Report Number:

2019-09-03

Pipelined Parallelism in a Work-Stealing Scheduler

Authors: Thomas Kelly

A pipeline is a particular type of parallel program structure, often used to represent loops with cross-iteration dependencies. Pipelines cannot be expressed with the typical parallel language constructs offered by most environments. Therefore, in order to run pipelines, it is necessary to write a parallel language and scheduler with specialized support for them. Some such schedulers are written exclusively for pipelines and unable to run any other type of program, which allows for certain optimizations that take advantage of the pipeline structure. Other schedulers implement support for pipelines on top of a general-purpose scheduling algorithm. One example of such an algorithm is "work stealing," a paradigm used by many popular parallel environments. The benefit of this approach is that it allows the user to arbitrarily combine pipeline and non-pipeline structures in their programs. This article is concerned with implementing support for pipelines in a work-stealing scheduler, and then attempting to optimize the scheduler in such a way that takes advantage of the pipeline structure, despite running in a general-purpose environment.
... **Read complete abstract on page 2.**

Pipelined Parallelism in a Work-Stealing Scheduler

Complete Abstract:

A pipeline is a particular type of parallel program structure, often used to represent loops with cross-iteration dependencies. Pipelines cannot be expressed with the typical parallel language constructs offered by most environments. Therefore, in order to run pipelines, it is necessary to write a parallel language and scheduler with specialized support for them. Some such schedulers are written exclusively for pipelines and unable to run any other type of program, which allows for certain optimizations that take advantage of the pipeline structure. Other schedulers implement support for pipelines on top of a general-purpose scheduling algorithm. One example of such an algorithm is "work stealing," a paradigm used by many popular parallel environments. The benefit of this approach is that it allows the user to arbitrarily combine pipeline and non-pipeline structures in their programs. This article is concerned with implementing support for pipelines in a work-stealing scheduler, and then attempting to optimize the scheduler in such a way that takes advantage of the pipeline structure, despite running in a general-purpose environment.

Pipelined Parallelism in a Work-Stealing Scheduler

Thomas Kelly
Computer Science and Engineering
Washington University in St. Louis

Abstract

A pipeline is a particular type of parallel program structure, often used to represent loops with cross-iteration dependencies. Pipelines cannot be expressed with the typical parallel language constructs offered by most environments. Therefore, in order to run pipelines, it is necessary to write a parallel language and scheduler with specialized support for them. Some such schedulers are written exclusively for pipelines and unable to run any other type of program, which allows for certain optimizations that take advantage of the pipeline structure. Other schedulers implement support for pipelines on top of a general-purpose scheduling algorithm. One example of such an algorithm is "work stealing," a paradigm used by many popular parallel environments. The benefit of this approach is that it allows the user to arbitrarily combine pipeline and non-pipeline structures in their programs. This article is concerned with implementing support for pipelines in a work-stealing scheduler, and then attempting to optimize the scheduler in such a way that takes advantage of the pipeline structure, despite running in a general-purpose environment.

1 Introduction

A common problem in parallel computing is scheduling parallel loops with cross-iteration dependencies. One example of such a program is Dedup, a file compressor from the PARSEC benchmark suite [1]. Algorithm 1 shows pseudocode for the program. The loop exhibits cross-iteration dependencies in that WRITE_TO_FILE (line 12) needs to be called on each file chunk, excluding some, in the same order as returned by GET_NEXT_CHUNK (line 3).

The majority of the work in each iteration is done by the COMPRESS method in line 10, which is unique in that it does not depend on (the same part of) any previous iteration. Therefore: we see potential for parallelism in Dedup, but it is necessary to describe a parallel structure in which certain segments of each iteration must be executed sequentially, while other segments can be executed in parallel. Such as a structure is henceforth referred to as a *pipeline*.

Algorithm 1 Dedup pseudocode

```
 1: procedure DEDUP
 2:     while !done do
 3:         chunk ← GET_NEXT_CHUNK                    ▷ Stage 0
 4:         if chunk = NULL then
 5:             done ← true
 6:         else
 7:             chunk.is_dup ← DEDUPLICATE(chunk)
 8:                                                   ▷ Stage 1
 9:             if !chunk.is_dup then
10:                 COMPRESS(chunk)                   ▷ Stage 2
11:             end if
12:             WRITE_TO_FILE(chunk)                  ▷ Stage 3
13:         end if
14:     end while
15: end procedure
```

1.1 The Pipeline DAG

It is common practice to describe a parallel program using a Directed Acyclic Graph, or DAG. Each node on the DAG represents a point in the control flow of the program. An edge from one node to another represents a dependency between those points in the control flow: that is, the point corresponding to the first node must be executed logically before the second.

In order to draw a DAG for Dedup, or any other pipeline, we first break up the loop into *stages*. The goal is to separate the parts of each iteration that must be executed sequentially from those that can be executed in parallel. Algorithm 1 breaks Dedup down into such stages. The only stage which can be executed in parallel in this case is Stage 2. We henceforth refer to this type of stage as a *parallel stage*, and other stages as *sequential stages*.

