Enabling Real-Time Irregular Data-Flow Pipelines on SIMD Devices

Tom Plano and Jeremy Buhler
{planot,jbuhler}@wustl.edu
Washington University
St. Louis, Missouri, USA

ABSTRACT

Streaming data-flow applications arise in many contexts where each item in a data stream must be processed within a bounded latency, or deadline, following its arrival. We consider applications whose behavior is irregular, in the sense that the application may reduce or amplify data volumes dynamically at various stages of its computation. Our implementation target for these applications is SIMD-capable processors such as GPUs. For such devices, organizing the computation so that a full-width SIMD vector of inputs can be processed at once makes the most efficient use of the processor. However, having parts of the computation wait while full vectors of input accumulate may incur more latency than the application's deadline allows.

We present a novel approach to scheduling irregular streaming applications with latency constraints on SIMD devices. After describing a model for executing such applications, we formalize the objective of efficient processor utilization and the constraints associated with bounded latency and adequate throughput to handle a stream of items arriving at a fixed rate. We introduce a strategy, enforced waiting, to optimize the objective subject to the constraints. We demonstrate empirically that, for a test application from bioinformatics, our strategy can effectively lower processor utilization relative to to a baseline approach that cannot introduce waits inside the application pipeline. Finally, we characterize the region of parameter space in which the new approach is likely to outperform the baseline.

CCS CONCEPTS

• **Computing methodologies** → **Vector / streaming algorithms**; • **Software and its engineering** → **Real-time schedulability**; • **Theory of computation** → **Streaming models**; • **Computer systems organization** → *Pipeline computing*.

KEYWORDS

irregular data-flow, bounded latency, scheduling, GPGPU

ACM Reference Format:
Tom Plano and Jeremy Buhler. 2021. Enabling Real-Time Irregular Data-Flow Pipelines on SIMD Devices. In *Workshop on Scheduling and Resource Management for Parallel and Distributed Systems, August 9, 2021, Chicago, IL.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/1122445.1122456

SRMPDS '21, August 9, 2021, Chicago, IL

ACM ISBN XXXX…$0.00
https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Applications that process streams of data arising from physical processes or instruments often must work under latency constraints. For example, a monitoring program in a chemical plant or a self-driving car might receive periodic readings from one or more sensors and must take action promptly when those readings indicate a problem or change in state. Similarly, an orbiting gamma-ray telescope might process a stream of incoming photons and must alert ground-based instruments when it detects a gamma-ray burst [6]. For such applications, it is desirable to guarantee that with high confidence, the application will finish processing an incoming event within a fixed deadline after its arrival.

Applications that process a data stream through a pipeline of computational stages can be described abstractly by the streaming data-flow (SDF) model [16]. When both the latency and data volume of each stage in a pipeline are fixed *a priori*, it is a simple matter to compute the end-to-end latency of the pipeline and determine whether it can meet a given deadline. However, many streaming applications of interest, including those with latency constraints, are *irregular*: even if the service time of an individual computational stage is fixed, the volume of output generated from a given amount of input to the stage is data-dependent and unpredictable. Irregular streaming applications processing sensor data arise in, e.g., network intrusion detection [22], biological sequence comparison [1], decision cascades in machine learning [26], and the aforementioned gamma-ray burst detection. For such applications, achieving bounded latency requires both careful attention to pipeline design and a model of the application's irregularity.

As the computational demands of streaming applications have grown, these computations have targeted more powerful processors. A common feature of such processors – whether modern general-purpose CPUs, GPUs, or even customized logic in an FPGA – is support for fine-grained parallelism, in particular SIMD operations that can perform the same computation on multiple data items at once. Prior work [9, 11, 21, 24] has described strategies for mapping irregular computations, and particularly streaming computations, to SIMD architectures to maximize their *throughput*; such mappings address the problem of dynamically reallocating work among SIMD lanes to efficiently use the full SIMD width of the processor. However, such mappings pay no attention to latency and cannot in general provide guarantees of meeting end-to-end processing deadlines.

This work explores strategies for mapping latency-sensitive, irregular streaming computations to SIMD processors. We describe

an execution model for streaming applications on such devices and consider how to implement these applications so as to achieve bounded end-to-end latency while efficiently exploiting the SIMD capability of the device. Specifically, we seek to meet a given latency bound while minimizing the fraction of time that the application must utilize the processor, which is tied to its ability to fill all SIMD lanes as it executes. We propose to enforce waiting times at each pipeline stage to accumulate inputs that can occupy more SIMD lanes and describe how to choose these times to minimize processor utilization. Finally, we compare our approach to a simpler, "monolithic" strategy that enforces waiting time only at the beginning of the pipeline. We show that adding waits internal to the pipeline can result in improved utilization compared to the monolithic approach while still incorporating the latency demands of the application.

