Adaptive Resource Utilization via Feedback Control for Streaming Applications *

Hasnain A. Mandviwala, Nissim Harel,
Umakishore Ramachandran
Georgia Institute of Technology
College of Computing
801 Atlantic Dr., Atlanta, GA
{mandvi, nissim, rama}@cc.gatech.edu

Kathleen Knobe
HP Labs - Cambridge Research Lab.
One Cambridge Center, Cambridge, MA
kath.knobe@hp.com

Abstract

A large emerging class of interactive multimedia streaming applications that are highly parallel can be represented as a coarse-grain, pipelined, data-flow graph. One common characteristic of these applications is their use of current data: A task would obtain the latest data from preceding stages, skipping over older data items if necessary to perform its computation. When parallelized, such applications waste resources because they process and keep data in memory that is eventually dropped from the application pipeline. To overcome this problem, we have designed and implemented an Adaptive Resource Utilization (ARU) mechanism that uses feedback to dynamically adjusts the resources each task running thread utilizes so as to minimize wasted resource use by the entire application. A color-based people tracker application is used to explore the performance benefits of the proposed mechanism. We show that ARU reduces the application's memory footprint by two-thirds compared to our previously published results, while also improving latency and throughput of the application.

1. Introduction

Resource Utilization is the efficient use of resources given to an application. Unlike Resource Management schemes, such as QoS and scheduling, which are concerned with *allocation* of resources to an application, Resource Utilization targets the economical *use* of resources already allocated to an application. For example, if a scheduler provides an application thread pool with a set of CPU time-slices, the rules by which resources are managed among these threads would be characterized as their resource utilization policy.

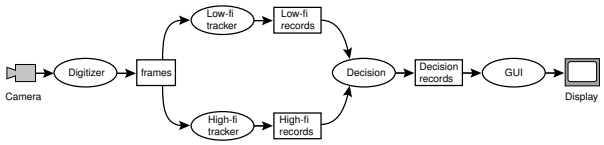

Figure 1. **Vision Application pipeline.**

Efficient resource utilization is considered primarily the responsibility of application developers, who have a vast set of static analysis methodologies and tools at their disposal. However, many applications such as streaming multimedia applications are affected by dynamic phenomena such as current load, for which such tools are inapplicable.

Our motivation to suggest a dynamic resource utilization mechanism stems from our unique perspective of reducing wasted resource usage. To better understand our goal, it is important to understand the characteristics, structure and dynamics of parallel streaming applications that we target. Figure 1 illustrates an example of a simple streaming vision application pipeline.

The most important characteristic common to this class of applications is the need for representing temporal data. Therefore, associating every piece of data with a *timestamp*, allows for an index into the virtual (or wall-clock) time of the application. Such a timestamp tag preserves the temporal locality of data, which is required during analysis by many multimedia algorithms and interactive applications. For example, a stereo module in an interactive vision application may require images with *corresponding* [1] timestamps from multiple cameras to compute its output, or a gesture recognition module may need to analyze a sliding

* The work has been funded in part by an NSF ITR grant CCR-01-21638, NSF grant CCR-99-72216, the Yamacraw project of the State of Georgia, and the Georgia Tech Broadband Institute. The equipment used in the experimental studies is funded in part by an NSF Research Infrastructure award EIA-99-72872, and Intel Corp.

1 Corresponding timestamps could be timestamps with the same value or with values close enough within a pre-defined threshold.

window over a video stream. Providing support at the run time system level for handling time can ease the programming effort required to write and maintain these applications.

In addition, application pipelines consist of independent tasks implemented by a single or a group of threads. The structure of each task typically consists of repeatedly getting data from input buffers, processing it, and producing new data. These tasks run independently in parallel but have input dependencies between them that are inherent in the application task-graph structure. Tasks also have variable execution rates based on their internal runtime complexity and resource availability in the parallel environment. Such variable execution rates create a production differential between tasks, creating a need for communication buffers between threads.

Stampede [19, 24, 23] is a run-time system designed to provide abstractions and system support to help write such streaming media applications. One available abstraction is called a *timestamp*, which is associated with every data item, produced or consumed by any application thread. Other abstractions, such as *Channels* and *Queues*, provide data elements with system-wide unique names, and serve as containers or buffers for storing time-sequenced data items. They facilitate inter-thread communication and synchronization, regardless of the physical location of threads. Since tasks in successive pipeline stages do not have the same rate of consumption/production, a task may have to drop or skip-over stale data to access the most recent data from its input buffers. Consuming fresh data helps ensure the production of fresh output, a necessary condition for interactive multimedia pipelines. Channels and Queues alleviate the problem of variable production/consumption rate by allowing non-FIFO and out-of-order access to data items sometimes required in such application pipelines.

Skipping of data may be important to preserve the interactive nature of applications, but as a side affect the system sustains data items that are eventually dropped. Efficient implementation of the application will attempt to minimize resource utilization wasted in maintaining such data. Stampede run-time system contains garbage collection (GC) algorithms [18, 6, 14] based on timestamp visibility that free data elements as soon as the run-time system can be certain that they are not going to be used by the application. These Stampede GC algorithms differ from traditional GC in their logic governing garbage collection in that they assert data will not be used in the future. Traditional GC concepts, on the other hand, regard a data element as a garbage only if it not *reachable* by any one of the threads that compose the computation.

GC algorithms may help alleviate the problem by freeing unwanted data after it has already been created, however it would be best if wasted items are never produced in the first place, saving both processing, networking, and memory resources. Static determination of unneeded items cannot be made due to the interactive nature of such applications. The only way to eliminate wasteful resource consumption is by dynamically controlling and matching the data production of each thread with the overall application rate.

