MEMORY OPTIMIZATIONS FOR DISTRIBUTED STREAM-BASED APPLICATIONS

A Thesis
Presented to
The Academic Faculty

by

Nissim Harel

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
College of Computing

Georgia Institute of Technology
December 2006

MEMORY OPTIMIZATIONS FOR DISTRIBUTED STREAM-BASED APPLICATIONS

Approved by:

Umakishore Ramachandran, Advisor
College of Computing
Georgia Institute of Technology

Kathleen Knobe
Programming Systems Lab
Intel Corporation

Mustaque Ahamad
College of Computing
Georgia Institute of Technology

Milos Prvulovic
College of Computing
Georgia Institute of Technology

Richard M. Fujimoto
College of Computing
Georgia Institute of Technology

Date Approved: 19 October 2006

To my parents, Edna and Haim.

To my sister, Iris, and my brother, Igi.

ACKNOWLEDGEMENTS

Although this dissertation bears the name of a single author, it would not have been possible without the aid of many people. I would like to take this opportunity to express my gratitude to those individuals that helped me complete this work.

First and foremost I would like to thank my advisor, Prof. Umakishore Ramachandran, for the optimism, enthusiasm, guidance, and the help he provided me and for the patience he showed through the many (too many...) years I spent at Georgia Tech as a graduate student. Without Prof. Ramachandran's continuous support it would have been impossible to arrive at this point. I would also like to express my deep appreciation to Vasanthi Ramachandran for the constant encouragement, and the many gatherings with all the delicious food. Yes, I will get married soon (I hope).

I worked very closely with Dr. Kath Knobe. Her insight and advice were invaluable to my work. Over the past eight years she has been a tremendous support to me both professionally and personally. I will always treasure her friendship.

I would also like to express my gratitude to the thesis committee members, Prof. Mustaque Ahamad, Prof. Richard Fujimoto, and Prof. Milos Prvulovic, who provided me with extremely helpful advice that greatly improved this work.

Special thanks to Hasnain Mandviwala with whom I worked closely over the past six years. It is rare to find such an enthusiastic colleague and an original thinker. I enjoyed every minute we worked together. This work would not have been possible without his input, for which I truly owe him a debt of gratitude.

Mrs. Smith (sometimes simply called Marci) helped me tremendously in translating my ideas into English and removing all the extra commas. This dissertation would have been incomprehensible (and with too many commas) without her. Special thanks to Tim and Joshua Smith for their friendship and patience while she was helping me.

I thank Viji Periapoilan, a true Tamil Pen, for her advice during this work, the introduction to the intricacies of the *pu* endings in the Tamil language, and the special cardamom tea that saved me from constant coughing during the springtime in Georgia.

I would like to express my deepest thanks to Dr. Vivekanand Vellanki, Dr. Paul Arnab, Dr. Sameer Adhikari, Namgeun Jeong, and Russ Keldorph for the pleasure of being their colleague and working closely with them.

I also would like to thank my workmates at the CERCS lab: Bikash Agarwalla, Yavor Angelov, Dr. Rajnish Kumar, Anand Lakshminarayanan, Toby Reyelts, and Dr. Matt Wolenetz.

During the years I spent as a graduate student, I also co-founded a company named ClickFox. On the one hand, this slowed the work on my dissertation. On the other hand, I met a talented group of people that also became my friends and my family. I would like to use this opportunity to thank Joe Carter, Kapil Chandra, Elizabeth Clause, Mike Greenberg, Al Hart, Irina and Vadim Kulya, Beth Leitch, Sharon Lynch, Pravin Mane, Keith Manning, Paul Molitor, Gayatri Ratnaparkhi, Sandra Schlosser, Dr. Teruyuki Shikano, David Stahl, Isabel Twyeffort, and Tai Toson.

Special thanks to Dr. Vijaykumar Krishnaswamy for his friendship through the Georgia Tech and the ClickFox years. I felt I could always count on his creativity and innovation whenever I thought we had reached a dead end. His enthusiasm towards abstract concepts (such as, specific variants of an operating system) is inspiring.

Dr. Dean Jerding, whom I met at ClickFox, has always been a very good friend and I admire his creativity, methodical thinking, organization and his calm demeanor while working through the many problems that are part of life at a startup company.

Thanks to Ami Feinstein for being both a great friend and a great person to work with. Special thanks for the jokes!

Sincere thanks to Srihari Govindharaj who is more than a colleague, he is a real brother, and a man with a lot of soul. Thanks for the valuable lesson that there is nothing that cannot be recycled.

I cherish Michael Chavez's friendship and support especially through difficult times. As

a token of my friendship I promise not set foot in a Celine Dion concert.

Thanks to Spencer Rugaber for his good advice, professionalism, and his ability to think out of the box.

Thanks to Nisim Gadot, Patricia Oliver, and to Ofra and Guy Tessler for the care they have showed me.

Thanks to Haim Alon for his tremendous help, his generosity, and his friendship through the many years I have known him.

Thanks to Prof. Jihad El-Sana for so gracefully showing how intelligence and a good heart go hand in hand. I wish I could look at the number of papers you have already written as a guide, but I am afraid it would be too much work!

Thanks also to Prof. Ofer Arieli for his friendship, his kindness, and steadfast support.

Thanks to Doug Britton, Selcuk Cimtalay, Wayne Daley, Stefan and Shannon Dancila, and to Monique Hossain for their friendship.

Many thanks to Oriya, Tav, Yif'a, Neri, Aharon-Hilel, Shir, and Yehuda for all the love they have given me and for reminding me how to play during hard times.

With enormous gratitude I would like to thank Tal and Elizabeth Cohen for being my family and providing me a second (and sometimes a first) home for the past eleven years.

It is always good to know that whatever happens, you always have a place to call home, and my sister Iris and her husband Tuvia are continuously providing me with such a place.

There are no words to express my gratitude to my brother, Igi, for his never-ending support, good advice, and his ability to have fun regardless of the circumstances. I feel extremely fortunate to have both him and Iris as my brother and sister.

Finally, thanks to my parents, Edna and Haim, for making all of this possible.

As one can see, it takes a village to write a dissertation, and I am blessed to be part of such a village!

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Distributed stream-based applications manage large quantities of data and, therefore, have large memory requirements. On the other hand, this class of applications has specific properties and exhibits unique production and consumption patterns that set these applications apart from general-purpose applications. This dissertation examines possible ways to harness the unique characteristics of stream-based applications to assist in creating efficient memory management schemes.

In particular, this dissertation looks at the memory reclamation problem. It takes advantage of special traits of streaming applications to extend the definition of the garbage collection problem for those applications and include not only data items that are not "reachable" but also data items that have no effect on the final outcome of the application. Streaming applications typically fully process only a portion of the data, and resources directed towards the remaining data items, that is data items that do not affect the final outcome, can be viewed as wasted resources that should be minimized. Two complementary approaches are suggested:

- Garbage Identification.
- Adaptive Resource Utilization.

Garbage Identification is concerned with an analysis of dynamic data dependencies to infer those items that the application is no longer going to access. Several garbage identification algorithms are developed and examined. Each one of the algorithms uses a set of application properties (possibly distinct from one another) to reduce the memory consumption of the application:

- The Dead timestamps based Garbage Collector algorithm (DGC) combines garbage identification and computation elimination. The DGC algorithm locally identifies dead items, that is, items that are of no interest to any of the application threads

and thus, will not be requested in the future. This information is then propagated to neighboring nodes.

- Additional GC algorithms for streaming applications (Keep Latest 'n Unseen, Propagation of Dead Sets, Out-of-Band Propagation of Guarantees, and Out-of-Band Propagation of Dead Sets) are also presented. These algorithms enable the runtime system to achieve a tighter control on memory used by streaming applications. They explore the potential benefits of providing more information to the runtime system either directly from the application writer or by disseminating information more efficiently within the runtime system.

The Adaptive Resource Utilization (ARU) algorithm predicts the capacity of the system to process data items. It attempts to adjust the rate new data items are introduced to the application in order to match the processing capacity of system. In that sense, ARU extends the definition of wasted data items to include those items that would be identified as garbage had they been produced. The ARU algorithm infers the capacity of the system by analyzing the dynamic relationships between the production and consumption of data items. It then adjusts the data generation accordingly. Thus, the ARU algorithm makes local capacity decisions based on global information.

This dissertation also presents a methodology to explore the design space of distributed software environments. The methodology involves expressing the attributes of the design space as "what-if" scenarios. The scenarios are then simulated and evaluated for their effect on the system's performance. The use of "what-if" scenarios makes it possible to define and express an ideal system. This ideal system serves as a reference point that helps in assessing how well implemented or simulated systems perform. The simulations are dependent on accurate measurements of events that are performed in a distributed environment. To support the simulations, a distributed and cycle-accurate event-logging measurement infrastructure is presented.

In this dissertation, the methodology and the measurement infrastructure are used to assess the performance of garbage identification algorithms. An Ideal Garbage Collector

is defined and used as a reference point to assess implemented (REF, TGC, and DGC) and proposed (OBPG, KLnU, PDS, and OBPDS) garbage identification algorithms. The proposed garbage identification algorithms are simulated and the best garbage collector is implemented. This work also serves as a case study that evaluates the validity of this methodology.

The results indicate that the garbage identification algorithms that achieve a low memory footprint (close to that of an ideal garbage collector) perform their garbage identification decisions locally; however, they base these decisions on best-effort global information obtained from other components of the distributed application.

The ARU algorithm is found to be as effective as the most successful garbage identification algorithm in reducing the memory footprint of stream-based applications, thus confirming the previous observation that using global information to perform local decisions is fundamental in reducing memory consumption of stream-based applications.

CHAPTER I

INTRODUCTION

1.1 Motivation

1.1.1 The Emergence of Streaming Applications

The physical and economic feasibility of capturing and processing a large number of data streams, from different sources in real-time, is a result of the confluence of various economic and technological trends: hardware has become less expensive, input and output devices smaller, processors faster, and memory and storage devices denser. These advances make it possible to utilize a smaller physical space while installing a larger number of different types of sensors (e.g., cameras and microphones in a lobby), and to perform complex tasks on the data they capture (e.g., tracking people in real-time). These new capabilities make it possible to develop and deploy a new class of applications, called *streaming applications*. These applications process data that is structured and processed in a continuous flow. The information, only partially available or too large to be fully delivered in a single installment, is broken into a stream of smaller packets of information that are interpreted, rendered, and processed as the packets arrive to the relevant application component.

1.1.2 Unique Requirements of Streaming Applications

Broadly speaking, streaming applications are organized as a series or a pipeline of tasks processing streams of data, e.g., starting with sequences of camera images, extracting higher and higher-level "features" and "events" at each stage, and eventually responding with outputs. The applications tend to involve the processing of large sets of different types of streaming sources of information at near real-time. Although the amount of computing power we currently have still allows us to process only a fraction of all the data captured, in many cases it is enough to get us sufficiently close to the desired result (i.e., what we would have achieved had we processed all the data). The reason lies with the fact that streaming applications try to attach a meaning to the information they acquire. The goal is not to

fully process all the data captured, but rather extract a specific meaning from the data.

The environments where streaming applications are deployed often require the extractions of multiple meanings in tandem that necessitate the running of many applications concurrently in the same environment.

1.1.3 Augmented Living: An Example of a Streaming Application Environment

Consider, as an example of a streaming application environment, the challenge of providing an augmented residence for the elderly population. This living environment enables seniors to remain independent in their homes thus delaying their placement in a care facility [34]. There is no single algorithm or a single task that can fully support such an augmented environment; rather, there is a need to build a system that continuously performs many different functions in parallel. Some of these functions are: video monitoring, health monitors (pulse, skin temperature, blood pressure, respiration, movement, gait, weight, posture, gestures), fall detection, and motion detection.

These environments are expected to acquire information about the activities and the well being of the occupants from a myriad of sensors (e.g., cameras, microphones, pressure mats, and biosensors). These environments are also forced to process data at different rates and different resolution levels due to the inherent nature of the signals and the sensors measuring them. Once these data streams are processed, the system can immediately recognize one or more crisis situations and act upon them accordingly. In addition, if no crisis situation is recognized, the system can assist in daily routines, as well as continue to monitor the health and the well being of the individual.

1.1.4 Gait Analysis: An Example of a Single Streaming Application

In addition to having particular requirements when working in tandem, each of the individual applications possess unique traits that set them apart from general-purpose applications. We can demonstrate these traits by looking at a single streaming application that performs gait analysis. This is one of the many applications required to support an augmented living environment.

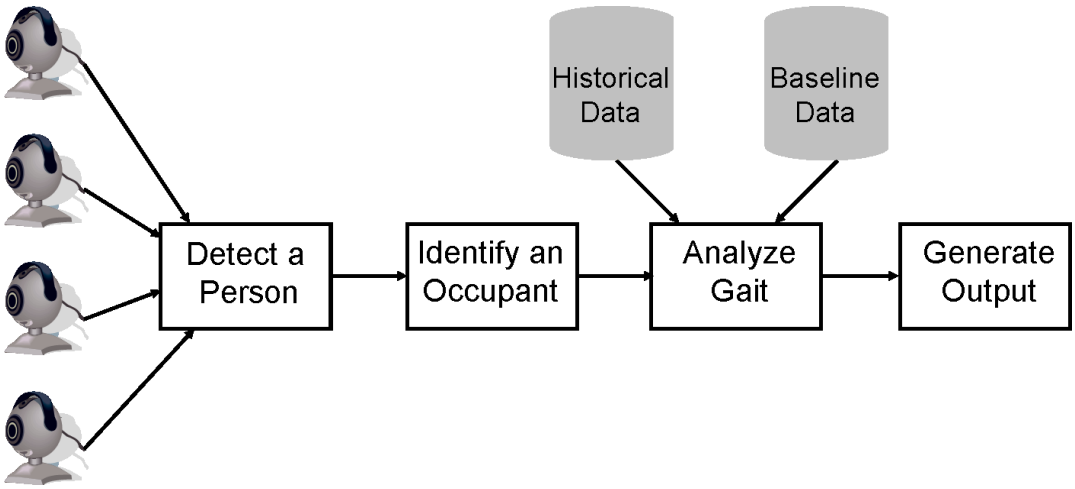

Figure 1: Gait Analysis Application: Cameras provide a continuous stream of images from various scenes to a module that looks for the presence of a person in the frames. If the system discovers the presence of a person, the information is passed to a module responsible for identifying a specific occupant. Once an occupant is identified, their gait is analyzed and compared to a baseline database and to their personal historical data. The system then generates an output and/or takes actions based on the analysis results. Typically, only a small portion of the images captured are fully processed. For example, if a person is not detected in a frame, there is no reason to pass this information to the module responsible for identifying an occupant.

A gait analysis application examines the movements and gait to detect advances in chronic conditions (e.g., Parkinson's disease, arthritis) as well as detect symptoms of critical conditions (e.g., heart attack, stroke). By placing cameras in strategic locations throughout the environment, images of the occupant walking would be captured and their gait analyzed. The system must first detect the presence of a person and determine if the person detected is one of the designated occupants. Once the system recognizes an occupant it begins analyzing the gait captured and compares it to previously recorded gaits for that occupant, and also to gait patterns of healthy people as well as to those gait patterns indicative of diseases. If, based on the gait, the system detects an emergency, it will send out the proper alerts. If one of the occupants' gait reflects deterioration in the occupant's condition, the system would assess the level of deterioration, notify the health care provider, and/or recommend appropriate treatment. Note that a large number of the images captured may not have a person present in the frame and thus will not be fully processed. Moreover, it is possible to fully assess the occupant's condition by processing only a fraction of the captured images.

Figure 1 illustrates a possible implementation of the gait analysis application. Multiple cameras capture concurrent streams of images. The system analyzes the image streams continuously seeking the presence of a person. Once a person is detected the system analyzes the relevant data streams in order to properly identify the occupant or occupants. Once occupants have been identified, the system then analyzes the gait based on the data streams related to each occupant, historical data stored, and baseline data. Lastly, the system generates reports, alerts, and/or suggests treatment.

1.1.5 Challenges in Supporting Streaming Applications

The gait analysis application (described in Chapter 1.1.4) illustrates how the class of streaming applications displays dynamic communication characteristics while being computationally demanding. The gait analysis application cannot determine *a priori* the set of cameras with relevant information. As the information required for a specific computation is constantly changing, dynamic communication patterns emerge among the processes responsible for identifying an occupant. The task of identifying an occupant is, in itself, complex, and computationally demanding. Further, once an occupant has been identified, the initial gait analysis determines the specific algorithms that should be deployed in order to focus the remaining analysis on the relevant outcomes. These algorithms exhibit different levels of complexity and bring into play different data elements to perform the analysis. Once more, the specific algorithms and what data these algorithms require are not known *a priori*; rather, they are based on external events that are, by nature, dynamic.

Gait analysis is only one of the many streaming applications that are needed to provide an augmented living environment for the elderly. In general, many of the streaming applications exhibit similar characteristics. These similarities create an opportunity to manage streaming applications differently than general-purpose applications, thus potentially improving the efficiency of the implementation. Specifically, memory and buffer management, the focus of this dissertation, play a vital role in the efficiency of streaming applications due to the large amount of data these applications are required to handle.

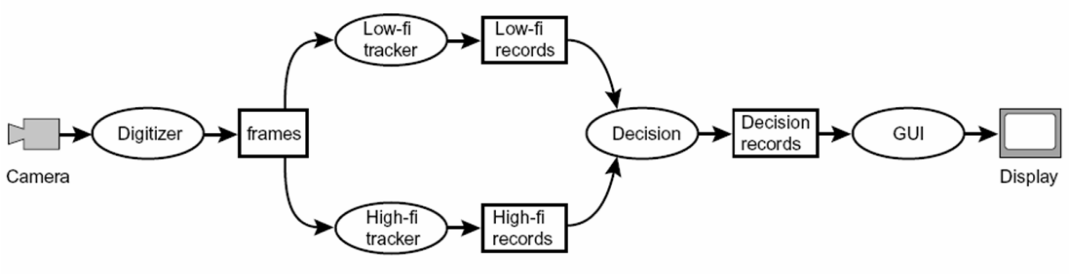

Figure 2: A Vision Application Pipeline: This application locates a specific object in a scene. The ovals in the figure represent computing modules, and the rectangles represent buffers, where intermediate data is temporarily stored before being requested by the relevant computing module. The application receives a stream of images from a single camera. The frames are then passed to two modules, a low-fidelity tracker and a high-fidelity tracker, that look for the presence of the specific object in an image. The results of the analysis are then passed to a decision module that uses this information to determine whether the object is present in the scene. As part of the decision protocol, the module may ask the tracker modules to process frames they had originally not processed to keep in-sync with real-time.

We will now use another, more basic example of a streaming application (a vision application pipeline, Figure 2) to illustrate some of the memory and buffer management challenges created by streaming applications. Unlike the gait analysis application, it has a single input and a single output point. The task-graph of the vision application pipeline is static, that is, the relationships between the various threads that constitute the application are known *a priori*, and do not change dynamically throughout the execution of the application. This simple vision application starts with a digitizer that captures and digitizes images every 1/30 of a second. The Low-fi tracker and the Hi-fi tracker analyze the frames produced by the digitizer for objects of interest and produce their respective tracking records. The decision module combines the analysis of such lower level processing to produce a decision output that drives the GUI that interacts with the user. From this example, it should be evident that even though the lowest levels of the analysis hierarchy produce regular streams of data items (in the vision application, for example, the digitizer, the lowest level of the analysis hierarchy, tends to have enough resources to ensure the mere capturing of an image every 1/30 of a second), other characteristics come into play and contribute to the complexity of buffer management as data moves to higher levels in the pipeline:

1. Streaming applications tend to process only a small portion of the input data. As we discussed earlier, streaming applications process data in order to attach meaning to it, not for the sake of processing all the data. In most cases, it is sufficient to process only a fraction of the data captured to receive meaningful results. These results tend to be close enough to the results the application would have produced had it been able to process all the data. In fact, finding the right trade-off between applying more resources to a streaming application and the benefit these additional resources provide is a challenge almost every streaming application system designer faces.

2. Streaming applications are required to process and attach meaning to data as close as possible to real-time, so as to give the opportunity to react to developing situations. For example, there is very little benefit in identifying potential stroke symptoms after help has been administered.

3. Threads may not access their input datasets in a strict stream-like manner. In order to follow the scene in real-time, a thread (e.g., the Hi-fi tracker) may prefer to receive the "latest" input item available, skipping over earlier items. This may even result in canceling activities initiated earlier, so that they no longer need their input data items. Consequently, producer-consumer relationships are hints and not absolutes, complicating efficient data sharing especially in a cluster setting.

4. Since computations performed on the data increase in sophistication as we move through the pipeline they also take more time to be performed. Consequently, not all the data that is produced at lower levels of the processing will necessarily be used at the higher levels. As a result, the datasets become temporally sparser and sparser at higher levels of processing because they correspond to higher and higher-level hypotheses of interesting events. For example, the lowest-level event may be: "a new camera frame has been captured", whereas a higher-level event may be: "John has just appeared at the bottom-left of the field view". Nevertheless, we need to keep track of the "time of the hypothesis" because of the interactive nature of the application.

5. Newly created threads may have to re-analyze earlier data. For example, when a

thread (e.g., a Low-fi tracker) hypothesizes human presence, this may create a new thread (e.g., a Hi-fi tracker) that runs a more sophisticated articulated-body or face-recognition algorithm on the region of interest, beginning again with the original camera images that led to this hypothesis. This dynamism complicates the recycling of data buffers.

6. Datasets from different sources need to be combined, correlating them temporally. For example, two vision pipelines may combine data from two or more cameras to track different angles of a scene. Other applications may work multi-modally by combining data from different input devices and sensors, e.g., video, audio, and motion detectors.

These characteristics bring up two requirements. First, data items must be meaningfully associated with time and, second, there must be a discipline of time that allows systematic reclamation of storage for data items (garbage collection).

Because these characteristics set streaming applications apart from scientific, as well as other traditional applications, there is a strong need for a novel system infrastructure to support the writing and managing of these applications. The search for new trade-offs to support and optimize streaming applications has generated considerable interest and spurred new research in the Computer Systems field. One of the support systems proposed is Stampede: a runtime system designed and developed to simplify the task of programming streaming applications by providing a simple and intuitive programming model. Stampede embeds an association between data items and time. In addition, the Stampede runtime system takes care of the synchronization and communication, as well as the memory and buffer management challenges, inherent in these applications.

Traditionally, memory management has been the responsibility of the programmer (e.g., in languages such as FORTRAN and C). The increased complexity of programs in general, and memory management in particular, has prompted a trend to move away from this model, towards a Garbage Collection (GC) model, where some of the responsibilities for memory management lie within the domain of the compiler and the runtime environment (e.g., in languages such as C# and Java). In both cases, a memory item that is no longer

participating in computations is identified as garbage if it is not referenced from anywhere in the application.

Environments for general-purpose applications must be conservative with their approach towards resource management in general, and memory management in particular, because they must assume no knowledge about a specific programming model or a specific class of applications. However, as stated earlier, streaming applications exhibit common characteristics that may allow a runtime system specific to these applications to harness these common traits and deploy more aggressive memory management policies.

1.2 Problem Statement

Streaming applications are required to handle large quantities of data, introducing a major challenge to their overall efficiency. Thus, effective memory and buffer management are vital in a successful deployment of streaming applications.

We explore ways to harness the unique characteristics of streaming applications to assist in creating a more efficient management of the memory associated with an application. These methods, in-turn, help the system achieve better control of memory resources required to sustain streaming applications and reduce the unnecessary usage of computational and network resources allocated to create and maintain superfluous data items that are not likely to affect the outcome of an application.

1.3 Description of Approach

Our main hypothesis is that streaming applications operate within a set of constraints that can then be used to generate more efficient memory allocation and management policies to support these applications. Thus, we explore ways to take advantage of these known characteristics of streaming applications to reduce the number of data items an application maintains and computes at any given time.

Using Stampede, a programming environment for supporting streaming applications as a test bed, we experiment with various approaches to optimize the memory management of streaming applications. We perform these experiments by extending the runtime capabilities of Stampede to support these various approaches to memory management.

We also develop an elaborate measurement infrastructure together with metrics and methodologies to assess the quality of the different memory optimization approaches. This measurement infrastructure also allows us to simulate suggested algorithms using "what-if" scenarios without actually implementing each and every variant of the proposed approaches, thus enabling us to quickly evaluate them and single out the most promising approaches for implementation.

In this dissertation we suggest to redefine the problem of garbage collection in streaming applications in the broader context of an application's memory consumption and resource utilization. We present two main approaches to memory optimizations in streaming applications. The first one explores ways to improve the garbage identification process. Each one of the proposed algorithms applies a different method to identify items that can be considered as garbage. The algorithms analyze the allocated data items to infer those items that would not be requested by any of the application threads. The second approach for memory optimizations in streaming applications infers the capacity of the system to process data so that the system can control the input rate of items as they are introduced into the system.

1.3.1 Garbage Identification

Timestamp-based garbage identification is performed by analyzing the allocated data items to infer those specific timestamped items that will be further processed and those items that will not. Timestamped items an application will not process further can be designated as garbage, and the memory associated with these items can be reclaimed. Garbage identification utilizes one of the characteristics of streaming applications: namely, that in many cases only a fraction of the data available either needs to be or ends up being fully processed and thus affects the outcome. The process used by the system to infer the specific data items the application will request can be greatly improved if application-level information is provided to the garbage collector. In the garbage collection algorithms presented, we also explore various ways the runtime system can use information that application writers may provide (such as uncovering dependencies inherent in an application) to make the garbage collection

more efficient. In addition, we examine the balance between giving the application writer the task of providing different types of application-level information, and the reduction in the memory pressure resulting from the incorporation of specific types of information into the garbage collection algorithm.

The first algorithm, REFerence count based garbage collector (REF, see Chapter 4.1), maintains a reference count that an application provides with each item, and collects it once the system can infer the item is not going to be asked for again, that is when its reference count reaches zero. The second algorithm, Transparent Garbage Collector (TGC, see Chapter 4.2), does not receive any information from an application. Rather, TGC analyzes the global state of an application and infers those specific data items that are not going to be requested by any one of the application threads. The third algorithm, Dead timestamp Garbage Collector (DGC, see Chapter 5) analyzes the local dependencies between various application threads and uses data guarantees provided by the application writer to determine the items that can be reclaimed. The rest of the algorithms (presented in Chapter 7) explore ways to reduce memory consumption in streaming applications either by taking advantage of additional information that can be provided to the runtime system and/or by dispensing information faster through the various nodes of the distributed application. The Out-of-Band Propagation of Guarantees (OBPG) algorithm allows for a faster propagation of information about guarantees. Keep Latest 'n Unseen (KLnU) uses information about dependencies between consumption and production of data within a specific channel to reclaim additional timestamps. Propagating Dead Sets (PDS) allows for a more comprehensive transfer of dead timestamp information throughout an application. Finally, Out-of-Band Propagation of Dead Sets (OBPDS) allows for a faster propagation of dead-sets in an application.

1.3.2 Adaptive Resource Utilization

While the application memory pressure can be reduced by uncovering data dependencies and propagating this information throughout the application threads, it still fails to make use of the unique way accuracy is evaluated in streaming applications. Financial and scientific

applications are required to process all the input data in order to produce accurate results. In contrast, streaming applications do not aim to process all input data. Rather, processing an input stream is a means to perform another task, such as, determining whether the gait of an elderly person indicates an onset of a stroke. The application can achieve this goal by processing only a fraction of the input data available. Moreover, it is not important that specific data items be fully processed, as long as the system can perform the task it was chartered with (for example, identify the occurrence of a stroke based on the gait analysis) in a timely manner. The next algorithm, Adaptive Resource Utilization (ARU, see Chapter 8) analyzes the application behavior as a whole to infer the capacity of the system to process data. Using this information, the ARU algorithm controls the rate at which items are introduced into the system. The ARU algorithm also holds the assumption (that is valid in many cases, but certainly not in all) that it can respond quickly enough to changes in the system's capacity to process items. In other words, the ARU algorithm is effective when changes in the system's capacity to process items occur at a lower frequency than the rate the ARU algorithm requires to propagate information about these changes in the application's processing rate to all the participating nodes. The ARU algorithm allows the system to predict how many items it can process, and to adjust the input rate accordingly, thus reducing the amount of resources devoted to data items that will not contribute to the output.