The remainder of this work is organized as follows. Section 2 describes our application and device models. Section 3 discusses related work. Section 4 describes how we add waits to the pipeline, while Section 5 describes the monolithic strategy. Section 6 describes empirical assessment of the two strategies on a simulation of a pipeline derived from computational biology, and Section 7 concludes and considers future work.

2 MODELS AND PERFORMANCE CRITERIA

We begin by describing an abstract model of irregular streaming applications and a system on which they are implemented. The system has a SIMD processor and exhibits regular, predictable scheduling behavior typically seen in real-time computation. We then describe our latency constraint and performance objective, which can be related to the application's SIMD-lane occupancy.

2.1 Application Model

A streaming application is composed of a pipeline of N computational stages, or *nodes*, connected by dataflow edges. Node n_i, $0 \leq i < N$, reads data items from its input edge and performs some computation on each, producing zero or more items per input on its output edge. Each edge is associated with a queue for items, whose capacity is theoretically unbounded (though in practice we will use only a small amount of queue space). When a node *fires*, it consumes inputs and produces a variable, data-dependent number of output items for each input item processed. With each node n_i, we associate an *average gain* g_i, which is the average number of outputs produced per input. We define the *total gain* G_i into node i as $G_i = \prod_{j=0}^{i-1} G_j$.

Data arrives to the application as stream of items of unbounded length. Items arrive regularly at a fixed rate of ρ_0 per unit time (e.g., the polling rate of a sensor). We define $\tau_0 = \frac{1}{\rho_0}$ as the time between consecutive item arrivals.

2.2 Implementation Model

An application is implemented on one single-threaded processor. Each node is assigned a fixed $\frac{1}{N}$ fraction of processor time, which it may utilize or yield as it chooses. We assume that nodes are scheduled preemptively at a fine granularity so that when a node wants to fire, it encounters negligible delay before it can do so.

The processor running the application has SIMD capabilities. Each node n_i can consume a vector of up to v input items each time it fires. These items are processed in parallel, requiring a fixed service time t_i for one input vector, whether it is full or not. (We note that the service time t_i is measured assuming that the node uses only its assigned $\frac{1}{N}$ fraction of the processor while firing.)

2.3 Performance Criteria

The application must meet a latency criterion, in the form of a deadline D. If an item arrives at the head of the pipeline at time t, all outputs associated with that item must exit the tail of the pipeline by time $t + D$; otherwise, we say that the item misses its deadline. An application should ideally not miss its deadline for any input; however, in the presence of stochastic application behavior, we may only be able to achieve a low frequency of misses, rather than guarantee absolutely that they do not occur.

To quantify an application's efficiency, we say that a node is *active* if it is firing or *waiting* otherwise. A waiting node yields its processor time to the system until it is again ready to fire. Our performance objective is to minimize the application's *active fraction* – the total time, over an entire stream of inputs, that any node is active, divided by the total time that any node is either active or waiting. A lower active fraction implies that the application yields more of its available processor time, which could be used, e.g., to support other applications running on the same system or to idle the processor to save energy.

For SIMD processors in particular, an application's active fraction can be decreased by having nodes fire less often but with input vectors closer to the maximum size v. In a throughput-oriented application, one can utilize the queues between nodes to straightforwardly schedule execution so as to ensure that essentially every firing of a node consumes a full vector of v inputs. However, when an application has a deadline to meet, a node may not be able to wait an arbitrarily long time for a full vector of input to accumulate before firing. The more empty SIMD lanes in a typical firing, the more firings (and hence, the more time spent active) are needed to process a given number of inputs, and the higher the resulting active fraction. To balance deadline and performance concerns, we will insert *bounded* waiting times at various points in the application so as to reduce the occurrence of empty SIMD lanes.

3 RELATED WORK

Irregularity is a well-documented feature of applications running on SIMD processors [7]. Overcoming irregularity to obtain high performance requires some form of dynamic data-to-SIMD-lane remapping [11, 12, 24]. Prior work on SIMD pipeline design and scheduling, including our own [9, 21] and numerous domain-specific contributions, has focused on maximizing the throughput of these applications. In contrast, our focus in this work is to explore the impact of latency constraints on the design problems arising for irregular streaming applications.

Much work over the past ten years addresses real-time scheduling for SIMD devices, particularly GPUs. Implementations include TimeGraph [15], GPUSync [10], and many others [8, 20, 23, 27]. The principal goal of these schedulers is to divide the GPU among separate, competing tasks, including both real-time and non-real-time work. In contrast, this work focuses on effectively scheduling a single real-time pipeline composed of multiple cooperating parts.

