Rate Control for Threads in Streaming Applications [*]

Hasnain A. Mandviwala [†]
mandvi@cc.gatech.edu,

Nissim Harel [†]
nissim@cc.gatech.edu,

Umakishore Ramachandran [†]
rama@cc.gatech.edu,

Kathleen Knobe [‡]
kath.knobe@hp.com

April 5, 2004

Abstract

A large emerging class of interactive multimedia streaming applications can be represented as a coarse-grain, pipelined, dataflow graph. Such applications are ideal candidates for execution on a high-performance cluster. Each node in the graph represents a task component of the streaming application, where each task is consuming data from preceding stages, and producing data for subsequent stages. The amount of data computation performed by a task, is dependent on a multitude of design issues such as task algorithm latency, data dependency, resource availability *etc.*One common characteristic of these applications is the use of current data: A task would obtain the latest data from preceding stages, skipping over older data if necessary to perform its computation. Such applications when parallelized, waste resources in terms of processing and memory on data that is eventually dropped from the application pipeline. To overcome this problem, we have designed and implemented a distributed *Rate Control* algorithm that dynamically adjusts the processing rate of each thread to meet application requirements. Optimizations incorporating application-level knowledge such as data-dependencies between producers and consumers, further improve performance. A color-based people tracker application is used to explore the performance benefits of the proposed Rate Control algorithm. We show that Rate Control reduces the application's memory footprint by 80% when compared to our previously published results. Optimizations further increase the application's throughput by 226%, and reduce latency by 40.5%.

[*]Submitted to ICPP 2004; 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.

[†]College of Computing, Georgia Institute of Technology, 801 Atlantic Drive, Atlanta, Georgia

[‡]HP Labs - Cambridge Research Laboratory, One Cambridge Center, Cambridge, Massachusetts

1 Introduction

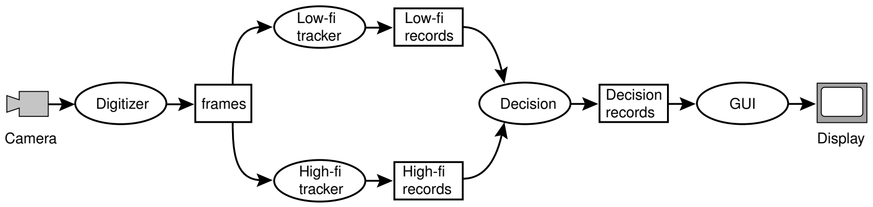

Figure 1: **Vision Application pipeline.**

There is an emerging class of streaming multimedia applications that are highly parallel and require system support and higher-level abstractions for improved performance and programming ease. Figure 1 illustrates an example of such a streaming vision application pipeline. The pipeline consists of independent tasks implemented by a single thread or a group of threads. These tasks are generally designed to contain a loop, where data is first consumed from a buffer, processed or mutated, and then emitted before allowing the task to loop again on new input (see figure 2). The structure of each task typically consists of repeatedly getting data from input buffers, processing the data, and producing data as a result of the processing (see figure 2). The tasks are independent of one another except for the input data dependencies pre-programmed in the task-graph, leading to differential production rates for the different tasks depending on the complexity of the processing.

To alleviate the problem of variable production/consumption rate along with the problem of non-FIFO and out-of-order access to items sometimes required in such application pipelines, tasks are connected by efficient, globally accessible, buffered communication abstractions. For example, *Stampede* [9, 11, 10] provides *Channels* and *Queues*, and APIs for performing I/O on these data abstractions.

Since tasks in successive pipeline stages do not have the same rate of consumption/production, a task may at times 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. To enable such behavior, systems such as Stampede provide tasks the ability to get the *latest* [9] item from an input channel.

Skipping of data may be important to keep the interactive nature of applications, but it is indirectly responsible for the performance degradation of applications. Since computation is used to produce all data, unused data reflects on computations wasted on producing it. The occurrence of wasted computation especially in a computationally demanding vision application is a cause of poor use of scarce resources, which in-turn causes a reduced application performance.

The best solution would be to never produced wasted items in the first place, saving both processing and memory resources. However, 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 the *rate* at which each task produces data. Such rate-control conserves resources and further improves application performance in resource-constrained environments.