We draw the DAG in a grid shape, where each column represents a single iteration of the loop, and each row represents a stage. Every node (except for the bottom in each column) has an edge to the node directly below it. This is because the different stages within a single iteration must be executed sequentially. Nodes may or may not have an edge to the node on their right, depending on whether they belong to a sequential or parallel stage. Figure 1 shows the DAG for Dedup.

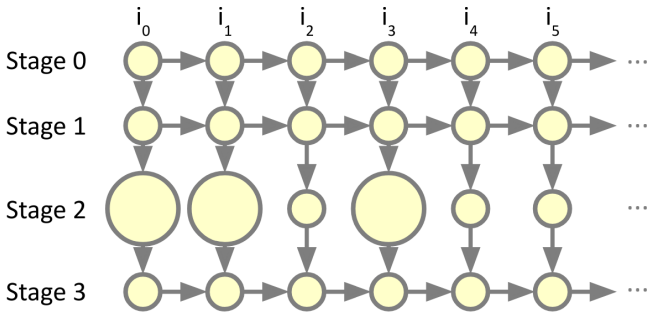

Figure 1: DAG representation of Dedup

1.2 Pipeline Syntax

Most parallel environments offer a unique syntax for defining a program's parallel structure. Often, this will be some variation of the "fork-join" paradigm. Halpern describes fork-join parallelism which would be implemented in Intel's Cilk Plus environment as the keywords "cilk_spawn" and "cilk_sync" [2] [5].

Unfortunately, it is not possible to describe a Pipeline DAG with the typical fork-join syntax. For this reason: pipelines cannot be executed in fork-join parallel environments, unless those environments offer specialized support for pipelines. In order to support pipelines, we first need a syntax which can describe them. The environment implemented in this article happens to use a syntax identical to that of Intel's Piper environment by Lee et al., described here for completeness [7].

Algorithm 2 Pipelined Dedup pseudocode

```
1:  procedure DEDUP
2:      pipe_while !done do
3:          chunk ← GET_NEXT_CHUNK
4:          if chunk = NULL then
5:              done ← true
6:          else
7:              STAGE_WAIT(1)
8:              chunk.is_dup ← DEDUPLICATE(chunk)
9:              STAGE(2)
10:             if !chunk.is_dup then
11:                 COMPRESS(chunk)
12:             end if
13:             STAGE_WAIT(3)
14:             WRITE_TO_FILE(chunk)
15:         end if
16:     end pipe_while
17: end procedure
```

The first construct offered is **pipe_while**. Syntactically, **pipe_while** is very similar to a standard while loop. It takes some conditional expression, and then offers a body for more code. The code inside of the **pipe_while** body represents a complete iteration of the pipeline, and it will keep executing iterations until the conditional expression evaluates to false.

All code inside of a **pipe_while** loop belongs to a stage, which necessarily has a stage number, and must be either parallel or sequential as previously explained. The evaluation of the loop condition, along with all code until the first STAGE or STAGE_WAIT statement, implicitly belongs to stage 0. As an implementation choice: stage 0 is implicitly a *sequential* stage.

The next constructs are STAGE and STAGE_WAIT. Both of these take an integer value as argument. It is required that the argument passed to these constructs always be greater than 0, and always be greater than the value passed in the last STAGE or STAGE_WAIT call. Any code *after* a call to STAGE or STAGE_WAIT, up until the next call, belongs to the stage whose number was passed as argument. A call to STAGE_WAIT indicates that the following code, up until the next STAGE or STAGE_WAIT, is a sequential stage. Likewise, a call to STAGE indicates a parallel stage.

As another implementation detail: just before the end of the loop body, there is an implicit sequential stage of an arbitrarily high number. This, combined with the fact that stage 0 must be sequential, guarantees that every iteration starts and ends on a sequential stage. In other words: no iteration can start or terminate before the previous iteration does so.

1.3 Existing Environments

A syntax is only one of two necessary components in a parallel environment; the other is a runtime. That is, the code which actually handles scheduling across multiple cores. There does already exist a number of environments that support pipelines, each with their own syntax and scheduling style. Examples include:

- Intel's Piper. Piper is an extension of Intel's Cilk Plus environment, which is based on the scheduling method work-stealing [6]. Piper is a general-purpose environment which, in addition to the pipeline syntax described here, also supports the standard fork-join paradigm. The benefit of this approach is that it allows users to arbitrarily combine pipelined and non-pipelined structure in their programs [7].

- Intel's Threading Building Blocks (TBB). TBB is another environment by Intel based on work-stealing, which natively supports pipelines as of a recent version. Like Piper, it is a general-purpose environment and allows its users to arbitrarily combine different DAG structures. However, unlike Piper, it does not support "on-the-fly" pipelines. What this means is that, in Piper, the number of stages in an iteration

can be determined at runtime, while in TBB the exact structure of the pipeline must be specified ahead-of-time [4].

- URTS. Mastoras et al. propose URTS, a unified runtime system for pipelines. URTS is an environment written exclusively for pipelines, and it cannot execute any other type of parallelism. The benefit of this is that it allows the runtime to take advantage of the pipeline structure in ways that its creators claim cannot be accomplished in a general-purpose runtime. URTS does not support "on-the-fly" pipelines- the number of stages must be known at compile time [8][7].

- Pipelite. Pipelite is another environment from the creators of URTS. Like URTS, it exclusively runs pipelines and is optimized to take advantage of their structure. Pipelite, however, supports on-the-fly pipelines [9].