Reducing the pipeline's active fraction frees up processor time that a system-wide scheduler like those mentioned above could use to support other tasks.

An important contribution of the prior work is to show that, even on a GPU device, appropriate driver support could make feasible a real-time scheduling loop supporting the model of Section 2. However, Otterness et al. [19] provide a note of caution, showing that effective real-time guarantees are limited by undocumented or poorly specified device behaviors. For this reason, we chose to initially develop and test strategies in simulation, as described in Section 6, rather than attempt to build infrastructure running on an actual GPU.

The work of Verner et al. [25], like our own, considers real-time deadlines to be associated with data items to be processed, as opposed to tying deadlines to compute tasks. However, that work does not consider streams composed of multiple processing stages, nor the case where those stages exhibit irregular dataflow.

In the presence of stochastic behavior, estimating the likely maximum time before an item exits the pipeline is an application of queueing theory. In particular, the SIMD processing characteristic of nodes corresponds to a queue with *bulk* or *batch service*, which was first analyzed in [2] and later in [5]. While these works study a single queue that is serviced in bulk, as in the approach of Section 5, later results on networks of bulk queues [13, 14, 18] make strong assumptions regarding temporal servicing behavior that seem a poor fit to SIMD processors. In the present work, we introduce model parameters to capture essential queueing behavior and then select these parameters empirically; future work will explore more theoretically sophisticated approaches.

4 USING ENFORCED WAITS TO REDUCE PROCESSOR UTILIZATION

We seek to lower an application's active fraction by improving its SIMD lane occupancy. Intuitively, since a full vector and a partial vector of inputs to node n_i require the same processing time t_i, we can increase occupancy by delaying n_i's firing to allow more time for inputs to accumulate – ideally until a full vector of v inputs is available. However, the permissible delay is limited by the need to meet the deadline for end-to-end latency.

For simplicity of analysis, we add to each node n_i a fixed delay w_i. Each time node n_i finishes firing, it waits exactly w_i units of time before firing again – regardless of how many or how few items accumulate in its input queue during that time. (This means that a node for which inputs accumulate slowly might sometimes fire with an empty vector; for ease of analysis, we still charge such firings as active time, though in practice they could be treated as a vacation for the node.) Hence, the time between firings of n_i is now exactly $t_i + w_i$.

The delays w_i are free design parameters, which we may choose so as to optimize performance. In what follows, we formulate the active fraction objective and execution constraints in terms of the w_i's and the properties of the system, creating a constrained optimization problem whose solution minimizes active fraction while avoiding deadline misses.

4.1 Objective Function

Our goal is to minimize the application's processor utilization, measured as the fraction of its allocated processor time that it spends executing the code of some node. Time not spent executing a node is returned to the system for other applications to use.

Suppose the application processes a stream of X input items. On average, node n_i of the application's pipeline will receive a total of XG_i inputs over a period of $\frac{X}{\rho_0}$ cycles. Assuming that n_i fires once per $t_i + w_i$ cycles, and that at most v inputs accumulate between firings, the total number of firings by node n_i needed to consume all its input is

$$\left\lceil \frac{X}{\rho_0(t_i + w_i)} \right\rceil.$$

Each of these firings requires active time t_i, and each is followed by a waiting time w_i. Hence, the total active time to process the entire stream is

$$\sum_{i=0}^{N-1} \left\lceil \frac{X}{\rho_0(t_i + w_i)} \right\rceil t_i,$$

while the sum of active and waiting time is

$$\sum_{i=0}^{N-1} \left\lceil \frac{X}{\rho_0(t_i + w_i)} \right\rceil (t_i + w_i).$$

In the limit of large X, removing the ceilings in these expressions has negligible impact. With this simplification, the second sum reduces to $\frac{NX}{\rho_0}$, and the ratio of active to active-plus-waiting time reduces to

$$\frac{1}{N} \sum_{i=0}^{N-1} \frac{t_i}{t_i + w_i}.$$

4.2 Constraints

To ensure that our pipeline can sustain an input arrival rate of ρ_0, it must be *stable*; that is, each node must fire often enough on average to prevent its input queue growing without bound. For the initial node n_0, this constraint can be stated as

$$(t_0 + w_0)\rho_0 \leq v,$$

since the node can consume up to v inputs if available each $t_0 + w_0$ cycles.