In this paper, we propose a distributed Rate Control algorithm for threads in interactive streaming multimedia applications. The algorithm uses the dynamic production rate of each stage of the pipeline to give feedback to earlier stages. This feedback helps each stage adapt its own production rate to suit the dynamic needs of the application. Optimizations incorporating application-level knowledge such as data-dependencies between consumers and producers further improve application performance in terms of throughput and latency. We use a color-based people tracker application to explore the performance benefits of the proposed Rate Control algorithm. We show that Rate Control reduces the memory footprint by 80% [1] for this application compared to our previously published results. Further, we show that the optimizations increase the throughput by 226%, and reduce the latency by 40.5% [1].

In section 2 we review related work and compare them with Rate Control. In section 3 we describe the run-time instrumentation that enables the Rate Control 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 Rate Control mechanism we propose strives to optimize available resources to best meet application needs. Prima facie, this mechanism seems similar to the notion of guaranteed Quality of Service (QoS) in multimedia systems. However, there are some important differences. 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 Rate Control mechanism, requires only minimal instrumentation in the programming runtime environment 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 [15, 1, 13, 7]). However, Rate Control requires little involvement by the application writer even at the application-level. Rate Control also exploits the periodic nature of streaming applications, and simplifies the application adaptation to incorporate changes in resources availability. Functionally, Rate Control does not guarantee a specific level of quality of service. Instead, it makes sure that application task components execute at an equilibrium rate such that resources are not wasted on computations producing data that are eventually thrown away. Theoretically, we consider Rate Control to be orthogonal to QoS provisioning.

The Rate Control mechanism is complementary to the problem of Garbage Collection (GC) [14, 3] in general, and GC in streaming applications [8, 2, 6] in particular. Both Rate Control 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. But, while GC algorithms deal with resource reclamation, Rate Control directs the pace of data production to match available system resources and application pipeline constraints. It should be noted, however, that the rate control mechanism does not eliminate the need to deal with garbage created during execution, although it reduces the magnitude of the problem.

3 Rate Control

In this section we present our distributed Rate Control algorithm.

[1]On a resource constrained environment where all application threads were run on single physical node.

3.1 Factors Determining Rate of Tasks

Pipelined streaming applications such as the one illustrated in figure 1, have similarities with systolic architectures [5]. 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. Systems such as Stampede associate 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 [2], we proposed algorithms for eliminating upstream computations using the virtual times of timestamped data requests made by downstream threads. However, such techniques have shown limited success [2]. The cause described is such that in many interactive application pipelines, upstream threads (such as a digitizer) are 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. This earlier work did not make use of the processing rate of threads. If the 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. Essentially, the Rate Control algorithm proposed here does just that by creating a balanced processing pipeline by the judicious use of computational resources.

The Rate Control mechanism we describe next is in the context of the Stampede programming system that provides abstractions called channels and queues that hold timestamped data items. Stampede provides *put/get* API calls for accessing data items in these abstractions. The reader is referred to earlier publications [10] for further details.

3.3 Distributed Rate Control

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

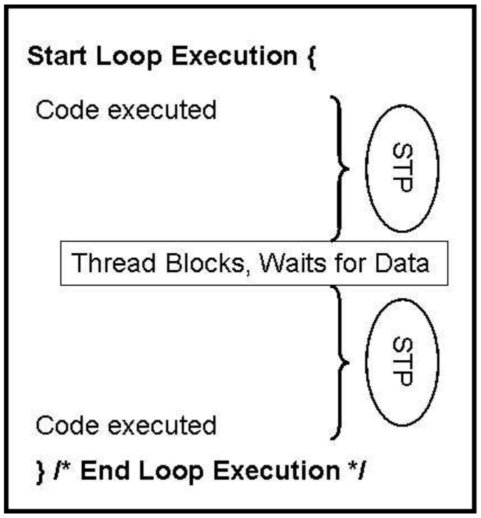

Figure 2: **Measuring the Sustainable Thread Period (STP)**

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 (see section 3.1) 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 STP value is used to compute the *summary STP* value described below, which is in-turn propagated back upstream to other tasks in the pipeline.