1.4 Contributions

This dissertation has three main contributions:

1. Extending the definition of the garbage collection problem in streaming applications to include not only data items that are not reachable but also data items that have no effect on the final outcome of the application. Streaming applications typically fully process only a portion of the data, and resources directed towards the remaining data items, that is data items that do not affect the final outcome, can be viewed as wasted resources that should be minimized.

2. Using the extended definition of garbage in streaming applications to develop novel

garbage collection algorithms and optimizations:

- The Dead timestamps based Garbage Collector algorithm (DGC) combines garbage identification and computation elimination. The DGC algorithm locally identifies dead items, that is, items that are of no interest to any of the application threads and thus, will not be requested in the future. This information is then propagated to neighboring nodes.
- Presenting additional GC algorithms for streaming applications (Keep Latest 'n Unseen, Propagation of Dead Sets, Out-of-Band Propagation of Guarantees, and Out-of-Band Propagation of Dead Sets). These algorithms enable the runtime system to achieve a tighter control on memory used by streaming applications. These algorithms explore the potential benefits of providing more information to the runtime system either directly from the application writer or by disseminating information more efficiently within the runtime system.
- The Adaptive Resource Utilization (ARU) algorithm is developed to regulate and match the producers' rate of production to the consumers' rate of consumption. The ARU algorithm predicts the capacity of the system to process data, and regulates the introduction of new data items to match this capacity. Thus, instead of attempting to identify items as garbage as soon as possible, the ARU algorithm prevents the system from generating items that would be classified as garbage in the future. By using this algorithm the system may achieve a tighter control on the utilization of resources already allocated to the application, and may direct these resources towards processing items that would affect the application outcome rather than items that will not be fully processed.

3. Development of a methodology and metrics to explore the design space of distributed software environments. The methodology involves the expression of attributes of the design space as "what-if" scenarios. The scenarios are then simulated and evaluated for their effect on the system's performance. The use of "what-if" scenarios makes it possible to define and express an ideal system. This ideal system serves as a reference

point that helps in assessing how well implemented or simulated systems perform. The simulations are dependent on accurate measurements of events that are performed in a distributed environment. To support the simulations, a distributed and cycle-accurate event-logging measurement infrastructure is presented. In this dissertation, we use the methodology and the measurement infrastructure to assess the performance of garbage collection algorithms. An Ideal Garbage Collector is defined and used as a reference point against which implemented (REF, TGC, and DGC) and proposed (OBPG, KLnU, PDS, and OBPDS) garbage collection algorithms are assessed. The proposed garbage collection algorithms are simulated and the best garbage collector is implemented. This work also serves as a case-study that evaluates the validity of this methodology.

1.5 Dissertation Outline

The following is the outline for this dissertation. In Chapter 2, we present Stampede, a runtime system that supports streaming applications. Stampede serves as a test bed for the memory optimization algorithms we propose. In Chapter 3, we present a methodology for exploring the design space of streaming applications. We use this methodology to evaluate the different memory optimization algorithms. In Chapter 4, we present two garbage collection algorithms: a REFerence count based garbage collector (REF) and a Transparent Garbage Collector (TGC). REF performs garbage identification locally and bases its decisions on local information. TGC, on the other hand, makes global garbage identification decisions based on global information. We examine these garbage collection algorithms, as well as their limitations. In Chapter 5, we present the Dead timestamps based Garbage Collector (DGC). This garbage collector uses a propagation of guarantees that are derived from data dependencies to perform local garbage identification decisions based on global information. In Chapter 6, we examine the performance of the DGC algorithm and compare it to REF, TGC, and the Ideal Garbage Collector (IGC). In Chapter 7, we propose additional algorithms for memory optimizations in streaming applications. The Keep Latest 'n Unseen (KLnU) algorithm is based on harnessing data dependencies among

produced items in a channel. KLnU makes local garbage collection decision based on local information. The Propagation of Dead Sets (PDS) algorithm is similar to DGC in that it propagates local information globally. However, it provides richer information than DGC, thus it has a greater potential for reducing the memory consumption of an application. The Out-of-Band Propagation of Guarantees (OBPG) and Out-of-Band Propagation of Dead Sets (OBPDS) algorithms examine the potential of relieving network delays on reducing an application's memory footprint. Using the methodology presented in Chapter 3, these optimizations are first simulated and then two of them, KLnU and PDS, are implemented. In Chapter 8, we examine another approach, that of the Adaptive Resource Utilization (ARU), in reducing the memory consumption of streaming applications. This approach attempts to match the introduction of new data items to the capacity of the system to process them, thus reducing the memory allocated to support data items that are not fully processed. In Chapter 9 we survey related work, and the conclusions are presented in Chapter 10.

CHAPTER II

STAMPEDE: A RUNTIME SYSTEM FOR STREAMING APPLICATIONS

Stampede is a runtime system that supports the development and execution of streaming applications by handling communication, synchronization, and buffer management, in-turn directing the application writer's attention away from these arduous and repeated tasks, and allows her to focus on the problem the application is set to solve. The Stampede runtime system is used in this study as a test bed for evaluating different memory optimization policies and algorithms.

2.1 Motivation

Application domains such as interactive vision, animation, and multimedia collaboration display dynamic scalable parallelism, and high computational requirements. These properties make them good candidates to execute on parallel architectures such as SMPs and clusters of SMPs. However, in the absence of suitable facilities, writers of streaming applications are left with the repeated and arduous task of implementing mechanisms at the application level for harnessing the full potential of parallel architectures. In the previous chapter we discussed some of the unique features of streaming applications. The following is a brief discussion of such characteristics that pose a burden on the application writer. First, *time* is an important attribute due to the interactive nature of these applications. In particular, streaming applications require the efficient management of temporally evolving data. For example, a stereo module in an interactive vision application may require images with corresponding timestamps from multiple cameras to compute its output. Another example is a gesture recognition module that may need to analyze a sliding window of frames over a video stream. Secondly, in addition to the concept of time in streaming applications, both the data structures as well as the producer/consumer relationships in these applications are *dynamic* and *unpredictable* and therefore cannot be determined at compile time.

Existing programming environments for parallel computing do not provide the application writer with adequate management and support for such temporal requirements.

An abstraction for parallel programming called Space-Time Memory (STM) [46] addresses the challenges programmers of streaming applications face during the development process. The STM abstraction is a dynamic concurrent distributed data structure for holding time-sequenced data. STM addresses the common parallel programming requirements found in most interactive applications, namely, inter-task synchronization, and meeting soft real-time constraints. These facilities are particularly helpful when streaming applications are implemented on an SMP or across clusters.

Stampede runtime system [37] implements the STM abstractions. It also provides additional support by managing and reclaiming the STM's time-sequenced data items. As discussed in Chapter 1, data management in the context of streaming applications is quite different from the traditional memory address-based garbage collection and therefore poses a unique and arduous challenge on the system's designer. Memory management is further complicated because computations, and data items associated with each of the computations, are spread across a cluster.

In this chapter, we present some of the concepts and abstractions that are part of the STM model and are provided by the Stampede runtime system. The focus of this discussion is expounding on the entities in Stampede related to memory and process managements. For a full description of the STM model and the Stampede runtime system, please refer to [48], [46], and [37].

2.2 Entities in Stampede

The Stampede runtime system consists mainly of a collection of *threads* that use *channels* to communicate time-associated items. Channels serve as memory buffers in which these items are stored and retrieved. Because data items stored in channels are time-stamped (see Figure 3), we also refer to the collection of channels as *Space-Time Memory*.

2.2.1 Threads

Threads are responsible for performing computation and processing data. Each thread operates in the context of a single *address space*. Stampede allows the creation of multiple address spaces in the cluster and an unbounded number of dynamically created application threads within each address space. The threading model within an address space is basically standard OS threads such as pthreads (POSIX threads) [15] on Tru64 Unix and Linux, and Win32 threads on Windows architecture.

Threads may be *sources* (only produce items), *intermediate* (both consume and produce items), or *sinks* (only consume items). An example of a source thread is one that captures image frames from a video camera at regular intervals and puts them into a channel (e.g., the Digitizer thread in the vision application pipeline presented in Figure 2). An example of a sink thread is one that renders images to a display device (e.g., the GUI thread in Figure 2). Most of the threads in a typical streaming application are intermediate, that is they both consume and produce items (e.g., the Low-fi tracker, the High-fi tracker, and the Decision threads in the vision application pipeline presented in Figure 2).

2.2.2 Time and Visibility Management in Stampede

The STM model associates every data item with a *timestamp*. Timestamps are integers that represent discrete points in real time. For example, image sequence numbers of captured images can be used as timestamps to index those images. In this example, a thread that captures images may use the order of their capture (i.e., their sequence) to represent the relationship of each and every image with real time and with the rest of the captured images. Although most of the threads do not capture data and thus lack a direct reference to real time, they can still refer to real time indirectly by preserving the relationship established when the data was originally captured. For instance, a source thread captures an image, x, and associates it with the timestamp t_{s1}. The captured image is then put into a channel. An intermediate thread may get an image-timestamp pair <x, t_{s1}> from the channel, perform a computation on it, and put a result <y, t_{s1}> into another channel, where y is a detected feature or an annotated image. The intermediate thread preserves the relationship

of the data it produces, y, and real time by maintaining the timestamp value (t_{s1}) that was associated with the data when it was originally captured. STM permits these different data items (x and y) to share the same timestamp value (t_{s1}) because they are associated with the same discrete point in real time. However, it is possible to distinguish between these two items because they represent different stages of processing the same captured data (the first one is represented by x, and the second one by y).

Another motivation for allowing this kind of sharing of timestamp values is that the programming of intermediate threads is greatly simplified if they do not have to manage the progress of time explicitly, especially when they process timestamps out of order. In most cases, if a thread is currently getting items from one or more channels with timestamps $t_{s1}, t_{s2}, \ldots, t_{sn}$, the timestamp of the corresponding output item can be simply derived from the input timestamps. In the frequently occurring case of a thread with a single input, the next output timestamp is usually just the current input timestamp.

In general, we want "time" to march forward, i.e., as a thread repeatedly gets increasing timestamps on its inputs, we want the outputs that it produces also to have increasing timestamps.

To capture this idea, STM introduces the notion of *visibility* for a thread: when a thread is processing inputs with timestamps $t_{s1}, t_{s2}, \ldots, t_{sn}$, it is only allowed to output items with timestamps greater than or equal to the minimum of the input timestamps. Another way to think about this example would be to assume that when a thread derives its "current time" from the timestamps of its current input items, it is not allowed to output an item that is "earlier" than its current time.

There are a few more common-sense rules associated with time management. When a thread creates a child thread, the newly created child thread is not allowed to input or output items with timestamps earlier than its creation time. When a thread attaches a new input connection to a channel it is not allowed to input items earlier than the time of attachment.

Although the ideas above allow automation of time management, there are few situations where threads need to manage their notion of time explicitly. Source threads, for example,

need to inject new timestamps, as they have no input timestamps to base their current time. Stampede maintains a state variable for each thread called *virtual time*, $VT(t)$, that helps determining the virtual time associated with the newly defined data items. An application may choose any application-specific entity as the virtual time. For example, in the vision pipeline (see Figure 2), the frame number associated with the camera image may be chosen as the virtual time.

Certain threads may even wish to "reserve the right" to look at older data. For example, a thread may use simple differencing between images to identify regions where change has occurred (for example, someone approaching the camera). When a sufficiently interesting difference is detected, the thread may re-analyze the region-of-interest in the last d images using a more sophisticated algorithm (for example, a face detector). Thus, threads require the ability to go d steps back in time.

The programming model implemented in Stampede, and, in particular, its rules for time and visibility are motivated by these observations. The rules allow "out of order" processing. For example, for performance reasons a task may be parallelized into multiple threads responsible for processing different frames while sharing common input and output channels. The rate at which these threads process inputs may be unpredictable, thus, outputs may be produced out of timestamp order. A downstream thread should also be allowed to process these results as they arrive, rather than in timestamp order. By introducing constraints around time management within the application, these visibility rules make it possible for the runtime system to deduce which items will not be used in the future by any one of the application threads. Thus, these constraints are crucial to any garbage collection scheme implemented.

2.2.3 Channels

STM *channels* are memory buffers that provide random access to a collection of time-indexed data items. STM channels can be envisioned as a two-dimensional table (see Figure 3). Each row, called a *channel*, has a unique system-wide ID.

A particular channel may be used as the storage area for an activity (e.g., a digitizer) to

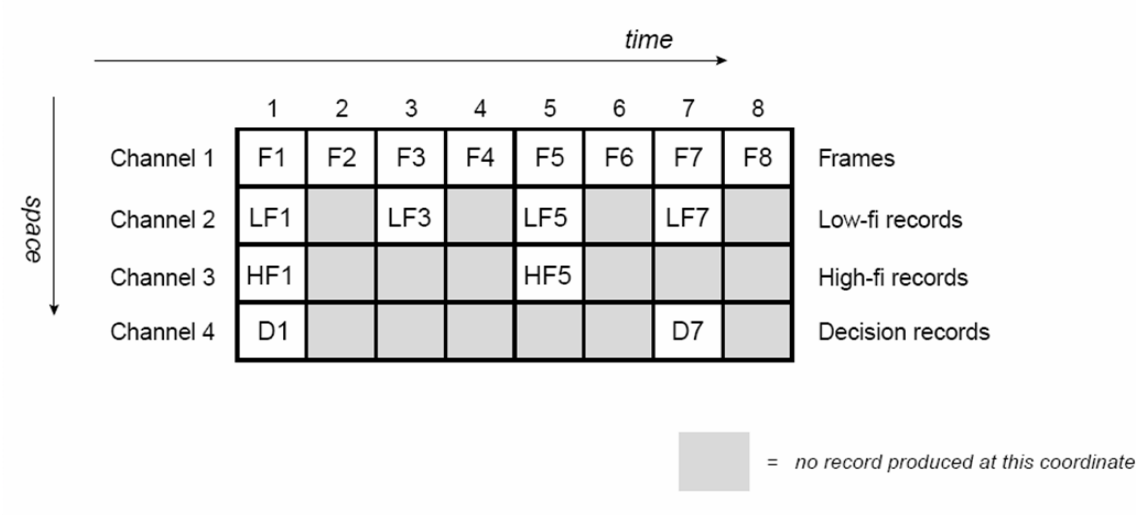

Figure 3: Mapping the Vision Application Pipeline to STM Channels: STM channels can be envisioned as a two-dimensional table of time (timestamps) and space (channels). This example depicts a snapshot of a possible instantiation of the vision application pipeline presented in Figure 2. Note that not all the data items produced by Channel 1 have corresponding equivalents in the rest of the channels.

place the time-sequenced data records that it produces. Every column in the table represents the temporally correlated output records of activities that comprise the computation. For example, in the vision pipeline in Figure 2, the digitizer produces a frame F_t with a timestamp t. The Low-fidelity tracker produces a tracking record LF_t analyzing this video frame. The decision module produces its output D_t based on LF_t. These three items are on different channels and may be produced at different real times, but they are all temporally correlated and occupy the same column t in the Space-Time Memory.

Similarly, all the items in the next column of the STM channel table have the timestamp $t+1$. Figure 3 shows an example of how the STM channels may be used to orchestrate the activities of the vision pipeline introduced in Figure 2. The rectangular box at the output of each activity in Figure 2 is an STM channel. The items with timestamp 1 (F_1, LF_1, HF_1, and D_1) in each of the four boxes in Figure 3, represent a column in the STM table.

2.3 Operations on the STM Channel Abstraction

The STM API has operations to create a channel dynamically, and for a thread to *attach* and *detach* a channel. Each attachment is known as a *connection*, and a thread may have

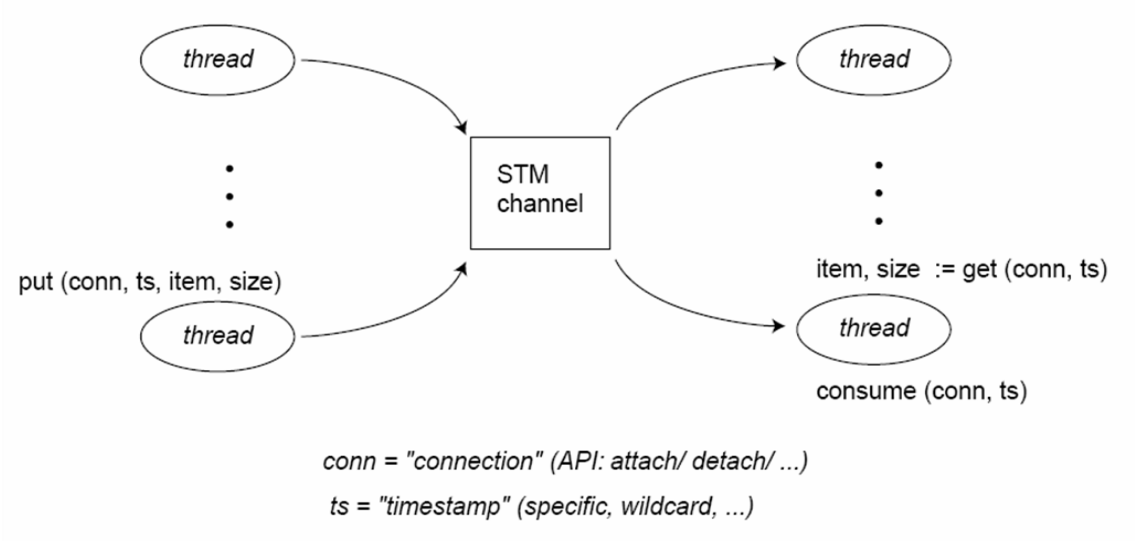

Figure 4: An Overview of Stampede Channel Usage: This figure depicts the relationship of a channel to threads. A thread sends an item to a channel via a *put* operation. The put call specifies the connection on which the operation is performed, the item, the timestamp associated with the item, and the size of the item. A thread receives an item from a channel via a *get* operation. The get call specifies the connection on which the operation is performed and the timestamp associated with the desired item. A thread communicates it has completed processing an item it had previously received on a specific connection via a *consume* operation. The consume call specifies the connection and the timestamp associated with the item to be consumed.

multiple connections to the same channel. Figure 4 shows an overview of how channels are used. A thread can *put* a data item into a channel via a given output connection using the call:

```
spd_channel_put_item (output_connection, timestamp, buffer_pointer,
                      buffer_size, ...)
```

The item is described by the pointer `buffer_pointer` and the item size (`buffer_size`) in bytes. A channel cannot have more than one item with the same timestamp, but there is no constraint that items be put into the channel in increasing or contiguous timestamp order. Indeed, to increase throughput, a module may contain replicated threads that pull items from a common input channel, process them, and put items into a common output channel. Depending on the relative speed of the threads and the particular events they recognize, it may happen that items are placed into the output channel out of order. Channels can be created to hold a bounded or unbounded number of items. The `put` call takes an additional

flag that allows it either to block or to return immediately with an error code if a bounded output channel is full.

A thread can *get* an item from a channel via a given connection using the call:

```
spd_channel_get_item (input_connection, timestamp,
                      & buffer_pointer, & buffer_size,
                      & timestamp_range, ...)
```

The `timestamp` can specify a particular value, or it can be a wildcard requesting, for instance, the latest value currently in the channel. As in the `put` call, a flag parameter specifies whether to block if a suitable item is currently unavailable, or to return immediately with an error code. The parameters `buffer_pointer` and `buffer_size` can be used to pass in a buffer to receive the item or, by passing `NULL` in `buffer_pointer`, the application can ask Stampede to allocate a buffer. The `timestamp_range` parameter returns the timestamp of the item returned, if available; if an item with the requested timestamp is unavailable, it returns the timestamps of the "neighboring" available items (assuming the latter exist).

The `put` and `get` operations are atomic. The semantics of `put` and `get` are copy-in and copy-out, respectively. Thus, after a `put`, a thread may immediately safely re-use its buffer. Similarly, after a successful `get`, a consumer can safely modify the copy of the object that it received without interfering with the channel or with other threads.

Put and get operations, with copying semantics, are of course reminiscent of message-passing. However, unlike message-passing, these are location- independent operations on a distributed data structure.

These operations are one-sided: there is no "destination" thread/process in a `put`, nor any "source" thread/process in a `get`. The abstraction is one of putting items into and getting items from a temporally ordered collection, concurrently, not of communicating between processes.

2.4 Summary

Stampede is an environment that is designed to simplify the programming of stream-based applications. The Stampede runtime system implements an abstraction for parallel programming called Space-Time Memory (or STM) and provides facilities that address some

of the challenges programmers face during the development process of stream-based applications.

In this dissertation, we use the Stampede runtime system as the test-bed for developing and evaluating the proposed mechanisms for optimizing the memory usage of stream-based applications.

CHAPTER III

A METHODOLOGY FOR EXPLORING THE DESIGN SPACE OF STREAMING APPLICATIONS

This dissertation suggests various approaches for optimizing the memory usage of streaming applications based on exploration and analysis of their design space. An analytical evaluation of these approaches is required to understand the effects of specific design space parameters on the memory usage of streaming applications. This evaluation also helps to compare different proposed optimizations as well as identify the most effective ones.

A detailed and rigorous evaluation of each suggested optimization algorithm requires a significant effort, and may prove to be futile if the optimization is found to provide little or no improvement over a non-optimized system. Thus, while this kind of effort is required as part of the final evaluation phase, it is superfluous at the early evaluation stage, when the target of the analysis is simply to classify and distinguish between promising and non-promising approaches. At this early stage it is sufficient to differentiate between effective and non-effective approaches, so that the non-effective approaches can be factored out and not explored any further. In other words, at the early design stage we are more interested in the relative performance of the suggested algorithms and less in the exact quantification of their performance.

In addition, and as part of the final evaluation stages, there is a need to evaluate the costs associated with each and every one of the suggested optimizations. These costs are comprised of:

- Additional system resources required to harness information readily available to the runtime system, and/or
- Additional information provided by the application writer that helps the runtime system in reducing the memory consumption of streaming applications.

While only the first type of cost is quantifiable, any evaluation of the design space of streaming applications has to take into account both of these costs as part of the process of understanding and assessing the tradeoffs associated with a suggested approach.

In this chapter we present the use of "what-if" scenarios as a method for a coarse-grain evaluation of memory optimization algorithms in streaming applications. In addition we present the measurement infrastructure that is used for both evaluating these scenarios, and the final assessment of the memory optimization algorithms we propose.

3.1 Motivation

First introduced during World Word II to crack German military codes, computers were quickly harnessed to solve a myriad of scientific, security, and business problems. The constant increase in computer capabilities, coupled with the reduction in their price, created an environment that enabled computers to provide solutions to new problems. Indeed, the process of providing computing solutions to more and more problems is the most significant characteristic of the computer age. By the last decade of the 20th century, micro-computers had enough resources, and were sufficiently inexpensive to make a new architecture, that of the *clusters*, economically feasible. Clusters tie the power of many computers together, and this theoretically infinite amount of power made it possible to provide solutions to many problems that were initially considered too expensive to solve using computers. However, designers of cluster solutions quickly realized that concepts, tradeoffs, and insights that were gained through analysis of traditional applications in traditional environments may not be applicable for distributed applications running on clusters. Therefore, when designing systems to support emerging applications in general, and streaming applications in particular, one must thoroughly understand the design space, examine competing design options and choose the ones that suit the application needs the best.

One approach for assessing competing design options is to build several flavors of the system, one for each design option. It is then possible to compare among the different design options and select the one that provides the best results. However, this approach is not flexible enough to enable exploring the design space, and, in many cases, significant

resources are needed for debugging and building each one of the flavors. We are seeking a more efficient way of understanding the design space.

3.2 Using "What-if" Scenarios to Evaluate Competing Design Options

The motivation for using "what-if" scenarios in the context of evaluating memory management policies for streaming applications comes from observing many similarities between this problem and the problem of assessing memory hierarchy management policies. In both cases, it is much easier to evaluate competing design options by modeling and simulating their behavior rather than implementing them and then analyzing their performance. In assessing competing cache policies, for example, researchers routinely use the Trace Driven Simulation method to decide on the design parameters (size, partition, eviction algorithm, etc.) of the cache to be implemented. In these cases, Trace Driven Simulation is based on a record of every main memory location referenced by a program during its execution. A model of the cache can then be simulated, and by varying parameters of the simulation model, it is possible to simulate directly any design parameter, such as cache size, placement, fetch or replacement algorithm, line size, etc. This method enables the cache designers to decide on the design parameters of the cache without actually implementing the cache.

One of the reasons for the effectiveness of the Trace Driven Simulation method in predicting cache behavior is that the evaluated design parameters are not directly affected by changes they create in the system. To illustrate this point, consider evaluating several replacement algorithms for a uni-processor cache. The replacement algorithms are evaluated by their effect on the hit ratio of the cache. The second-order effects on execution time as a result of differences in cache hit ratio have very little influence (if any) on the order or the location of memory accesses. A memory trace that might be generated using an implementation of the simulated system would, therefore, be very similar to the one generated by the original system. As a result, the simulated cache hit ratio would be very close (if not identical) to the implemented cache hit ratio.

Keeping the principle of evaluating parameters that are not directly affected by the simulated system made it possible to apply the Trace Driven Simulation method to many other research areas in Computer Science. As far as hardware design is concerned, simulations are taken even further to analyze hypothetical parameters in order to explore the system's design space. For example, processor caches are often simulated as having infinite size. This impractical scenario helps researchers to understand design limits and provides a measure to assess how close to ideal a particular realistic algorithm is.

We suggest that hypothetical, "what-if" scenarios are not just limited to hardware systems but can also be applied when exploring the design space of distributed software environments. These scenarios enable designers to propose a hypothetical question that explores one or more design space parameters and allow designers to better understand the design space tradeoffs. In addition, "what-if" scenarios enable designers to define and examine ideal conditions, void of implementation or other constraints. Examining ideal "what-if" scenarios enables system designers to investigate the limits of the performance gains expected from a suggested change to the system's design parameters and to better understand the usefulness of a certain approach.

Trace Driven Simulation in the context of streaming applications measures performance parameters and generates a trace of a current system implementation. Using the current implementation as a base-line, it is then possible to analyze "what-if" scenarios that examine particular design options or the influence of specific facets of the current implementation. It enables a designer to explore the design space, and define an ideal scenario, void of constraints. The performance of an implemented or a simulated system can then be compared against the ideal scenario to understand how well different implementations and simulations perform.

We believe this method is also appropriate for the problem we present of understanding memory management policies tradeoffs for streaming applications in distributed environments. The objective is not to quantify the performance of each and every design option precisely, but rather to qualitatively understand the performance differences between one algorithm to another. The comparison to an "ideal" system provides a measure as to how

well the system performs under different design assumptions and helps to assess if under these assumptions there is a potential to push the performance envelope even further. Performance assessment of a possible design option is, thus, mostly based on an existing implementation, and only a small part of it on actual simulation. In addition to achieving more accurate assessments, this hybrid approach is faster to implement than a full-fledged simulation.

The accuracy of the "what-if" scenarios relies heavily on a precise and effective measurement infrastructure. In this sense, streaming applications present particular challenges. Some of the events measured in these applications are very short and may require only few tens of instructions to complete. Traditional, operating system-provided, clocks do not supply a sufficient level of granularity to support these types of measurements. Even if they did, measurements may require so many resources, that the time and resources needed to record an event may be greater than the event itself. Thus, traditional clocks may perturb the system greatly to make the measurements (and any analysis based on them) useless.