This project concerns implementing support for pipelines in an existing work-stealing environment. Specifically, we will be using a Cilk-like environment called Cheetah by Lee et al. Cheetah is a work-stealing environment similar in structure to Cilk Plus, but meant to be simpler and lighter-weight. Similarly, we will be adding features to base Cheetah which are intended to be a simpler, lighter-weight version of Piper. Additionally: we will be investigating the claim made by Mastoras et al. that work-stealing schedulers cannot be optimized for pipelines. We will be adding features to Cheetah which are meant to mimic the optimizations in URTS and Pipelite, and then analyze the results. The resulting environment is henceforth referred to as Cheetah-Piper.

2 Work Stealing

We present here a brief overview of the Cheetah work-stealing scheduler. This is prior work, and the summary presented here is not meant to be an extensive report on the runtime, but rather to highlight the points in the runtime which will need to be changed to support pipelines.

2.1 Worker Threads

The core idea behind work-stealing schedulers is to balance work between a fixed number of threads. Usually, the runtime creates as many threads as there are cores available. Applications that do not make use of a fully-fledged parallel environment may instead choose to accomplish their parallelism by manually instantiating pthreads. This approach incurs unnecessary overhead, and it leaves the work of scheduling to the operating system. The worker thread approach saves the user from having to interact with their operating system's thread interface at all.

In Cheetah (and Cilk), each thread maintains a *ready-deque*. That is: a doubly-ended queue which, as its name suggests, contains pieces of user code which are ready to be executed. These pieces of user code are called stacklets- in base Cheetah they contain a logically sequential chain of function calls, starting with a spawned function.

When a worker completes execution of its current stacklet, it pulls a new one off the *bottom* of its own ready-deque. If that worker's readydeque is empty, then the work-stealing aspect of the scheduler comes into play: the worker becomes a thief, and selects a victim (another worker thread) at random. The thief attempts to steal a stacklet from the *top* of the victim's deque [6].

2.2 Closures

In Cheetah, functions can spawn off other functions and then be stolen by another worker, leaving the spawned child behind on the original worker's deque. The original worker still needs to know where to return (or if it is necessary to return) when the child terminates. The data structure that keeps track of this information is called a Closure.

The top stacklet on every worker's deque is automatically wrapped inside of a closure. When a thief steals the top stacklet (and its closure), it wraps the left-behind child in a new closure and stores a link to the stolen closure. In this way, the runtime maintains a tree of closures which stores the parent/child relationships between function calls [6].

2.3 Suspension

Upon a "cilk_sync" keyword, a worker thread must halt execution if any spawned children have not yet returned. This is where suspension comes into play: the worker will store its context into a jump buffer (from the C standard library setjmp) and then remove the running Closure from its own deque. When a Closure is suspended, it is not on any worker's deque, but maintains its parent/child relationships via the Closure tree. When a spawned child returns, its worker will check to see if the parent is suspended. If so, it will check if all of its spawned children have returned (i.e., if this returning function is the last to do so). If this is the case, the worker will install the parent Closure on its own deque and resume its execution [6].

An important note is that, in base Cheetah, suspension *only* happens upon hitting a sync. Furthermore: the runtime relies on the fact that a suspended Closure has spawned children, at least one of which will eventually resume the Closure. These assumptions, however, don't hold up in Cheetah-Piper.

3 Implementing Pipelines

3.1 Pipe Frame and Iteration Frames

The pipe_while loop itself transpiles (via preprocessor macros) into code which does not look syntactically different from ordinary Cilk Plus.

Algorithm 3 Simplified transpiled pipe_while loop

```
1: procedure TRANSPILED PIPE WHILE
2:     while !done do
3:         CILK_SPAWN(iter_func)
4:         THROTTLE
5:     end while
6: end procedure
```

Algorithm 3 demonstrates a simplified version of what the loop body might look like with a standard cilk_spawn construct. A serial while loop executes with the condition supplied by the user, and at each iteration uses cilk_spawn to spawn off a new instance of the iteration body (the iteration body is stored and executed as a C++ lambda). Given the way that Cheetah works, there exists a stack frame and (eventually) Closure for this while loop, which is henceforth referred to as the *Pipe Control Frame* or PCF. Likewise, there are frames corresponding to the iterations themselves which, in Cilk Plus terms, are *spawned children* of the PCF.

Before spawning each iteration, the loop must pass a "throttle" method. This will be explained in detail below.

Where this differs from base Cheetah is not in the syntax but in the runtime: the PCF is stored in a special data structure distinct from the ordinary Cheetah stack frame. Among other pertinent data such as the loop's current iteration and condition, the PCF is linked to a fixed-size (more on this below) buffer of iteration frames. Iteration frames contain data pertinent to a particular iteration, namely: a *stage counter* for that iteration, and links to the iteration frames on the "left" and "right" (before and after the current iteration).

3.2 Throttling

An important aspect of pipeline scheduling not yet discussed is throttling. Spawning new iteration introduces overhead, and so we wish to prevent "runaway" pipelines. That is: a pipeline which repeatedly spawns off new iterations before executing the existing ones. This is possible if, say, the new iterations repeatedly suspend (more on this below) on stage 0 and the PCF keeps being stolen by new workers. To combat this possibility: Cheetah-Piper restricts the allowed number of spawned iterations to a user-specified parameter.