3.3.2 Computation of Summary STP and Backward Propagation

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

The summary STP value computation at a node can be a *min* or a *max* operation[2]. In the example shown in figure 3, node A has output connections to nodes B-F. The downstream nodes B-F report STP values of 337, 139, 273, 544, and 420, respectively. Consider the pipeline depicted

[2]A user-defined function can also be used here. It can be constructed with the knowledge of the application topology but is bounded in performance by the *min* and *max* functions

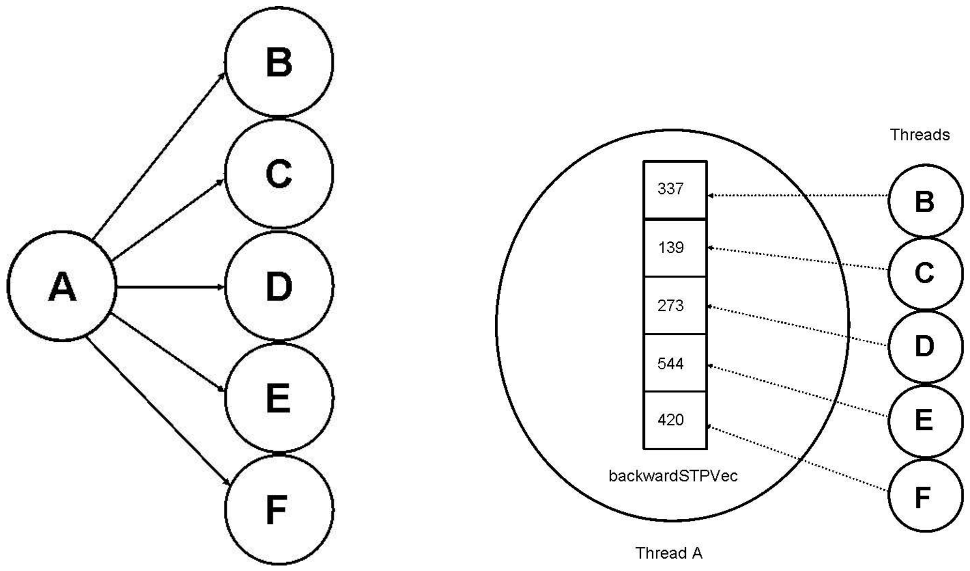

Figure 3: **STP propagation in the pipeline (left) using the backwardSTPVec (right)**

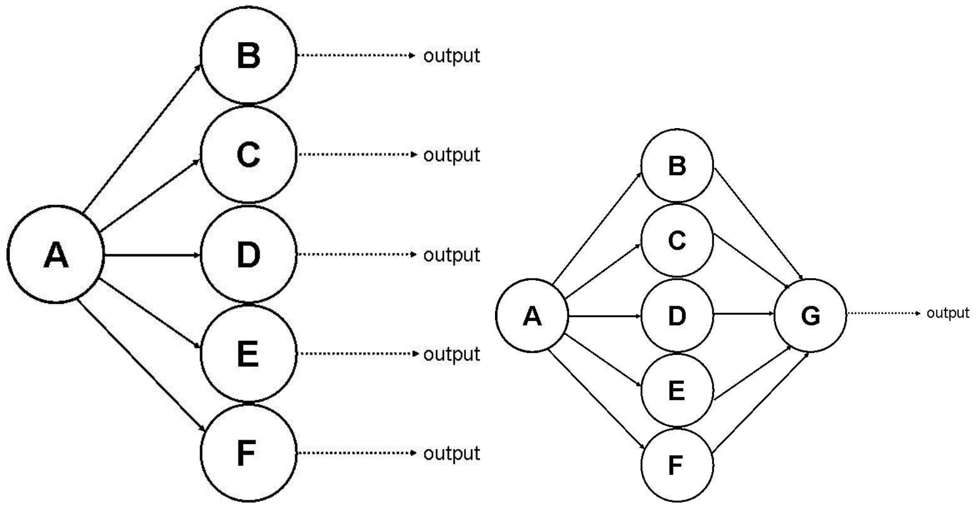

Figure 4: **Min (left) and Max (right) operators**

in the left half of figure 4, in which nodes B-F are end points of the computation. In this case, Node A uses a *min* operation to compute the summary STP to sustain the fastest consumer (C) that has the smallest STP. Consider the pipeline shown in the right half of figure 4. In this case, A is a thread connected to data abstractions represented by nodes B-F. Node A uses a *max* operation on its own STP value and those in the *backwardSTPVec* to compute the summary STP value for propagation.