For $i > 0$, node n_i must fire often enough to consume output from node n_{i-1} as fast as it is produced. Hence, n_i must consume at least vg_{i-1} items in the time that node n_{i-1} consumes v items (and so produces vg_{i-1} outputs):

$$(t_i + w_i)g_{i-1} \leq t_{i-1} + w_{i-1}.$$

The above constraints enforce sustainable average-case behavior, in that they do not permit any queue to grow without bound over time. To account for transient deviations from average-case behavior, we will assume that the input queue for node n_i attains some maximum size $b_i v$ while processing a stream. Under this assumption, an input queued for node n_i may not produce its corresponding outputs until b_i firings of the node have elapsed, i.e., until $b_i(t_i + w_i)$ cycles have elapsed. To meet our deadline D, we therefore require that

$$\sum_{i=0}^{N-1} b_i(t_i + w_i) \leq D.$$

Free variables: wait times $w_0 \ldots w_{N-1} \geq 0$.

$$\min\ T(\vec{w}) = \frac{1}{N}\sum_{i=0}^{N-1}\left(\frac{t_i}{t_i + w_i}\right) \text{ s.t.}$$
$$(t_0 + w_0)\rho_0 \leq v$$
$$(t_i + w_i)g_{i-1} \leq t_{i-1} + w_{i-1} \quad \text{for } 1 \leq i < N$$
$$\sum_i b_i(t_i + w_i) \leq D$$

Figure 1: Optimization problem with enforced waiting times at each pipeline node.

Combining the above objective and constraints results in the optimization problem of Figure 1. We address the question of how to choose the parameters b_i empirically in Section 6.

5 ALTERNATIVE: BATCH PROCESSING WITH MONOLITHIC PIPELINE

The approach of the previous section assumes that it is possible to allocate fractions of processor time to individual pipeline nodes and to manage their execution behavior in detail. Suppose instead that we have only a throughput-optimized implementation of the pipeline, with no ability to wait between nodes. If the pipeline's average end-to-end latency for a single input is at most $\frac{1}{\rho_0}$, we can pass items to the pipeline and process them individually as they arrive. More generally, if the average end-to-end latency for a block of M inputs is at most $\frac{M}{\rho_0}$, we can repeatedly accumulate and process blocks of M items. For large enough M, processing large quantities of input at once will likely make more efficient use of the processor's SIMD lanes than processing one item at a time. In what follows, we refer to this approach as "monolithic," since it schedules the entire pipeline as a unit rather than controlling delays for individual nodes.

While a larger block size M is preferable for a throughput-oriented pipeline, it also increases the time needed to accumulate a block and the time to process all of its elements. Eventually, M becomes too large to ensure that an arriving item will on average be completely processed by its deadline, and the system becomes unstable. Hence, M is restricted by the application's latency constraint.

As in the previous section, we may attempt to minimize processor utilization for the monolithic implementation through our choice of M. If the application spends some of its time accumulating a block of inputs after having finished the previous block, that additional time may be yielded to the system, decreasing the application's active fraction.

We now describe how to tune this simpler monolithic strategy to minimize active fraction. We will compare the performance to this strategy to that of per-node enforced waits in Section 6.3.

Free variable: block size $M > 0$.

$$\min \frac{\rho_0 \overline{T}(M)}{M} \text{ s.t.}$$
$$\overline{T}(M) \leq \frac{M}{\rho_0}$$
$$b\frac{M}{\rho_0} + \hat{T}(M) \leq D$$
where
$$\overline{T}(M) = \sum_i \left\lceil \frac{MG_i}{v} \right\rceil t_i$$
$$\hat{T}(M) = S\,\overline{T}(M)$$

Figure 2: Optimization problem for monolithic approach.

5.1 Performance Objective and Constraints of Monolithic Application

For a fixed block size M, we may model the monolithic application's processor utilization as follows. A block of M inputs requires total time $\frac{M}{\rho_0}$ to accumulate. During this time, the application first processes some previously accumulated block of input, then waits while the next block finishes accumulating.

The average-case active time to consume a block of M items is simply

$$\overline{T}(M) = \sum_i \left\lceil \frac{MG_i}{v} \right\rceil t_i,$$

that is, the time for each node to consume all its input, assuming that each node produces the average amount of output per input. The corresponding average-case active fraction is $\frac{\rho_0 \overline{T}(M)}{M}$. To ensure that the pipeline is stable, we require that

$$\overline{T}(M) \leq \frac{M}{\rho_0}.$$

It remains to consider the constraint imposed by the per-item deadline D and the *worst-case* processing time $\hat{T}(M)$ incurred for M items, which may exceed the average $\overline{T}(M)$. As for the previous strategy, we introduce a multiplier b to an input item's waiting time to reflect the possibility of a long equilibrium queue size, for which a newly arrived item may find $b-1$ full blocks of M items ahead of it in the queue. Such an item may wait for up to time $b\frac{M}{\rho_0}$ before being processed by the pipeline; hence, we require