In this paper, we propose an Adaptive Resource Utilization (ARU) mechanism that uses feedback to influence utilization among application threads and minimizes the amount of wasted resources consumed by interactive streaming multimedia applications. The mechanism provides the production rate of each pipeline stage as feedback to earlier stages. This feedback helps each stage adapt its own production rate to suit the dynamic needs of the application.

We use a color-based people tracker application to explore the performance benefits of the proposed algorithm. We show that compared with our former published results, the ARU mechanism reduces the memory footprint by nearly two-thirds while simultaneously improving performance measures such as latency and throughput.

In section 2 we review and compare related work. In section 3 we describe the run-time instrumentation that enables the ARU optimization. In section 4 we explain our performance evaluation methodology. We present comparative results in section 5, and conclusions in section 6.

2. Related Work

The Adaptive Resource Utilization (ARU) mechanism we propose strives to optimize resource usage to best meet resource availability. Prima facie, this mechanism seems similar to the notion of Quality of Service (QoS) in multimedia systems but is quiet different. Firstly, ARU deals with optimizing *resource utilization*, as opposed to QoS, which we categorize as a *resource management* system. In other words, ARU does not guarantee a specific level of service quality like QoS. Instead, it makes sure that threads execute tasks at an equilibrium rate such that resources are not wasted on computations and on producing data that would eventually be thrown away. Therefore, while ARU allows an application to voluntarily reduce its resource consumption on the inference that using more resources would *not improve* performance, QoS forces a reduction in resource consumption if it cannot meet a certain service level. QoS is not concerned with inefficient use of resources as long as a service quality is maintained. Thus, we can consider ARU to be orthogonal to QoS provisioning.

Most QoS provisioning systems work at the level of the operating system, *e.g.,*reserving network bandwidth for an application or impacting the scheduling of threads. Our ARU mechanism, resides in the programming runtime en-

vironment above the OS-level. QoS provisioning typically requires the application writer to understand, and in many cases specify, the application's behavior for different levels of service (see [32, 1, 29, 16]). However, ARU requires little developer involvement.

Similarities to ARU can also be drawn from the extensive work done in the domain of real-time scheduling, especially from those that use feedback control. However, there are fundamental differences between them. First and foremost, similar to QoS, real-time scheduling is a resource management mechanism. The primary aim of real-time scheduling is to ensure that data processing complies with application deadlines. ARU, on the other hand, attempts to minimize wasted resources by exploiting available knowledge about data dependencies. ARU therefore takes advantage of inter-application dependencies whereas real-time scheduling make no such exploit and uses only global information about all applications. The global knowledge is readily available in the OS, where real-time scheduling is typically performed. However, internal data-dependencies can be found only within an application or a run-time system designed for a particular class of applications (such as Stampede). ARU makes use of this information from the Stampede runtime implicitly derived by the input/output connections made between threads.

Real-time scheduling also includes reservation style scheduling works where different threads must first reserve their CPU time to be allowed use of the resources [30, 8, 28, 27, 17]. These differ from our approach in (1) they are all scheduling techniques, (2) are instrumented in the kernel and (3) require expert developers to supply accurate period/proportion reservation. There are systems that alleviate the requirement of accurate reservation information by using feedback: The Real-Rate [27] mechanism removes the dependency on specifying the rate of real-time tasks.

More traditional real-time schedulers such as EDF [12], RM [12] and Spring [33] are all open-loop static scheduling algorithms that require complete knowledge about tasks and their constraints. Variants like FC-EDF [13], similar to Adaptive Earliest Deadline (AED) [7] in real-time databases, use feedback to improve miss-ratio in resource constrained environments by increasing priority (AED) or increasing CPU utilization (FC-EDF).

Other hybrid schedulers such as SMART [17], and BERT [2] handle both real-time and non-real-time applications simultaneously. Both use feedback to allow for the coexistence of real-time and non-real-time applications types by stealing resources from non-real-time applications to give to real-time algorithms.

Recent *adaptive* scheduling works consider resource constrained environments other than limited CPU, *e.g.,*high memory pressure [20]. Here, threads are put to sleep to prevent thrashing while experiencing high memory pressure. Although our approach also involves sleeping of threads, we do so not to avoid resource constraints but to avoid using resources on computing unneeded data altogether. Avoiding wasted computation indirectly reduces memory pressure by using less resources to begin with.

Massalin and Pu [15] introduced the idea of using feedback control loops similar to hardware phase locked loops in real-time scheduling. Their approach deals with adaptively giving more unused resources to threads that require them or give priority to threads that need them more based on feedback information. The Swift toolbox was also developed [21, 3, 5] to allow portability of the feedback control mechanism from its OS test bed, the Synthesis Kernel [22]. However, these mechanisms try to improve upon scheduling (resource management) and do not try to eliminate wasted resource usage (resource utilization). There is also a special need for application modification to use these feedback mechanisms as shown in works [4, 26] where as our mechanism, instrumented in the Stampede runtime system, is available by default to application writers that use the runtime. In addition, the feedback control loop work of Massalin and Pu [15] deals with feedback *filters*, where feedback information is first filtered before propagated back to the algorithm. Currently, ARU does not include the notion of filters, although it is a natural extension of our work.