Because the number of iterations is fixed, we can store our iteration frames in a circular buffer of a fixed size. Before spawning off a new iteration, the PCF must pass the throttle method, which checks if the frame for the current iteration is currently in use. If so, the PCF is suspended.

Suspending the PCF is very similar to suspending a Closure for sync in base Cheetah; we know that the PCF has at least one active child, by virtue of the fact that there is a currently executing iteration. For this reason, we can rely on that child and Cheetah's existing resumption logic to eventually resume the PCF.

3.3 Suspending Iterations

As discussed: the iteration body is spawned just as any other function frame. The iteration itself will therefore execute sequentially, as written. The downward edges in the Pipeline DAG are handled implicitly for this reason.

The only effect of a call to STAGE is to update the current iteration's stage counter to the supplied value (there is no need to check any other iteration's current stage when about to execute a parallel stage). STAGE_WAIT, on the other hand, is where the real scheduling logic occurs. Upon a call STAGE_WAIT(j) on iteration i, the worker will check if the left $(i-1)$ iteration's stage counter is *greater than* j. This implies that iteration $i-1$ has completed the sequential stage j and iteration i should be free to advance.

If this is not the case, then iteration i needs to be suspended. This presents a problem: the assumptions that base Cheetah relies on no longer apply. An iteration does *not* necessarily have any spawned children, so the existing Cheetah logic is not guaranteed to resume the suspended iteration.

Piper experiences the same problem. Its solution is to have a *returning* iteration– i.e. one that has gotten past its final (implicit) STAGE_WAIT– check if the *next* iteration is suspended, and if so resumes it. Doing so guarantees that any suspended iteration will eventually be resumed: if the earlier iteration reaches this checking point first, the later one won't suspend because all stages in the earlier iteration have completed. If the later iteration suspends first, we know the earlier iteration must still be executing and will eventually resume the later one [7].

Cheetah-Piper's solution is similar, but intended to better take advantage of the pipeline structure.

4 Optimizing for Pipelines

URTS and Pipelite are environments that exclusively run pipelines, and meant to take advantage of their structure in ways that general-purpose environments cannot. The structure of these runtimes is very different from that of Piper or Cheetah-Piper, and it is difficult to precisely port these optimizations. Roughly, they can be summarized as follows [8] [9]:

- Prioritize scheduling earlier iterations. If a thread is in need of work, and there are suspended iterations, attempt to resume the earliest one.
- Synchronize using atomic operations, to reduce data contention.

4.1 Suspended Iteration List

The second optimization is possible for URTS and Pipelite because worker threads can always depend on the existence of a pipeline. These runtimes keep an atomic "ticket counter" in a location accessible by all workers, and assign new iteration numbers according to its value. By nature of the work-stealing environment, though, we want our workers to remain naive to parallel structure of the pipeline except as needed [8] [9].

The first optimization, however, is possible within this framework. Our goal here is to write a concurrent data structure which can store suspended iterations, and allows a worker to inexpensively retrieve the earliest one.

If we require that the earliest iteration be retrievable in $O(1)$ time, it becomes necessary that we store some kind of reference to the current earliest iteration, and that retrieving the earliest iteration provides us with a reference for the *next* suspended iteration. Maintaining such a chain means having to perform $O(n)$ worst-case work somewhere in the runtime, where n is the iteration buffer size (we could potentially use a concurrent priority queue to reduce this to $\log(n)$, but such an approach seems overly complex for this application). This is acceptable, though, if we keep the $O(n)$ work to the suspending worker. This is in accordance with the *work-first principle*. We prioritize speed in retrieving the earliest suspended iteration because it is the more frequently executed task. Suspension is already a fairly costly operation by nature of the setjmp library.

To accomplish these speed goals, we store the suspended iterations in what is effectively a concurrent linked list, but with one caveat. There is a field in each iteration frame that stores the *iteration number* of the next suspended iteration (this field's value has no meaning if its containing iteration frame is not suspended) rather than a pointer. We access the appropriate iteration frame by taking the modulus of this number with the buffer size, and using the result as index. This approach has the benefit of locality, since all of our elements are contiguous in memory, while still providing constant-time access to our earliest iteration.

The suspended linked list is concurrent and non-blocking, its synchronization mechanisms based on atomic Compare-and-Swap (CAS) primitives. Specifically, we use the design proposed by Harris [3].

With this linked list, a worker can retrieve the earliest suspended iteration in constant time via a "head" pointer in the PCF. In order to insert a suspended iteration, a worker traverses the list until finding the appropriate place based on iteration number.

4.2 Optimization Points

Broadly speaking, almost any inefficiency in a work-stealing scheduler comes from "idle" time. This is time that a worker spends without any work to execute, perhaps due to failing steals. Idle time is essentially the gap between a work-stealing scheduler in practice and the ideal greedy scheduler, which instantaneously executes work as soon as it becomes available.

In order to avoid idle time, a worker should immediately execute any work that it knows to be available. Work-stealing is one of the best possible paradigms when the runtime has no inherent knowledge about the parallelism in the program. However, while we wish to maintain the general-purpose nature of Cheetah, Cheetah-Piper can rely on certain guarantees when executing pipelines.