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 information to adjust their rate of data item production. This cascading effect indirectly adjusts the production rate of all upstream threads.

Both the computation and propagation of STP values occur in a distributed manner in the pipeline, *i.e.,*the summary STP computation is completely local to a node, and STPs are exchanged with neighboring nodes by piggy-backing them on every *put/get* operation made to *channels* and *queues*.

3.4 Discussion

The Rate Control 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 Rate Control 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.

Note that the computation of summary STP values in a distributed manner has no implication on application correctness. It simply impacts the performance and usage of resources. The disparity between an *optimal* usage of resources predicted by an oracle and the *actual* achieved by the Rate Control algorithm depends on how quickly the summary STP values are propagated to neighbors. At the same time, it is essential in such interactive applications that the performance (*e.g.,*delivered frame rate in a vision pipeline) is maximized commensurate with resource availability. For example, a temporary increase in load may result in a large STP for a particular stage of the pipeline. So it is imperative that communication of STP values among neighbors happen often enough that the performance does not become sub-optimal. However, aggressive communication of STP values leads to overhead that can also hurt performance.

In our Rate Control algorithm, we assume that propagation only occurs on *put/get* operations. Thus resource usage and performance are expected to be slightly worse than in an optimal situation which can be discerned by an oracle.

4 Implementation and Performance Evaluation

We have used the *Stampede* distributed programming environment as the test-bed for our Rate Control algorithm deployment. Implemented in C, Stampede is available as a cluster program-

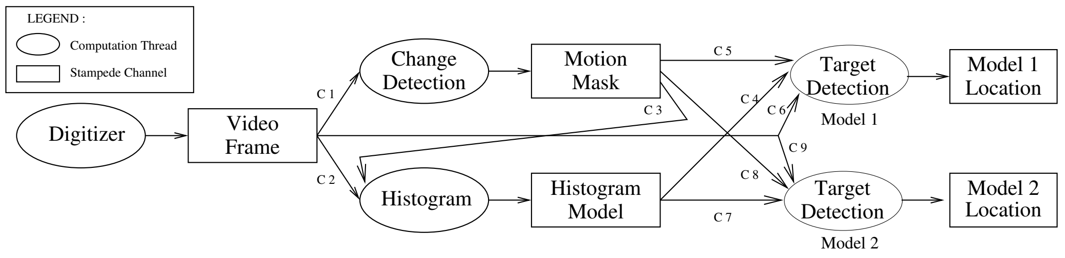

Figure 5: Color-based People **Tracker** application pipeline

ming library for a variety of platforms including x86-Linux and x86-WIN-NT. We modified the Stampede runtime to compute and piggy-back STP values on API calls to channels and queues.

A color-based people tracker application (figure 5) developed at Compaq CRL [12] is used to evaluate the performance benefit of the Rate Control 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 *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 Rate Control algorithm.

The performance of the application is measured using the following metrics: *latency*, *throughput*, and *jitter*. Latency measures the time for 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 time difference between successive output frames. The average jitter and standard deviation are measures of the smoothness of the output frame rate.

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 (Figure 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 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 by all the stages of the entire pipeline; it is simply the cumulative execution times of all the pipeline stages the application (excluding sleep time). Correspondingly, wasted computation is the cumulative execution times on items that did not make their way through the entire 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.

We have an elaborate measurement infrastructure for recording these statistics in the Stampede runtime. A post mortem 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 with in Stampede [8, 2, 4]. Among these techniques, the most resource saving is found in the *Dead Timestamp Garbage Collector* (DGC) [2]. DGC is based on *dead timestamp identification*, a unifying concept that simultaneously identifies both dead items (memory) and unnecessary computations (processing). We use DGC as the baseline and add Rate Control to that scheme to understand the performance improvement due to Rate Control.