It is important to note that giving more resources to bottleneck threads in the pipeline would improve the performance of the overall pipeline application. However, we deal with scenarios where the option of more resources, *e.g.,*CPU or threads to the bottleneck task has been exhausted. Such cases, *i.e.,*problems of dynamic *resource management* are handled by feedback based scheduling algorithms [15, 21, 4, 3, 26, 5] where bottleneck threads are given more resources to improve their throughput.

Both ARU and GC are similar in that they are dynamic in nature, and have the common goal of freeing resources that are not needed by an application. However, the ARU mechanism is complementary to both traditional GC [31, 9] and Timestamp based GC in streaming application [18]. Traditional GC algorithms consider a data item to be garbage only if it is not "reachable" by any thread in the application. On the other hand, Timestamp based GC algorithms such as Dead Timestamp GC (DGC) [6] use virtual time inferences to define garbage. These are data items that the application will not use in the future. ARU goes one step further, and attempts to prevent the creation of data items that will not be used at all by examining the consumption/production patterns of the application. Unlike GC algorithms, ARU directly affects the pace of data production and matches it with available system resources and application pipeline constraints. It should be noted, however, that the ARU mechanism does not eliminate the need to deal

with garbage created during execution, although it reduces the magnitude of the problem.

3. ARU via Feedback Control

In this section we present our distributed ARU via a Feedback Control Loop that minimizes the creation of unused data in streaming applications.

3.1. Factors Determining Rate of Tasks

Pipelined streaming applications such as the one illustrated in figure 1, have similarities with systolic architectures [11]. It is therefore useful to talk about a *rate* of execution for the entire pipeline. This is the rate at which a processed output is emitted from the right end of the pipeline as fresh input is being provided from the left end. Ideally, every pipeline stage should operate at the *same* rate such that no resources are wasted at any stage. However, in contrast to a systolic architecture, the rate is different at each pipeline stage of a streaming application. Intrinsically, the rate of each pipeline stage is determined by the changing size of the input data, and the amount of processing required on it. Since computation is data-dependent (for example, looking for a specific object in a video frame), the execution time of a task for each loop iteration depicted in figure 2 may vary. Additionally, the actual task execution time is subject to the vagaries of OS scheduling and computational load on the machine. Unfortunately, these parameters are fully known only at run time.

3.2. Eliminating Wasted Resources

As discussed earlier, skipping over unwanted data may allow an application to keep up with its interactive requirements, but it does not allow savings on computations already executed to produce such data. We use the term *wasted computation* to denote task executions that produce data eventually unused by downstream threads. Unfortunately, *a priori* knowledge of parameters described earlier (section 3.1) is required to eliminate wasted computation. Even though the future cannot be determined at any point in time, *virtual time* (VT) systems such as Stampede, allow inferences to be made about the *future local virtual time* using task-graph topology. This technique is used to eliminate irrelevant resource usage. Stampede associates a notion of virtual time with each thread in a pipelined application. Furthermore, data produced by each thread is tagged with a virtual timestamp. In our earlier work [6], we proposed GC algorithms for eliminating upstream computations (*i.e.,*computations performed at earlier stages of the task-graph) using the virtual times of timestamped data requests made by downstream threads. However, such techniques have shown limited success [6]. The cause for this phenomenon is that in many interactive application pipelines, upstream threads (such as the digitizer in figure 1) tend to be quicker than downstream threads (such as an image tracker). As a result, it generally becomes too late to eliminate upstream computations based on local virtual time knowledge. There is, however, another piece of information that is embedded in the task-graph that can help the run-time system to predict wasted computations. If processing rate of downstream stages were made available to the runtime system, it would become possible to control the rate of production of timestamped items in earlier stages. This would retroactively eliminate unwanted computation *before* data production.

3.3. Distributed ARU

We now describe a distributed algorithm whereby tasks constantly exchange local information to change their rate of data item production.

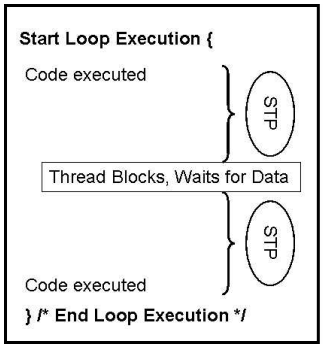

Figure 2. **Measuring the Sustainable Thread Period (STP)**

3.3.1. Sustainable Thread Period We define *sustainable thread period (STP)* as the time it takes to execute one iteration of a thread loop. STP is dynamically computed locally by a thread with clock readings taken at the end of each loop iteration (see figure 2). Since the STP is measured at runtime, it captures all factors affecting the execution time of a thread. It is important to note that blocking time (*i.e.,*time spent waiting for an upstream stage to produce data) is not included in the STP. In essence, a *current-STP* value captures the minimum time required to produce an item given present load conditions. This *current-STP* is used as *feedback* to compute the *summary-STP* described below, which is in-turn propagated back upstream as more feedback to other tasks in the pipeline.

3.3.2. Computation of Summary-STP and Backward Propagation For generality in the ARU algorithm, a *node*

may either be a *thread*, *channel*, or a *queue*. Each node has a *backwardSTP* vector that contains *summary-STPs* received from downstream nodes (see figure 3). Using this vector, along with the *current-STP* generated by the node itself (if this is a thread node), each node computes a *summary-STP* value *locally*, that is then propagated to upstream nodes on every *put/get*[2] operation.