We've already covered how a worker can retrieve the earliest suspended iteration. For correctness (not necessarily performance), we call this method in the same place where Piper checks the *next* suspended iteration. That is: when an iteration returns. There are other points in the runtime, though, where a worker may find itself without work, and yet it still has access to at least one of the pipeline-related structures. In other words: these are situations where the runtime can depend on the existence of a pipeline, and therefore can take advantage of its structure. Piper actually includes its own optimizations here as well: just like when an iteration is returning, it checks the *next* iteration [7]. But in our case, we dictate that the worker should resume the *earliest* suspended iteration if possible.

This approach is preferable to jumping back into the runtime and relying on stealing to find more work, as the latter is more uncertain and could always lead to idle time. With that in mind: Cheetah-Piper attempts to resume the earliest suspended iteration at all of the following points:

- Upon an iteration returning (this is necessary to guarantee completion of the pipeline)
- Upon an iteration suspending
- Upon trying to steal from another worker, and finding a currently-executing iteration (and no parents on the deque to steal)

At that last point in particular, Cheetah-Piper attempts to optimize in another way: by resuming the PCF if it is suspended (such as for throttling), but only *after* checking the suspended iterations. In this way, we satisfy both the greedy property and the principles of URTS/Pipelite: we resume an early iteration if available,

but otherwise we attempt to continue spawning new iterations. In either case, we avoid idle time by not attempting another steal. The exact control flow in this case is as follows:

- Attempt to steal from a worker currently executing an iteration
- Resume the earliest suspended iteration, unless none are available
- Resume the PCF if suspended

5 Testing

5.1 benchmarks

We present benchmarking results for three different runtimes: Piper, Cheetah-Piper without the pipeline optimizations (except for those also implemented in Piper), and Cheetah-Piper with the pipeline optimizations. We use as benchmarks three different programs, each meant to represent a different use case:

- Dedup. As previously mentioned, Dedup is a file compressor from the PARSEC benchmark suite. Dedup is a "coarse-grained" pipeline in the sense that individual stages are a non-trivial amount of work. Compared to the other two benchmarks utilized here, Dedup has relatively little intrinsic parallelism. The dataset being used with Dedup is the "native" file input provided by PARSEC [1].
- Ferret. Ferret is an image comparison program, also from the PARSEC suite. Ferret is coarse-grained, but has much more intrinsic parallelism than Dedup. Again, the dataset being used is "native" from PARSEC [1].
- Pipe-Fib. Pipe-Fib is a fibonacci number calculator by Lee et al., originally intended as a benchmark for Piper. Pipe-Fib iteratively calculates Fibonacci numbers where each number represents an iteration in the pipeline, and the stages are represented by individual bits of the arithmetic operations. It is therefore a very "fine-grained" pipeline [7].

5.2 Timing

The three runtimes in question are instrumented to record three different types of timing data:

- Working time. This is the time spent on the actual user code within the program. Ideally, the total aggregate work time across all threads should always be constant for a given program (a parallel environment distributes work amongst the worker threads but shouldn't actually be able to change the amount of it). When working time does change, the phenomenon is referred to as "work inflation." Work inflation may be caused by differences in cache locality within the schedulers, any work that the runtime may add within the scope of user code, or simply flaws in the instrumentation design.
- Scheduling time. This is the time spent within the scheduling logic of a runtime. For the three runtimes, this does include time spent in stage waits, throttling, etc.
- Idle time. This time is technically also spent within the scheduling logic, but is meant to demonstrate the time that a worker spends without any work to do. To be precise: this is the total amount of time that workers spend on *unsuccessful* steals. When a steal is successful, that time is instead added to scheduling time.

Model name	Intel Xeon CPU E5-2665 0 @ 2.40 GHz
Cores	16
CPU MHz	2394.286
NUMA nodes	2
Cores per node	8
Threads per core	1

Table 1: Hardware specifications of the machine used for benchmarking

5.3 hardware

Table 1 shows the hardware specifications of the CPU which these tests were conducted on. This model of CPU supports hyperthreading, but it was disabled for these benchmarks.

6 Results

Here we present the results in three forms:

- The processing time (real time) spent by each benchmark on various numbers of cores, Figures 2 through 4.
- The speedup exhibited when increasing the number of cores, Figures 5 through 7. This is based on the same timing data as in the previous type of chart, but here each data point is the time value associated with the serial (one-core) execution, divided by the time value associated with the variable number of cores.

