe, Jennifer M. Anderson, Monica S. Lam, and Chau-Wen Tseng. 1995. An Overview of the SUIF Compiler for Scalable Parallel Machines. In *SIAM Conference on Parallel Processing for Scientific Computing*.

[2] Utpal K. Banerjee. 1988. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers.

[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 *Proc. of the 17th International Conference on Parallel Architectures and Compilation Techniques.* 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.* 161–171.

[6] William Blume, Rudolf Eigenmann, Jay Hoeflinger, David Padua, Paul Petersen, Lawrence Rauchwerger, and Peng Tu. 1994. Automatic Detection of Parallelism: A Grand Challenge for High-Performance Computing. *IEEE Parallel Distrib. Technol.* 2, 3 (1994), 37–47.

[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 *Proc. of the 5th Symposium on Principles and Practice of Parallel Programming*. 207–216.

[8] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *Proc. of the 29th Conference on Programming Language Design and Implementation.* 101–113.

[9] Aydın Buluç, Timothy Mattson, Scott McMillan, Jose Moreira, and Carl Yang. 2017. The GraphBLAS C API Specification, version 1.0. http://www.graphblas.org

[10] 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 *Proc. of the 10th International Symposium on Code Generation and Optimization.* 84–93.

[11] Dimitrios Chasapis, Marc Casas, Miquel Moretó, Raul Vidal, Eduard Ayguadé, Jesús Labarta, and Mateo Valero. 2015. PARSECSs: Evaluating the Impact of Task Parallelism in the PARSEC Benchmark Suite. *ACM Trans. Archit. Code Optim.* 12, 4 (2015), 41:1–41:22.

[12] Younghyun Cho, Camilo A. Celis Guzman, and Bernhard Egger. 2018. Maximizing System Utilization via Parallelism Management for Co-located Parallel Applications. In *Proc. of the 27th International Conference on Parallel Architectures and Compilation Techniques.* 14:1–14:14.

[13] Younghyun Cho, Surim Oh, and Bernhard Egger. 2017. Adaptive Space-Shared Scheduling for Shared-Memory Parallel Programs. In

Job Scheduling Strategies for Parallel Processing. 158–177.

[14] C. Dave, H. Bae, S. Min, S. Lee, R. Eigenmann, and S. Midkiff. 2009. Cetus: A Source-to-Source Compiler Infrastructure for Multicores. *Computer* 42, 12 (2009), 36–42.

[15] David del Rio Astorga, Manuel F Dolz, Luis Miguel SÃąnchez, J Daniel GarcÃŋa, Marco Danelutto, and Massimo Torquati. 2018. Finding parallel patterns through static analysis in C++ applications. *Int. J. High Perform. Comput. Appl.* 32, 6 (2018), 779–788.

[16] 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 Process. Lett.* 21, 02 (2011), 173–193.

[17] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The Implementation of the Cilk-5 Multithreaded Language. In *Proc. of the 1998 Conference on Programming Language Design and Implementation*. 212–223.

[18] Dalvan Griebler, Marco Danelutto, Massimo Torquati, and Luiz Gustavo Fernandes. 2017. SPar: A DSL for High-Level and Productive Stream Parallelism. *Parallel Process. Lett.* 27 (2017), 1740005:1–1740005:20.

[19] Dalvan Griebler, Renato B. Hoffmann, Marco Danelutto, and Luiz Gustavo Fernandes. 2018. High-Level and Productive Stream Parallelism for Dedup, Ferret, and Bzip2. *Int. J. Parallel Program.* 47 (2018), 253–271.

[20] Renato B. Hoffmann, Júnior Löff, Dalvan Griebler, and Luiz G. Fernandes. 2022. OpenMP as Runtime for Providing High-Level Stream Parallelism on Multi-Cores. *J. Supercomput.* 78, 6 (2022), 7655–7676.

[21] Zia Ul Huda, Ali Jannesari, and Felix Wolf. 2015. Using Template Matching to Infer Parallel Design Patterns. *ACM Trans. Archit. Code Optim.* 11, 4 (2015), 64:1–64:21.

[22] Intel. 2012. Threading Building Blocks Reference Manual.

[23] Intel. 2013. Intel Cilk Plus Language Extension Specification. Version 1.2.

[24] Md Kamruzzaman, Steven Swanson, and Dean M. Tullsen. 2013. Load-balanced Pipeline Parallelism. In *Proc. of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.

[25] Edward A. Lee. 2006. The Problem with Threads. *Computer* 39, 5 (2006), 33–42.

[26] 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.

[27] Aristeidis Mastoras. 2019. *Efficient Execution of Linear Pipelines*. Ph. D. Dissertation. ETH Zurich.

[28] Aristeidis Mastoras, Sotiris Anagnostidis, and Albert-Jan N. Yzelman. 2022. Design and Implementation for Nonblocking Execution in GraphBLAS: Tradeoffs and Performance. *ACM Trans. Archit. Code Optim.* 20, 1 (2022), 6:1–6:23.

[29] Aristeidis Mastoras, Sotiris Anagnostidis, and Albert-Jan N. Yzelman. 2022. Nonblocking execution in GraphBLAS. In *Proc. of the 2022 IEEE International Parallel and Distributed Processing Symposium Workshops*. 230–233.