In an earlier work [6], we introduced an *Ideal Garbage Collector (IGC)* [6]. IGC gives a theoretical lower limit for the memory footprint by performing a post mortem analysis of the execution to eliminating 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. To determine how close the results are to the ideal, the Rate Control algorithm is compared to IGC.

5 Experimental Results

The hardware platform is a cluster of 8-way SMPs (500MHz Intel Pentium III Xeon processors) running Redhat Linux (Linux Kernel 2.4.20). The machines are interconnected by gigabit Ethernet. Two different configurations were used for mapping application tasks to physical nodes. In configuration 1, all tasks were mapped to a single physical node, whereas in configuration 2, all five tasks were mapped to distinct physical nodes. Each item emitted by the different pipeline stages are of the following sizes: Digitizer 738 *kB*, Background 246 *kB*, Histogram 981 *kB* and Target-Detection 68 *Bytes*.

5.1 Application Performance

	$Config\ 1:\ 1\ node$			$Config\ 2:\ 5\ nodes$		
	$Latency$ $(usec)$ $mean$	$Latency$ $(usec)$ STD	$Throughput$ $(frames/sec)$ $mean$	$Latency$ $(usec)$ $mean$	$Latency$ $(usec)$ STD	$Throughput$ $(frames/sec)$ $mean$
$DGC\ w/o\ RC$	671,721	57,665	2.71	940,529	207,917	3.37
$DGC\ w/\ RC-min$	665,607	78,967	2.84	951,509	106,250	3.46
$DGC\ w/\ RC-max$	272,145	2,731	6.13	1,000,886	120,230	3.19

Figure 6: **Latency and Throughput** of the Color-based People tracker application.

Latency and Throughput: Figure 6 summarizes the latency and throughput results for the tracker. For single node configuration (configuration 1), the *max* operator applied at each node of the task graph results in the best average latency. The *min* operator does not significantly help in reducing the average latency compared to the baseline DGC. The throughput more than doubles with the max operator compared to the baseline DGC. Note that each node is an 8-way SMP. Thus in principle, one could assume that each task would get its own processor to execute on. Thus the improvement in latency and throughput is mostly from lessening the load on the memory subsystem due to the Rate Control mechanism.

The 5-node configuration (configuration 2) does not offer any advantage to the Rate Control algorithm [3]. The reason is quite simple: As can be seen from figure 6, the baseline DGC performance for the 5-node configuration is slightly worse compared to the 1-node version. This increase is due to the network communication costs when the tasks are distributed to different nodes. There is adequate processing and memory bandwidth in each node since only a single task is mapped to each node. The bottleneck for the pipeline is the tracker task with the data in and out of that task across the network. Thus quenching some of the eager tasks (such as the digitizer) from producing more items does not help either the latency or throughput. This experiment demonstrates that as far as latency and throughput are concerned, Rate Control is meaningful only in a resource constrained setting.

	$Config\ 1:\ 1\ node$		$Config\ 2:\ 5\ nodes$	
	$Jitter$ $(usec/frame)$ $mean$	$Jitter$ $(usec/frame)$ STD	$Jitter$ $(usec/frame)$ $mean$	$Jitter$ $(usec/frame)$ STD
$DGC\ w/o\ RC$	368,149	30,423	319,281	197,497
$DGC\ w/\ RC - min$	349,578	43,632	286,581	10,707
$DGC\ w/\ RC - max$	162,578	3,497	293,999	11,360

Figure 7: **Jitter** of the Color-based People tracker application. Jitter is the time difference between two successful output frames.

Jitter: Figure 7 shows the jitter results for the application. The Standard Deviation (STD) is dramatically reduced for the max-operator in both configurations. This illustrates the effectiveness of Rate Control for smoothening the flow of stream data.

5.2 Resources Usage

Next we analyze the resource usage of the Rate Control mechanism in comparison to the baseline DGC as well as IGC.