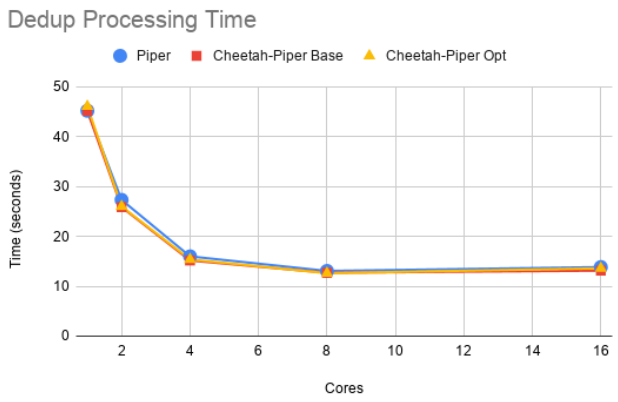

Figure 2: Processing time for the Dedup benchmark

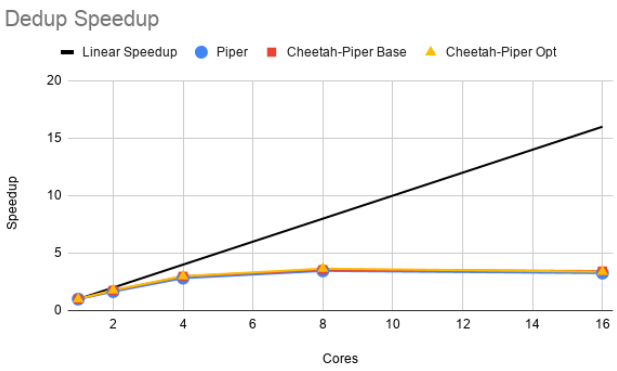

Figure 5: Speedup across different amounts of cores for the Dedup benchmark

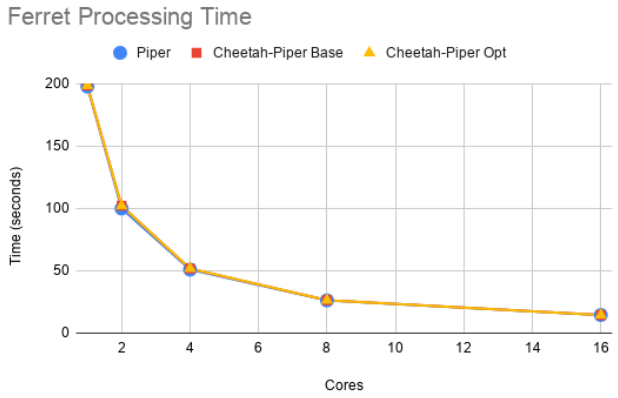

Figure 3: Processing time for the Ferret benchmark

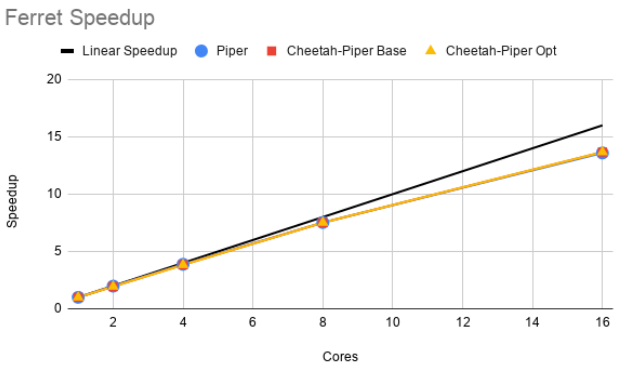

Figure 6: Speedup across different amounts of cores for the Ferret benchmark

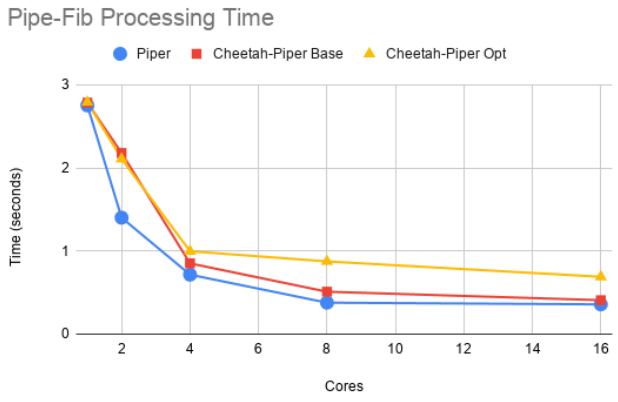

Figure 4: Processing time for the Pipe-Fib benchmark

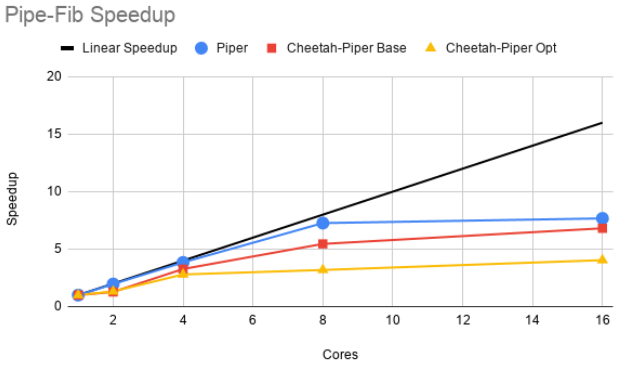

Figure 7: Speedup across different amounts of cores for the Pipe-Fib benchmark

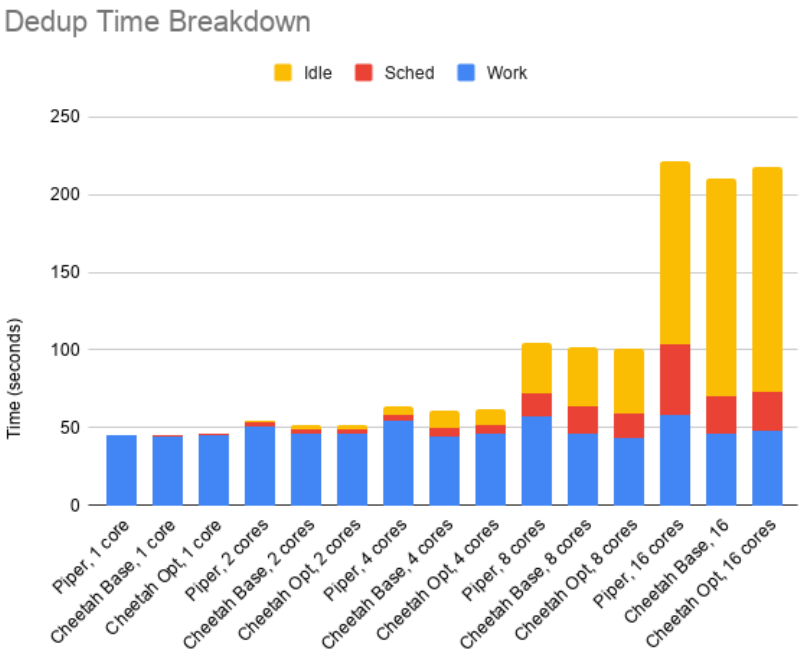

Figure 8: Breakdown of processing time by category for the Dedup benchmark