[30] Aristeidis Mastoras and Thomas R. Gross. 2016. Unifying Fixed Code and Fixed Data Mapping of Load-Imbalanced Pipelined Loops. In *Proc. of the 21st Symposium on Principles and Practice of Parallel Programming*. 53:1–53:2.

[31] Aristeidis Mastoras and Thomas R. Gross. 2018. Understanding Parallelization Tradeoffs for Linear Pipelines. In *Proc. of the 9th International Workshop on Programming Models and Applications for Multicores and Manycores*. 1–10.

[32] 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.

[33] Aristeidis Mastoras and Thomas R. Gross. 2019. Chunking for Dynamic Linear Pipelines. *ACM Trans. Archit. Code Optim.* 16, 4 (2019), 44:1–44:25.

[34] Aristeidis Mastoras and Thomas R. Gross. 2019. Efficient and Scalable Execution of Fine-Grained Dynamic Linear Pipelines. *ACM Trans. Archit. Code Optim.* 16, 2 (2019), 8:1–8:26.

[35] Aristeidis Mastoras and Thomas R. Gross. 2019. Load-Balancing for Load-Imbalanced Fine-Grained Linear Pipelines. *Parallel Comput.* 85, C (2019), 178–189.

[36] Aristeidis Mastoras and George Manis. 2015. Ariadne – Directive-Based Parallelism Extraction from Recursive Functions. *J. Parallel Distrib. Comput.* 86, C (2015), 16–28.

[37] A. Moreno, E. Cesar, A. Guevara, J. Sorribes, and T. Margalef. 2012. Load Balancing in Homogeneous Pipeline Based Applications. *Parallel Comput.* 38, 3 (2012), 125–139.

[38] Niall Murphy, Timothy Jones, Robert Mullins, and Simone Campanoni. 2016. Performance Implications of Transient Loop-Carried Data Dependences in Automatically Parallelized Loops. In *Proc. of the 25th International Conference on Compiler Construction*. 23–33.

[39] OpenMP Architecture Review Board. 2018. OpenMP Application Program Interface. Version 5.0.

[40] Antoniu Pop and Albert Cohen. 2013. OpenStream: Expressiveness and Data-flow Compilation of OpenMP Streaming Programs. *ACM Trans. Archit. Code Optim.* 9, 4 (2013), 53:1–53:25.

[41] William Pugh. 1991. The Omega Test: A Fast and Practical Integer Programming Algorithm for Dependence Analysis. In *Proc. of the 1991 Conference on Supercomputing*. 4–13.

[42] Arun Raman, Hanjun Kim, Taewook Oh, Jae W. Lee, and David I. August. 2011. Parallelism Orchestration Using DoPE: The Degree of Parallelism Executive. In *Proc. of the 32nd Conference on Programming Language Design and Implementation*. 26–37.

[43] Arun Raman, Ayal Zaks, Jae W. Lee, and David I. August. 2012. Parcae: A System for Flexible Parallel Execution. In *Proc. of the 33rd Conference on Programming Language Design and Implementation*. 133–144.

[44] Easwaran Raman, Guilherme Ottoni, Arun Raman, Matthew J. Bridges, and David I. August. 2008. Parallel-stage Decoupled Software Pipelining. In *Proc. of the 6th International Symposium on Code Generation and Optimization*. 114–123.

[45] Ram Rangan, Neil Vachharajani, Guilherme Ottoni, and David I. August. 2008. Performance Scalability of Decoupled Software Pipelining. *ACM Trans. Archit. Code Optim.* 5, 2 (2008), 8:1–8:25.

[46] Jim Sukha. 2013. Piper: Experimental Support for Parallel Pipelines in Intel Cilk Plus. https://www.cilkplus.org/sites/default/files/experimental-software/PiperReferenceGuideV1.0_0.pdf Reference Guide, Version 1.0, Intel.

[47] M. Aater Suleman, Moinuddin K. Qureshi, Khubaib, and Yale N. Patt. 2010. Feedback-directed Pipeline Parallelism. In *Proc. of the 19th International Conference on Parallel Architectures and Compilation Techniques*. 147–156.

[48] Herb Sutter and James Larus. 2005. Software and the Concurrency Revolution. *Queue* 3, 7 (2005), 54–62.

[49] Yuzhe Tang and Bugra Gedik. 2013. Autopipelining for Data Stream Processing. *IEEE Trans. Parallel Distrib. Syst.* 24, 12 (2013), 2344–2354.

[50] Hans Vandierendonck, Kallia Chronaki, and Dimitrios S. Nikolopoulos. 2013. Deterministic Scale-free Pipeline Parallelism with Hyperqueues. In *Proc. of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 32:1–32:12.

[51] M. Wolfe and C. W. Tseng. 1992. The Power Test for Data Dependence. *IEEE Trans. Parallel Distrib. Syst.* 3, 5 (1992), 591–601.

[52] A. N. Yzelman, D. Di Nardo, J. M. Nash, and W. J. Suijlen. 2020. A C++ GraphBLAS: specification, implementation, parallelisation, and evaluation. (2020). http://albert-jan.yzelman.net/PDFs/yzelman20.pdf Preprint.