$$b\frac{M}{\rho_0} + \hat{T}(M) \leq D.$$

We assume that $\hat{T}(M) \leq S\overline{T}(M)$, where S is a scale parameter. If worst-case time arises because of occasional bursty behavior in the data, we expect that as M becomes large, $S \longrightarrow 1$ as local variations in the input stream are averaged out. But S may be larger if the stream exhibits sustained non-average-case behavior over longer stretches.

To summarize, Figure 2 describes the problem of minimizing active fraction for the monolithic approach.

6 EMPIRICAL INVESTIGATION OF STRATEGIES

We empirically compared the behavior of the enforced waiting and monolithic strategies in an irregular streaming pipeline. The pipeline we used is drawn from an implementation of the NCBI BLAST biosequence comparison tool [1] developed for the Mercator framework for irregular streaming applications on GPUs [9].

6.1 Application Description

The BLAST pipeline consists of four nodes, three of which produce at most one output per input and one of which (stage 1) may expand its input by a factor of up to $u = 16$. Table 1 shows the service time t_i (measured on an NVidia GTX 2080 GPU) and the average gain g_i of each stage. Times and gains were measured on a comparison of the human genome vs. a 64-kilobase microbial query sequence. The assumed SIMD vector width for all stages was $v = 128$, consistent with the Mercator implementation.

The average gains g_i do not completely describe the range of behavior observed for the various pipeline stages. Rather than gather a detailed empirical distribution of each gain, we assumed that for each input, nodes 0, 2, and 3 produce one output with probability g_i or 0 otherwise. For node 2, we assumed that the number of outputs per input is Poisson with mean g_i, censored at the upper limit u.

We considered a range of possible values for the input inter-arrival time $\tau_0 = \frac{1}{\rho_0}$ and the deadline D. τ_0 was varied from 1 to 100 cycles, while D was varied from 2×10^4 to 3.5×10^5 cycles. Values of D below 2×10^4 cycles resulted in no feasible (that is, substantially miss-free) realizations of the pipeline by either approach tested, while the upper limits for both parameters were chosen because observed behavior was largely unchanged for larger values.

6.2 Applying the Strategies

For each of the enforced-wait and monolithic strategies, we implemented the optimizations of Figures 1 and 2 in the AMPL solver language. All pipeline parameters, as well as values for τ_0 and D, were supplied as described above. Given these values, as well as parameters describing the worst-case behavior of each pipeline stage (b_i's for the enforced-waits strategy, or b and S for the monolithic strategy), we inferred optimal values for the free variables (w_i for the enforced-waits strategy, M for the monolithic strategy) using the BONMIN [3, 4, 17] open-source solver for non-linear mixed-integer programs.

Node	t_i (cycles)	g_i
0	287	0.379
1	955	1.920
2	402	0.0332
3	2753	N/A

Table 1: Properties of the NCBI BLAST streaming pipeline. Gain for the final stage is omitted because it does not impact the design choices made by optimization or the inferred cost of the computation. For this application, we used $v = 128$.

A crucial question in applying each strategy is how to choose parameter values to adequately capture the worst-case behavior of the pipeline. While it is possible in principle to estimate these parameters from the known gain distributions using queueing theory, such estimation is challenging for the enforced-waits model, which represents a tandem network of bulk-service queues with non-exponentially distributed service times. We instead took an empirical approach as follows. We developed a discrete-event simulation of pipeline execution on the system described in Section 2. The simulator is capable of processing a long stream of simulated inputs using either of our two strategies and determining how many inputs, if any, incur a deadline miss. We began with optimistic choices for for the worst-case parameters ($b_i = \lceil g_i \rceil$ and $b = 1$, $S = 1$, essentially asserting that the worst-case behavior closely matches the average), then used the optimizer to implement each strategy and checked how often the simulator reported deadline misses over 100 runs with different random seeds. If frequent misses were observed for any tested values of D and τ_0, we raised one or more parameters, re-optimized, and tried again. We note that the active fractions measured in the simulator closely matched those predicted by the optimizer for each approach and set of parameters tested.

For the enforced-waits strategy, the following parameters resulted in no misses in at least 95% of random trials for any combination of D and τ_0 tested, over simulated execution on streams of 50000 inputs: $b_0 = 1, b_1 = 3, b_2 = 9, b_3 = 6$. For most ($D$,$\tau_0$) combinations, no misses were observed in at least 98% of random trials, and the number of inputs incurring a miss was fewer than 1% of all inputs. Smaller values for the b parameters empirically incurred much more frequent deadline misses.