Memory Footprint: Figure 8 shows the mean memory footprint in bytes when Rate Control is applied to baseline DGC. Recall that the mean memory footprint is the amount of memory consumed by all items in the various channels in the pipeline averaged over the occupancy time for the items. The IGC row shows the theoretical limit for mean memory footprint with an ideal garbage collector. Figure 9 shows the same data in a graphical form as a function of time. By

[3]Actually, the min operator does slightly better than the baseline DGC, and the max operator slightly worse than the baseline.

	Config 1 : 1 node			*Config 2 : 5 nodes*		
	Memory usage (B) mean	*% w.r.t. IGC*	*Memory usage (B) STD*	*Memory Usage (B) mean*	*% w.r.t. IGC*	*Memory usage (B) STD*
DGC w/o RC	19,027,447	715	2,474,005	41,678,704	371	17,069,621
DGC w/ RC − min	4,335,774	162	733,551	17,432,479	155	2,290,592
DGC w/ RC − max	3,717,205	139	480,643	17,243,738	153	2,091,459
IGC	2,662,321	100	927,039	11,219,351	100	2,406,524

Figure 8: **Memory Footprint Statistics** for the Color-based People tracker application using Rate Control in comparison with memory footprint performance with the Ideal Garbage Collector (IGC).

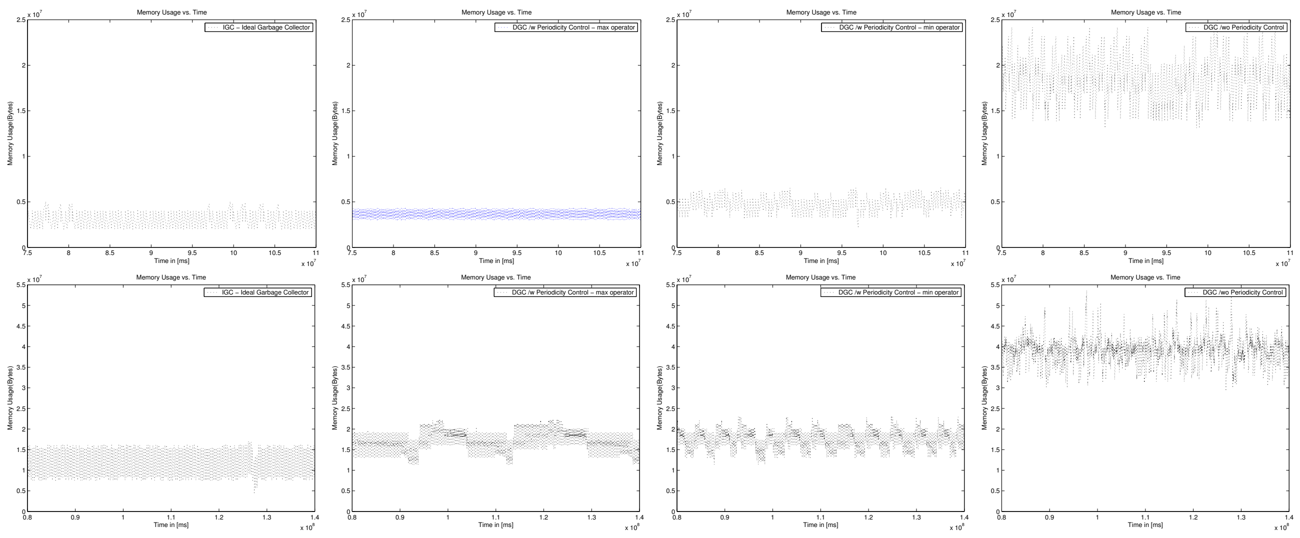

Figure 9: **Memory Footprint** (Top: single node config 1, Bottom: five node config 2) The four graphs represent the memory footprint of the application under different runtime implementations. (left to right)(a) Ideal Garbage Collector (IGC), (b) DGC with Rate Control - Max Operator, (c) DGC with Rate Control - Min Operator, (d) Dead Timestamp Garbage Collector (DGC) . All graphs have the same scale, with the y-axis showing memory use (bytes x 10^7), and the x-axis representing time (microseconds).

eliminating wasted computations, Rate Control dramatically reduces the memory footprint needed for this application in both the 1-node and 5-node configurations. In fact, the results for the max operator are quite close to the ideal garbage collector. For example, for 1-node configuration, Rate Control with the max operator reduces the mean memory footprint by 80.5% compared to baseline DGC; and the memory footprint is just 39% over IGC.