Additionally, the events recorded are distributed, and may start in one node and continue on another. Measuring distributed events requires at least a certain level of clock synchronization.

We therefore introduce a low-cost, yet accurate, distributed measurement infrastructure that addresses these challenges to increase the accuracy level of the traces that form the basis of the simulations.

3.3 Measurement Infrastructure

The measurement infrastructure consists of two mechanisms: *event logging*, and *postmortem analysis* [40]. The event logging mechanism is responsible for defining, capturing, and recording events. It also addresses the challenges of accurately measuring short events in a distributed environment. The postmortem analysis uses the event logs to calculate statistics of an application run. It also provides the infrastructure to simulate different "what-if" scenarios, including simulating ideal scenarios. One of the advantages of this measurement infrastructure is its ability to accurately accumulate the time spent in any

and every piece of code segment. This allows us to assess the time and resources used by code segments as they migrate from one machine to another. Moreover, it allows us to account for computation associated with a message, and to track the message and the resources allocated to process it across machines in a cluster.

3.3.1 Event Logging

The event logging mechanism goal is to be able to accurately accumulate the time spent in any and every piece of code segment for the purposes of assessing design options in general, and memory management policies in particular. The basic idea is to declare *events* that are names meaningful to the code that is being timed. For example, if we want to accumulate the time spent in a procedure body, we place a call to the timing function at the entry (start time) and exit (end time) points. The event logging mechanism records the start and end times along with a mnemonic user-specified event name that uniquely identifies the code fragment that is being timed.

Event recording should have the least possible perturbation to the original application. Thus, events are recorded in a log that is maintained as a buffer within the main memory. Each log entry consists of the unique event name, and the start and end times for that event. The time for recording the log for an event is assumed to be small in comparison to the code fragments that need to be timed. The ability to declare events enables to minimize the events being logged to the one we are interested in measuring, thus reducing chances to perturb the system. The logging subsystem maintains two in-memory buffers. When one buffer is approaching fullness, the logging subsystem switches to the other buffer. Then it flushes out the first buffer via DMA to the disk in binary form. While this flushing does not use processor cycles, it could perturb the normal application execution since there could be contention for the memory bandwidth. In order to keep this perturbation to a minimum, we pick a large enough buffer size so that the number of disk I/Os during the application execution is kept to a minimum. Typically event log records are about 16 bytes. With a 4MB buffer size, we can ensure that there are only 4 disk I/Os to generate a log file of a million events. These data structures may result in some cache pollution, but that is

unavoidable in any logging infrastructure that does not employ additional hardware.

Events are recorded alongside with the time they occur and several methods can be used to determine it. Any time determination methodology should impart least possible perturbation to the original application. When events are relatively long, standard system supported timing calls (such as the Unix `gettimeofday` call) may be sufficiently accurate. However, some of the events we wish to measure are very short (as small as few tens of instructions). These standard system supported timing calls may be inappropriate because their granularity is not fine enough for our purpose, and even if they were, system calls are quite expensive, and may involve cache pollution. Fortunately, most modern processors such as the Intel Pentium line have a CPU cycle counter that is accessible as a program readable register. This allows us to implement the time measurement part of the event logging mechanism using very inexpensive inline assembly commands. We can, therefore, assume that even for short events, the time for recording the log for an event (a few memory write instructions) is small in comparison to the code fragments that need to be timed. In any case, within the limitations of a software-based measurement infrastructure, cycle reading is the least intrusive method of recording these events.

3.3.2 Postmortem Analysis

A postmortem analysis program has no effect on the application performance because it defers the execution of many operations that are required to generate measurements and statistics until after the application itself has terminated. The postmortem analysis program combines logs from different machines, reads in the records from the log file on the disk, and computes the wall-clock time for macro events of interest. When direct measurements are not possible, the postmortem analysis program can also make use of these macro event to infer event measurements as well as to generate inferred events.

Figure 5 demonstrates the use of known macro events to make inferences about events that cannot be measured directly. It shows a single operation that spans two address spaces, such as a request for a remote item. The request for an item is generated in address space 1, and is actually fulfilled in address space 2. Each point in the figure represents a single record

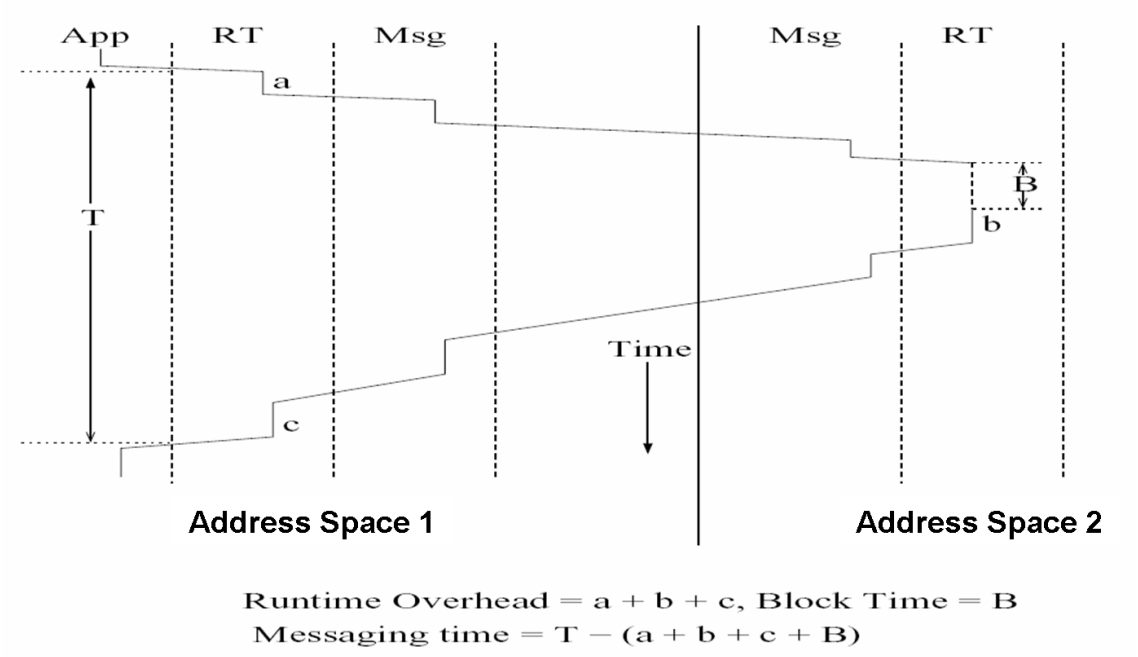

Figure 5: Timeline for a Remote Operation Spanning Two Address Spaces: RT, Msg, and App designate the Runtime system, the Messaging Layer, and the Application logic, respectively. Each point represents a logged event. The runtime overhead as well as the messaging time can be inferred from the logged events.

that the logging mechanism records. The postmortem analysis program uses these logged records to calculate macro events. The postmortem analysis program can then use these macro events to make inferences about events that span across multiple address spaces. These events cannot be measured directly without synchronizing the clocks in both address spaces.

In the example depicted in Figure 5, the postmortem analysis program helps to infer the wall-clock time of the run-time system overhead and the messaging time. An application thread, running on Address Space 1 can measure event `T`. The runtime system in Address Space 1 can measure events `a` and `c` that are associated with the overhead of the runtime system on Address Space 1. The runtime system in Address Space 2 can measure events `B` and `b`. Event `b` is associated with the overhead of the runtime system in Address Space 2. Event `B` is associated with the blocking event related to this message in Address Space 2. The event logging mechanism allows to associate all events measured (`a`, `b`, `c`, `B`, and `T`) with the same event name. As can be seen from the figure, it is possible to determine the

blocking time in Address Space 2 (`B`) and the runtime system overhead in Address Space 2 (`b`). It is also possible to calculate the runtime system overhead in Address Space 1 (`a+b`). The association of events with an operation enables the postmortem analysis program to calculate the runtime system overhead associated with this operation (`a+b+c`), and to infer macro events that are not measured directly, such as the messaging time (`T−a+b+c+B`). Please note that although these two latter macro events span multiple address spaces, in these cases the postmortem analysis program does not require clocks in the different address spaces to be synchronized as the deduction of the macro events is based on the duration each event took in each one of the address spaces.

3.4 Summary

The expression and analysis of "what-if" scenarios and the measurement infrastructure presented in this chapter enables a more efficient and precise exploration of the design space of runtime systems for streaming applications. This exploration leads to a formation of algorithms that use memory more efficiently, thus reducing the memory footprint of streaming applications. A by-product of the memory reduction is a decrease in other resources streaming applications consume (e.g., computation, network) and as a result an improvement in performance.

In the following chapters we present several garbage collection algorithms specifically designed to reduce the memory footprint of streaming applications. We use the methodology and the measurement infrastructure presented in this chapter to assess the performance of these algorithms and to better understand the design space of memory optimizations in the context of streaming applications.

CHAPTER IV

GARBAGE COLLECTION ALGORITHMS IN STAMPEDE

In this chapter we present two garbage collection algorithms: the Referenced Based Garbage Collector (*REF*) and the Transparent Garbage Collector (*TGC*) [36]. Although REF and TGC are implemented in Stampede, the concepts behind these algorithms are general, and can be applied to any runtime system designed to support streaming applications. Memory based garbage collectors reclaim storage of heap-allocated objects (data structures) when they are no longer "reachable" from the computation. By contrast, REF and TGC identify timestamped items as garbage if the particular algorithm can deduce those specific items that will not be used by the application, regardless of whether they are reachable or not. Indeed, the main difference between REF and TGC is the technique employed to identify those data items the application will not process further.

4.1 Reference Count Based Garbage Collector (REF)

This is a simple garbage collector that employs a method similar to the garbage collection of heap-allocated memory using reference counts. As the name implies, the algorithm uses a reference count to determine whether or not an item can be considered as garbage. This reference count is associated with an item by a *producer* thread and indicates the number of consumers that will receive the item. Every time a consumer reads the item, the reference count number is decremented by one. The item can be considered as garbage once the reference count reaches zero. The garbage-collection condition to determine whether an item is garbage or not is simply: collect an item when the reference count for this item is zero.

The REF garbage collector can work only if the consumer set for an item is known *a priori* at the time an item is tagged with a reference count. An example of where this requirement is satisfied would be when an application can be described as a static data flow graph, where all the connections are fully specified at application startup time.

In the context of Stampede, a *producer* thread associates a reference count with an item before it is *put* in a channel. The reference count is decremented whenever a *get* operation is performed on the item. Stampede is programmed to garbage collect the item when the reference count associated with the item reaches zero. The application programmer uses the *put* operation to supply the reference count to the runtime system.

4.2 Transparent Garbage Collector (TGC)

As described in Chapter 2.2.2, Stampede associates every data item with a timestamp. These timestamps are integers that represent discrete points in real time and their value increases with the progress of time. Each Stampede thread can determine its "interest" in timestamp values. For example, as the execution progresses, a thread may determine that it has no interest in receiving and processing items with timestamp values that are smaller or equal to 20. The thread can then communicate its "interest" in timestamp values to the runtime system. The runtime system receives this data from all application threads. It can then analyze this data and generate an interest set that applies to all application threads. The runtime system can then garbage collect any item with a timestamp that lies outside this interest set.

The runtime system uses a single interest set that is generated based on *all* thread-level interest sets. Thus, the garbage collection decision is global in nature and can be generated solely by the runtime system. In addition, the thread-level interest sets can also be generated by the runtime system without any intervention from the application writer. As a result, the garbage collection identification and collection can be performed by the runtime system and is transparent to the application writer, hence it is called Transparent Garbage Collector, or TGC.

TGC uses a single number, *Global Virtual Time* (GVT) that is associated with the application and represents the minimum timestamp value that is of interest to every thread in the entire application. TGC employs a distributed algorithm to compute the GVT.

In the context of Stampede, the GVT is guaranteed to advance so long as each thread:

1. Advances its virtual time (by analyzing the timestamps on its input and output connections).

2. Consume items on its input connections that are earlier than its virtual time.

By definition, items with timestamps less than GVT are not to be accessed by any thread and can therefore be safely garbage collected.

The garbage collection condition for this technique uses the GVT to determine whether or not an item can be considered as garbage. If the timestamp of the item is less than the GVT the item can be garbage collected; otherwise there is still a possibility that one of the application threads will access the item and therefore the item must be retained. Thus, GVT calculation is the centerpiece of the Transparent Garbage Collection algorithm.

To calculate the GVT, the TGC algorithm maintains two *state variables* on behalf of each thread:

1. *Thread virtual time*, denoted as $VT(thread)$, gives each thread individual control on timestamp values that it can associate with items it may produce on its output connections. As a thread performs a *set virtual time* operation (an API call in the Stampede programming system), the thread virtual time advances as well.

2. *Thread keep time*, denoted as $KT(thread)$, is the minimum timestamp value of items that this thread is still interested in getting on its input connections. Thread keep time advances as items are consumed.

The minimum of these two state variables produces a lower bound for timestamp values that are of interest to a thread. TGC then determines the GVT by computing the lower bound on the minimum timestamp values that are of interest to *all* the application threads. Once GVT is determined, the GC condition is simple: channel items whose timestamps are below the GVT are considered garbage and can be reclaimed.

The details of how the GVT value is computed are beyond the scope of this document, and can be found in [36].

4.3 Applicability and Limitations

Both REF and TGC present a different approach to the garbage collection problem than the one taken by memory based garbage collectors. Traditional garbage collectors reclaim storage of heap-allocated objects (data structures) when they are no longer "reachable" from anywhere in the computation. By contrast, REF and TGC analyze timestamp entities and identify those data items that would not be used by the streaming application in the future and reclaim these items as garbage. However, TGC and REF differ in the method they apply to identify garbage items.

REF makes local decisions that are purely based on information the application writer provides regarding consumption patterns. While producer/consumer relationships may change dynamically, the runtime system has no way of knowing these changes have occurred. It is the sole responsibility of the application (and the application writer) to encode the appropriate reference count for an item when it is produced. Thus, REF is most suitable for static application graphs, where producer/consumer relationships are known *a priori*. REF can be more aggressive than TGC by identifying garbage items earlier in the execution cycle, because of the local nature of its decision making. In addition, the algorithm and the computations involved in establishing that an item is garbage are simple and straightforward.

The Transparent Garbage Collector (TGC), on the other hand, eliminates the need for the application to supply a reference count for each and every item produced. It implements a distributed algorithm to determine a global low water mark for timestamp values of interest to the application as a whole. TGC allows any level of application dynamism since it is independent of the application details. However, this flexibility comes at a price. The global low water mark calculation requires information from all the threads that constitute the streaming application. The global nature of this calculation and the inherent network delays force TGC to make more conservative decisions regarding garbage identification. In addition, because TGC involves maintaining and processing information from all the threads, it requires more computational resources than REF.

While TGC and REF manage to redefine the garbage collection problem in terms of

streaming applications constraints, they fall short of exploring other, perhaps more aggressive, approaches that take advantage of many streaming applications characteristics. TGC makes global decisions, and does not take advantage of local information to quickly identify garbage items. REF, on the other hand, makes local decisions, but does not propagate the local decisions globally to other parts of the application. In the next chapter we will present another algorithm, Dead timestamps based Garbage Collector (DGC), that makes local decisions and does propagate this information throughout the streaming application pipeline.

CHAPTER V

DEAD TIMESTAMPS BASED GARBAGE COLLECTOR (DGC)

5.1 Introduction

The TGC algorithm calculates a global value of the minimum "observable" timestamp as a mechanism to deal with sparse production and consumption of timestamps in a distributed and global garbage collection environment. REF, on the other hand, makes decisions based on local data; however it does not propagate this information to other nodes of the application graph, thus limiting the benefits of identifying garbage items quickly to the immediate producer/consumer pairs. The Dead timestamps based Garbage Collector (DGC) algorithm takes these mechanisms a step further, and introduces the concept of a *local timestamp guarantee*. The DGC algorithm analyzes local conditions at threads and channels, in conjunction with application knowledge, to generate a local timestamp guarantee. Using this guarantee, the runtime system determines whether a timestamped item can be labeled as garbage. This information is then propagated to other nodes of the application graph and enables them to incorporate this knowledge into their localized decision process of identifying items for garbage collection.

The DGC algorithm models the application as a directed graph where nodes in this graph correspond to the application's threads and channels, while edges correspond to input and output connections. During the application execution, the DGC algorithm calculates timestamp guarantees on each and every application node and propagates these guarantees to the rest of the nodes through the application pipeline. Each application node uses these guarantees received from neighboring nodes, along with local guarantees to calculate the local timestamp guarantee. The latter is used to identify garbage items on that node.

5.2 Live and Dead Timestamps

The memory management and resource allocation challenges that streaming applications face are exacerbated because different threads may process timestamps at different speeds.

In particular, earlier threads (typically faster threads that perform simple low-level processing) may be producing items *dropped* by later threads performing more complicated, high-level processing at a slower rate. Only timestamps that make it all the way through the entire pipeline affect the output of the application, while a timestamp that is dropped by any thread during the application execution is *irrelevant* to the final outcome of the application.

The metric for efficiency in these systems is not the rate the application processes data as a whole, but rather, the rate the application processes *relevant* timestamps that affect the outcome. The work related to processing irrelevant timestamps, that is timestamps that do not affect the final outcome, represents an inefficient use of processing resources.

The DGC algorithm computes a timestamp guarantee for each node. The guarantee is a time marker such that all timestamps below its value are irrelevant and considered *dead*. As execution proceeds and more data items are being processed, the value of this time marker increases. A timestamp may be considered relevant or *live* at one node and concurrently irrelevant or dead at another node. Also, a timestamp may still be considered live at a certain execution time but dead at a later time on the same node. Dead timestamps are interpreted differently depending on the node type. If the node is a channel, items in that channel with dead timestamps are identified as garbage and can be immediately reclaimed. If the node is a thread, dead timestamps that have not yet been produced by the thread represent dead computations and can be eliminated. Note that dead computation elimination is different from dead code elimination by a compiler [56]. It is not the static code that is eliminated, but rather an instance of its dynamic execution. Thus, DGC provides a unified framework for garbage collection and dead computation elimination.

Next, we describe the use of application knowledge embodied as properties on task graph edges (i.e., connections between threads and channels) to help determine more efficient guarantees.

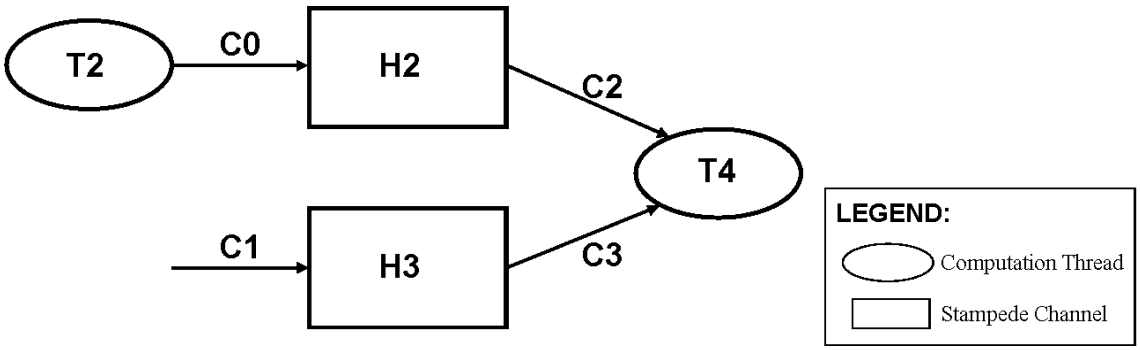

Figure 6: An Example of a Dependent Task Graph: thread $T4$ gets a timestamp from connection $C2$ only if it gets the same timestamp from connection $C3$. Connection $C2$ is said to be locally dependent on connection $C3$.

5.3 Progress of Virtual Time, Dependent Connections, and Monotonicity

Connections in streaming applications can be associated with a virtual time range corresponding to the timestamps they may receive (*get* from a channel) or send (*put* to a channel). The upper bound of this range is simply the highest timestamped item that may be gotten or put on the connection. The lower bound, below which items may be garbage collected, can be determined according to data dependencies at the node's input connections. For example, consider the common case of a thread's input connection, where the thread issues a command to get the latest timestamp (T) on an input connection. As part of managing its own virtual time, the thread may issue a command that guarantees it is no longer interested in any timestamp below T on that connection. Such a guarantee from a thread on an input connection indicates that timestamps less than T are irrelevant (and can be eliminated from the channel) so far as this input connection is concerned.

The upper and lower bounds of the virtual time range may increase as new timestamped items are gotten over the connection; however, in any case, both upper and lower bounds do not decrease.

Dependent Connection: Dependencies among connections are a common characteristic of streaming applications. For example, consider a stereo vision application, where a thread gets the latest data item from one channel and checks other channels for an item

with a matching timestamp. Figure 6 illustrates an example of these dependent connections. Assume that thread $T4$ gets a timestamp from connection $C2$ only if it gets the same timestamp from connection $C3$. Connection $C2$ is said to be locally dependent on connection $C3$. This relationship is not commutative, i.e., the relationship "$C2$ depends on $C3$" does not imply that "$C3$ depends on $C2$". Indeed, in this case only $C2$ is dependent on $C3$.

Monotonicity: Monotonicity is a connection property guaranteeing that transfers of timestamps across a connection are always performed in the forward direction of time, i.e., items that are transferred across the connection will have an increasing timestamp value. One may consider Monotonicity as a static attribute for all input connections (i_conn) and output connections (o_conn) of a thread t. We can define a Boolean variable that can be queried to determine whether a connection is monotonic:

read_monotonic (ic or oc | ic ← i_conn(t), oc ← o_conn(t)) = {true|false}

Monotonicity may also be viewed as a type of dependency where a monotonic connection C is loosely dependent on itself. The next timestamp from C to be processed must be greater than the last one processed. But this view is not strictly limited to monotonic connections. Every connection is, therefore, locally dependent, either on itself or on some other connection. A local dependency results in a *local guarantee*. Dependencies in general and monotonicity in particular form the basis of the DGC algorithm, which takes local guarantees and combines and propagates them to produce *transitive guarantees*.

5.4 Forward and Backward Processing

The input to the DGC algorithm is the application specified task graph in the form of input and output connection information between thread and channel nodes, along with the associated monotonicity and dependence properties of these connections. The algorithm has two components: forward and backward processing of guarantees.

Forward processing at a node N computes the forward guarantee as the *lower bound* on timestamps that are likely to leave N. Similarly, backward processing at a node N computes the backward guarantee as the lower bound on timestamps that are dead so

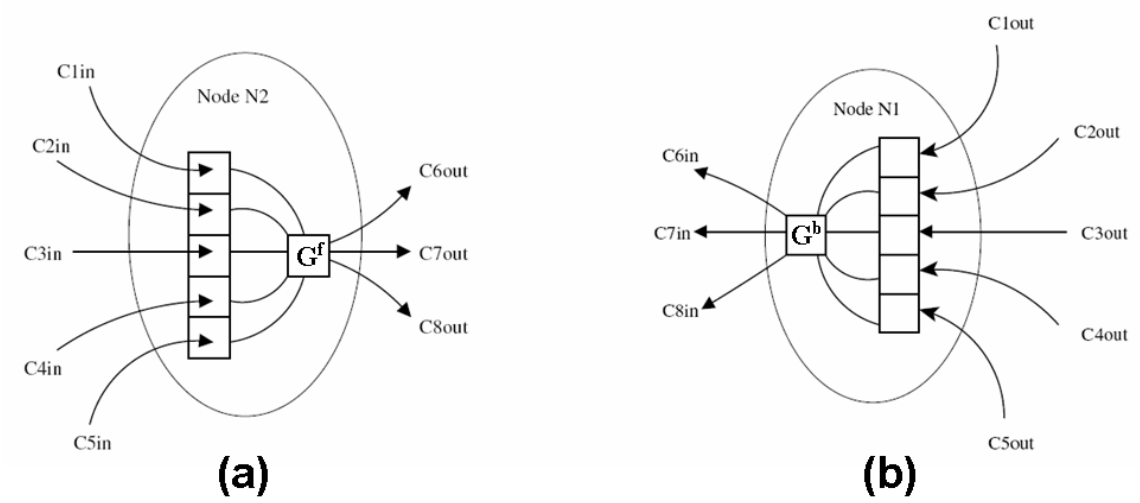

Figure 7: Forward Guarantee Vector $GVec_N^f$ (a) and Backward Guarantee Vector $GVec_N^b$ (b).

far as N is concerned. These bounds act as a *timestamp marker* that separates relevant timestamps (with higher value) from irrelevant timestamps (equal or lower value). Forward and backward processing algorithms generate the guarantees by using these locally available markers on the connections that are incident at each node. In the Stampede runtime system, these algorithms execute at runtime when an item is transferred during put and get operations. Thus, the process of updating guarantees is associated with the flow of items through an application pipeline. In particular, as a timestamped item is transferred from node $N1$ to node $N2$, the algorithm updates the forward guarantee at node $N2$ and the backward guarantee at node $N1$. This enables continual dead timestamp identification at both nodes.

Figure 7 illustrates the components involved in processing forward and backward guarantees. On Figure 7 (a), node $N2$ has input connections, $C1_{in}$ - $C5_{in}$ and output connections $C6_{out}$ - $C8_{out}$. Each node maintains a vector of forward guarantees $GVec_N^f$. There is a slot in this vector for each input connection (in this case the connections involved are $C1_{in}$ - $C5_{in}$). Each slot of the vector, $GVec_N^f[i]$, holds the last forward guarantee communicated to the node over Ci_{in}. These guarantees are simply the timestamp markers associated with the respective connections. Forward processing at a node N involves computing the MIN of the elements of this vector and maintaining it as the Forward Guarantee for this node

N, labeled G^f.

Figure 7 (b) illustrates the components involved in processing a backward guarantee. The arrows indicate the direction the backward guarantees flow, this direction is opposite to the direction data items flow. Node $N1$ has input connections, $C6_{in}$ - $C8_{in}$ and output connections $C1_{out}$ - $C5_{out}$. Each node maintains a vector of backward guarantees $GVec^b_N$. There is a slot in this vector for each output connection (in this case, the output connections involved are $C1_{out}$ - $C5_{out}$). Each slot of the vector, $GVec^b_N[i]$, holds the last backward guarantee communicated to the node over Ci_{out}. These guarantees are the timestamp markers associated with the respective connections. Backward processing at a node N involves computing the MIN of the elements of this vector, $GVec^b_N$, and maintaining it as the *Backward Guarantee*, G^b, for this node N.

The Backward Guarantee for node N identifies dead timestamps for this node. If the node is a channel, items in the channel with timestamps that are dead can instantly be identified as garbage items. Timestamps that arrive at a channel where they have been previously determined to be dead are *dead on arrival* and are not placed in the channel. If the node is a thread, dead timestamps that have not yet been computed by that thread are dead computations and are not computed.

The forward guarantee, G^f_N , at node N is calculated using the *guaranteed lowest forward timestamp*, GLF^f_N , which is computed by applying a MIN operation on the Forward Guarantee Vector, $GVec^f_N$. If the node is a thread, its virtual time, $VT(t)$, provides an opportunity to make an even tighter bound on the forward guarantee. As described in Chapter 4.2, the thread controls the timestamps it can produce on its output connections by setting the thread virtual time, $VT(t)$. The runtime system can set a more strict forward guarantee bound for a thread by taking the maximum between GLF^f_N and the thread virtual time, $VT(t)$. If the node is a channel, on the other hand, the forward guarantee, G^f_N, has to take into account items not yet consumed on the channel in addition to GLF^f_N. The forward guarantee in this case is set to the minimum of the two. Listing 5.1 describes the calculation of the forward guarantee.