For the monolithic strategy, we observed no deadline misses in simulation even with $b = 1$, $S = 1$. The likely explanation is that, for large enough M, the throughput-oriented implementation aggregates many node firings into the active time for each M inputs, which tends to suppress occasional departures from average-case behavior.

6.3 Performance Comparison

Figure 3 compares the optimized active fractions for the enforced-wait and monolithic strategies on our BLAST pipeline. Qualitatively, it is immediately evident that the two approaches exhibit complementary sensitivities – the enforced-wait strategy's active fraction is insensitive to τ_0 except at the smallest sizes but scales inversely with D, while the monolithic strategy is mostly insensitive to D but scales inversely with τ_0.

For the enforced-wait strategy, a longer deadline D relaxes a constraint on the times $t_i + w_i$, so that the total time spent waiting can rise, and hence the active fraction can become smaller, as D rises. This behavior illustrates that enforced waiting can effectively exploit "deadline slack" to insert waits that improve SIMD occupancy and so improve processor utilization. In contrast, in the monolithic strategy, raising D allows the block size M to grow, but the active fraction tends to a constant in the limit of large M. Hence, the monolithic strategy's ability to exploit additional deadline to improve utilization is limited.

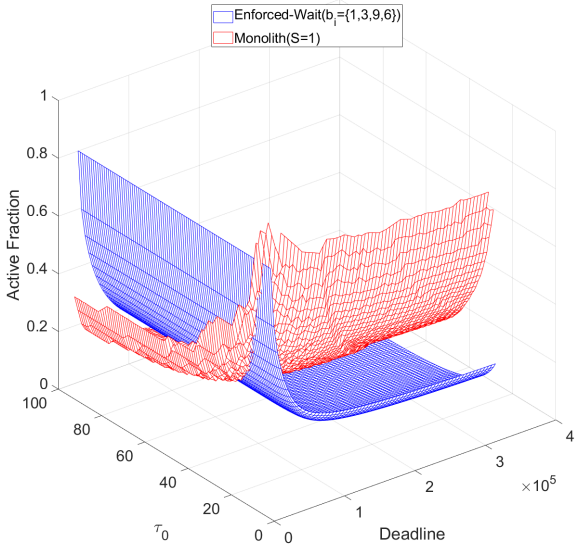

Figure 3: Comparison of enforced-wait and monolithic strategies on BLAST pipeline, illustrating complementary sensitivities to τ_0 and D in the two approaches.

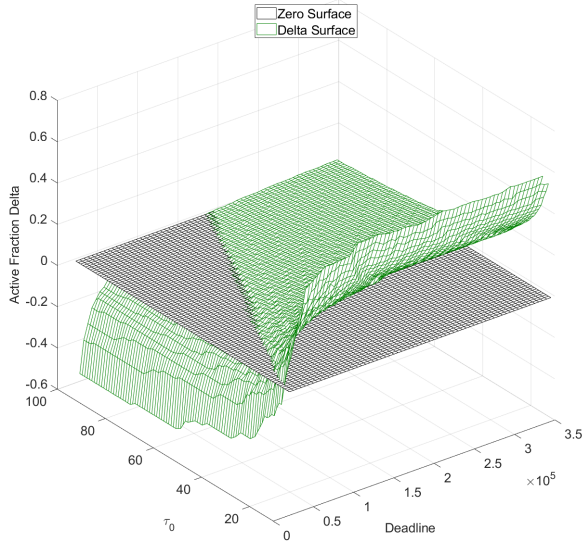

Figure 4: Difference between active fractions for the two strategies (monolithic minus enforced-waits). Zero plane is shown in black. Enforced waits outperform the monolithic strategy in the region above the zero plane.

The opposite picture emerges with respect to sensitivity to τ_0. For the enforced-wait strategy, the average-case constraints on $t_i + w_i$ scale inversely with ρ_0 and hence linearly with τ_0, so limiting behavior emerges as τ_0 rises. In contrast, the expression for the monolithic strategy's active fraction scales linearly with ρ_0 and hence inversely with τ_0; hence, this strategy can better exploit slower input arrival rates to improve utilization.