Given below is the algorithm for propagating and computing the *summary-STP*:

- Receive *summary-STP* value from output connection *i* from downstream nodes (figure 3).
- Update *backwardSTP[i]* with received *summary-STP* value.
- Compute *compressed-backwardSTP* value by applying *min/max* operator to *backwardSTP* vector.
- If node is a thread, compute *summary-STP* = *max(compressed-backwardSTP, current-STP)*
- Else (node is a channel or a queue and therefore does not generate *current-STP* values) *summary-STP* = *compressed-backward-STP*.
- Propagate *summary-STP* to nodes earlier in the pipeline.

The computation of the *compressed-backwardSTP* value represents compressing the execution rate knowledge of consumer nodes. This computation can be either done by using the default *min* operator, which is a conservative approach, or with the help of a user-defined function that captures data-dependencies between consumer nodes. For complete data-dependency between *all* consumers nodes, the *max* operator can be used (figure 4). Any function other than the default *min* operator requires the application writer to understand the data-dependencies that exist between consumer nodes so as to decide which nodes dictate the *compressed-backwardSTP* value without hurting the current-node throughput. The *min* operator is the default operator as it does not affect throughput and is safe to use in all data-dependency cases.

In the example shown in figure 3, node A has output connections to nodes B-F. The downstream nodes B-F report *summary-STP* values of 337, 139, 273, 544, and 420, respectively to node A. Consider such a pipeline where nodes B-F are end points of the computation. In this case, node A sustains the fastest consumer (C) with the smallest *summary-STP* by using a *min* operation to compute the *compressed-backwardSTP*. Consider the pipeline shown in

2 In the specific context of Stampede, *put/get* operations allow inserting and retrieving timestamped data to/from globally accessible Stampede buffers called *channels* and *queues*. However, such operations could be generalized to *write/read* operations on any given buffer data-structure.

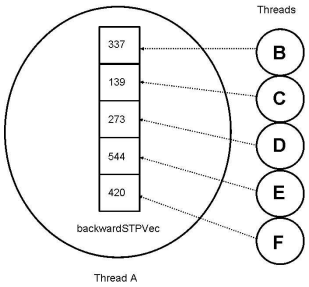

Figure 3. **STP propagation using the backwardSTPVec**

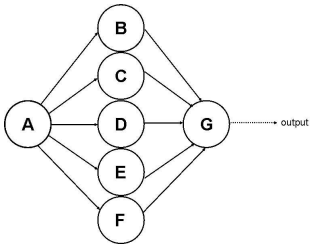

Figure 4. **Using the Max() operator**

figure 4. In this case, A is a thread connected to data abstractions represented by nodes B-F, which are in turn connected to a consumer G. With this data-dependency knowledge, node A can use a *max* operation on the *backwardSTPVec* to get the highest *summary-STP* value and therefore get an aggressive reduction in production rate to match the slowest consumer. This is acceptable in a pipeline where node G dictates the throughput of the entire pipeline and producing more data would only be wasteful.

The *summary-STP* value is then computed by applying a *max* operator between the *compressed-backward-STP* value and the *current-STP* value of the node. Note only thread nodes generate *current-STP* feedback values. This allows a thread with a larger period than its consumers to insert its execution period into the *summary-STP*.

Once the *summary-STP* value is computed, it is propagated to upstream nodes. Source threads, *i.e.,*threads on the left of the pipeline in figure 1, use the propagated *summary-STP* information to adjust their rate of data item production. Our results show that this cascading effect indirectly adjusts the production rate of all upstream threads.

Both the computation and propagation of *summary-STP* values occur in a distributed manner in the pipeline, *i.e.,*the computation is completely local to a node, and values are exchanged with neighboring nodes by piggy-backing them on every *put/get*[2] operation. This mechanism has scalability advantages over a centralized approach used elsewhere. *e.g.,*in scheduling and QoS systems (section 2) management is handled by a central entity such as a scheduler. However,

a distributed mechanism does raise issues of system reaction time. The worst case propagation time for a *summary-STP* value to reach the producer from the last consumer in the pipeline is equal to the time it takes for an item to be processed and be emitted by the application (*i.e.,*latency). This is due to the fact that as data items propagate forward in the processing pipeline, *summary-STP* values propagate one stage backwards on the same *put/get* operation.

One stability problem that we encounter is noise in the *summary-STP* values emitted by consumers. This results in non-smooth production rate for producer threads. Recall that the *summary-STP*, or the execution time for a task iteration run by a thread, is largely affected by the amount of resources (such as CPU) given to the thread by the underlying OS. Variances in the OS scheduling of threads result in variances in the execution time of task iterations run by these threads. We observe that consumer tasks intermittently emit large or small *summary-STP* values. Such noise can be smoothed out by applying *filters* also used by other feedback systems [21, 3, 5]. Filters to smooth *summary-STP* noise have currently not been implemented in ARU and is left for future work.

3.3.3. Assumptions The ARU algorithm is predicated on the following two assumptions:

- Threads always request the latest item from its input sources; and
- To achieve optimal performance, the application task graph is made available to the runtime system.

No additional application information is needed for the ARU algorithm. It is possible that application defined functions for computing the *summary-STP* values for each pipeline stage may lead to better performance and/or resource usage. However providing such knobs to the application increases programming complexity and hence is not considered in this study.