The backward guarantee or the lower bound on timestamps that are dead, G^b_N, for a

```
GLF_N^f = min{GVec_N^f}
if (N is a thread t)
    G_N^f = max{GLF_N^f , VT(t)}
else (N is a channel c)
    G_N^f = min{GLF_N^f , UNCONSUMED ts in c}

return G_N^f
```

Listing 5.1: Forward Guarantee

node N is described in Listing 5.2.

If node N is a channel, then the backward guarantee, G_N^b, at the node is calculated by applying a MIN operation on the Backward Guarantee Vector, $GVec_N^b$.

If node N is a thread, the backward guarantee computation is more complicated as it also involves information derived from dependent input connections. This information, captured by the variable $T_{interest}$, allows the algorithm to make the backward guarantee even tighter. Changes in the states of input connections are done only in conjunction with get and put operations, and affect only a single input connection at a time. Therefore, it is sufficient to analyze how changes in that specific input connection (Cin) affect the information derived from dependent input connections and as a result also the backward guarantees.

The algorithm undergoes several steps, each time incorporating more information to hone a tighter value for the backward guarantee, G_N^b. First, the algorithm computes a lower bound on the items of interest to the thread incorporating information available on this particular connection. Any item with a timestamp less than the minimum value of interest to the thread on this connection, or a timestamp that is not going to be produced by the thread can be considered irrelevant, or dead, as far as the thread is concerned. As such, we can express the first lower bound of G_N^b as the maximum of:

1. $KT(Cin)$ - the keep time value on Cin. This is the minimum timestamp value of items that the thread, N, is interested in getting on its input connection, Cin.

2. $GVec^f[Cin]$ - the last forward guarantee (i.e., the lower bound on timestamps that are likely to leave the connection) over connection Cin.

```
if N is a channel
   G^b_N = min{GVec^b_N}
else if N is a thread
   if(read_monotonic(C_in) == true)
      T_interest = max{KT(C_in), GVec^f[C_in]}, max{SEEN(C_in}}
   else
      T_interest = max{KT(C_in), GVec^f[C_in]}

   C'_in = locally_dependent_connection(C_in)

   if(C_in != C'_in)
      if(read_monotonic(C'_in)
         T'_interest = max{T_interest, max{SEEN(C'_in)}}
      else
         T'_interest = max{T_interest, KT(C'_in)}

   else
      T'_interest = T_interest

   GLA^b_N = min{GVec^b_N}
   G^b_N = max{GLA^b_N, T'_interest}

return G^b_N
```

Listing 5.2: Backward Guarantee

If Cin is monotonic it is possible to achieve a tighter guarantee by incorporating information about the maximum timestamp seen by the connection (max{SEEN(C_{in})}). As stated earlier (Chapter 5.3), monotonicity is the property of an entity (in this case, a connection) to constantly pass data items with non-decreasing values of timestamps. If a timestamp value has already been seen on a monotonic connection, then an item with a lesser value will not pass through that connection.

Incorporating information from dependent connections has the potential of tightening the value of the backward guarantee even further, regardless of whether or not the connection is monotonic. This information is captured by $T'_{interest}$, a variable that updates the value of $T_{interest}$ calculated earlier. The function *locally_dependent_connection (Cin)* is user defined and returns an input connection, $C'in$, on which Cin is dependent. If a dependent connection is identified, the algorithm checks for the maximum timestamp value

this dependent connection is guaranteed not to produce. KT(C'_{in}), the keep time, or the minimum timestamp value of interest, on the dependent connection, C'_{in}, serves as a bound for all types of dependent connections (monotonic and non-monotonic). If the dependent connection is monotonic, this initial bound can be made even tighter by incorporating max{SEEN(C'_{in})}, the maximum timestamp seen on this dependent monotonic connection. If there is no dependent connection, however, the bound on G^b_N is not changed during this phase.

Lastly, G^b_N incorporates the minimum value of the backward guarantee vector, $GVec^b_N$. This is the only operation performed in cases where node N is a channel and not a thread.

5.5 Transfer Functions

The application task graph describes dependencies between the output of one thread and the input of another. We use this knowledge to propagate the forward and backward guarantees between threads. However, there are also dependencies within a given thread between its own input connections and its own output connections. These are manifested by the computation within the thread. For example, it is conceivable that not all input connections to a thread node play a role in determining the timestamps on one of its output connection. If this application knowledge is made available to the algorithm determining forward and backward guarantees, then the guarantees produced would be more aggressive. DGC algorithm can harness application knowledge, encoded as transfer functions, to further hone the algorithm's ability to generate forward and backward guarantees thus determining tighter bounds on irrelevant timestamps.

The mechanism used to capture this application knowledge, as previously mentioned, is *transfer functions*. These functions describe the dependencies between a thread and its input and output connections. There are two types of transfer functions: a *forward* and a *backward* transfer function. A forward transfer function (T^N_f) is defined for each output connection from a node and a backward transfer connection (T^N_b) is defined for each input connection to a node. $T^N_f(Cout)$ is the set of input connections of node N that influence the output connection $Cout$. Similarly, $T^N_b(Cin)$ is the set of output connections of node

```
GLF_N^f = min{GVec_N^f[Ci] iff Ci ∈ T_N^f(Ci_out)) ∀ o_conn Ci_out}
if (N is a thread t)
    G_N^f = max{GLF_N^f, VT(t)}
else (N is a channel c)
    G_N^f = min{GLF_N^f, UNCONSUMED ts in c}

return G_N^f
```

Listing 5.3: Transfer Function Effect on Forward Guarantee Calculation

N that are influenced by the input connection Cin.

If node N is a thread, T_f^N and T_b^N are made available to the runtime system by the application developer. However if N is a channel, $T_f^N(Cout)$ is the set of all input connections to N, and $T_b^N(Cin)$ is the set of all output connections from N. If transfer functions are not provided for thread nodes, we conservatively assume that all input connections influence all output connections. Listing 5.3 describes the effect a transfer function has on the forward guarantee calculation.

The following example illustrates how transfer functions are used by the forward and backward processing algorithms to generate tighter bounds for dead timestamps. In Figure 6, assume that input connection $C2$ depends on connection $C3$ in the following manner. when thread $T4$ gets the latest timestamp from connection $C3$ (say this is t), it performs a *get* operation from connection $C2$ for the same timestamp t. Thus, the timestamps the application gets on connection $C3$ affect the timestamps the application gets on connection $C2$. Note that the fact that $C3$ is dependent on $C2$ does not necessarily mean $C2$ is dependent on $C3$. Quite the contrary. In this case $C2$ is not dependent on $C3$. A backward transfer function can capture this dependency on $C3$ and allow $C2$ to eliminate data items based on timestamps that are gotten on connection $C3$. Figure 8 illustrates a snapshot of a dynamic state of the application depicted in Figure 6. Under these circumstances, the highest timestamp on channel $H3$ is 14. Channel $H2$ contains timestamps 7, 8 and 9. Thread $T2$ is about to compute timestamp 10. When thread $T4$ gets timestamp 14 from connection $C3$ it will then wait for timestamp 14 from connection $C2$. The backward transfer function will help backward guarantee processing at thread $T4$ to compute the backward guarantee

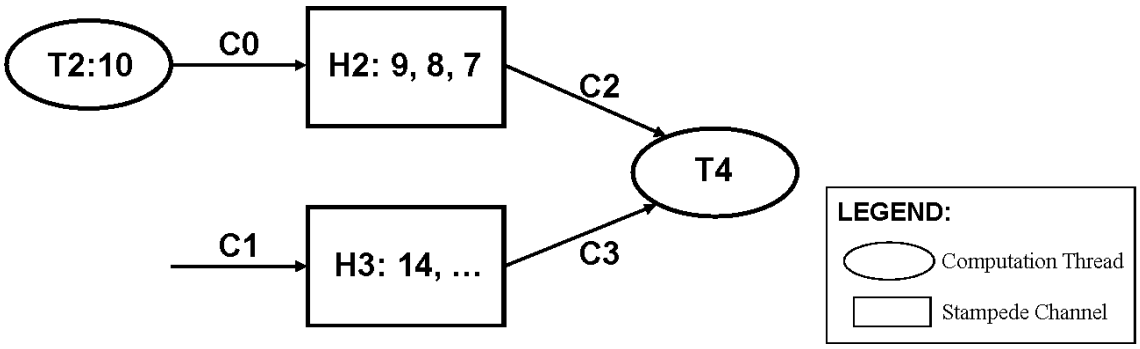

Figure 8: Example of dead timestamp elimination: This example illustrates a snapshot of a dynamic state of an application. Connection $C2$ is dependent on connection $C3$. According to this dependency, once channel $H3$ puts a data item with timestamp 14, connection $C2$ accepts only a data item with timestamp 14. Thus channel $H2$ can eliminate any data item with a timestamp below 14. When this information is passed to thread $T2$, it can refrain from generating data items with timestamp below 14, thus creating the conditions for computation elimination.

on connection $C2$ as 14, thus allowing channel $H2$ to denote timestamps less than 14 as garbage (i.e., timestamps 7, 8, and 9) and eliminate them; this information in turn will tell thread $T2$ to eliminate thread steps that produce timestamps 10, 11, 12 and 13 as dead computations.

5.6 Garbage Collection

The forward and backward processing algorithms utilize the connection properties (monotonicity and dependency) together with the (optional) transfer functions to locally determine the node guarantee G_N. Each node uses G_N as the GC condition for garbage collection. A timestamp that is less than G_N is considered garbage and can either be collected (if the node is a channel) or not be computed (if the node is a thread).

5.7 Implementation

The dead timestamp identification algorithm, as described earlier, is implemented in the Stampede runtime system. This implementation allows a node (which can either be a channel or a thread) to propagate timestamp values of interest forward and backward through the dataflow graph (of channels and threads) that represents the application. The DGC implementation assumes that the application dataflow graph is fully specified at application

startup time. Applications with static characteristics certainly satisfy this condition. Additionally, applications where a maximal form of the dataflow graph can be fully specified at application startup time also satisfy this condition. This maximal form represents each and every data flow possible during any program execution regardless of the circumstances. The application may activate or deactivate specific data flow connections during the program execution depending on the dynamic conditions of the application. Thus, unlike REF, this implementation of the DGC algorithm supports application dynamism, although in a limited form compared to TGC.

It is possible, however, to extend the level of application dynamism the DGC algorithm supports by incorporating the forward and backward guarantees into the rules that determine the virtual time of a thread. The forward and backward guarantees create constraints upon the timestamp values that a thread may generate. The current DGC implementation does not account for the effects these guarantees may have on the virtual time of a thread. As long as the application graph is fully known at the application startup time there is no danger of violating the thread virtual time rules. However, any dynamic change in the graph (e.g., adding a connection to a thread) may change the virtual time that is associated with a thread. Thus, in order to fully support dynamic graphs, the runtime system has to incorporate the changes that forward and backward guarantees may exert on threads virtual times upon any change in the application graph.

Forward propagation is instigated by the runtime system upon a put or a get operation on a channel. For example, when a thread performs a put on a channel, a lower bound value for timestamps that the thread is likely to generate in the future is enclosed by the runtime system and sent to the channel. Similarly upon a get from a channel, the runtime system calculates a lower bound for timestamp values that could possibly appear in that channel and piggybacks that value on the response sent to the thread.

Backward propagation is similarly instigated by a put or a get operation. In fact, backward propagation is likely to be more beneficial in terms of performance due to the properties of monotonicity and dependence on other connections, which we described in Chapter 5.3. These properties come into play during a get operation on a channel.

The Stampede API is extended to enable a thread to inquire the forward and backward guarantees so that it may incorporate these guarantees in its computation.

The dead timestamp identification algorithm requires the application writer to provide monotonicity and dependency information to the runtime system. This information assists in identifying dead timestamps.

This extended implementation of Stampede creates only a very minimal burden at the application level. Specifically, the application is tasked with providing the monotonicity and the dependency information of a given connection (if any) on other connections. In addition, the application has the option to provide handler functions that the runtime system can then call during execution to determine the forward and backward transfer functions for a given connection.

DGC algorithm offers two specific avenues for performance enhancement compared to REF and TGC. First, it provides a unified framework for both eliminating unnecessary computation from the thread nodes and the unnecessary items from the channel nodes as compared to the REF and TGC algorithms that perform only the latter. Secondly, the DGC implementation allows the runtime system to eliminate items from the channels more aggressively compared to REF and TGC using the application level guarantees of monotonicity and dependency for a connection.

CHAPTER VI

EVALUATING DEAD TIMESTAMPS BASED GARBAGE COLLECTOR (DGC)

We evaluate the dead timestamps based garbage collector (DGC) by comparing its performance with REF and TGC.

6.1 Environment

All three GC strategies are implemented in the Stampede runtime system [45]. As mentioned in Chapter 2, the Stampede programming abstractions are implemented as a C runtime library on top of a standard operating system and messaging libraries. The Stampede runtime system is available on a variety of platforms along with the x86-Linux platform used for this study. Specifically, we use a 17-node cluster of SMPs interconnected by Gigabit Ethernet. Each node is an 8-way SMP, comprising of 550 MHz Intel Pentium III Xeon processors with a 4GB of physical memory at each node. The Linux kernel used is Redhat 2.4.20, along with a reliable UDP messaging library called CLF (Cluster Language Framework). More details regarding CLF are provided at [38].

6.2 Metrics and Methodology

We define the following metrics for the performance evaluation of the GC algorithms.

Average Channel Occupancy Time - This metric quantifies the average time spent by items with the *same* timestamp in all channels before they are garbage collected[1].

Average Pipeline Latency - This metric quantifies the average latency experienced by items with the same timestamp that make their way through the entire application pipeline. Latency is measured from the time an item enters the first pipeline stage, until it is leaves the last pipeline stage[2].

[1]Note that the focus is on a given timestamp making its way through the pipeline of channels. The item bearing this timestamp morphs as it moves through different stages of processing in the pipeline

[2]Note that the focus is on a given timestamp making its way through the pipeline of channels. The item

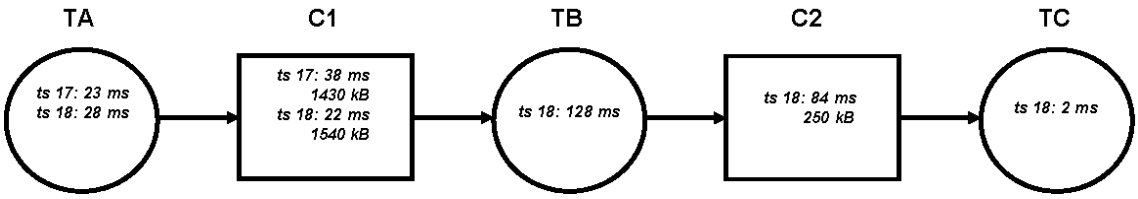

Figure 9: Example of Metrics on a Simple Pipeline: The pipeline has three threads (TA, TB, and TC) and two channels ($C1$ and $C2$).

Average Memory Usage - This metric quantifies the aggregate memory used by the application in all the channels averaged over the execution time of the application. It is a measure of the average memory pressure exerted by the application during the course of the execution. The more aggressive GC strategy the system uses, data items are reclaimed faster and the memory pressure exerted on the system is reduced.

Figure 9 provides an example of a simple pipeline that has three threads: TA, TB, and TC, and two channels: $C1$ and $C2$. The figure illustrates a snapshot of the computation and memory consumption of two items with timestamps 17 and 18. Thread TB always requests the latest item from channel $C1$ and receives the item with timestamp 18. The total occupancy time of the item with timestamp 17 is 38[ms]. The total occupancy time for timestamp 18 is the sum of the time the item spent in channels $C1$ and $C2$, which is: 106[ms]. The average occupancy time for both items, is therefore, 72[ms] per item. The average memory usage is calculated in a similar fashion, and is: 1610[kB]. The pipeline latency can be calculated here only for timestamp 18, as the item with timestamp 17 did not reach the end of the pipeline. This latency includes all the time the item spent in the threads and channels of the pipeline. In this case, the pipeline latency is: 264[ms].

Comparison with IGC - We define an Ideal Garbage Collector (IGC) [29] as one that garbage collects all data as soon as possible. It also knows *a priori* those items that will not reach the end of the application pipeline, and thus can be considered irrelevant. IGC assumes these items are not produced in the first place, and does not account for any memory associated with them. Therefore, the memory footprint associated with the IGC equals the memory that is required to buffer only relevant items of the application. Thus

bearing this timestamp morphs as it moves through different stages of processing in the pipeline

the memory footprint recorded by IGC represents a lower bound for the memory application consumes, and therefore may serve as a "gold" standard against which different garbage collection algorithms and strategies can be compared.

Obviously, an Ideal Garbage Collector cannot be implemented, as it requires predicting whether an item will be considered relevant. This assertion can only be made after an item successfully reaches the end of the application execution pipeline. Hence, the application or the runtime system cannot know whether an item is relevant for a majority of the time an item is in the pipeline and considered to be live and not garbage. Nevertheless, we can simulate the behavior of a hypothetical IGC with the aid of runtime event logs and a post-mortem analysis technique. This strategy is akin to using trace-driven simulation in system studies. In these studies, trace-driven simulations are used to evaluate possible improvements to a system as well as predict the performance gains expected from these improvements.

For the purpose of simulating IGC, we used an execution trace of an actual application run using DGC. This execution trace records events such as get, put, and consume in trace logs with their respective times of occurrence. The logs also contain the GC times for all items on all channels. IGC can be simulated using this data by considering only the events related to relevant timestamps, that is, the exact set of timestamps successfully reaching the end of the application pipeline.

6.3 Applications

6.3.1 Color Tracker

The color tracker application (Figure 10), developed at Compaq CRL [47], tracks the locations of multiple moving targets on the basis of their color signatures. This application performs a real-time color analysis of video frames and compares the results against multiple color models. It operates on a live video stream, which continuously captures images. It then digitizes these images, and passes them down the pipeline for analysis and target matching.

A digitizer produces a new image every 30 milliseconds. In the setting of this experiment,

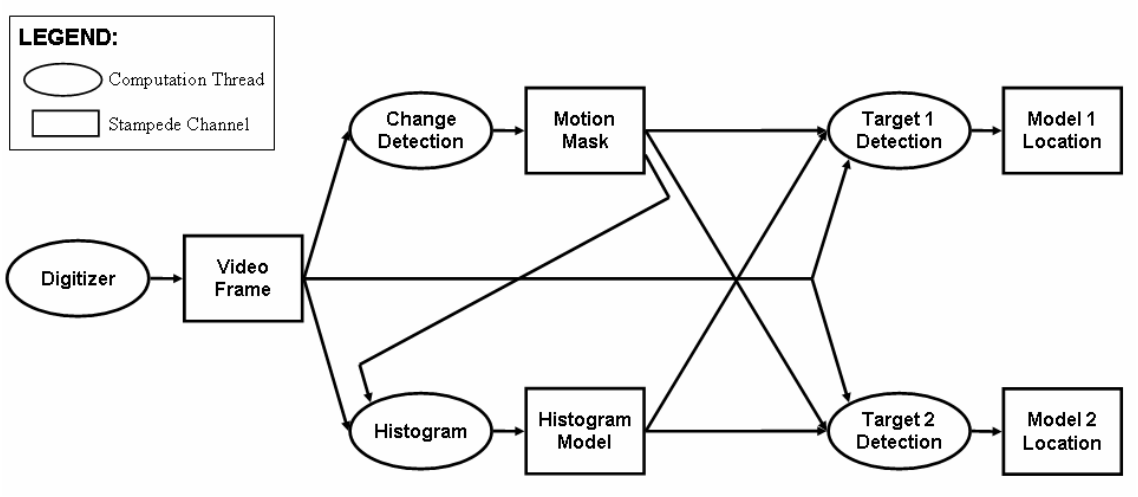

Figure 10: Color Tracker Task Graph: the Color Tracker tracks specific models in a scene. The tracked models are predefined and are provided through the Target Detection threads. A Digitizer generates video frames that are put in the Video Frame channel. The remaining threads: the Change Detection, the Histogram, and the Target Detection threads, require these video frames to complete their task. Thus, the Color Tracker is an example of a highly connected application.

the digitizer reads a pre-recorded set of images from a file to have a fair comparison among the three GC algorithms. A digitized image is then passed to both a motion detection thread and a histogram thread. The motion detection thread subtracts the current image from the former image to detect changes in the scene. The histogram thread receives input from both the digitizer and the motion detection threads and generates a color histogram of the image received. The target detection thread uses input from the digitizer, the motion detection, and the histogram threads to locate a pre-determined target in the image. This thread is the slowest stage in the application pipeline and cannot keep up with the digitizer's rate of frame production. Thus, not every image the digitizer produces will propagate through the entire pipeline, and successfully reach the end. Every pipeline stage gets the latest available timestamp from their respective input channel connections. A real deployment of this application may use multiple target detection threads (one per target being tracked). For our experiments, we use two target detection threads, each searching for a different object in the same frame using a color-histogram model unique to the object. With this workload, the average message sizes delivered to the digitizer, motion mask, histogram, and target detection channels are 738KB, 246KB, 981KB, and 67B, respectively.

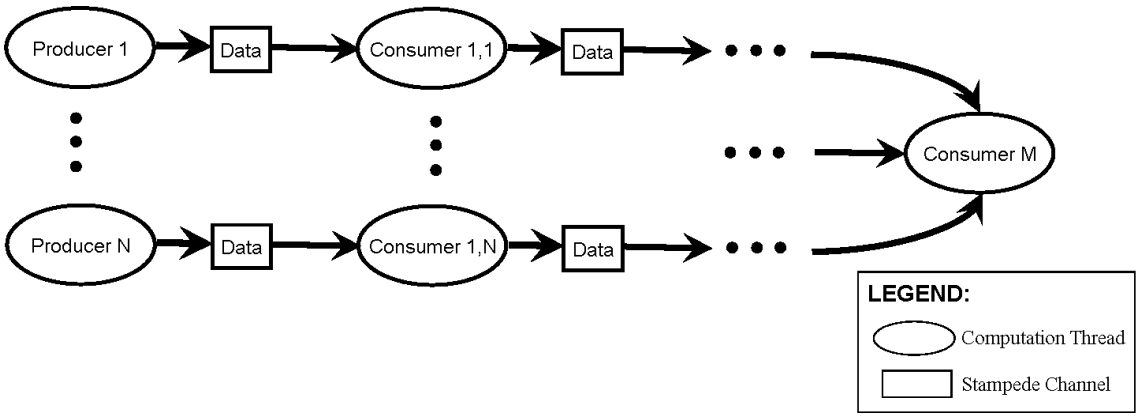

Figure 11: Generic Pipeline Application Task-graph: M Producer/Consumer stages and N-pipelines

6.3.2 Generic Pipeline Application Suite

Figure 11 depicts a generic application pipeline that serves as a synthetic workload generator [44]. It has N number of pipelines and M number of stages in each pipeline. The number of pipelines and stages can vary, and each (M, N) set results in a new synthetic application. In addition to the number of pipelines and stages, one can control the rate the producer threads introduce data items into the application pipeline, and the amount of time each consumer requires to work on the data it receives. The structure of the synthetic application lends itself to stress testing the scalability of the GC algorithms. DGC, for example, depends on the propagation of guarantees to neighboring stages; therefore, the length of the pipeline determines the delay experienced for such propagation. Similarly, TGC requires computation of a global virtual time (GVT) that involves communication among all stages, and thus is affected by the total number of pipelines and stages in the application.

6.3.3 A Synchronized Data-Stream Application

As we mentioned earlier in Chapter 5.2, DGC can aid both in eliminating dead items from channels and dead computations from threads. Dead computations can only result if later stages of the pipeline can indicate to earlier stages their lack of interest for some specific timestamp values. Figure 12 shows a particular instance of a synthetic application that results in computation elimination [44]. The task graph simulates an application

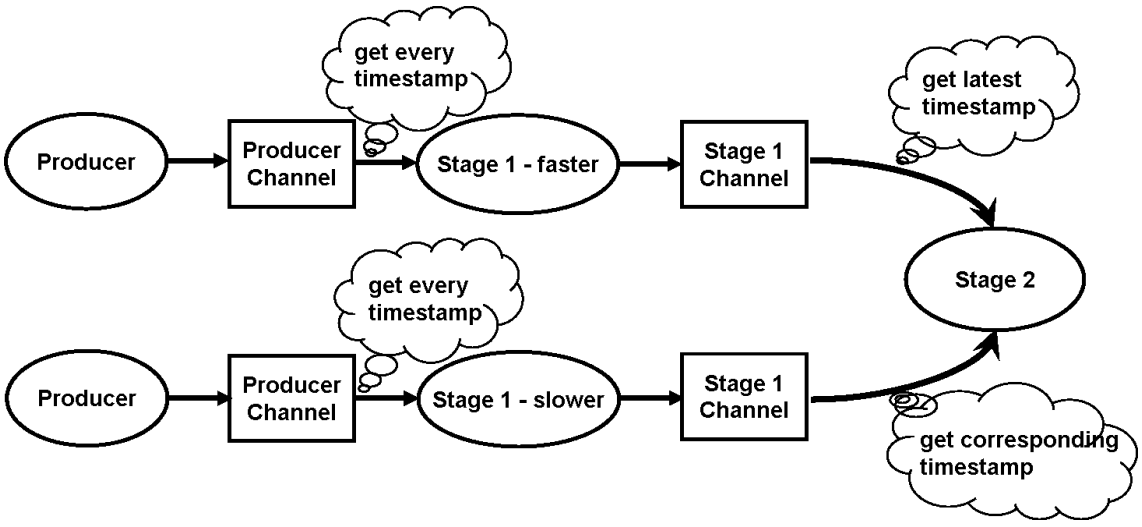

Figure 12: A Synchronized Data Stream Application: Stage 2 must receive corresponding timestamps from both Stage 1 threads. This application setup creates the conditions for dead computation elimination. Stage 2 receives the latest timestamp from the faster Stage 1 thread, and the corresponding timestamp item from the slower Stage 1 thread. The Stage 2 thread receives a timestamp with higher value from the faster Stage 1 thread, and requests the corresponding timestamp from the slow Stage 1 thread. As this thread is slower, it still processes an item with a lower timestamp. Once the slower Stage 1 thread receives the information about the requested higher timestamp from the Stage 2 thread, it can eliminate the processing of all items up to the item with the timestamp that corresponds to the item Stage 2 got from the faster Stage 1 thread, thus performing computation elimination.

where two independent data streams (e.g., two video images, or a video image with an audio sample) need to be correlated. The application has two independent pipelines that include a producer and a first stage processing thread. Results from the two pipelines are combined at the second stage, which processes items with the same timestamp value from both pipelines. The difference in production rate between the two stage-1 threads allows the faster thread to communicate information about dead items to the slower thread, thus creating the conditions for dead timestamps elimination.

6.4 Experimental Setup

6.4.1 Color Tracker

We use two configurations of the color tracker application:

1. 1-node configuration - All threads execute on one physical node within a single address space. The channels are also mapped to the same physical node. No network-related

communication occurs in this configuration.

2. 5-node configuration - Each one of the five threads (and their corresponding output channels) is mapped to a distinct node in a cluster. As a result, the application is distributed over five physical nodes of the cluster. This configuration maximizes the network-related communication of the application.

The 1-node configuration does not perform any network communication, thus it does not require the runtime system to use the messaging layer. Also, as all channels and threads are mapped to the same node, items in channels are directly accessible to each and every thread. Therefore, threads are not required to create a copy of channel items they access, and cache them locally.

The 5-node configuration must use the messaging layer when data is exchanged across node boundaries, hence network latencies affect the application performance. In addition, threads cannot access directly items that are located in channels that are mapped to a different node. In this case, threads locally cache a copy of the item they access. These cached copies increase the memory footprint of the application, and have a direct effect on the garbage identification potential prevalent in the application. Successful items that manage to reach the end of the application pipeline are cached more times (and consume more memory) than items that the application drops. As a result, under the 5-node configuration the application allocates a larger portion of the memory to successful items compared to the 1-node configuration. As a result, we expect the garbage identification algorithms to have a smaller effect on the memory footprint on the 5-node configuration.