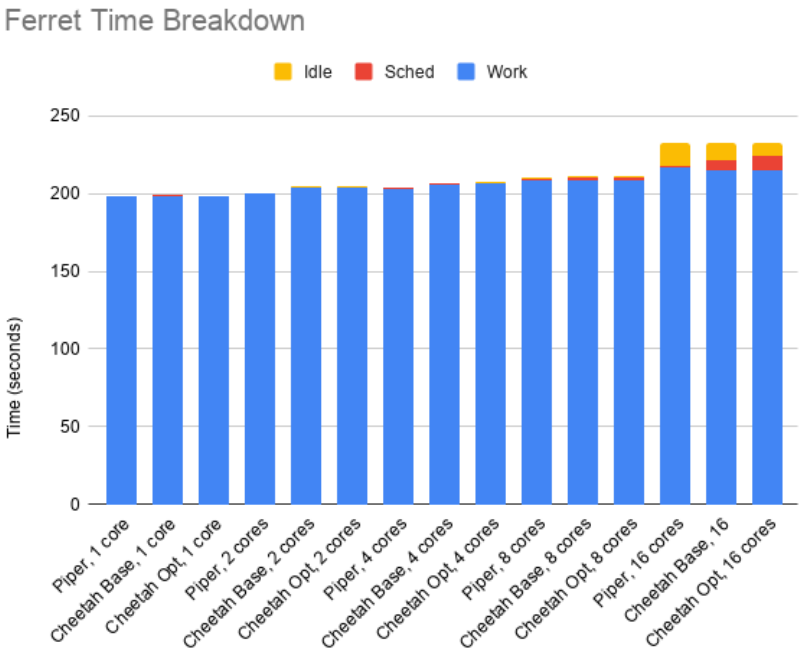

Figure 9: Breakdown of processing time by category for the Ferret benchmark

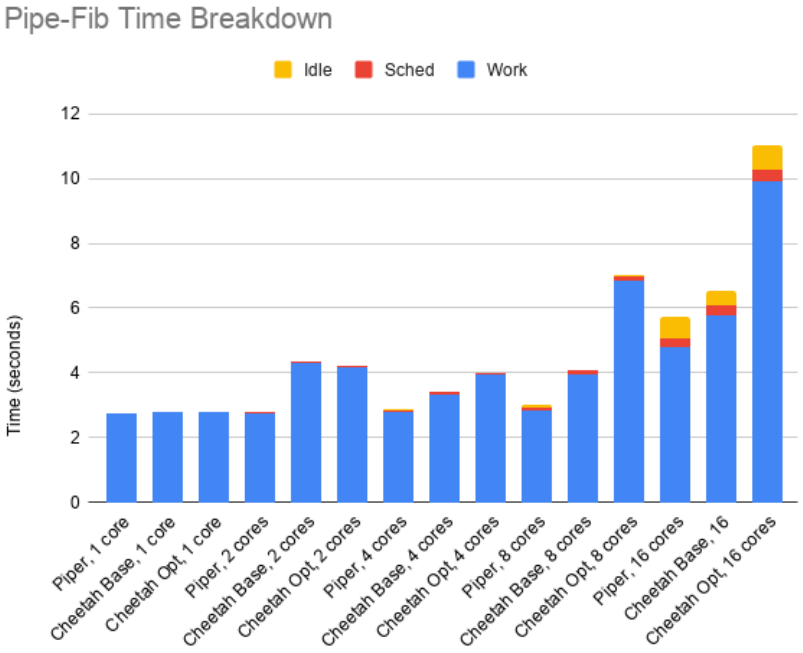

Figure 10: Breakdown of processing time by category for the Pipe-Fib benchmark

- The categorical breakdown of the aggregate time spent by all cores, Figures 8 through 10. As previously discussed, there are three time categories. These charts show the total time spent in each one. Note that, because this is based on agrregate time and not real-time as in the other two chart types, we expect these values to strictly increase with the number of cores. If we were to divide the total height of each bar by the associated number of cores, we would arrive at the same time values used in the other two charts.