4. Implementation and Performance Evaluation Methodology

We have used the *Stampede* distributed programming environment as the test-bed for our ARU mechanism. Implemented in C, Stampede is available as a cluster programming library for a variety of platforms including x86-Linux and x86-WIN-NT. The implementation of ARU included adding a special API call (*periodicity_sync()*) to the Stampede runtime. This call computes the *current-STP* value for a specific thread. Each thread is required to call this function at the end of every thread iteration loop. In addition we modified the Stampede runtime to piggyback the *summarySTP* values on existing put/get calls to channels and queues. To allow the application to specify producer/consumer dependencies to the underlying ARU mechanism, a parameter was added to all channel/queue and thread creation APIs (*e.g.,. spd_chan_alloc()*). As described earlier, this graph dependency information is optional, and the default conservative *min* operator assumes no graph dependencies and allows producers to slow down to the faster consumer. Other user-defined dependency-encoded operators, such as the *max* operator, can be used to increase the aggressiveness of reducing wasted resource utilization.

A color-based people tracker application (figure 5) developed at Compaq CRL [25] is used to evaluate the performance benefit of the ARU algorithm. The tracker has five tasks that are interconnected via Stampede channels. Each task is executed by a Stampede thread. The application consists of (1) a *Digitizer* task that outputs digitized frames; (2) a *Motion Mask* or *Background* task that computes the difference between the background and the current image frame; (3) a *Histogram* task that constructs color histogram of the current image; (4) a *Target-Detection* task that analyzes each image for an object of interest using a color model; and (5) a *GUI* task that continually displays the tracking result. Note that there are two target-detection threads in figure 5 where each thread tracks a specific color model. The color-based people tracker application with its fairly sophisticated task-graph provides a realistic environment to explore the resource-savings made possible by the ARU algorithm.

The performance of the application is measured using the following metrics: *latency*, *throughput*, and *jitter*. Latency measures the time it takes an image to make a trip through the entire pipeline. Throughput is the number of successful frames processed every second. Jitter, a metric specifically suited for streaming applications, indicates the average change in the time difference between successive output frames. Mathematically, jitter is represented as the standard deviation of the time difference between successive output frames. Jitter therefore is a measure of the smoothness of the output frame rate or throughput.

The resource usage of the application is measured using the following metrics: *memory footprint*, *percentage wasted memory*, and *percentage wasted computation*. Memory footprint provides a measure of the memory pressure generated by the application. Intuitively, it is the integral over the application memory footprint graph (Figures 8,9). Mean memory footprint is the memory occupancy for all the items in various stages of processing in the different channels of the application pipeline averaged over time. The mean memory footprint is computed as: $MU_\mu = \Sigma(MU_{t_{i+1}} \times (t_{i+1} - t_i))/(t_N - t_0)$

Standard deviation of the memory footprint metric is a good indicator of the "smoothness" of the total memory consumption; the higher the deviation the higher the expected peak memory consumption by

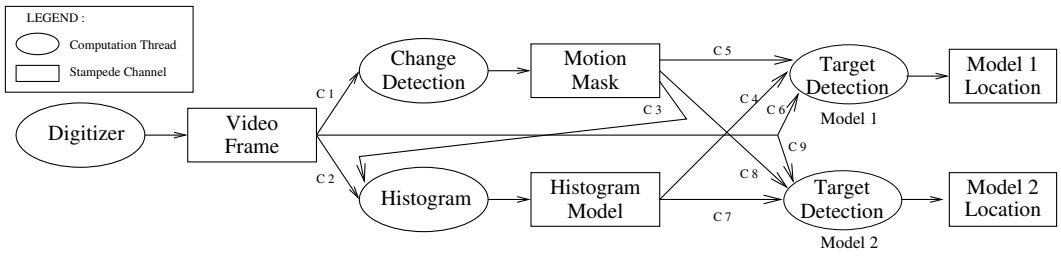

Figure 5. Color-based People Tracker application pipeline

the application. This metric is computed as: $MU_\sigma = \sqrt{\Sigma((MU_\mu - MU_{t_{i+1}})^2 \times (t_{i+1} - t_i))/(t_N - t_0)}$

Total computation is simply the work done (execution time) by all tasks in the different stages of the application pipeline (excluding blocking and sleep time). Correspondingly, wasted computation is the cumulative execution times spent on items that were dropped at some stage in the pipeline. Therefore, the *percentage wasted computation* is a ratio between the wasted computation and the total computation. Similarly, the *percentage memory wasted* represents the ratio between the wasted memory (integrated over time just as mean memory footprint) and the total memory usage of the application. These percentages are a direct measure of efficient resource usage in the application.

Please note that we do not directly account for the overhead of ARU in the metrics above. We consider the overhead to be negligible relative to the resources used by the application. For example, the *summary-STP* values that are piggy backed with each item are only 8 bytes long, very small compared to the size of each item (typically in the order of several hundred kilobytes). Also, the cost of computing the *summary-STP* value is minuscule. The computation involves a simple *min/max* operations on very small vectors (order *n*, where n is the number of output connections from a node). This computation is done only once at the end of each data production iteration by a thread, and at every put/get call on buffers.

We have an elaborate measurement infrastructure for recording these statistics in the Stampede runtime. Each interaction of an item with the operating system (*e.g.,*allocation, deallocation, *etc.*) is recorded. Items that do not make it to the end of the pipeline are marked to differentiate between wasted and successful memory and computations. A postmortem analysis program uses these statistics to derive the metrics of interest presented in this paper.

A number of garbage collection and scheduling strategies have been implemented and experimented within Stampede [18, 6, 10]. Among these techniques, the most resource saving is found in the *Dead Timestamp Garbage Collector* (DGC) [6]. DGC is based on *dead timestamp identification*, a unifying concept that simultaneously identifies both dead items (memory) and unnecessary computations (processing). Each node (be it a thread, a channel, or a queue) propagates information about locally dead items to neighboring nodes. These nodes use the information in turn to determine which items they can garbage collect.