6.4.2 Generic Pipeline Application Suite

These synthetic applications are run under resource-hungry and resource-rich conditions. A resource-hungry setting is the most common scenario for a streaming application, where more items are available for processing than can be feasibly processed with the available system resources. To maintain current output, every application stage drops older data items to process only the *latest* item. An experiment using a resource-hungry setting exposes the benefits of an aggressive garbage collector that is faster in identifying irrelevant

timestamps; Items spend less time in channels thus reducing the channel occupancy time. To create a resource-hungry setting, the production rate of each producer thread is set to one item every 30 milliseconds, with each of the Mth stage threads requiring $30 * M$ milliseconds of processing time.

This contention does not exist in a resource-rich environment, where every item can be processed in real-time without creating a backlog of older unprocessed items. Under these conditions, GC is not required to identify and reclaim skipped items, but merely to collect processed items. Again, the speed at which timestamps are identified as garbage helps reduce channel occupancy time. To create such an environment, we adjust the producer threads to output an item every 60 milliseconds, and allow each Mth stage to simply copy a data item from a previous stage.

The number of stages, M, is varied between three (one producer, one processing stage, and one output stage) and eight (one producer, six processing stages, and one output stage). The number of parallel pipelines, N, is varied between one and eight. Each pipeline stage is mapped to one physical node and each item size is set to 64KB at each stage.

6.4.3 A Synchronized Data-Stream Application

The synchronized data-stream application is executed on a single physical node. This application structure, coupled with the significant difference between the processing times of stage-1 threads, creates the condition for dead computation elimination.

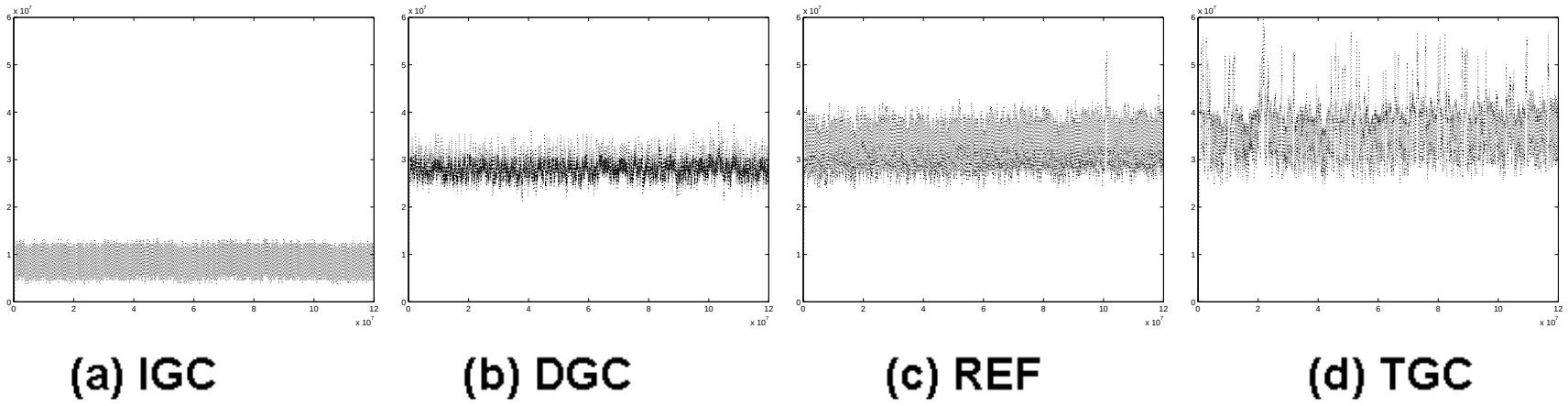

Figure 13: Color Tracker Application: Memory Footprint (5 node configuration) for the (a) Ideal Garbage Collector (IGC); (b) DGC; (c) REF; and (d) TGC.
The y-axis represents the memory usage (in bytes x 10^7); while the x-axis represents the time (in ms).
DGC consumes less memory than REF and TGC for the color tracker application. Moreover, DGC exhibits a more stable memory consumption pattern while REF and TGC exhibits bigger fluctuations from their average memory consumption. However, the Ideal Garbage Collector exhibits a much lower memory consumption than DGC, suggesting the possibility of developing other algorithms that may harness information prevalent in the system, and perform better than DGC and closer to the Ideal Garbage Collector.

6.5 Experimental Results

6.5.1 Memory Pressure Performance

Table 1: Tracker Application Performance Under TGC, REF, DGC, and IGC: DGC consumes 30% and 25% less memory on average than TGC for 1-node and 5-node configurations, respectively, while having almost no affect on the latency. However, DGC still uses 3-4 times more memory than IGC, suggesting that there may exist a potential to decrease the average memory usage even further.

	Config 1: 1 node				
	Pipeline Latency (ms)	Average Memory usage (MB)	Memory usage STD	% w.r.t. IGC	% w.r.t. DGC
TGC	492	24.7	6.3	616	144
REF	480	23.3	6.2	585	135
DGC	506	17.2	4.1	429	100
IGC	N/A	4.0	1.0	100	23
	Config 2: 5 node				
	Pipeline Latency (ms)	Average Memory usage (MB)	Memory usage STD	% w.r.t. IGC	% w.r.t. DGC
TGC	555	36.8	5.7	444	131
REF	557	32.9	4.7	397	117
DGC	558	28.1	2.2	339	100
IGC	N/A	8.3	2.9	100	30

Table 1 provides the latency results of processed timestamp items for the color tracker application. Although latency is higher with DGC, due to inline execution of transfer functions on get and put operations, the percentage increase of latency is marginal. TGC pipeline latency is 492 and 555 [ms] for 1-node and 5-node configurations, respectively. REF pipeline latency is 480 and 557 [ms] for 1-node and 5-node configurations, respectively. DGC pipeline latency, on the other hand, is 506 and 558 [ms] for 1-node and 5-node configurations, respectively. These results represent a 2.8% and a 0.5% increase in average latency compared to TGC and a 3.3% and a less than 0.1% increase in average latency compared to REF for 1-node and 5-node, respectively.

Although the latency is slightly increased, DGC exhibits superior memory footprint performance compared to REF and TGC as can clearly be seen in Figure 13.

In addition, Table 1 shows a low mean for memory usage in DGC compared to both TGC and REF (TGC uses 44% and 31% more memory than DGC, REF uses 35% and 17% more memory than DGC for 1-node and 5-node configurations, respectively). The low memory usage compared to TGC is expected due to the aggressive nature of DGC. However, the performance advantage of DGC compared to REF is quite interesting as both use local information to determine whether or not an item is garbage and should be collected. REF makes local decisions on items in a channel once the consumers have explicitly signaled a set of items as garbage. DGC has an added advantage over REF in that it propagates guarantees to upstream channels thus enabling dead timestamps to be identified much earlier, resulting in a smaller footprint compared to REF. Yet, even DGC, with the least memory usage, trails far behind IGC (on average, DGC uses 4.29 and 3.39 times more memory than IGC for 1- and 5-node configurations, respectively). These results indicate a potential to incorporate additional information regarding dependencies among items produced by an application to achieve a memory footprint closer to IGC (see Chapter 7).

6.5.2 Scalability of GC Algorithms

Figure 14 shows that in a resource-rich environment, timestamps linger in channels more than twice as long for TGC compared to either DGC or REF. The occupancy times for the latter two are roughly unchanged with the number of pipelines, whereas the occupancy time increases for TGC. This is to be expected since, unlike REF and DGC, the communication the TGC algorithm requires is proportional to n^2, where n is the number of address spaces in the experiment. The global nature of TGC, which requires information to propagate to all application nodes in order to reach a garbage identification decision, makes this decision sensitive to network delays. These delays are exacerbated by network contentions that are a direct result of the large increase in the amount of messages generated as the number of address spaces increases. These results suggest TGC does *not* scale well in comparison to DGC and REF. Again, this is primarily due to the local nature of both DGC and REF and the global nature of the TGC algorithm, which is far more conservative. DGC identifies garbage using the *local timestamp guarantee* G_N at node N, and REF uses a local

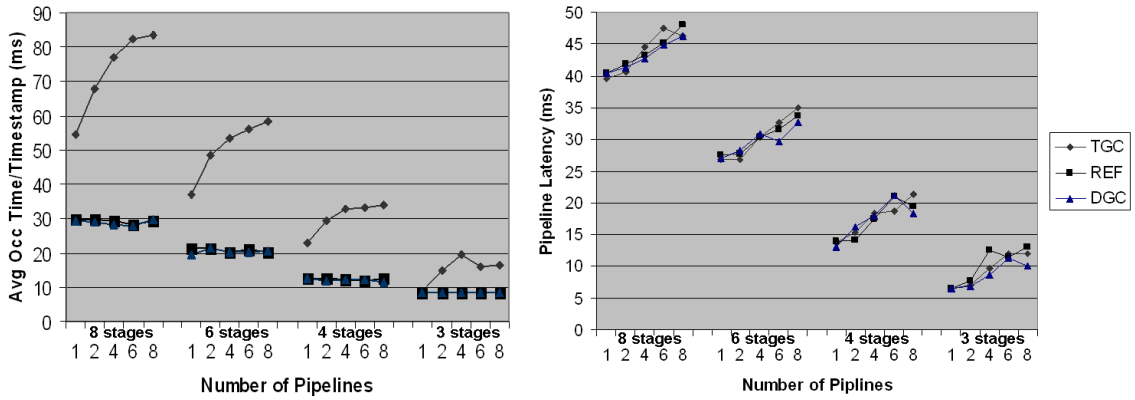

Figure 14: Generic Pipeline Application (resource-rich environment): Average Channel Occupancy Time (left) and Pipeline Latency (right). REF and DGC, that perform local decisions, scale well. TGC, on the other hand, makes global decisions based on global information and does not scale well.

consumer count defined for each timestamped item. Such local mechanisms allow DGC and REF to identify irrelevant timestamps faster, with a minimal number of messages exchanged between nodes. Note that the in-line nature of DGC and REF does not affect performance as can be seen in the latency results (see Figure 14).

As we mentioned earlier, in a resource-hungry environment timestamps are skipped since every stage attempts to get the latest timestamp from its input channel. Figure 15 reveals similar trends in a resource-hungry environment to those observed in the resource-rich environment. An interesting result is seen in the latency graph: Initially, as the number of pipelines increases, the latency actually *reduces* for DGC. This is a result of the synthetic application structure that enables dead computation elimination to occur under DGC. In every iteration, Consumer M in Figure 11 requests the *latest* timestamp from pipeline 1 (top pipeline), and the consequent guarantee G_N^b is propagated to all other pipelines. This results in computation elimination for the lower pipelines. The successful timestamps that make it to Consumer M from these lower pipelines incur less latency thus lowering the average pipeline latency for successful items. However, when the number of pipelines increases, the guarantee G_N^b from Consumer M reaches the threads of higher order (Nth) pipelines too late to cause dead computation elimination to occur. Consequently, the average pipeline latency *increases* beyond a given number of pipelines (8 pipelines).

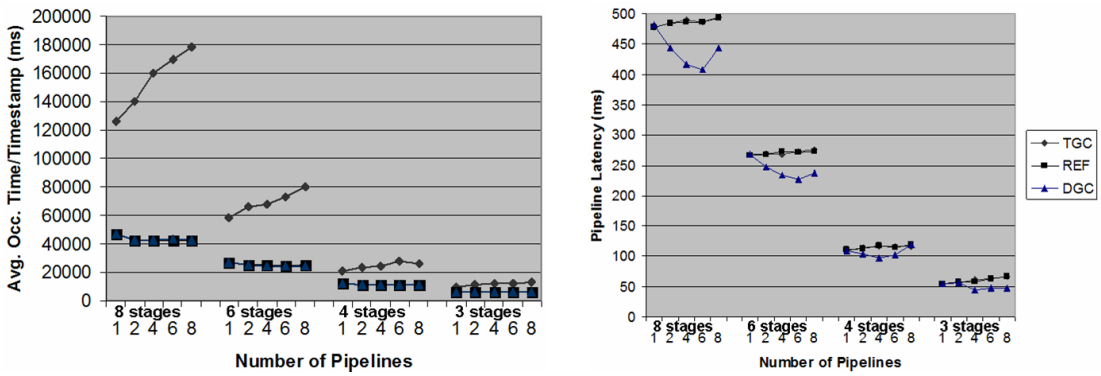

Figure 15: Generic Pipeline Application (resource-hungry environment): Average Channel Occupancy Time (left) and Pipeline Latency (right). REF and DGC, that perform local decisions, scale well. TGC, on the other hand, makes global decisions based on global information and does not scale well.

6.5.3 Dead Computation Elimination Results

Table 2: Synchronized data-stream application: The effects of Dead Computation Elimination (DCE) on the performance of the application. DCE affects not only the average memory usage, but also the average latency, as resources are directed away from computations that will be dropped in the future, and will not reach the end of the pipeline.

Parameter	DCE Active	DCE Inactive
Timestamps produced	2,343	2,088
Timestamps fully processed	999	81
Avg. Pipeline Latency	88,419 [ms]	1,735,125 [ms]
Avg. Memory Usage	248 [KB]	470 [KB]

This experiment has relevance only for the DGC algorithm and it demonstrates the significance of dead computation elimination. The DGC algorithm can be viewed as having two components:

1. Dead item GC that identifies and reclaims dead items on channels; and,

2. Dead computation elimination that identifies and refrains from computing dead items on threads.

For this experiment we execute the synchronized data-stream application (Figure 12) in two configurations: one with both dead item garbage collection and dead computation elimination enabled, and, the other with only dead item GC activated.

The results (Table 2) demonstrate the role dead computation elimination plays in improving the overall performance of this application. When dead computation elimination is active, 999 out of 2,343 (or 42.6%) timestamped items manage to make it through the application pipeline, as oppose to only 81 out of 2,088 (or 3.9%) when dead computation elimination is inactive. Other performance metrics (such as latency and average memory usage) confirm the significance of dead computation elimination.

6.6 Conclusions

Each one of the three GC algorithms we presented so far (REF, TGC, and DGC) represents a trade-off between the level of application knowledge available to the runtime system and the aggressiveness of the garbage collection algorithm. TGC uses no application knowledge, and implements a distributed algorithm to determine a low water mark for timestamp values of interest to the application. REF uses a statically encoded per item reference count as the only application knowledge. DGC uses information about dependencies between input and output data streams to make runtime decisions on timestamp values of interest to a thread or a channel.

TGC allows flexibility in application dynamism since it is not sensitive to application details, but makes conservative decisions regarding GC. Both REF and DGC make aggressive decisions regarding GC but, cannot transparently accommodate dynamic changes to the thread-channel graph. Further, DGC unifies elimination of unneeded computation and storage in one single framework. We have shown that this aggressiveness pays off: DGC improves the memory footprint of the tracker application by over 30%, and like REF, also shows better scalability with increased application size, compared to the global alternative of TGC. We have also described conditions that allow dead computation elimination to occur with DGC algorithm, and have quantified the performance gains with a synthetic application.

The choice of a specific algorithm is a trade-off between programming ease and performance gain. Further, an application may choose to use all three algorithms simultaneously. TGC represents the maximum programming ease as the runtime system transparently takes

care of the garbage collection. REF is not always applicable due to the dynamic properties that are prevalent in some streaming applications. DGC requires the application writer to provide the runtime system with more application knowledge; however, this effort is beneficial in reducing application resource usage.

While DGC succeeds in reducing the memory pressure substantially compared to TGC and REF, it still consumes three to four times more memory than the Ideal Garbage Collector, as defined in [29]. The major difference between the two may suggest that DGC falls short of fully utilizing information that is either available, or can be provided to the runtime system to identify garbage items expediently thus reducing the memory pressure the application exerts on the runtime system. In the next chapter we investigate additional approaches that build on DGC, and may employ more aggressive methods to claim items earlier, and to further reduce the application memory pressure.

CHAPTER VII

BEYOND DGC: EXPLORING ADDITIONAL METHODS FOR MEMORY OPTIMIZATIONS IN STREAMING APPLICATIONS

7.1 Approaches

A comparison between the Ideal Garbage Collector (IGC) simulation and any one of the Garbage Collection algorithms presented so far, reveals that even under the best performing garbage collector, DGC, an application consumes around four times more memory than under IGC. Although IGC is a hypothetical garbage collector that serves merely as an indicator of the garbage collection potential prevalent in the system, this large difference suggests the prospect of developing other garbage collection algorithms that may further reduce the overall application memory footprint. In this chapter, we explore two general approaches:

1. Improve the flow of dependency and dead timestamp information throughout the application pipeline. The DGC algorithm is limited by the speed at which the dead timestamps information can reach the various application nodes. First, dead timestamp guarantees are transmitted only upon get and put operations. Secondly, the information is transmitted to neighboring nodes only, and has to propagate to the rest of the application nodes upon additional get and put operations. Beyond the aforementioned limitations, the DGC algorithm delivers the guarantees as a single number that serves as a watermark. The runtime system does not transmit information about timestamps that are above the watermark, and thus considered dead. Although the runtime system may have enough information to consider some items as garbage, the reclamation of these items must be delayed until they are included in the timestamp range below the watermark. We will suggest possible strategies to overcome these limitations, and explore their potential for reaching a memory footprint closer to that of the ideal garbage collector.

2. Under the DGC algorithm, a timestamp classification, as live or dead, incorporates only certain aspects of the application data dependency information. We explore the potential of using additional information that an application writer may provide. As before, we assess the results by comparing them to the application memory footprint recorded under DGC and the simulated IGC algorithms.

7.2 Proposed Garbage Collection Algorithms

In this section we present four algorithms to optimize the memory footprint of streaming applications. The Keep Latest 'n Unseen (KLnU) involves the application writer providing additional data-dependency information to the runtime system to allow for a more efficient garbage identification. The Propagation of Dead-Set (PDS) algorithm takes advantage of local information about dead items readily available to the runtime system and propagates this information to neighboring nodes creating a tighter classification of items as garbage. The last two algorithms, Out-of-Band Propagation of Guarantees (OBPG), and Out-of-Band Propagation of Dead-Set (OBPDS) explore the potential for alleviating network limitations by simulating an instant transfer of information regarding garbage items.

7.2.1 Keep Latest 'n Unseen (KLnU)

The first algorithm takes advantage of one of the characteristics of streaming applications; namely, that in many cases downstream computations are interested only in the *latest* items produced by an upstream computation. In addition, upstream computations tend to be lighter than downstream computations. This trait of streaming applications result in a large number of items becoming irrelevant. For example, in the experiment involving the color tracker application (presented in Chapter 6.3.1), only one in eight items the digitizer produces succeeds in reaching the end of the application pipeline. The proposed optimization is to associate an *attribute* with a channel that allows it to discard all but the *latest* items. When a producer puts a new item, the channel may immediately denote any item with an earlier timestamp as garbage if that item has not been gotten up until this point on any one of the channel connections. We refer to this set of earlier timestamps identified as garbage items as a *dead set* (see Chapter 7.2.2 for further discussion regarding dead sets).

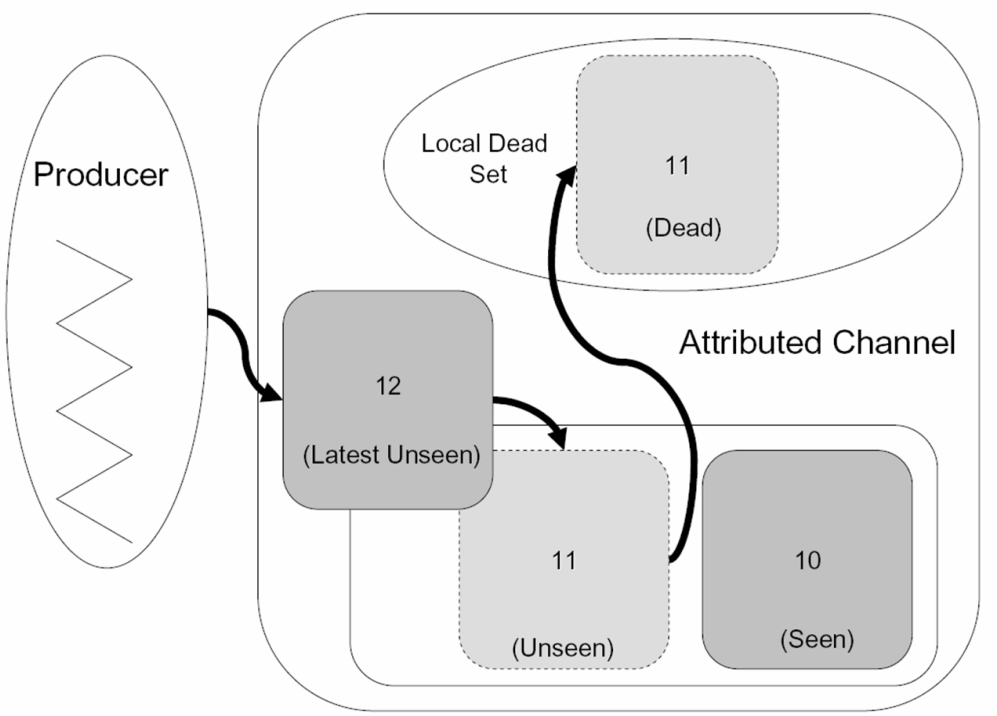

Figure 16: An Attributed Channel: An example of the KLnU optimization, where n=1.

Since a Stampede application depends on timestamp causality, this optimization will not allow an item with an earlier timestamp to be garbage collected even if *one* connection has gotten that item from this channel. We generalize this attribute and call it *Keep Latest 'n Unseen (KLnU)*, to signify that a channel retains only the last n items. The value of n is specified at channel creation time and different channels may have a different value for n. Although, this attribute gives the channel a local control to garbage collect items that are deemed irrelevant, it can only be implemented as an addition to whatever system-wide garbage collection mechanism (e.g., DGC, TGC, etc.) is already in place.

Figure 16 illustrates an example of an attributed channel operating under the KLnU algorithm, where n=1. The producer thread has just produced an item with timestamp 12 while items with timestamps 10 and 11 are already present in the channel. The item with timestamp 10 has been gotten by a consumer, while the item with timestamp 11 has not been gotten by any consumer so far. In other words, the item with timestamp 11 is "unseen" to all of the consumer threads connected to this channel. Thus, upon put of an

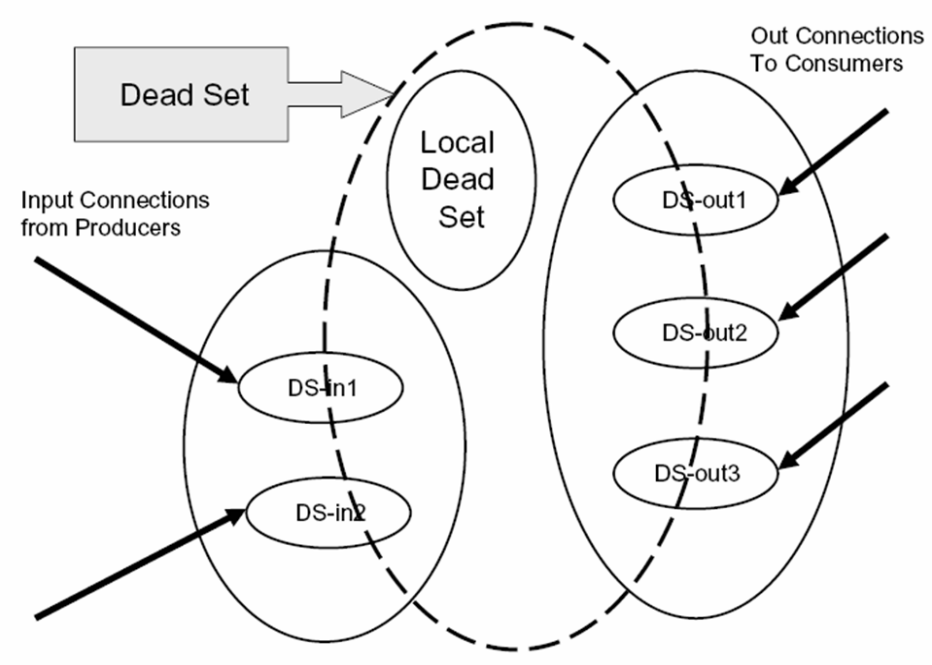

Figure 17: Propagation of Dead Sets: The arrows indicate the direction of flow of dead sets into a given node.

item with timestamp 12, the "unseen" item (that is, the item with timestamp 11) can be added to the dead set of this channel and garbage collected while the "seen" items (in this case, the item with timestamp 10) is retained in the channel.

7.2.2 Propagating Dead Sets

KLnU, presented in Chapter 7.2.1 is just one way to incorporate application knowledge to help in identifying garbage items in a channel more efficiently and generate local dead sets. One may think of other attributes and policies that may fit other data-dependency scenarios and that may generate dead sets within a channel in different ways. However, under each of these possible algorithms, the dead set information stays within a channel and does not propagate to other threads and channels in the application.

The *Propagation of Dead Set* (*PDS*) optimization propagates *local* dead-set information from one node to other parts of the application similar to the way the DGC algorithm propagates forward and backward guarantees (see Chapter 5).

Figure 17 shows the state of a node in a given task graph. This node updates its dead set by incorporating local data dependency information via mechanisms such as KLnU

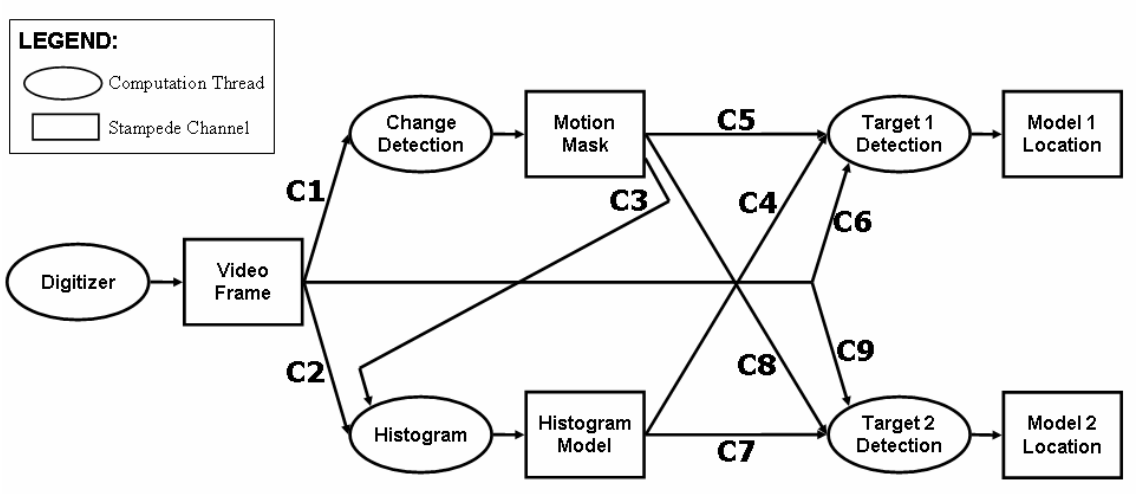

Figure 18: Color Tracker Task-graph and Out Connections

(see Chapter 7.2.1). It also includes propagated dead-set information received from all the output connections (out-edges) due to backward propagation. Similarly, it adds dead-set information to the local set from all the input connections (in-edges) due to forward propagation. The dead-set for a node is computed as the *union* of the local dead-set and the *intersection* of the dead set information of all the in and out-edges (or connections) incident at that node.