Qualitatively, then, we expect and observe that enforced waits are more effective when the deadline is larger relative to the arrival rate, while the monolithic strategy wins in the opposite case. Figure 4 quantifies this effect by plotting the difference between the monolithic and enforced-waits active fractions. The enforced-waits strategy decreased active fraction vs. the monolithic strategy over a large portion of the arrival rate/deadline parameter space. The difference is particularly large – at least 0.4 in absolute terms, or several-fold better for enforced-waits – in the region of the fastest arrival rates and sufficient deadline slack. Conversely, the monolithic strategy dominates by a similar amount for slow arrivals and little deadline slack.

Overall, the two approaches tested display complementary strengths. As the rate of item arrival increases, the more complex enforced-waits model is better able to effectively leverage any available gap between total service time and deadline to improve SIMD processor utilization. While neither strategy can absolutely guarantee that all deadlines will be met in the face of stochastic behavior, enforced-waits is more sensitive to stochastic changes in gain at each stage than the monolithic approach, which tends to average together the behavior of many vectors of inputs. It therefore proved empirically more difficult to eliminate all misses with enforced-waits. However, the frequency of misses can still be driven to a low level for moderate values of the parameters b_i that substantially lower processor utilization.

7 CONCLUSION AND FUTURE WORK

Streaming applications with latency constraints and irregular data flow are challenging to schedule effectively. For SIMD processors, scheduling trades off the ability to use the processor efficiently, by gathering data into full-width SIMD vectors, against the ability to meet deadlines by processing inputs rapidly. We have shown how to balance these concerns by introducing enforced waits at each node of a streaming pipeline. Our approach can substantially reduce the time that the application occupies the processor while respecting latency requirements. We have demonstrated for a test application that, for a broad range of arrival rates and deadlines, enforced waits incur less processor utilization than a simpler strategy that treats the pipeline as monolithic for scheduling purposes.

Future work will focus on better *a priori* modeling strategies for departures from an application's average behavior. Empirical observation of a full pipeline's execution would permit better modeling of the distribution of service times for the monolithic model, which should permit application of prior work on bulk-service queues [2, 5] to derive reasonable values for the queue multiplier b and/or directly estimate the time an item spends in the system. The enforced-waits model has less supporting queueing theory, but we will investigate approximations using better-understood models such as (non-bulk) Jacksonian networks. We note that these

models generally assume Poisson arrivals, which is a reasonable generalization of the fixed arrival rate assumed in this work.

We will seek additional real-time streaming applications to improve validation, including an implementation of the gamma-ray burst detection application mentioned earlier. We will also investigate how to realize our scheduling model on GPU devices, either exactly (which will likely require driver modifications) or approximately by extending our timing model to accommodate cooperative or otherwise more coarse-grained division of processor time between pipeline stages. Even if the closed architecture and hardware limitations of GPUs prove resistant to such scheduling, many other devices, including general-purpose multicores, increasingly offer wide SIMD support and have much more developed system scheduling that could accommodate our approach.

REFERENCES

[1] Stephen F Altschul, Warren Gish, Webb Miller, Eugene W Myers, and David J Lipman. 1990. Basic local alignment search tool. *Journal of Molecular Biology* 215, 3 (1990), 403–410.

[2] Norman T. J. Bailey. 1954. On queueing processes with bulk service. *J. Royal Statistical Society, Series B (Methodological)* 16 (1954), 80–87.

[3] Pierre Bonami, Lorenz T. Biegler, Andrew R. Conn, Gérard Cornuéjols, Ignacio E. Grossmann, Carl D. Laird, Jon Lee, Andrea Lodi, François Margot, Nicolas Sawaya, and Andreas Wächter. 2008. An algorithmic framework for convex mixed integer nonlinear programs. *Discrete Optimization* 5, 2 (2008), 186–204.

[4] Pierre Bonami, Mustafa Kilinç, and Jeff Linderoth. 2012. Algorithms and software for convex mixed integer nonlinear programs. In *Mixed Integer Nonlinear Programming*. Springer, 1–39.

[5] G. Briére and M. L. Chaudhry. 1989. Computational analysis of single-server bulk service queues, $M/G^Y/1$. *Advances in Applied Probability* 21 (1989), 207–225.

[6] James Buckley, Lars Bergstrom, Bob Binns, Jeremy Buhler, Wenlei Chen, Michael Cherry, Stefan Funk, Dan Hooper, John Mitchell, Georgia De Nolfo, et al. 2019. The Advanced Particle-astrophysics Telescope (APT). *Bulletin of the American Astronomical Society* 51, 7 (2019), 78.

[7] M. Burtscher, R. Nasre, and K. Pingali. 2012. A quantitative study of irregular programs on GPUs. In *Proc. 2012 IEEE Int'l Symp. on Workload Characterization*. 141–151.