The goals of ARU and Garbage Collection (GC) are orthogonal (section 2) as ARU tries to reduce wasted use of resources whereas GC tries to reclaim resources already used in the application. To use a GC mechanism in the Stampede runtime, we use our latest DGC algorithm and add the ARU mechanism to understand the extent of wasted resource reduction and subsequent performance improvement in applications due to ARU.

In an earlier work [14], we introduced an *Ideal Garbage Collector (IGC)* [14]. IGC gives a theoretical lower limit for the memory footprint by performing a postmortem analysis of the execution trace of an application. IGC simulates a GC that can eliminate all unnecessary computations (*i.e.,*computations on frames that do not make it all the way through the pipeline) and associated memory usage. Needless to say, IGC is not realizable in practice since it requires future knowledge of dropped frames. To determine how close the results are to the ideal, the ARU mechanism is compared to IGC.

5. Experimental Results

The hardware platform used is a 17 node cluster over Gigabit Ethernet of 8-way SMPs with 550MHz Intel Pentium III Xeon processors with 2MB L2 cache. Each SMP node has 3.69 *GB* of physical memory and is running Redhat Linux (2.4.20). All experiments reported in this section are done in two configurations. In configuration 1, a single physical node is used for all tasks, where each task is mapped to an individual thread. Every thread runs on its own address space. All global channels are allocated on this node as well. In configuration 2, five physical nodes are used with all five tasks mapped to distinct threads in turn running on separate nodes. Channels in this case are allocated on nodes where their corresponding producer threads are mapped. The data items emitted to channels by the

different threads are of the following sizes: Digitizer 738 *kB*, Background 246 *kB*, Histogram 981 *kB* and Target-Detection 68 *Bytes*.

The performance results given below are average statistics over successive execution runs of the tracker application.

5.1. Resources Usage

	Config 1: 1 node			Config 2: 5 nodes		
	Mem. use (MB)		% wrt	Mem. use (MB)		% wrt
	STD	mean	IGC	STD	mean	IGC
No ARU	4.31	33.62	387	6.41	36.81	341
ARU-min	2.58	16.23	187	2.94	15.72	145
ARU-max	0.49	12.45	143	0.37	13.09	121
IGC	0.33	8.69	100	0.33	10.81	100

Figure 6. **Memory Footprint for the tracker application in comparison with the Ideal Garbage Collector (IGC).**

	Config 1: 1 node		Config 2: 5 nodes	
	% of Mem. Wasted	% of Comp. Wasted	% of Mem. Wasted	% of Comp. Wasted
No ARU	66.0	25.2	60.7	24.4
ARU-min	4.1	2.8	7.2	4.0
ARU-max	0.3	0.2	4.8	2.1

Figure 7. **Wasted Memory Footprint and Wasted Computation Statistics for the tracker application.**

Memory Footprint: Figure 6 shows the mean memory footprint in megabytes when ARU is applied to the baseline tracker application. Recall that the mean memory footprint accounts for memory consumed by all items in application channels. The IGC row shows the theoretical limit for mean memory footprint with an ideal garbage collector. By eliminating wasted computations, ARU dramatically reduces the memory footprint the application requires, both in 1-node and 5-node configurations. In fact, results for the max operator are quite close to the ideal garbage collector. For example, in the 1-node configuration, ARU with the max operator reduces the mean memory footprint of the tracker by almost two-thirds when compared to the tracker footprint without the ARU mechanism. Figures 8 and 9 show the same data in a graphical form as a function of time. It provides a qualitative perspective, as all graphs are shown side by side and share the same axes. One can observe not only how close ARU is to IGC, but also how ARU reduces fluctuations in the application memory pressure over time.

Percentage of Wasted Resources: Figure 7 shows the amount of wasted memory and computation in the tracker application with and without ARU mechanism. When not using ARU, more than 60% of the memory footprint is wasted as opposed to only less than 5% wasted with the ARU-max operator. Substantial savings are visible for computation resources as well. Thus the ARU mechanism succeeds in directing almost all resources towards useful work.

5.2. Application Performance

	Throughput (fps)		Latency (ms)		Jitter (ms)
	mean	STD	mean	STD	
	Config 1: 1 node				
No ARU	3.30	0.02	661	23	77
ARU-min	4.68	0.09	594	9	34
ARU-max	4.18	0.10	350	7	46
	Config 2: 5 nodes				
No ARU	4.27	0.06	648	23	96
ARU-min	4.47	0.10	605	24	89
ARU-max	3.53	0.15	480	13	162

Figure 10. **Latency, Throughput and Jitter of the tracker**

In addition to reducing resource waste, the ARU mechanism also succeeds in improving application performance by decreasing jitter and latency, and increasing throughput (figure 10).

One can observe that even though ARU-max reduces latency compared to no ARU, it performs worse in terms of throughput (5 node configuration). This is not due to a high cost of the ARU mechanism, but is an artifact of the aggressiveness with which the *max* operator slows down producers to remove wasted resources. The less aggressive ARU-min mechanism manages to maintain a higher throughput at the expense of higher latency and greater resource usage.