In Chapter 5, we introduced the notion of dependency among input connections incident at a node. This information allows timestamp guarantees to be *derived* for dependent connections. For example, if connection A to a channel is dependent on connection B to another channel, and if the guarantee on B is T (that is no timestamps *less than* T will be gotten on B), then the guarantee on A is T as well. Similarly, we propose dependency among output connections. For example, let A and B be output connections from a given channel, and let A be dependent on B. If the timestamp guarantee on B is T (that is no timestamps *less than* T will be gotten on B), then the guarantee on A is T as well.

Table 3 illustrates the connection dependency for the color-tracker application (see Figure 18). As can be noticed from the table, for the video frame channel the output connections to the histogram ($C2$) and target-detection threads ($C6$ and $C9$) are dependent on the output connection to the change-detection thread ($C1$). This dependency implies that timestamps in the dead-set of the change-detection thread would never be gotten by either

Table 3: Color-tracker out connection dependency The dependencies stated here are for the color-tracker application illustrated in Figure 18.

Channel Name	Out Connection	Dependent Connections	Channel Name	Out Connection	Dependent Connections
Video Frame	C1	C2, C6, C9	Motion Mask	C3	C5, C8
Video Frame	C2	no dep.	Motion Mask	C5	no dep.
Video Frame	C6	C1	Motion Mask	C8	no dep.
			Hist. Model	C4	no dep.
			Hist. Model	C7	no dep.

the histogram or the tracker threads. Therefore, the dead-set from the change-detection thread serves as a shorthand for the dead-sets of all three output connections emanating from the video frame channel. Thus the dependent connection information allows faster propagation of dead-set information to neighboring nodes during get and put operations.

7.2.3 Out-of-Band Propagation of Guarantees (OBPG)

The current implementation of DGC propagates forward and backward guarantees by piggybacking them on get and put operations. A limitation of this approach is that a node guarantee cannot be propagated along the graph until either

- a put operation is performed by a thread node, or
- a get operation is performed on a channel node.

In any case, DGC propagates guarantees only to neighboring nodes. The guarantees will further be propagated to the rest of the application pipeline only upon additional get and put operations. More aggressive garbage collection would be possible if these guarantees are instantly made available to all application nodes. It is conceivable to use out-of-band communication among nodes to disseminate these guarantees. However, there will be a consequent increase in runtime system overhead for such communication. To understand this trade-off, the *Out-of-Band Propagation of Guarantees* (*OBPG*) optimization evaluates the following hypothetical question:

How can the memory footprint be reduced by instantly disseminating the node guarantees to all other application nodes?

To explore this question we assume zero cost in terms of network delays and resources to disseminate the information instantly. Naturally, it is not possible to implement either instantaneous propagation of guarantees or out-of-band propagation of guarantees at zero cost; however, such an inquiry allows us to explore the potential performance gains and decide whether embedding a flavor of out-of-band propagation of guarantees within a garbage collection algorithm is worth pursuing.

7.2.4 Out-of-Band Propagation of Dead Sets (OBPDS)

The PDS optimization (see Chapter 7.2.2) assumes that the dead-set information is propagated only upon get and put operations and only to neighboring nodes. Under the *Out-of-Band Propagation of Dead Sets* (*OBPDS*) algorithm we explore the potential for more aggressive garbage collection when local dead-set information is instantaneously disseminated globally. As with out-of-band propagation of guarantees, we investigate the potential for memory footprint reduction when out-of-band communication of the dead-set information is instantly disseminated to all nodes in the graph at zero cost.

7.3 Methodology

The brute-force method of implementing each and every one of the garbage collection algorithms we propose in order to assess their performance is an elaborate and time-consuming process. Furthermore, some of the algorithms are expressed as "what if" scenarios to explore the potential of overcoming a bottleneck or a limitation and cannot be fully implemented. Any practical implementation of such algorithms is only partial, and a large investment may be required to merely get closer, yet perhaps not near an ideal implementation. Therefore, we simulate the proposed garbage collection algorithms and based on the simulation findings, implement only the most promising ones, i.e., those algorithms that show the highest potential for reducing the memory footprint of streaming applications.

To perform the simulations, we use a measurement infrastructure that records pertinent events that occur during an application execution (for more details see Chapter 3). Events

of interest are logged at runtime in a pre-allocated memory buffer that is written to disk upon a successful termination of the application. Some of the interesting events that are logged include memory allocation times, channel get and put times, and memory free (or garbage collection) times. A post-mortem analysis program then uses these logged events to reconstruct the memory usage of the application, as a function of time, and generate metrics of interest such as the application's mean memory footprint, channel occupancy time for items, and latency in processing.

We have already described (see Chapter 6) the use of a subset of the events recorded to reconstruct an execution profile of an Ideal Garbage Collector. To simulate IGC, the post-mortem analysis program considers only the events that are related to relevant timestamps, that is, timestamps that reach the last stage of the application pipeline. Similarly, to simulate the proposed garbage collection algorithms the post-mortem analysis program includes only those events that would have occurred under the corresponding GC algorithm being simulated. In addition, and whenever needed, the post-mortem analysis program alters the recorded GC event times for certain items. The new time assigned to each event reflects the GC time that would have been recorded had the corresponding GC algorithm been implemented.

The simulation results for the different GC algorithms are received, analyzed, and compared to IGC. These results enable us to set apart those algorithms that display the most promising potential based on the simulation. These algorithms are the first candidates for implementation.

Once the selected algorithms are implemented and run, it is possible to compare their actual performance with the simulation predictions. This comparison helps in understanding the limitations of simulations in general, and the limitations of this simulation in particular, in predicting the performance of suggested GC algorithms.

7.4 Simulation Setup and Metrics

We use the *memory footprint* metric to evaluate the proposed optimization strategies. Memory footprint is the amount of memory the application uses for buffering timestamped items

in channels as a function of real time, and is indicative of the instantaneous memory pressure the application exerts. To associate real numbers with the memory footprint, we also present the *mean memory usage* of the application.

We use the color tracker application (presented in Chapter 6.3.1) for assessing the various garbage collection optimizations.

All experiments are carried out using the experimental settings described in Chapter 6.1. The operating system used is Redhat Linux with a 2.6.9 kernel. We conduct our experiments using two configurations. In the first configuration all threads and channels shown in Figure 18 execute on one node within a single address space. In the second configuration, all five threads and their corresponding output channels are distributed over five nodes of the cluster. This setup is discussed in more detail in Chapter 6.4.1.

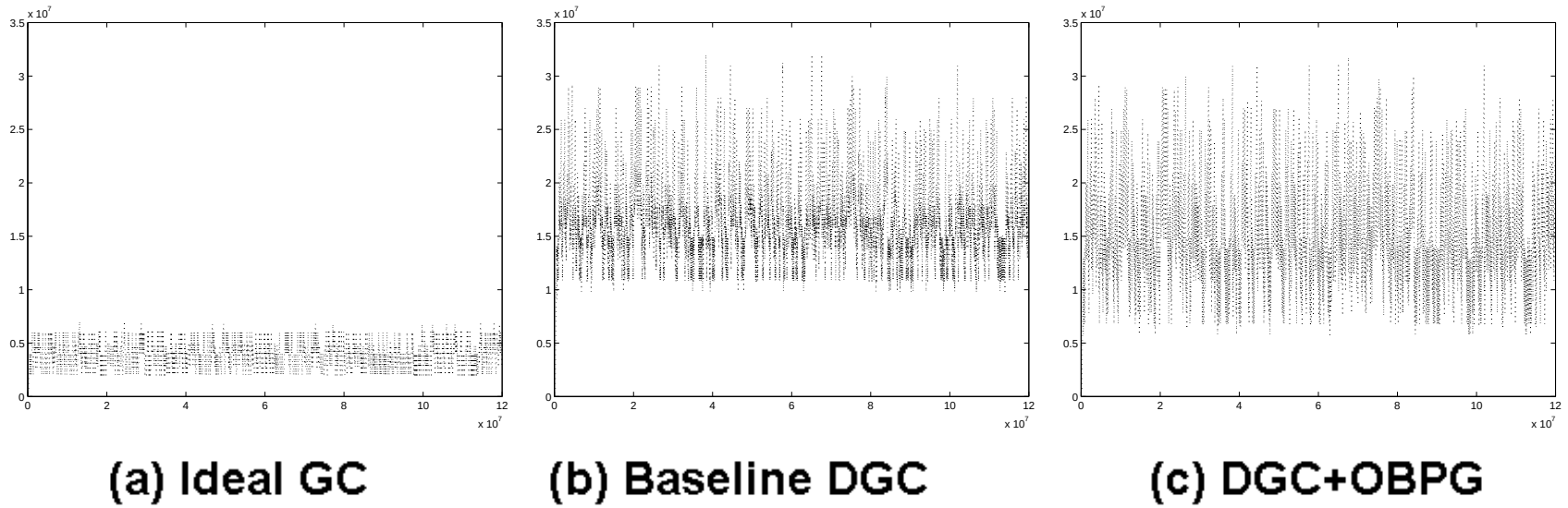

Figure 19: Memory Footprint of DGC, Simulated IGC, and OBPG: The simulation results are based on the color tracker application running on a single address space (and on a single node). The memory footprint graphs above show results for (from left to right): (a) Ideal GC - Lower Bound (IGC), (b) the baseline DGC implementation, and (c) DGC optimized with Out-of-Band Propagation of Guarantees (OBPG) All three graphs share the same scale, with the y-axis showing memory use (bytes x 10^7), and the x-axis representing time (milliseconds).

7.5 Simulation Results

In this section we present the simulation results of the proposed algorithms described above (see Chapter 7.2). The simulation results indicate the performance potential of each of the proposed algorithms and provide guidance to select those algorithms that are most promising for implementation. As we explained in Chapter 7.2, KLnU differs from the rest of the three optimizations presented in that it does not necessarily require DGC as the underlying garbage collector. However, we use the DGC algorithm as the common underlying garbage collector for evaluating all four optimization schemes to facilitate the comparison among the optimizations.

7.5.1 Simulated Performance of Out-of-Band Propagation of Guarantees (OBPG)

Figure 19 (c) shows the results of the OBPG optimization. As we mentioned in Chapter 7.2.3, zero time and zero cost are assumed for the out-of-band dissemination of the timestamp guarantees for this optimization. The average memory usage of OBPG shows a relatively small reduction of 4% for the 5-node configuration, and 16% for 1-node configuration (see Tables 4 and 5) compared with the DGC baseline. Qualitatively, Figure 19 (b) shows that the peak memory usage of OBPG is lower than the one DGC exhibits; however, the difference is not significant. There are two possible explanations for these observations. The first, relates to a common property of streaming applications; later stages of the application pipeline tend to perform more computations than earlier stages. Therefore, backward guarantees are more useful in hastening garbage identification than forward guarantees. The second possible explanation takes into account a characteristic that is prevalent in streaming applications, namely the utilization of highly connected task graphs. As a result later threads require direct input from earlier channels. In the case of the color tracker application, for example, the target detection thread requires input from the video frame channel, the first channel in the application task graph (see Figure 10). As a consequence, garbage identification information from later stages of the pipeline is fed back via a direct connection to earlier stages regardless of the OBPG optimization, and thus reduces

its potential benefits.

Recall that the OBPG simulation assumes ideal conditions, which cannot be implemented in practice. In that sense, the simulation results present an upper limit to the benefits OBPG can provide. These benefits cannot be realized under real conditions. Most importantly, even with these ideal assumptions, the application average memory consumption with an OBPG collector is still 2.5 to 4 times higher than with an Ideal Garbage Collector and very close to the baseline DGC, therefore this optimization may not be a good candidate for implementation.

7.5.2 Simulated Performance of Keep Latest 'n Unseen (KLnU)

For this optimization, we associate the KLnU attribute ($n = 1$, i.e., a channel buffers only the most recent unseen item) with all the channels in the color tracker application pipeline. Figure 19 and Figure 20 (a) show the effect of the KLnU optimization over the DGC baseline. Compared to the baseline implementation of DGC the KLnU optimization achieves only a modest improvement in average memory usage (2% - 17% reduction). This is surprising since we expect this optimization to enable each channel to be more aggressive in eliminating garbage locally. Recall that even though the buffering is limited to just the most recent item, a channel still cannot garbage collect earlier items that have been gotten by at least one output connection to preserve timestamp causality. This surprisingly small reduction in memory usage is a consequence of the application accessing most of the data items before new timestamp items are produced. As a result, the KLnU optimization cannot identify these data items as garbage, and cannot reclaim them.

7.5.3 Simulated Performance of Propagating Dead Sets (PDS)

Figure 20 (b) shows the results of the PDS optimization. Compared to the baseline DGC, the reduction in memory usage is significant (38% - 60%). Moreover, PDS simulation results exhibit an increase of 54% - 83% in average memory usage over the Ideal Garbage Collector. Although these results may suggest that there is still a substantial garbage identification potential prevalent in the system, PDS is, in fact, relatively close to the best achievable garbage collection goal (please recall that the Ideal Garbage Collector cannot be

implemented, and serves as a lower bound for the memory footprint of the application; see Chapter 7.6 for a more complete discussion).

The major improvement observed is quite surprising at first glance and can be attributed to the combination of two effects: first, nodes propagate the dead set information forwards and backwards using the application task graph; second, a channel aggressively incorporates the incoming dead set information using the dependency information on its output connections. Analysis of the application runtime logs reveals that most of the improvement is associated with the latter effect. For example, the video frame channel can use the dead set information that it receives from the change-detection thread immediately without waiting for similar notifications from both the histogram and target-detection threads (see Figure 18).

7.5.4 Simulated Performance of Out-of-Band Propagation of Dead Sets (OBPDS)

This optimization is similar to OBPG, with the difference that dead set information is disseminated out-of-band in addition to the guarantees. As in OBPG, we assume that this dissemination requires zero cost and zero time to assess the limit of the performance gains that can be expected from OBPDS. Table 4 shows a small reduction (less than 7%) for the 1-node configuration in memory usage compared to PDS without out-of-bound propagation of both guarantees and dead sets. As we observed with the OBPG optimization, the relatively small impact of this optimization is due to the high level of connectivity of this particular application task graph. On the other hand, Table 5 shows a more significant reduction of 20% in average memory usage for the 5-node configuration compared to PDS without out-of-bound propagation of both guarantees and dead sets. However, these reductions are based on ideal assumptions that cannot be realized in practice, and may serve as an upper limit to the expected performance. Overall, the combined effect of the proposed optimizations is a reduction in memory usage to 37% on the 1-node configuration and 49% on the 5-node configuration compared to the baseline DGC.

Table 4: Simulation Results for 1-Node Configuration: Simulated performance of different GC optimizations for the color tracker application. Percentage Mean Memory usage of optimizations with respect to that of the baseline DGC and IGC are also presented in the table.

Setup	Config 1: 1 node			
	Mean Memory Usage (B)	Memory Usage STD	% w.r.t. DGC	% w.r.t. IGC
DGC baseline	16,362,587	5,534,513	100	488
DGC + OBPG	13,679,554	5,488,904	84	408
DGC + KLnU	13,651,817	4,225,333	83	407
DGC + PDS	6,577,808	1,253,143	40	196
DGC + OBPG + OBPDS	6,134,006	1,579,959	37	183
IGC (Lower Bound)	3,353,929	864,361	20	100

Table 5: Simulation Results for 5-Node Configuration: Simulated performance of different GC optimizations for the color tracker application. Percentage Mean Memory usage of optimizations with respect to that of the baseline DGC and IGC are also presented in the table.

Setup	Config 2: 5 nodes			
	Mean Memory Usage (B)	Memory Usage STD	% w.r.t. DGC	% w.r.t. IGC
DGC baseline	25,033,964	3,729,229	100	247
DGC + OBPG	24,026,015	3,487,294	96	237
DGC + KLnU	24,463,924	3,723,629	98	242
DGC + PDS	15,609,572	3,268,030	62	154
DGC + OBPG + OBPDS	12,387,970	3,670,903	49	122
IGC (Lower Bound)	10,123,131	3,286,813	40	100

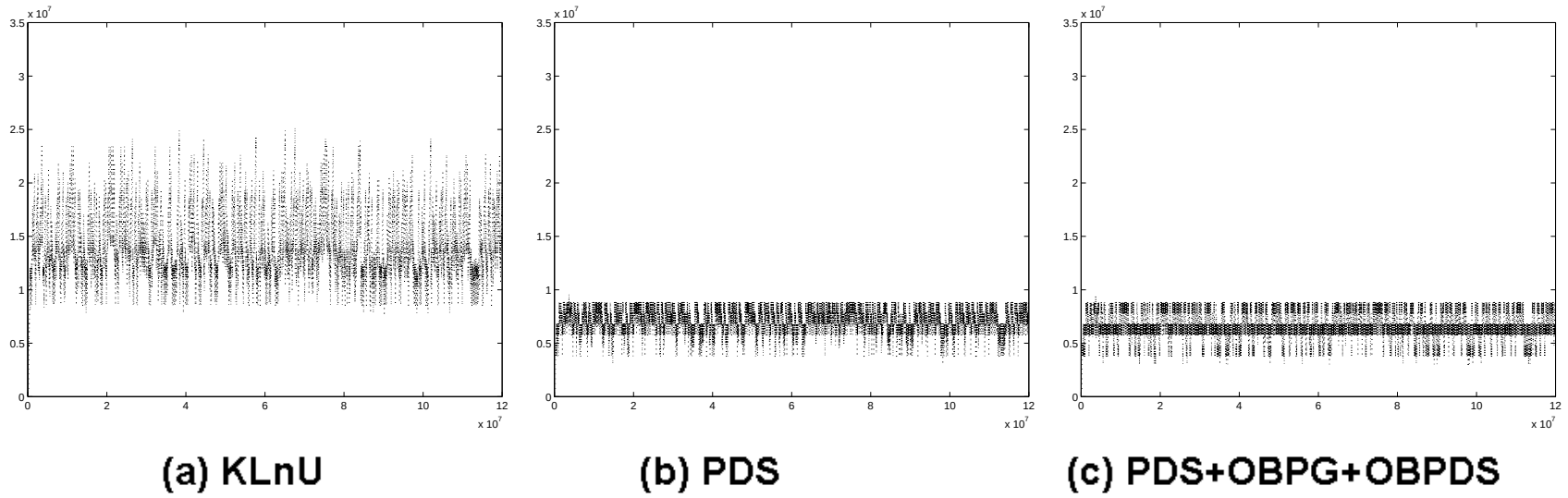

Figure 20: Memory Footprint of GC Optimizations Simulations: The simulation results are based on the color tracker application running on a single address space (and on a single node). The memory footprint graphs above show results for (from left to right): (a) DGC with KLnU, (b) DGC with PDS, and (c) DGC with OBPG and OBPDS. All three graphs share the same scale, with the y-axis showing memory use (bytes x 10^7), and the x-axis representing time (milliseconds).

7.6 Discussion of Simulation Results

A number of optimizations are evaluated in terms of their potential to improve the performance of garbage collection in streaming applications. Of the ones considered, it is possible to fully implement KLnU and PDS while it is not possible to fully implement OBPG and OBPDS. The latter two were presented mainly to better understand the limit of the expected performance gains and also to serve as a guide to more realistic strategies. The KLnU optimization is dependent upon a requirement from the channel to provide only latest items, and upon the application writer providing this information to the runtime system. The performance benefits of the PDS optimization are sensitive to several factors that are application-specific: the number of attributed channels, the value of n for the KLnU attribute, the connectedness of the task graph, and the dependencies among the input and output connections to nodes in the application.

Our performance study using the color tracker application revealed some counter intuitive results. First, disseminating the timestamp guarantee of DGC or of the dead-set information of PDS to all nodes did not result in substantial savings. In hindsight, this seems reasonable given the connectedness of the color tracker task graph. Second, the KLnU optimization in itself is not sufficient to obtain a substantial reduction in memory usage. The propagation of dead-set information and the use of the dependency information on the output connections of channels is the key to achieving most of the performance benefits.

While the optimizations succeed in substantially reducing the memory footprint of the application, even the best performing optimizations still use significantly more memory than the Ideal Garbage Collector. For example, with the PDS optimization, the application uses 96% and 54% more memory on average than the Ideal Garbage Collector for the 1-node and the 5-node configurations, respectively. While one cannot rule out the existence of more aggressive algorithms than PDS, it may be relatively close to the best achievable reduction of the memory footprint using garbage identification techniques. First, the Ideal Garbage Collector is assumed to be an oracle. It knows *a priori* which items are going to be identified as garbage, and does not include the memory associated with these items in the application

memory footprint. In practice, garbage identification algorithms can not hope to achieve this knowledge. They can classify items as garbage only after these items are produced and put into a channel. Moreover, in the most successful garbage identification schemes nodes at earlier stages of the application pipeline incorporate information received from nodes at later stages of the application pipeline. In these cases, a node at an early stage of the application pipeline defers the identification of a specific item as garbage until this item is processed by later stages of the application pipeline, and until the information related to this item propagates to that node. During this time, the item continues to reside in the channel and consume memory. The effect this deferred decision has on the application memory footprint is exacerbated by the fact that items at earlier stages of the pipeline tend to require more memory than items at later stages of the pipeline.

Additionally, there is a risk in generalizing the expected performance benefits of these optimizations simply based on the results of one application. Nevertheless, it appears that knowledge of the dependency information on the output connections of channels is a crucial determinant to the performance potential of these optimizations. A question worth investigating is the performance potential for incorporating output connection dependency in the original DGC algorithm. Another question worth investigating is the extent to which the REF and TGC algorithms will benefit from attributed channels.

7.7 Implementation

The simulation results indicate that algorithms involving immediate propagation of information throughout the application have a relatively small influence on the memory footprint even under ideal and unrealistic conditions. The PDS algorithm, on the other hand, has the biggest potential of reducing the memory footprint of an application, and as such, it is the first candidate chosen to be implemented. KLnU is the second candidate for implementation because dead sets generated by the KLnU algorithm are the ones that the PDS algorithm propagates. In addition, and despite the small reduction expected in mean memory usage according to the simulation, implementing KLnU can help in evaluating the validity of the usage of Trace Driven Simulation techniques in the context of assessing

memory consumption in distributed stream-based applications.

Both KLnU and PDS do not replace the garbage collector, but instead optimize the performance of the runtime system's garbage collector. KLnU is capable of identifying some data items as garbage earlier than the existing garbage collector does. PDS is able to propagate information about these data items to other nodes in the application faster than any baseline garbage collector in the runtime system.

The comparison results of the three garbage collector algorithms: REF, TGC, and DGC (see Chapter 6) indicate that DGC exhibits the lowest memory footprint. As such, it is the best candidate to serve as the baseline garbage collector, on top of which KLnU and PDS are implemented.

7.7.1 Keep Latest 'n Unseen (KLnU)

Recall that KLnU optimization associates an attribute with a channel that allows it to discard all but the latest items. Put operations to a channel have the potential of activating the KLnU optimization, and discard those items that were previously transmitted to the channel, but have not yet been gotten by any one of the relevant threads. Therefore, KLnU can be implemented by adding a function within the Stampede *put* operation that checks whether the channel holds items that were not gotten by any connection. These items are tagged as garbage and discarded.

The Keep Latest n if Unseen (KLnU) optimization keeps only the last n unseen items in a channel, where n is a parameter that the application writer may set as required by the application. This parameter provides the application with an n-sized window "into the past". In the case of the color tracker application, however, the application is interested only in the latest item, and thus n can be set to 1.

7.7.2 Propagation of Dead Sets (PDS)

PDS is implemented by maintaining a list of dead items. Every time a new item is identified as garbage, it is added to the list of dead items. This list is then propagated upon Stampede *put* operations to channels earlier in the application pipeline. Figure 21 depicts such communication. The dead set from channel B is transmitted to channel A. As can be seen

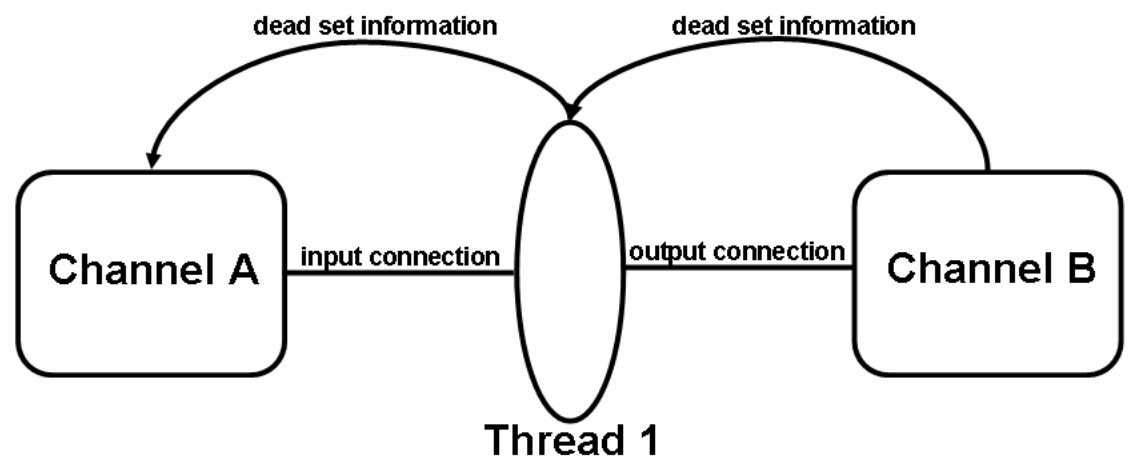

Figure 21: Flow of Dead Set Information Under the PDS Optimization: The dead set information flows from Channel B to Channel A via Thread 1.

from the figure, there is no direct connection between the two channels. Thus, the runtime system transfers the dead set information with the mediation of a thread. In this case, the runtime system transmits the dead set of channel B to thread 1 via their common output connection. Once this information reaches thread 1, the runtime system can transmit it back to channel A via the input connection common to thread 1 and channel A. Please note that the connections are described from the point of view of the thread. Thus, an output connection is one on which data items are outwardly transmitted from a thread to a channel. Similarly, an input connection is one on which data items are inwardly transmitted from a channel to a thread.

As items are added to the dead set, PDS maintains the set to prevent it from growing indefinitely. PDS uses the low water mark of the underlying garbage collector to trim items that are already considered as garbage by the garbage collector. In our case, where DGC serves as the underlying garbage collector, items with timestamps that are below the node guarantee (see Chapter 5.6) can be removed from the dead set, as the garbage collector has already labeled them as garbage to be reclaimed.

7.8 Setup and Metrics

The study is performed using the same setup described in Chapter 7.4.

The baseline garbage collector on top of which the KLnU and PDS optimizations are implemented is the Dead timestamps based Garbage Collector (DGC) presented in Chapters 5 and 6.

We use the following metrics to evaluate the KLnU and the PDS optimizations:

Average Memory Usage - This metric quantifies the aggregate memory used by the application in all the channels averaged over the execution time of the application (see Chapter 6.2 for more details).

Total Channel Occupancy Time - This metric quantifies the total time spent by all items in channels before they are garbage collected. In addition, we distinguish between *used occupancy time*, time spent on "successful" items, i.e., those items that make it through the application pipeline and affect the output, and *wasted occupancy time*, time spent on items that do not reach the end of the application pipeline and therefore have no influence on the end result.

Average Pipeline Latency - This metric quantifies the average latency experienced by items that make their way through the entire application pipeline and is described in more detail in Chapter 6.2.

Average Throughput - This metric measures the average number of items that reach the end of the pipeline, and thus affects the final outcome. The larger the throughput the more items the application can process (on average) in a given time.

7.9 Results

The implementation results presented in Table 6 show that both KLnU and PDS optimizations reduce the amount of memory consumed by the application. When KLnU is implemented on top of DGC there is a 4% and a 14% reduction in average memory usage for the color tracker application in the 5-node and the 1-node configurations, respectively. When PDS is implemented on top of DGC there is a 31% and a 60% reduction in average memory usage for the color tracker application in the 5-node and the 1-node configurations, respectively.