Percentage of Wasted Resources: Figure 10 shows the amount of wasted memory and computation with and without Rate Control over the baseline DGC. For baseline DGC, nearly 85% of the memory footprint is wasted for the 1-node configuration; contrast this with the max operator Rate Control where only 5.4% is wasted. Similar spectacular savings are accrued for the computation resource as well. Thus the Rate Control mechanism succeeded in directing most of the resources towards useful work.

	$Config\ 1: 1\ node$		$Config\ 2: 5\ nodes$	
	$\%of$ $Memory$ $Wasted$	$\%of$ $Computation$ $Wasted$	$\%of$ $Memory$ $Wasted$	$\%of$ $Computation$ $Wasted$
$DGC\ w/o\ RC$	84.94	37.85	73.20	32.63
$DGC\ w/\ RC - min$	13.30	3.10	6.35	2.11
$DGC\ w/\ RC - max$	5.39	1.37	1.97	0.77

Figure 10: **Wasted Memory Footprint and Computation Statistics** for the Color-based People tracker application using Rate Control.

6 Conclusion

Interactive multimedia applications are computationally intensive and are good candidates for execution on high performance computing engines. Such applications can usually be represented as a coarse-grain dataflow pipeline. To ensure the production of fresh output, these applications are designed to drop data when resources become insufficient. Dynamic adjustment of data production rate is a better approach than dropping data, since it is less wasteful of computational resources, and in-turn leads to better performance. We have presented a dynamic distributed Rate Control algorithm for dealing with the variances in application processing times as well as the vagaries of operating system scheduling. We have implemented this algorithm in the Stampede cluster programming framework. Using a color-based people tracker application, we show that the Rate Control algorithm achieves significant reduction in wasted resources in terms of both computation and memory.

7 Acknowledgments

A number of people have contributed to the Stampede project. Rishiyur Nikhil and Jim Rehg contributed to the space-time memory abstraction. Bert Halstead, Chris Jeorg, Leonidas Kontothanassis, and Jamey Hicks contributed during the early stages of the project. Members of the "Ubiquitous Presence" group at Georgia Tech continue to contribute to the project. Sameer Adhikari, Arnab Paul, Rajnish Kumar, Bikash Agarwalla, Matt Wolenetz, Phil Hutto, Namgeun Jeong, Junsuk Shin, Yavor Angelov, and Xiang Song deserve special mention. Ansley Post, Durga Devi Mannaru, Russ Keldorph and Anand Lakshminarayanan participated in aspects of the Stampede project during their tenure at Georgia Tech.

References

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

[2] N. Harel, H. A. Mandviwala, K. Knobe, and U. Ramachandran. Dead timestamp identification in stampede. In *The 2002 International Conference on Parallel Processing (ICPP-02)*, Vancouver, BC, Canada, August 2002.

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

[4] 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, November 1999. Technical paper.

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

[6] 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.

[7] C. Mourlas. A framework for creating and playing distributed multimedia information systems with qos requirements. In *Proceedings of the 2000 ACM symposium on Applied computing*, pages 598–600. ACM Press, 2000.

[8] R. S. Nikhil and U. Ramachandran. Garbage Collection of Timestamped Data in Stampede. In *Proc. Nineteenth Annual Symposium on Principles of Distributed Computing (PODC 2000)*, Portlan, Oregon, July 2000.

[9] 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 *Proc. Eleventh Intl. Wkshp. on Languages and Compilers for Parallel Computing (LCPC 98)*, Chapel Hill, NC, August 7-9 1998.

[10] 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. To Appear.

[11] 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 *Proc. Principles and Practice of Parallel Programming (PPoPP'99)*, Atlanta, GA, May 1999.

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

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

[14] P. R. Wilson. Uniprocessor garbage collection techniques, Yves Bekkers and Jacques Cohen (eds.). In *Intl. Wkshp. on Memory Management (IWMM 92)*, pages 1–42, St. Malo, France, September 1992.

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