The jitter caused by variations in the *summary-STP* causes jitter in the production rate as well. Due to the aggressive slowing of producers in ARU-max, coupled with the jitter in production, certain iterations of producer tasks are made slower than their consumers. This causes consumer threads to wait for data on buffers. Wait for consumer thread inadvertently decreases throughput for the application pipeline. However, as consumers are waiting for data in buffers, items never spend time in buffers themselves. This causes the observed reduced latency for ARU-max. It is clear from these results that a balance between aggressiveness of slowing producers and the amount

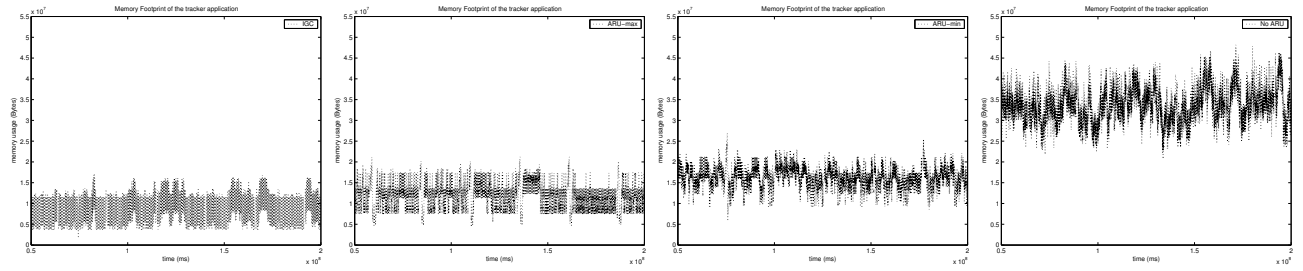

Figure 8. **Memory Footprint config. 1, single node: All graphs have the same scale. Y-axis: memory use (bytes x 10^7); X-axis: time (microseconds). (left to right)(a) Ideal Garbage Collector (IGC), (b) Dead-timestamp GC (DGC) with ARU - Max Operator, (c) DGC with ARU - Min Operator, (d) DGC without ARU.**

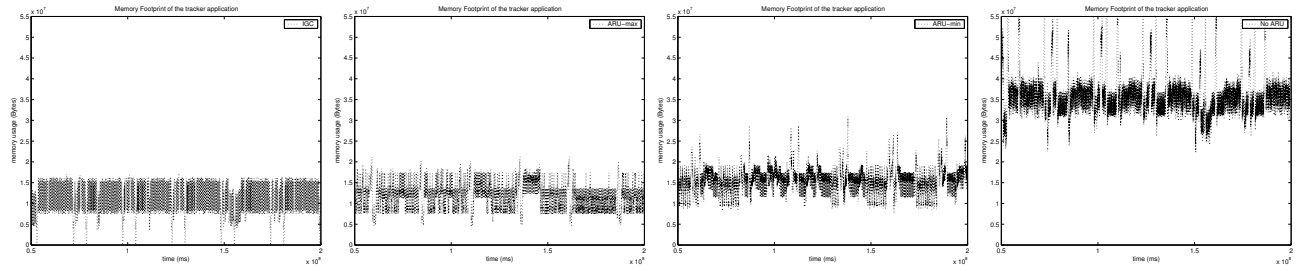

Figure 9. **Memory Footprint config. 2, five nodes: Scale and graph order same as for figure 8.**

resource usage needs to be maintained. We plan to explore this relationship further in future work.

6. Conclusion and Future Work

In this work we present an Adaptive Resource Utilization (ARU) mechanism that uses feedback to reduce wasted resource usage between a group of threads within an application. This mechanism is unique in its approach from similar work in traditional resource management, which have been dedicated to improving resource *allocation* to threads as oppose to incorporating application feedback on wasted production. Our mechanism targets parallel streaming multimedia applications that are computationally intensive and are designed to drop data when resources are insufficient so as to ensure the production of current output. With the ARU mechanism we show that dynamic adjustment of data production rate is a better approach than dropping data, since it is less wasteful of computational resources. Our ARU mechanism is implemented completely in user-space in the Stampede cluster programming framework. Using a color-based people tracker application, we show that the ARU algorithm achieves significant reduction in wasted resources in terms of both computation and memory while sustaining and even improving, application performance. The throughput increase in ARU-min clearly shows that ARU can improve performance. However, the ARU-max results show that being over aggressive saves more wasted resources and improves latency but at the expense of throughput. It is therefore important to find the right balance between wasted resource usage and application performance. Preliminary investigation indicates this is a viable avenue to pursue for future work.

References

[1] P. Ackermann. Direct manipulation of temporal structures in a multimedia application framework. In *International Conference on Multimedia*, pages 51–58. ACM Press, 1994.

[2] A. Bavier, L. Peterson, and D. Mosberger. Bert: A scheduler for best effort and realtime tasks. Technical Report TR-587-98, Princeton University, Aug. 1998.

[3] S. Cen. A software feedback toolkit and its applications in adaptive multimedia systems. Technical Report Ph.D. Thesis, Oregon Graduate Inst. of CS and Engr., Aug. 1997.

[4] S. Cen, C. Pu, R. Staehli, C. Cowan, and J. Walpole. A distributed real-time $mpeg$ video audio player. In *International Workshop on Network and Operating System Support for Digital Audio and Video*, pages 151–162, April 1995.

[5] A. Goel, D. Steere, C. Pu, and J. Walpole. Adaptive resource management via modular feedback control. Technical Report CSE-99-03, Oregon Graduate Institute of Computer Science and Engineering, Jan. 1999.

[6] N. Harel, H. A. Mandviwala, K. Knobe, and U. Ramachandran. Dead timestamp identification in stampede. In *Inter-*

national Conference on Parallel Processing, Vancouver, BC, Canada, Aug. 2002.