The KLnU and PDS optimizations not only reduce the average memory usage, but also

Table 6: GC Optimizations Performance: DGC, KLnU, and PDS performance in terms of average memory usage, latency, and throughput. The results confirm the simulation predictions: Both optimizations reduce the average memory usage over the baseline garbage collector (DGC), and PDS outperforms KLnU. A positive side effect that can be observed only when the optimizations are implemented is the reduction in latency and the improvement in throughput, as the run time system directs more resources towards items that are successfully processed by the application pipeline.

Setup	Config 1: 1 node			Config 2: 5 nodes		
	Memory Usage (MB)	Latency (ms)	Throughput (fps)	Memory Usage (MB)	Latency (ms)	Throughput (fps)
DGC	16.36	389	4.52	25.03	628	4.12
DGC+KLnU	14.06	374	4.74	24.06	580	4.27
DGC+PDS	6.50	362	4.77	17.29	634	4.23

improve the application performance in terms of latency and throughput. As the run-time system reduces its overhead (maintaining items that will not reach the end of the application pipeline) and directs its resources towards productive work (that is, towards items that will successfully reach the end of the application pipeline), data items are able to reach the end of the pipeline faster and more frequently. This performance improvement offsets the additional overhead that KLnU and PDS introduce. Under KLnU, the garbage identification decision is made by the channel and is based on information readily available to the channel. Thus, the overhead associated with KLnU is relatively small. PDS, on the other hand, has higher overhead because it propagates additional information to neighboring nodes. This overhead increases as communication costs increase. As a result, the PDS overhead is even greater in the 5-node configuration. In summary, the overall performance of the application is determined by a balance between the benefits of the reduced overhead by maintaining a smaller number of items, and the increased overhead of the implemented KLnU and PDS optimizations.

Throughput under both optimizations increases rather than decreases. Table 6 shows that KLnU increases the application throughput from 4.52[fps] to 4.74[fps] (or by 5%) for the 1-node configuration, and from 4.12[fps] to 4.27[fps] (or by 4%) for the 5-node configurations. PDS increases the application throughput from 4.52[fps] to 4.77[fps] (or by

6%) for the 1-node configuration, and from 4.12[fps] to 4.23[fps] (or by 3%) for the 5-node configuration. Similarly, latency does not increase but rather decreases. Table 6 shows that under KLnU the latency decreases from 389[ms] to 374[ms] (or by 4%) for the 1-node configuration, and from 628[ms] to 580[ms] (or by 8%) for the 5-node configuration. Under PDS the latency decreases from 389[ms] to 362[ms] (or by 7%) for the 1-node configuration; however, the latency increases slightly from 628[ms] to 634[ms] (or by less than 1%) for the 5-node configuration. This is the result of the relatively large overhead associated with PDS under the 5-node configuration.

Table 7: GC Optimizations Performance: Channel occupancy time for the 5-node configuration. Both KLnU and PDS optimizations achieve a more efficient garbage identification process, and as a result show a reduction in memory resources directed towards unsuccessful items compared to the baseline DGC. A positive side effect is the reduction in resources directed towards successful items, which is the result of the less time items spend waiting to be processed.

Setup	Wasted Occupancy Time (sec)	Used Occupancy Time (sec)	% Used	Compared to DGC	
				% Wasted	% Used
DGC	2,913	3,637	55.53%	100.00%	100.00%
DGC+KLnU	2,685	3,296	55.11%	92.16%	90.62%
DGC+PDS	875	3,633	80.59%	30.04%	99.89%

Tables 7 and 8 present the total occupancy time (that is, the total time items spent in the channels) for the 5-node and the 1-node configurations, respectively. They break the total occupancy time into wasted and used categories. Wasted occupancy time is the aggregate time items that do not reach the end of the pipeline spend in channels. This occupancy time does not contribute to the application end result, and thus, can be viewed as wasted. Used occupancy time, on the other hand, is the aggregate time successful items spend in channels. These items successfully reach the end of the pipeline, and affect the application output. Therefore, it is constructive to direct resources towards managing these data items.

In both configurations the two optimizations exhibit a reduction in the occupancy time, a result that confirms the reduction in average memory usage observed in Table 6. PDS

Table 8: GC Optimizations Performance: Channel occupancy time for 1-node configuration. As in the 5-node configuration, both optimizations show a reduction in memory directed towards unsuccessful items compared to the baseline DGC due to the more efficient garbage identification, and also a reduction in resources directed towards successful items. The improvement is more dramatic than the one observed in the 5-node configuration, where part of the wasted occupancy time is masked by network delays.

Setup	Wasted Occupancy Time (sec)	Used Occupancy Time (sec)	% Used	Compared to DGC	
				% Wasted	% Used
DGC	3,784	991	20.75%	100.00%	100.00%
DGC+KLnU	2,386	995	29.43%	63.08%	100.46%
DGC+PDS	604	949	61.11%	25.30%	95.35%

uses only 69% and 33% of the occupancy time consumed by DGC in the 5-node and the 1-node configurations, respectively. In addition, most of this reduction is related to wasted occupancy time (that is, occupancy time related to items that do not reach the end of the pipeline), which falls to only 30% and 25% of its value compared to DGC for the 5-node and the 1-node configurations, respectively. This confirms the success of the PDS optimization in transmitting the information about dead items much faster than the baseline DGC can. The 1-node configuration is more efficient than the 5-node configuration at reclaiming these items because PDS optimization relies on the transmission of dead set information to other nodes in the application. The 1-node configuration does not have to overcome network delays, thus the dead set information reaches the relevant nodes faster, and items no longer needed by the application are reclaimed faster. The relatively small reduction (up to 10%) in the used occupancy time can be attributed to the improvements in throughput and latencies observed in Table 6.

The KLnU optimization displays a similar trend, however it is less effective compared to PDS, as local information about dead items is not made known to other nodes in the application. KLnU shows only 9% and 29% reduction in total occupancy time over the baseline DGC for 5-node and 1-node configurations, respectively. The 1-node configuration shows a substantial increase in the percentage of occupancy time devoted to successful timestamped items (29.4%) over the baseline DGC (20.7%). The breakdown of occupancy

time into used and wasted shows practically no change in the occupancy time directed at successful timestamped items, and a substantial reduction (37%) in the occupancy time devoted to unsuccessful timestamped items. On the other hand, the 5-node configuration shows a fairly even reduction of 8%-9% in the total occupancy time for both successful and unsuccessful timestamped items. These results are also reflected in the percentage of occupancy time devoted to used items, which is almost identical to the one observed in the baseline DGC (around 55%). These results demonstrate how a reduction in resources directed at unsuccessful items may result in a reduction in the resources directed at successful items. This point is further demonstrated in Chapter 8 during the discussion regarding the Adaptive Resource Utilization algorithm performance.

7.10 Comparison between Simulation and Implementation

The baseline trace, on which the simulation is based, includes events that are associated with allocation or deallocation of specific items. A simulation is allowed to alter only the deallocation time of an item, thus affecting only the time an item is being garbage collected and removed from the memory of the application. It is not allowed to change other attributes related to the execution of an item, such as whether a specific item is being created, its creation time, its processing time, and the nodes that are involved processing the item. Thus, the only changes the simulation methodology permits are performed once the application has already furnished the item with all the computation and network resources it requires, and the item is waiting to be garbage collected. The simulation can then decide whether, based on the specific garbage collection optimization simulated, the item would have been garbage collected earlier. In this case, the simulation alters the deallocation time of the item to correspond to the logic of the garbage collection optimization simulated. The simulation does not take into account two consequences of the implementation:

1. The additional overhead associated with the implementation of the optimization.

2. A possible reduction in the overhead of the runtime system due to maintaining a smaller number of items concurrently.

These two consequences have a small influence on the overall application performance; moreover, they affect the performance of the application in opposite directions, so that the net effect on the application performance is small. Therefore, the simulation results are expected to be close to the implementation results.

Tables 9 and 10 present a comparison between the simulation predictions and the actual implementation results in terms of memory footprint reduction for the 1-node and 5-node configurations, respectively.

Table 9: Comparison Between Simulation and Actual Implementation for 1-node Configuration: Simulation results are almost identical to the actual implementation results.

Setup	Config 1: 1 node		
	Simulation Memory Usage (MB)	Actual Memory Usage (MB)	Difference
DGC	16.36	16.36	—
DGC + KLnU	13.65	14.06	+3.01%
DGC + PDS	6.58	6.50	-1.17%

Table 10: Comparison Between Simulation and Actual Implementation for 5-node Configuration: Simulation results are within a 10% range from the actual implementation results.

Setup	Config 2: 5 node		
	Simulation Memory Usage (MB)	Actual Memory Usage (MB)	Difference
DGC	25.03	25.03	—
DGC + KLnU	24.46	24.06	-6.64%
DGC + PDS	15.61	17.29	+10.77%

The average memory usage of the implementation for the 1-node configuration is practically identical to the simulation predictions. The implemented KLnU algorithm consumes 3% more memory than the simulation predicts, while the implemented PDS algorithm consumes 1% *less* memory than the simulation predicts. These results are better than expected,

because the simulation does not take into account overheads associated with the actual implementation. We can attribute this improvement to the reduction in overheads associated with maintaining a larger number of items concurrently and the diversion of resources towards processing successful data items. As a consequence, items that reach the end of the pipeline spend less time in channels, and consume less memory. Indeed, the actual performance of the KLnU algorithm shows a 4% reduction in latency, while the PDS algorithm shows a 7% reduction in latency (see Table 6). As mentioned earlier, the simulation does not take into account the second-order effects that are the result of the increased overhead associated with the implemented optimization, and the reduced overhead associated with the smaller number of items that the runtime system maintains concurrently. The net effect of these second-order improvements is a larger than expected reduction in average memory usage.

Differences between the simulation and the actual implementation are larger for the 5-node configuration (see Table 10). The implemented KLnU shows a decrease of 7% over the simulated KLnU, while the implemented PDS shows an increase of 11% over its simulated version. Recall that the KLnU algorithm is based on local decisions, while PDS performance is also network dependent. Upon a put operation, a KLnU attributed channel has enough information to decide whether to identify older items as garbage. An older item can be reclaimed if it is considered to be unseen for all channel connections because the attributed channel guarantees no thread will ask for it in the future. On the other hand, PDS is based on transmitting dead set information to all application nodes. Thus, network latencies come into play and are responsible for the slight increase in average memory usage observed in the PDS implementation. The simulation does not take into account network delays associated with propagating this information, thus, the memory footprint predictions of the simulation are lower than the memory footprint of an actual implementation.

7.11 Conclusions

In this chapter, we have presented four garbage collection algorithms that optimize memory usage in streaming applications. A common feature of these algorithms is the utilization of

dependency information that is either available or provided to the runtime system by the application writer.

KLnU is based on attaching an attribute to a channel that helps the runtime system assess whether older timestamped items can be reclaimed. This attribute is provided to the runtime system by the application writer, who is required to understand and then code the data dependencies prevalent within the channel.

PDS is based on sharing dead set information with other nodes in the application.

OBPG and OBPDS are based on instantaneous transmission of garbage identification information to all application nodes.

We have also presented a methodology to predict the performance of garbage collection algorithms prior to their implementation. This methodology of applying Trace Driven Simulation techniques to assist in exploring the design space of memory optimizations in the domain of streaming applications allows a rapid assessment of various design options without going through the labor of actually implementing them. The worthiness of implementing a specific algorithm is made based on the simulation results.

The study demonstrates the effectiveness of this approach. Although Trace Driven Simulation techniques cannot adequately model second-order effects, their accuracy level is impressive. The differences observed between simulation and actual implementation are of the order of 10%. More importantly, the use of an Ideal Garbage Collector is significant in understanding how successful an algorithm is in harnessing the garbage collection potential prevalent in the system. The simulation results demonstrate the success of the PDS algorithm in reducing the average memory usage of the application and in closely reaching the performance of an ideal garbage collector.

However, as successful as this methodology is in predicting the performance of the four garbage collection algorithms presented here, this study also demonstrates the limitations of this approach. The performance advantage of each one of the four algorithms is a result of faster and more efficient identification of garbage items. Thus, it is possible to simulate the performance of these algorithms by removing events or by altering the time specific events occur to reflect the reality of each one of the simulated algorithms. In addition, the

second-order effects of these algorithms are relatively minor, and mainly involve the effects of a slight reduction in latency that is a result of directing the limited resources the runtime system has towards successful data items.

The ARU algorithm, presented in the next chapter, takes a different approach to reduce the memory usage of streaming applications: it analyzes the system's capacity to process data items, and allows the introduction of new inputs to the system only when the system has the capacity to fully process these new items. If the ARU algorithm is successful in achieving its goal, the expected performance of the algorithm will be similar to the Ideal Garbage Collector, because ARU allows the introduction of new data items into the system only when there are enough resources available to process them. Therefore, simulation cannot help in understanding the expected effectiveness of the ARU algorithm. The actual success of the ARU algorithm in reducing the average memory usage is dependent mainly on the implementation details.

CHAPTER VIII

ADAPTIVE RESOURCE UTILIZATION (ARU)

8.1 Motivation

Chapters 4-7 addressed the problem of reducing memory usage of distributed streaming applications by reclaiming produced data as soon as possible. The myriad of garbage collection algorithms proposed differ in the phase they recognize a data item as garbage and reclaim it. In this chapter, we will propose a different approach that strives to maximize the overall *Resource Utilization* of an application. The term "Resource Utilization" means the efficient use of resources given to an application. Unlike Resource Management schemes, such as Quality of Service (QoS) and scheduling that target *allocation* of resources to an application, Resource Utilization focuses on 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 that manage these resources among the threads would be characterized as their resource utilization policy.

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, including distributed streaming applications, are affected by dynamic phenomena, and these tools are incapable of analyzing these effects. One example of a dynamic phenomenon would be when the application processing requirements are sensitive to changes in input data. Another example would be when the application is executed in an environment where the system load changes abruptly.

Four of the characteristics of streaming applications make Resource Utilization an appealing alternative to consider:

1. Typically, streaming applications are not required to process all input data to achieve their goal. These applications commonly extract meaning out of many streams of data, an outcome that can usually be realized by processing only a fraction of the

input data.

2. Streaming applications tend to function in a resource-hungry environment, as they require substantially more resources than available.

3. Streaming applications are required to produce data items that pace in correlation with real-time.

4. Threads that are part of the data processing and production pipeline are characterized by producer/consumer relationships. Tasks in successive pipeline stages do not have the same rate of consumption/production.

The need to produce data items that pace in correlation with real-time, coupled with resource-scarcity, prevents the application from fully processing all the data items. The rate differences between the various producer/consumer pairs in the pipeline forces data items that were already processed in earlier stages to be dropped, and not be processed further. Indeed, producers tend to drop or skip-over stale data to access the most recent data from their input buffers. The result is that limited resources are wasted on processing data that will not reach the end of the application pipeline and affect the application outcome.

Efficient implementation of an application will attempt to minimize resource utilization wasted in maintaining such data. The garbage collection (GC) algorithms presented in the previous chapters are based on timestamp visibility that frees data elements as soon as the runtime system can be certain that they are not going to be used by the application. These timestamp-based GC algorithms differ from the conventional GC algorithms in their logic governing garbage collection. Traditional GC algorithms deploy a conservative mechanism to identify garbage items. They only regard a data element as garbage if it is not reachable by any one of the threads that compose the computation. The timestamp-based GC algorithms, on the other hand, are able to assert that specific data items that will not be used in the future, and reclaim these items as well as the items are unreachable.

Such GC algorithms succeed in helping alleviate the memory usage problem by freeing unwanted data after it has already been created; however it would be best if wasted items were never produced in the first place, saving both processing, networking, and memory

resources. Static determination of unneeded items cannot be made due to the inherent dynamic nature of such applications.

Dynamically controlling and matching the data production of each stage in the pipeline with the overall application rate provides another approach to direct the scarce resources available towards processing pertinent data.

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

8.2 ARU via Feedback Control

8.2.1 Factors Determining Execution Rate

Pipelined streaming applications (for example, the one presented in Figure 22) share similarities with systolic architectures. It is therefore useful to talk about a rate of execution for the entire pipeline. This is the rate a processed output is emitted from the end of the pipeline. 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 it requires. Since computation is data-dependent (e.g., looking for a specific object in a video frame), the execution time of a task for each iteration may vary. Additionally, the actual task execution time is subject to external circumstances such as the vagaries of OS scheduling and computational load on the system. Unfortunately, these parameters are fully known only at run time.

8.2.2 Eliminating Wasted Resources

Skipping over unwanted data may allow an application to keep up with its interactive requirements, but it does not eliminate resources already allocated to produce and maintain such data. The term *wasted computation* denotes thread executions that produce data items

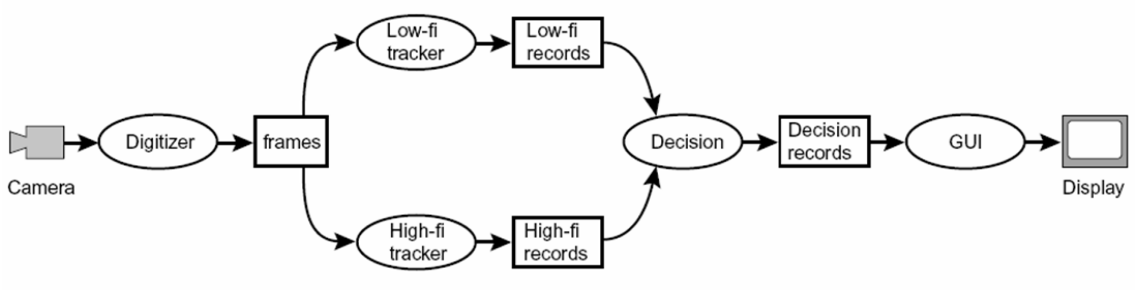

Figure 22: A Vision Application Pipeline: This application locates a specific object in a scene. Ovals are tasks or threads, and rectangles are "streams" or "channels".

that subsequent (or downstream) threads will not use. Unfortunately, *a priori* knowledge of parameters that determine the processing rates of the threads is required to eliminate wasted computations.

Even though this future knowledge cannot be determined absolutely, systems where data items are associated with time can utilize this information, combined with knowledge about task-graph topologies as well as data dependencies, to infer whether downstream threads would require a specific data item. This deduction can then be used to eliminate irrelevant resource usage. Stampede, for example, associates a notion of virtual time with each thread in an application pipeline. Furthermore, data produced by each thread is tagged with a virtual timestamp. The GC algorithms, proposed in chapters 4- 7, for eliminating upstream computations (i.e., computations performed at earlier stages of the task-graph) use the virtual times associated with data item requests made by downstream threads. However, as we saw, such techniques have limited success. This phenomenon is observed in many interactive application pipelines because upstream threads (such as the digitizer in Figure 22) tend to be quicker than downstream threads (such as the decision threads). As a result, it generally becomes too late to eliminate upstream computations based solely on local virtual time knowledge. There is, however, another piece of information that is embedded in the task-graph that can help the runtime system to predict wasted computations. If processing rates of downstream stages were made available to the runtime system, it would become possible to control the rate of production of time stamped items in earlier stages. In this

manner, data items that are not going to be used by downstream threads would not be produced in the first place. This would retroactively eliminate unwanted computation *before* data production.

8.3 Distributed ARU

The ARU algorithm [30] is predicated on the following two assumptions:

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

The first assumption is common to streaming applications, and the second imposes only a small burden on application writers. Furthermore, information about graph topology can be deduced from information that is readily available to the runtime system. Indeed, alleviating the application from providing the graph topology and incorporating this information into the ARU algorithm is one of the possible extensions to this algorithm.

The ARU algorithm does not require additional application information; however, as will be discussed later, limited information regarding data dependencies may help to further reduce wasted resources.

We will now describe a distributed algorithm whereby tasks constantly exchange local information to change their rate at which data items are produced.

8.3.1 Sustainable Thread Period (STP)

We define sustainable thread period (STP) as the time it takes to execute a single iteration of a thread loop. A thread dynamically computes STP locally with a clock reading taken at the end of every one of its loop iterations (see Figure 23). 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 a preceding 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 the present load conditions. This current-STP is used

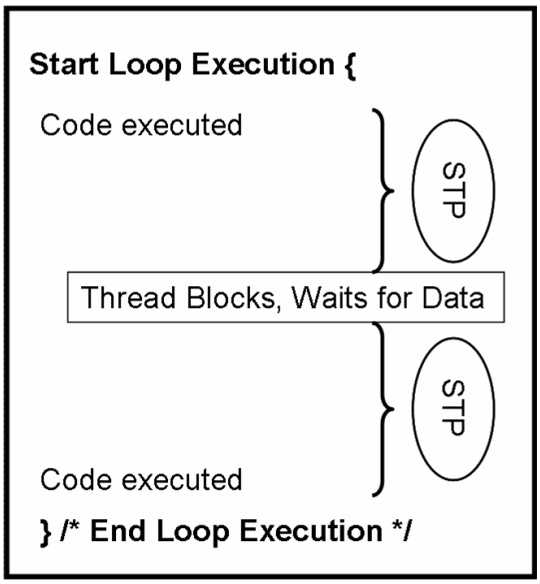

Figure 23: Measuring the Sustainable Thread Period (STP): Note that the STP does not include block times, when the thread waits for input from other threads. Thus, STP captures the minimum time required to produce an item given the present load conditions.

as feedback to compute the *summary-STP* described below that in turn is propagated as a feedback to preceding tasks in the pipeline.

8.3.2 Computation of Summary-STP and Backward Propagation

For generality in the ARU algorithm, a *node* may either be a thread, or a channel. Each node has a *backwardSTP* vector that contains summary-STP values received from downstream nodes (see Figure 24). 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 get and put operation.

Below is the algorithm for propagating and computing the summary-STP:

- Receive *summary-STP* value from output connection i from downstream nodes (Figure 24).
- Update $backwardSTP[i]$ with received *summary-STP* value.
- Compute *compressed-backwardSTP* value by applying MIN operator to backward-STP vector (note: refer to the discussion below for the use of other operators).

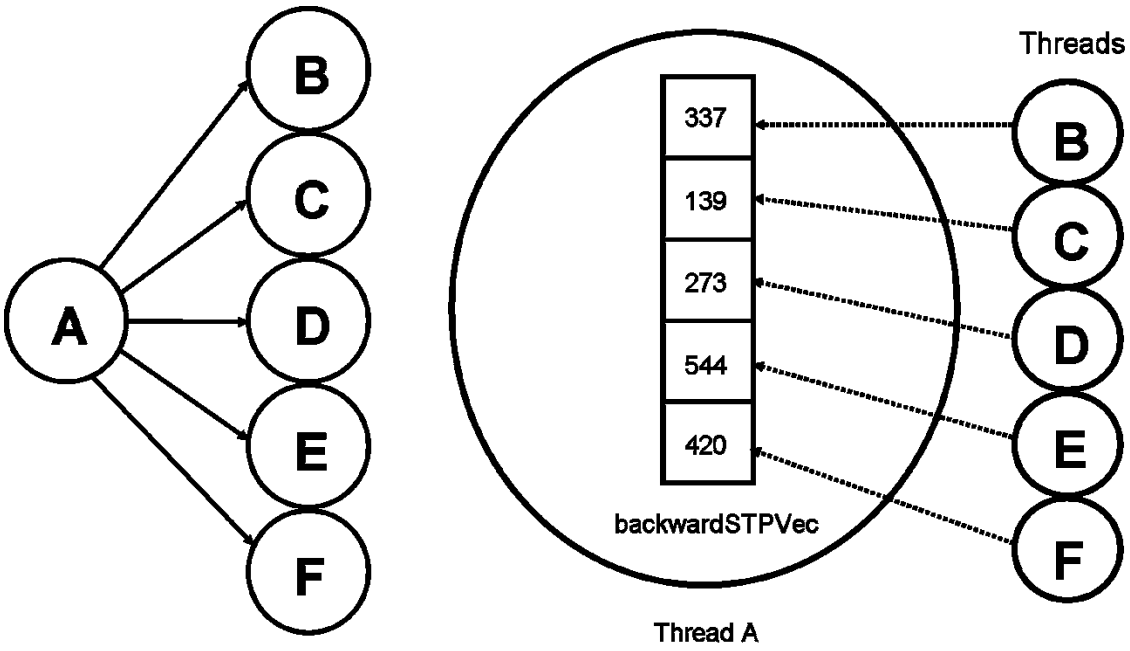

Figure 24: STP Propagation: STP propagation in the pipeline (left) using the backwardSTPVec (right)

- If node is a thread, compute *summary-STP* = *max*(*compressed-backwardSTP*, *current-STP*).
- Else (node is a channel and therefore does not generate *current-STP* values) *summary-STP* = *compressed-backward-STP*.
- Propagate *summary-STP* to preceding nodes in the pipeline.

The computation of the *compressed-backwardSTP* value represents reduction of the execution rate knowledge of consumer nodes (available at the producer node) into a single number that can then be propagated back to nodes in previous stages of the pipeline. This computation can either be done by using the default *min* operator (that represents a conservative approach and assumes no knowledge about data dependencies among the consumer threads), or with the help of a user-defined function that captures data-dependencies among consumer nodes. Applications may have different kinds of dependency flavors, corresponding to the type of data dependencies among the consumer nodes. A complete data-dependency with *all* consumer nodes, for example, can be expressed using the *max*

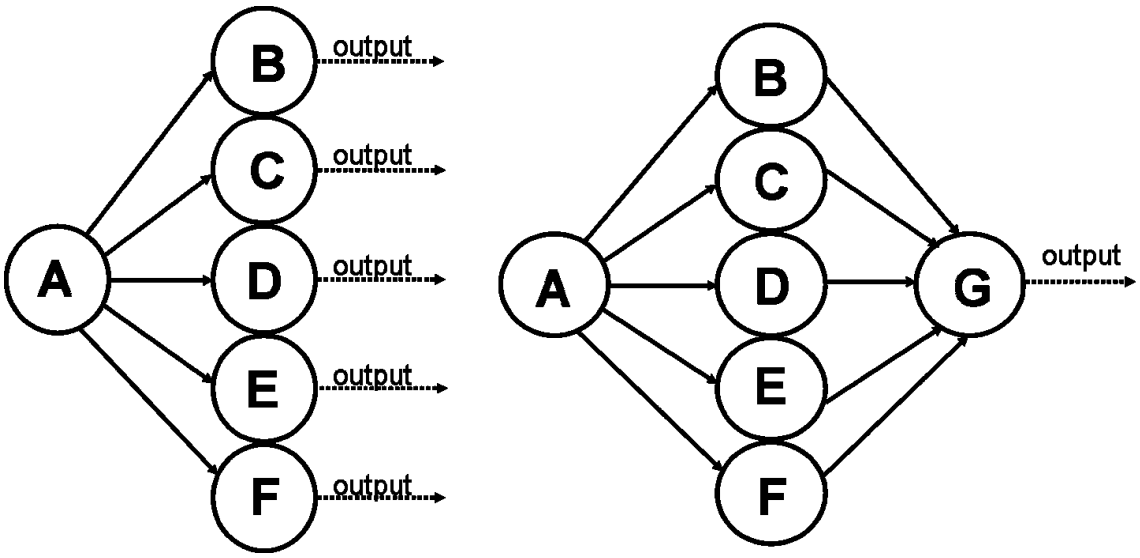

Figure 25: ARU: Examples of Task-graph Topologies with Min (left) and Max (right) Operators Applied to Them

operator (Figure 25). Any function other than the default *min* operator requires the application writer to have some understanding of data-dependencies prevalent in the application because the application writer needs to determine the specific nodes that dictate the *compressed-backwardSTP* value without hurting the current node throughput. The *min* operator is the default operator as it does not affect the throughput at the node and is safe to use in all data-dependency cases.