[8] Guoyang Chen, Yue Zhao, Xipeng Shen, and Huiyang Zhou. 2017. EffiSha: A Software Framework for Enabling Effficient Preemptive Scheduling of GPU. *SIGPLAN Not.* 52, 8 (Jan. 2017), 3–16.

[9] Stephen V. Cole and Jeremy Buhler. 2017. MERCATOR: A GPGPU Framework for Irregular Streaming Applications. In *2017 International Conference on High Performance Computing Simulation (HPCS)*. 727–736.

[10] Glenn A. Elliott, Bryan C. Ward, and James H. Anderson. 2013. GPUSync: A Framework for Real-Time GPU Management. In *2013 IEEE 34th Real-Time Systems Symposium*. 33–44.

[11] Kshitij Gupta, Jeff A. Stuart, and John D. Owens. 2012. A study of Persistent Threads style GPU programming for GPGPU workloads. In *2012 Innovative Parallel Computing (InPar)*. 1–14.

[12] K. Gupta, J. A. Stuart, and J. D. Owens. 2012. A study of persistent threads style GPU programming for GPGPU workloads. In *Innovative Parallel Computing (IEEE InPar)*. 1–14.

[13] W. Henderson and P. G. Taylor. 1990. Product form in networks of queues with batch arrivals and batch services. *Queueing Systems* 6 (1990), 71–88.

[14] W. Henderson and P. G. Taylor. 1991. Some new results on queueing networks with batch movement. *J. Applied Probability* 28 (1991), 409–421.

[15] Shinpei Kato, Karthik Lakshmanan, Raj Rajkumar, and Yutaka Ishikawa. 2011. TimeGraph: GPU scheduling for real-time multi-tasking environments. In *Proc. USENIX ATC*. 17–30.

[16] E.A. Lee and D.G. Messerschmitt. 1987. Synchronous data flow. *Proc. IEEE* 75, 9 (1987), 1235–1245.

[17] Robin Lougee-Heimer. 2003. The common optimization interface for operations research: Promoting open-source software in the operations research community. *IBM Journal of Research and Development* 47, 1 (2003), 57–66.

[18] Mihaela Mitici, Jasper Goseling, Jan-Kees van Ommeren, Maurits de Graaf, and Richard J. Boucherie. 2017. Tandem queue with batch service and its applications in wireless sensor networks. *Queueing Systems* 87 (2017), 81–93.

[19] Nathan Otterness and James H Anderson. 2021. Exploring AMD GPU Scheduling Details by Experimenting With "Worst Practices". In *Proceedings of the 29th International Conference on Real-Time Networks and Systems*.

[20] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. 2015. Chimera: Collaborative Preemption for Multitasking on a Shared GPU. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (Istanbul, Turkey) *(ASPLOS '15)*. Association for Computing Machinery, New York, NY, USA, 593–606.

[21] Tom Plano and Jeremy Buhler. 2020. Scheduling irregular dataflow pipelines on SIMD architectures. In *Proceedings of the 2020 Sixth Workshop on Programming Models for SIMD/Vector Processing*. 1–9.

[22] Martin Roesch et al. 1999. Snort: Lightweight intrusion detection for networks.. In *Lisa*, Vol. 99. 229–238.

[23] Ivan Tanasic, Isaac Gelado, Javier Cabezas, Alex Ramirez, Nacho Navarro, and Mateo Valero. 2014. Enabling Preemptive Multiprogramming on GPUs. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture* (Minneapolis, Minnesota, USA) *(ISCA '14)*. IEEE Press, 193–204.

[24] Stanley Tzeng, Anjul Patney, and John D. Owens. 2010. Task Management for Irregular-Parallel Workloads on the GPU. In *High Performance Graphics*, Michael Doggett, Samuli Laine, and Warren Hunt (Eds.). The Eurographics Association, 9 pages.

[25] Uri Verner, Assaf Schuster, Mark Silberstein, and Avi Mendelson. 2012. Scheduling Processing of Real-Time Data Streams on Heterogeneous Multi-GPU Systems. In *Proceedings of the 5th Annual International Systems and Storage Conference* (Haifa, Israel) *(SYSTOR '12)*. Association for Computing Machinery, New York, NY, USA, Article 8.

[26] Paul Viola and Michael Jones. 2001. Rapid object detection using a boosted cascade of simple features. In *Proceedings of the 2001 IEEE computer society conference on computer vision and pattern recognition. CVPR 2001*, Vol. 1. IEEE, I–I.

[27] Bo Wu, Xu Liu, Xiaobo Zhou, and Changjun Jiang. 2017. FLEP: Enabling Flexible and Efficient Preemption on GPUs. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA, 483–496.