All data points shown are averages taken from 5 independent trials.

To make a general statement about the results: it is difcult to discern any difference between the three runtimes, especially in terms of their overall processing times and speedups. Some subtle differences do become apparent in the breakdowns, though.

In dedup's case, all three runtimes appear almost indistinguishable in terms of processing time and speedup. (See Figures 2 and 5.) Unsurprisingly, the speedup falls off below the linear reference in Figure 5 rather quickly. Note, however, the rightmost three columns in Figure 8. While all three are roughly the same height, (as could already be inferred from Figure 2) a smaller portion comes from scheduling time in the case of Cheetah. Since this occurs for both base Cheetah-Piper and optimized Cheetah-Piper, we can't attribute this smaller scheduling time to any of the optimizations. It does give us a good baseline on the difference between Piper's scheduling logic and Cheetah's logic, though. While not conclusive, it supports the notion that Cheetah (not necessarily Cheetah-Piper) is a lighter-weight scheduler. The smaller scheduling time, however, is offset by a larger idle time. Perhaps this owes to an intrinsic lack of parallelism in Dedup- i.e., we are hitting the maximum speedup possible.

The next benchmark is Ferret. Just like in the case of Dedup, the three runtimes perform almost identically in terms of processing time and speedup (see Figures 3 and 6). Figure 6 demonstrates the high amount of intrinsic parallelism in Ferret- the speedup curves stay much closer to the linear reference. This is also evident in Figure 9. Note that a much larger portion of each column's height comes from working time relative to Dedup. Interestingly: our observations from Dedup regarding scheduling time seem to be inverted. Cheetah-Piper exhibits greater scheduling time than Piper, and optimized Cheetah-Piper exhibits greater scheduling time than base Cheetah-Piper. The latter could possibly owe to the cost associated with the suspended linked list operations. While we cannot yet say that the optimizations have improved performance, this does perhaps support the notion that they can reduce idle time. In this case, if such a phenomenon occured it was offset by the cost of scheduling. If Figure 6 is any indication, though, we aren't hitting Fer-

ret's maximum speedup at 16 cores. We could perhaps reduce scheduling time by further performance-engineering the scheduling mechanisms, and idle time wouldn't necessarily increase to meet the change, thus resulting in better overall performance. But again: this is only conjecture. In reality, the implications of these data are inconclusive.

Lastly, we have Pipe-Fib. Here is where real-time performance finally begins to differ, and not in Cheetah-Piper's favor (see Figures 4 and 7): Piper outperforms Cheetah-Piper, and base Cheetah-Piper outperforms optimized Cheetah-Piper, in both processing time and speedup. Furthermore: here is where the effects of work inflation are most prevalent. The working times in Figure 10 vary rather drammatically, making it difficult to draw conclusions regarding the effectiveness of each scheduler. At this time, the only conclusion we can draw from these data is that Cheetah-Piper needs to be further optimized for fine-grained pipelines.

7 Conclusions and Future Work

In its current form, Cheetah-Piper seems to be unable to offer a performance improvement upon Piper. The most promising application is perhaps in coarse-grained pipelines with ample parallelism such as Ferret, but even so, further work is required.

Possible future work includes:

- Investigate concurrent data structures other than the CAS-based linked list for suspended iterations. We could, for example, implement a concurrent priority queue based on iteration number.
- We could follow the lead of URTS and Pipelite and implement an atomic ticket counter system within the PCF, and furthermore restructure Cheetah such that a worker can install a child frame on a parent without actually owning the parent [8] [9]. That way, upon encountering an iteration frame, a worker could begin a new iteration without needing to steal the PCF. This approach would, however, require some major changes to Cheetah.

References

[1] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, Princeton, NJ, USA, 2011. AAI3445564.

[2] Pablo Halpern. Strict fork-join parallelism. Technical report, Intel Corporation, 2012.

[3] Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the 15th International Conference on Distributed Computing*, DISC '01, pages 300–314, London, UK, UK, 2001. Springer-Verlag.

[4] Intel Corporation. *Threading building blocks reference manual*, 2011.

[5] Intel Corporation. *Intel Cilk Plus Language Extension Specification*, 2013.

[6] Balaji V. Iyer, Robert Geva, and Pablo Halpern. Cilk plus in gcc, 2012.

[7] I-Ting Angelina Lee, Charles E. Leiserson, Tao B. Schardl, Jim Sukha, and Zhunping Zhang. On-the-fly pipeline parallelism. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '13, pages 140–151, New York, NY, USA, 2013. ACM.

[8] A. Mastoras and T. R. Gross. Unifying fixed code mapping, communication, synchronization and scheduling algorithms for efficient and scalable loop pipelining. *IEEE Transactions on Parallel and Distributed Systems*, 29(9):2136–2149, Sep. 2018.

[9] Aristeidis Mastoras and Thomas R. Gross. Efficient and scalable execution of fine-grained dynamic linear pipelines. *ACM Trans. Archit. Code Optim.*, 16(2):8:1–8:26, April 2019.