[7] J. R. Haritsa, M. Livny, and M. J. Carey. Earliest deadline scheduling for real-time database systems. In *IEEE Real-Time Systems Symposium*, pages 232–243, 1991.

[8] M. B. Jones, D. Rosu, and M.-C. Rosu. CPU reservations and time constraints: Efficient, predictable scheduling of independent activities. In *Symposium on Operating Systems Principles*, pages 198–211, Oct. 1997.

[9] R. Jones and R. Lins. *Garbage Collection : Algorithms for Automatic Dynamic Memory Management*. John Wiley, Aug. 1996. ISBN: 0471941484.

[10] K. Knobe, J. M. Rehg, A. Chauhan, R. S. Nikhil, and U. Ramachandran. Scheduling constrained dynamic applications on clusters. In *Proc. SC99: High Performance Networking and Computing Conf.*, Portland, OR, Nov. 1999.

[11] H. T. Kung. Why systolic architectures? *IEEE Computer*, 15(1):37–46, 1982.

[12] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[13] C. Lu, J. A. Stankovic, G. Tao, and S. H. Son. Design and evaluation of a feedback control edf scheduling algorithm. In *Real-Time Systems Symposium*, Dec. 1999.

[14] H. A. Mandviwala, N. Harel, K. Knobe, and U. Ramachandran. A comparative study of stampede garbage collection algorithms. In *The 15th Workshop on Languages and Compilers for Parallel Computing*, College Park, MD, July 2002.

[15] H. Massalin and C. Pu. Fine-grain adaptive scheduling using feedback. *Computing Systems*, 3(1):139–173, 1990.

[16] C. Mourlas. A framework for creating and playing distributed multimedia information systems with qos requirements. In *Symposium on Applied Computing*, pages 598–600. ACM Press, 2000.

[17] J. Nieh and M. S. Lam. A smart scheduler for multimedia applications. *ACM Transactions on Computer Systems*, 21(2):117–116, 2003.

[18] R. S. Nikhil and U. Ramachandran. Garbage Collection of Timestamped Data in Stampede. In *Symposium on Principles of Distributed Computing*, Portland, OR, July 2000.

[19] R. S. Nikhil, U. Ramachandran, J. M. Rehg, R. H. Halstead, Jr., C. F. Joerg, and L. Kontothanassis. Stampede: A programming system for emerging scalable interactive multimedia applications. In *Intl. Wkshp. on Languages and Compilers for Parallel Computing*, Chapel Hill, NC, Aug. 7-9 1998.

[20] D. S. Nikolopoulos and C. D. Polychronopoulos. Adaptive scheduling under memory constraints on non-dedicated computational farms. *Future Geneation Computer Systems*, 19(4):505–519, 2003.

[21] C. Pu and R. M. Fuhrer. Feedback-based scheduling: A toolbox approach. In *Proceedings of 4th Workshop on Workstation Operating Systems*, Oct. 1993.

[22] C. Pu, H. Massalin, and J. Loannidis. The synthesis kernel. *Computing Systems*, 1(1):11–32, Winter 1989.

[23] U. Ramachandran, R. Nikhil, J. M. Rehg, Y. Angelov, S. Adhikari, K. Mackenzie, N. Harel, and K. Knobe. Stampede: A cluster programming middleware for interactive stream-oriented applications. *IEEE Transactions on Parallel and Distributed Systems*, 2003.

[24] U. Ramachandran, R. S. Nikhil, N. Harel, J. M. Rehg, and K. Knobe. Space-Time Memory: A Parallel Programming Abstraction for Interactive Multimedia Applications. In *Principles and Practice of Parallel Programming*, Atlanta, GA, May 1999.

[25] J. M. Rehg, M. Loughlin, and K. Waters. Vision for a Smart Kiosk. In *Computer Vision and Pattern Recognition*, pages 690–696, San Juan, PR, June 17–19 1997.

[26] D. Revel, D. McNamee, C. Pu, D. Steere, and J. Walpole. Feedback-based dynamic proportion allocation for disk i/o. Technical Report CSE-99-001, Oregon Graduate Institute of Computer Science and Engineering, Jan. 1999.

[27] D. C. Steere, A. Goel, J. Gruenberg, D. McNamee, C. Pu, and J. Walpole. A feedback driven proportion allocator for real-rate scheduling. In *Symposium on Operating Systems Design and Implementation*, pages 145–158. Usenix, Feb. 1999.

[28] I. Stoica, H. Abdel-Wahab, and K. Jeffay. On the duality between resource reservation and proportional share resource allocation. In *Multimedia Computing and Networking*, number Volume 3020. SPIE Proceedings Series, Feb. 1997.

[29] V. Sundaram, A. Chandra, P. Goyal, P. Shenoy, J. Sahni, and H. Vin. Application performance in the qlinux multimedia operating system. In *International conference on Multimedia*, pages 127–136. ACM Press, 2000.

[30] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation*, pages 1–11, Nov. 1994.

[31] P. R. Wilson. Uniprocessor garbage collection techniques. In *Proc. Intl. Wkshp. on Memory Management*, pages 1–42, St. Malo, France, Sep. 1992.

[32] D. K. Y. Yau and S. S. Lam. Adaptive rate-controlled scheduling for multimedia applications. *IEEE/ACM Transactions on Networking*, 5(4):475–488, 1997.

[33] W. Zhao, K. Ramamritham, and J. A. Stankovic. Preemptive scheduling under time and resource constraints. *IEEE Transactions on Computers*, C-36(8):949–960, Aug. 1987.