In the example shown in Figure 24, 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*. The pipeline shown in the right-hand side of Figure 25 depicts a different pattern of data dependency. In this case, A is a thread connected to nodes B-F, that 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 consumer 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 that only thread nodes generate *current-STP* feedback values. This allows a thread with a larger execution period than its consumers to insert its 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 22, 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 *get* and *put* operation.

The ARU algorithm has scalability advantages over a centralized approach used in other cases. For example, in scheduling and QoS systems, 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 scenario for a *summary-STP* value to propagate from the last consumer in the pipeline to the producer would be 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 *get* or *put* 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 allocation) given to the thread by the underlying operating system. 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 (e.g., in [31]). Filters to smooth *summary-STP* noise have not yet been implemented in ARU algorithm in Stampede and

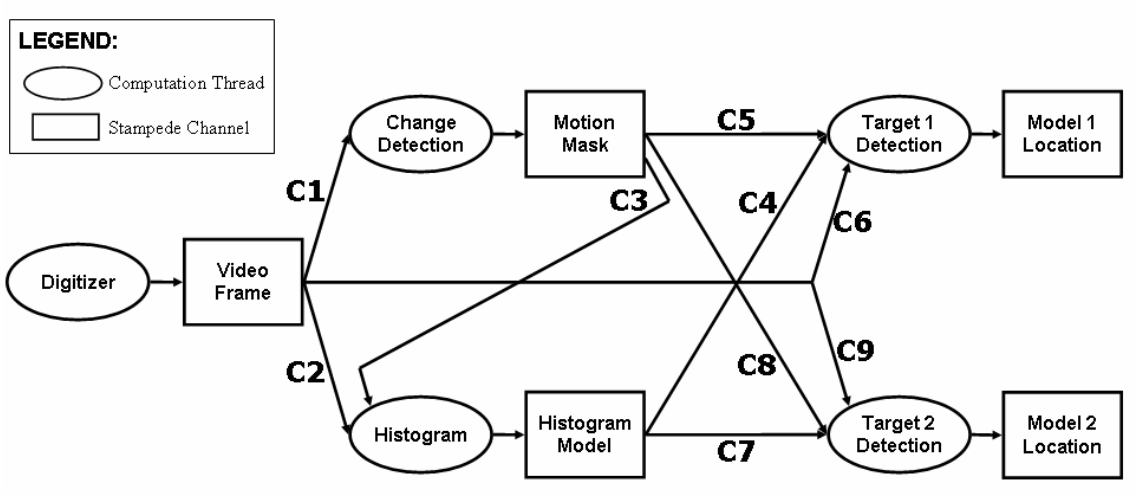

Figure 26: Color Tracker application pipeline

are left for future work.

8.4 Implementation and Performance Evaluation Methodology

We use the *Stampede* distributed programming environment as the test bed for our ARU mechanism. The implementation of ARU includes adding a special API call: (*periodicity_sync*()) to the Stampede runtime system. 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 the Stampede runtime system is modified to piggy-back the *summarySTP* values on existing *get* and *put* calls to and from channels. A parameter is added to all channel and thread creation APIs (e.g., *spd_chan_alloc*()) so that the application can specify producer/consumer dependencies to the underlying ARU algorithm. As mentioned earlier, this data dependency information is optional. The default conservative *min* operator assumes no data dependencies and allows producers to slow down and match the fastest consumer. Although not required, other user-defined dependency-encoded operators (such as the *max* operator) can be used to more aggressively reduce wasted resource utilization.

We use the color tracker application (Figure 26) developed at Compaq CRL [47] to evaluate the performance benefit of the ARU algorithm. This application was described in Chapter 6.3.1.

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

The resource usage of an 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.

Mean memory footprint is the time-average of memory occupancy for all the items in various stages of processing in the different channels of the application pipeline. 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 sum of actual execution time required 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

an 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 eight bytes long, a size that is 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* or *max* operation on very small vectors (order n, where n is the number of output connections from a node; the maximum value of n in this application is three). This computation is done only once by a thread at the end of each data production iteration, and at every *get* and *put* call on channels.

We use the measurement infrastructure described in Chapter 6.2 for recording these statistics in the Stampede runtime system. Each interaction that affects the memory usage of an item (e.g., allocation and de-allocation of items, etc.) is recorded. Items that do not make it to the end of the pipeline are marked to enable a differentiation between wasted and successful memory and computations. A postmortem analysis program uses these statistics to derive the metrics previously discussed in Chapter 3.

The ARU algorithm is implemented in Stampede together with the Dead Timestamp Garbage Collector (DGC), described in Chapter 5. It should be noted that the goals of ARU and Garbage Collection (GC) are orthogonal as ARU tries to reduce the use of wasted resources whereas GC tries to reclaim resources already allocated by the application. Comparing the performance of applications with DGC and ARU to that of a set-up where only DGC is present, allows us to understand the extent of wasted resource reduction and subsequent performance improvement in applications due to ARU. DGC optimizations (described in Chapter 7) are not included in this experiment. These optimizations require the application writer to better understand the application characteristics and data dependencies than the level of understanding required to support DGC and ARU mechanisms. In addition these optimizations can only be applied to a subset of this class of streaming applications.

We introduced an *Ideal Garbage Collector* (*IGC*) when we evaluated the various garbage collection algorithms (Chapter 6). 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 garbage collector 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 frames that will eventually be dropped. The ARU algorithm is then compared to IGC to determine how close the results are to an ideal garbage collector.

8.5 Results

The hardware platform used is the cluster environment described in Chapter 6.1. All experiments are performed using two configurations. In configuration 1, a single physical node is used and all tasks threads are run on this node. All global channels are allocated on the same node as the threads. In configuration 2, five physical nodes are used with all five threads (and their corresponding output channels) mapped to distinct nodes. This setup is discussed in more details in Chapter 6.4.1.

8.5.1 Resources Usage

Table 11: Memory Footprint for the color tracker application under DGC, DGC + ARU with the *min* operator, DGC + ARU with the *max* operator, and in comparison with the Ideal Garbage Collector (IGC).

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

Memory Footprint: Table 11 presents 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 the mean memory footprint with an ideal garbage collector. By eliminating wasted computations, ARU dramatically reduces the memory footprint the application requires, both in the 1-node and the 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 color tracker by almost two-thirds when compared to the color tracker footprint without the ARU mechanism. Figures 27 and 28 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, that is also reflected by the standard deviation of the memory usage in ARU.

Table 12: Wasted Memory Footprint and Wasted Computation Statistics for the color tracker application using ARU.

Setup	Config 1: 1 node		Config 2: 5 nodes	
	% of Memory Wasted	% of Computation Wasted	% of Memory Wasted	% of Computation Wasted
DGC (No ARU)	66.0	25.2	60.7	24.4
DGC w/ ARU-min	4.1	2.8	7.2	4.0
DGC w/ ARU-max	0.3	0.2	4.8	2.1

Percentage of Wasted Resources: Table 12 presents the amount of wasted memory and computation in the color tracker application with and without the ARU mechanism. When not using ARU, more than 60% of the memory footprint is wasted as opposed to 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 computations, that is computations that are directed to items that succeed in reaching the end of the application pipeline.

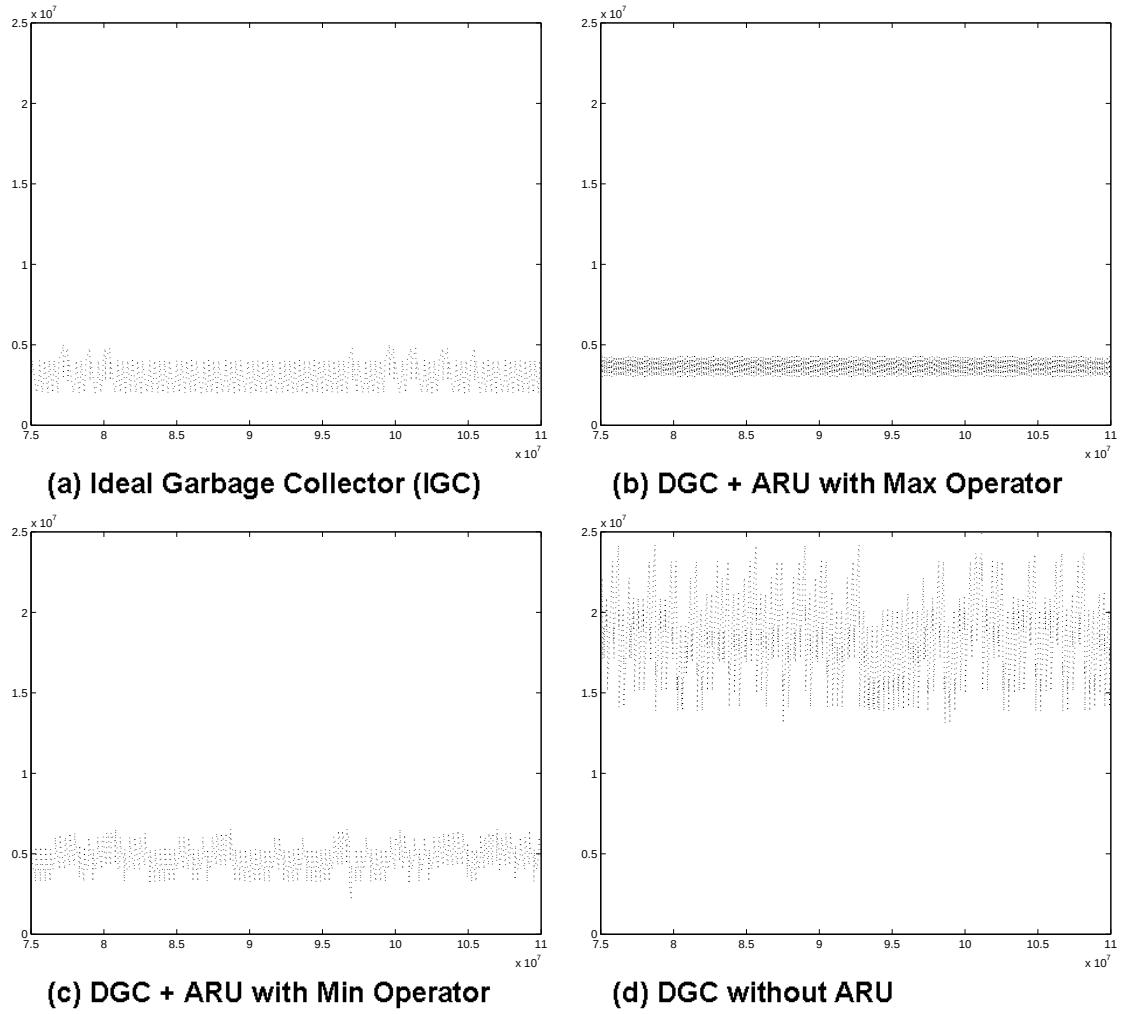

Figure 27: Memory Footprint for 1-node Configuration: All graphs have the same scale. Y-axis: memory use (bytes x 10^7); X-axis: time (microseconds).

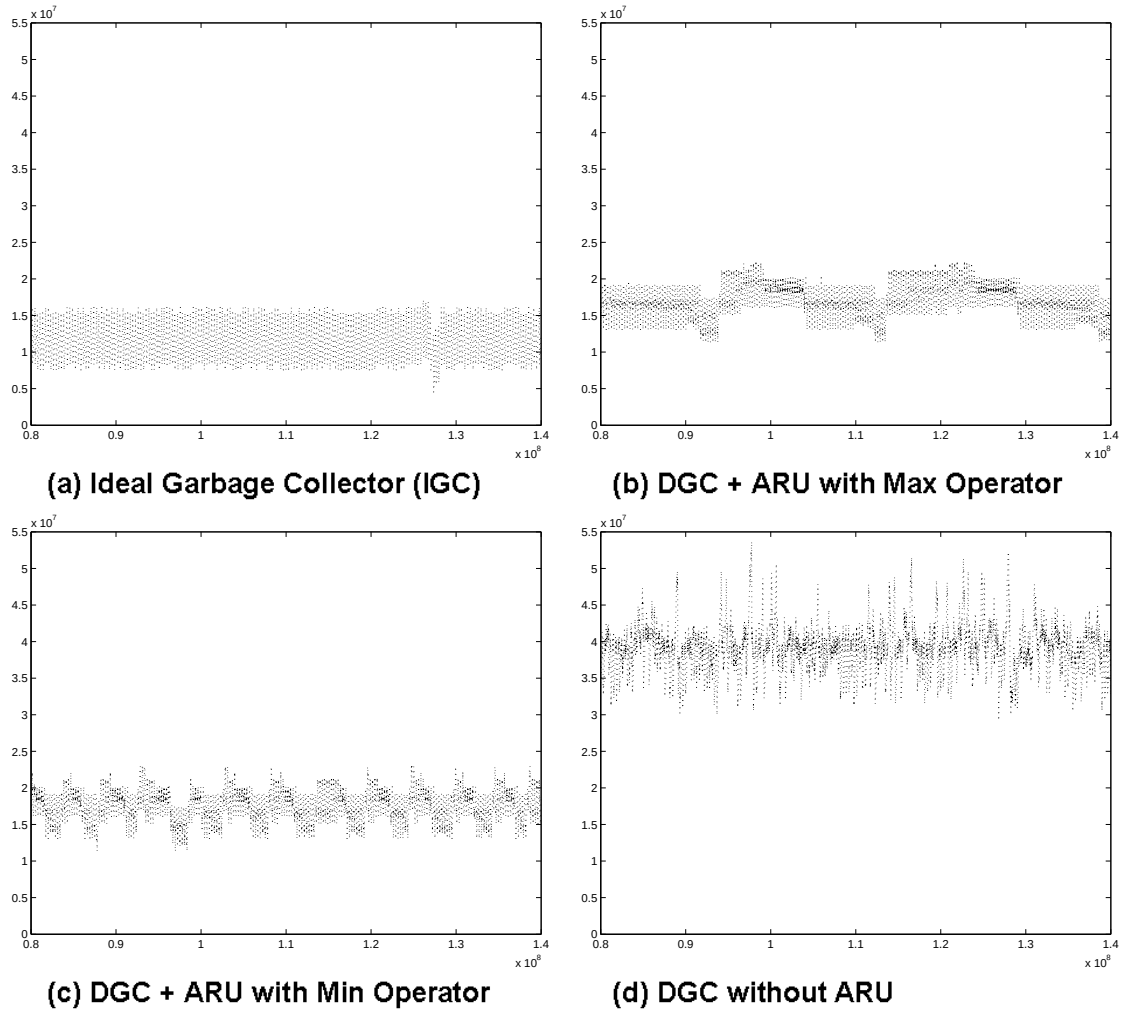

Figure 28: Memory Footprint for 5-node Configuration: All graphs have the same scale. Y-axis: memory use (bytes x 10^7); X-axis: time (microseconds).

Table 13: Latency, Throughput and Jitter of the color tracker application. Jitter is the time difference between two successful outputs

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

8.5.2 Application Performance

In addition to reducing resource waste, the ARU mechanism also succeeds in improving application performance by decreasing jitter and latency, and increasing throughput (see Table 13.

One can observe that even though ARU-max reduces latency compared to no ARU, it performs worse in terms of throughput (5 node configuration). The smaller throughput is not due to a high cost of the ARU mechanism, but is an artifact of the aggressiveness of the *max* operator that 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.

Variations in the *summary-STP* cause 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. As a result, consumer threads are forced to wait for data on buffers. Waiting for consumer thread inadvertently decreases throughput for the application pipeline. However, as consumers are waiting for buffers to have data items ready for consumption, items themselves never spend time waiting in buffers. The observed reduced latency is a consequence of the zero wait time of items in

buffers.

These results suggest that ARU-min is a better candidate for most streaming environments. ARU-min manages to increase the throughput over the baseline DGC, to reduce substantially the application memory consumption, and to reduce the jitter in production. ARU-max manages to reduce the average memory consumption even further, but at a price of decreased throughput and increased jitter in production compared to ARU-min. Thus, ARU-max should be preferred only when memory is the primary scarce resource.

It is clear from these results that a balance between aggressiveness of slowing producers and the amount of resource usage needed to be maintained. We plan to explore this relationship further in future work.

8.6 Conclusions

In this chapter we have presented an Adaptive Resource Utilization (ARU) algorithm, that takes a different approach towards reducing memory usage in streaming applications. Rather than targeting data dependency analysis and locating excess data items as a means of reducing the memory an application consumes, the ARU algorithm analyzes the capacity of the system to successfully process data items. It then uses this information to control the introduction of new data into the application pipeline so that this rate matches the capacity of the system to process data, thus preventing the creation of excess data in the first place.

While substantially reducing the memory usage of the application, the ARU algorithm does not replace the garbage collector. Instead, it operates on top of the garbage collector provided by the runtime system. In fact, the ARU algorithm does not impose any limitations on the type of garbage collector the runtime system uses and any one of the garbage collectors presented so far can be used in conjunction with the ARU algorithm.

While any streaming application can work with the ARU algorithm, there are applications that are more suited to operate in tandem with it. The key for the success of the ARU algorithm is its ability to predict the capacity of the system to successfully process data items. This prediction is based on an analysis of the application's past capacity and

is influenced by:

1. The application's workload - depending on the data, the application may need to perform more complicated calculations or activate different algorithms that may require different processing time. These changes may affect the capacity of the system to successfully process data items.

2. The resources available to the application - the system may incur changes in the resources it can allocate to the application, which may also affect the rate of data production.

Therefore, the ARU algorithm works best when there are no changes in the application workload or the resources available to the application. However, this is an unrealistic scenario for the vast majority of streaming applications. As mentioned in Chapter 1, most streaming applications operate under conditions of a continuously changing workload that affects the system's capacity to process data items. In addition, the resources the system can allocate to an application also tend to fluctuate, and these fluctuations also affect the capacity of the system to successfully process data items.

A more realistic scenario is when the application or the resources change from time to time, but not continuously. Under these circumstances, the ARU algorithm works well in the steady state periods, when the workload and/or the resources available are stable. When substantial changes to the application's throughput occur, the ARU algorithm lags behind until it updates its data structures with information about the new throughput.

In cases where an application can increase its throughput, the ARU algorithm keeps the application working at the original, lower, throughput. It enables the application to increase its throughput only when the information regarding the new conditions manages to propagate throughout the application pipeline. During this propagation time the application underutilizes the available resources, thus failing to increase and match its throughput to the level possible.

In cases where the application has to decrease its throughput, the ARU algorithm keeps the application working at the original, higher, throughput. It enables the application to

decrease its throughput only when the information about the new conditions manages to propagate throughout the application pipeline. During this propagation time, the application over utilizes the available resources, and generates data items that system does not have the capacity to fully process. As a result, the application wastes memory and computational resources until the ARU algorithm manages to readjust the application's throughput to the capacity of the system.

Therefore, the effectiveness of the ARU algorithm is dependent on the frequency of these capacity changes. The various garbage collection algorithms, on the other hand, are not sensitive to these throughput changes. They concentrate on harnessing existing data dependencies to uncover data items that are not going to be fully processed. These items are reclaimed to reduce the memory footprint of an application. Most of the garbage collection algorithms do not directly target computational resources, and the influence they have on the allocation of computational resources is marginal. By matching the introduction of new data to the capacity of the pipeline, the ARU algorithm manages to substantially reduce both the amount of memory the application consumes and the computation resources it requires. In addition, the ARU algorithm succeeds, in some cases, to increase the throughput of the application.

CHAPTER IX

RELATED WORK

9.1 Programming Models for Writing Distributed Applications

Several models have been proposed to ease programming of distributed applications. Tuple Space [11], for example, is a logically global, associative object memory. The basic data structure of a Tuple Space, a tuple, is an ordered list of data objects. The global nature of this space allows for the tuples to be accessed from any application node. In addition, the tuple space allows for concurrent access to tuples, and provides interprocess communication and synchronization logically independent of the underlying computer or network. Linda [11], [5] is a parallel programming language that implements Tuple Spaces. It is based on C (C-Linda) and Fortran (Fortran-Linda) and it provides programmers with operations to manipulate tuples. Linda creates a virtual shared memory system on heterogeneous networks, thus enabling programmers to write parallel programs that can be executed on a wide range of computing platforms. Linda's virtual shared memory environment enables processors to view the memory as a single global memory space, thus allowing for different parts of the data to reside on different processing nodes. The Linda programming environment suits the master/worker model. Under this scheme, task and workers are independent of each other. The *master* divides the work into discrete units of work called *tasks*, and *workers* retrieve these tasks and execute them. More recent implementations of middle-ware packages for ubiquitous computing, such as Sun's JavaSpaces [52] and IBM's TSpaces [25], are based on Linda, and provide a general infrastructure for distributed applications.

Space-Time programming model [46], on the other hand, was developed specifically for stream-based applications. These applications are different from general distributed applications in that they manipulate data that involve time. The Space-Time memory supports any type of data structure. The model associates a time attribute with each and every data item. This time attribute may affect data production and consumption patterns

because streaming application are required to operate within soft real-time constraints and they can achieve their objective by processing only a subset of the data captured. Thus, incorporating the time attribute into the programming model enables Space-Time not only to provide the programmer with an easy and intuitive programming environment, tailored for streaming applications, but also to manage the system's resources more efficiently. In particular, this set of applications has unique characteristics that pose a challenge yet provide an opportunity for much more aggressive garbage collection.

The Stampede runtime system [45] implements the Space-Time memory model. The association between a data item and a time attribute allows Stampede to support the master/worker model [21], a model supported by Linda. It also enables Stampede to support streaming application models, and in particular data and computation dependencies prevalent in these applications.

9.2 Garbage Collection Algorithms

The traditional GC problem (on which there is a large body of literature, for example, [55, 19]) concerns reclaiming storage for heap-allocated objects (data structures) when they are no longer "reachable" from the computation. The "name" of an object is a heap address, i.e., a pointer, and GC concerns a transitive computation that locates all objects that are reachable starting with names in a symbol table. In most safe GC languages, there are no computational operations to generate new names (such as pointer arithmetic) other than the allocation of a new object. Space-Time's GC poses an orthogonal problem. The "name" of an object in a channel is its timestamp, i.e., the timestamp is an index or a tag. Timestamps are simply integers, and threads can compute new timestamps. GC of timestamps is concerned with determining when a timestamped item will not be used any more (regardless of whether it is reachable) and thus, storage associated with all or some items that are tagged with this timestamp can be reclaimed.

The problem of determining the interest set for timestamp values in the Space-Time programming model has similarity to the garbage collection problem in Parallel Discrete Event Simulation (PDES) systems [10], yet it is less restrictive. Unlike the Space-Time

programming model, PDES systems require that repeated executions of an application program using the same input data and parameters produce the same results [9]. To ensure this property, every timestamp must *appear* to be processed *in order* by the PDES system. A number of synchronization algorithms have been proposed in the PDES literature to preserve this property. First attempts to perform GC were based on conservative assumptions. Algorithms such as Chandy-Misra-Bryant (CMB) [4, 8], for example, process the timestamps strictly in order, exchanging null messages to avoid potential deadlocks. Decisions are made locally, and there is no reliance on any global mechanism or control. Optimistic algorithms, such as Time Warp [16], assume that processing a timestamp out of order by a node is safe. However, if this assumption proves false, then the node rolls back to the state prior to processing the timestamp. To support such a roll back, the system has to keep around state, which is reclaimed based on calculation of a Global Virtual Time (GVT). The tradeoff between the conservative (CMB) and optimistic (Time Warp) algorithms is space versus time. While the former is frugal with space at the expense of time, the latter does the opposite.

On the other hand, the Space-Time programming model does not require in-order execution of timestamps, nor does it require that every timestamp be processed. Consequently, it does not have to support roll backs. If nothing is known about the application task graph, then similar to PDES, there is a necessity in the Space-Time programming model to compute GVT to enable garbage collection. The less restrictive nature of this programming model allows conception of different types of algorithms for GVT calculation like the one described in Chapter 4. Garbage collection can be even more aggressive if application level information is provided to the run-time system. The mechanisms described Chapters 5 and 7 use application-level knowledge enabling garbage collection based entirely on local events with no reliance on any global mechanism. Like CMB, they are frugal in space; however, application-level knowledge enables them to reduce latency while reducing space requirements. Gaining performance in both fronts is achieved by breaking the boundary between the application and the run-time system. Whether it is possible to achieve similar gains in PDES systems is an interesting problem worthy of investigation.

9.3 Adaptive Resource Utilization

9.3.1 ARU and Quality of Service

Quality of Service (QoS) mechanisms support the allocation of resources so that users and/or applications requirements are met. The Q-RAM, or the QoS-based Resource Allocation Model [43], for example, provides quality of service support along multiple dimensions such as timeliness, reliable delivery schemes, cryptographic security and data quality. Q-RAM assumes a system with multiple concurrent applications, each of which can operate at different levels of quality, based on the system resources available to it. Although the problem Q-RAM tackles is NP-hard, near-optimal polynomial algorithms have been developed [22]. An extension to the Q-RAM [23] model solves the problem of apportioning *multiple* finite resources to satisfy the QoS needs of multiple applications along *multiple* QoS dimensions.

The Rialto OS [17] presents a modular OS approach, with the goal of maximizing the user's *perceived* utility of the system, instead of maximizing the performance of any particular application. A QoS manager in the RT-mach OS is used to allocate resources to applications, each of which can operate at any resource allocation point within minimum and maximum thresholds [24].

Several proposals, such as the end-host architecture for QoS-adaptive communication [1], and the control theoretical model for QoS adaptations [26], incorporate a feedback control to help providing a distributed multimedia communication.

The Adaptive Resource Utilization (ARU) mechanism we present in Chapter 8 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; however, the two mechanisms are substantially different from each other. Firstly, ARU deals with optimizing *resource utilization*, as opposed to QoS, that can be categorized 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. Unlike ARU, 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 environment above the OS-level. QoS provisioning typically requires the application writer to understand, and in many cases to specify, the application's behavior for different levels of service (see [57], [2], [53], and [33]). However, the default configuration of ARU does not require any developer involvement, yet a minimal involvement may optimize the performance of the application even further.

9.3.2 ARU and Real-Time Scheduling

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 goal 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 using available knowledge about data dependencies. ARU therefore takes advantage of intra application dependencies whereas real-time scheduling only uses global information about all applications. The global knowledge is readily available at the operating system level, where real-time scheduling is typically performed. However, internal data-dependencies can be found only within an application or a run-time system such as Stampede, designed for a particular class of applications.

Real-time scheduling also includes reservation style scheduling where different threads must first reserve their CPU time to be allowed the use of resources (e.g., [54], [18], [51], and [35]). Reservation style scheduling techniques differ from our approach in that

1. They are all scheduling techniques

2. They are instrumented in the kernel, and,

3. They require application developers to supply accurate period/proportion reservation.

9.3.3 ARU and Feedback Based Real-Time Scheduling

There are systems that alleviate the requirement of accurate reservation information by using feedback: The Real-Rate [50] mechanism removes the dependency on specifying the rate of real-time tasks.

More traditional real-time schedulers such as *Earliest Deadline First* (EDF) [27], *Rate Monotonic* RM [27] and Spring [58] are all open-loop static scheduling algorithms that require complete knowledge about tasks and their constraints, and do not support dynamic workloads well. Variants like FC-EDF [28], use feedback to reduce miss-ratio as much as possible and to achieve high CPU utilization especially in resource constrained environments. Under this scheme, real-time scheduling parameters, such as period and proportion are provided statically at start-up time. However, the feedback mechanism refines these parameters to reduce the miss-ratio according to the system's dynamics.

FC-EDF uses PID (Proportional Integral-Derivative) as the basic feedback control technique for feedback control scheduling. The miss-ratio performance metric is periodically fed back to the PID controller. The PID controller maps the miss-ratio of accepted tasks to the change in requested utilization. The required control action (that is, increasing or decreasing the CPU resources allocated to the task) is computed according to the PID control formula.

Although FC-EDF uses a feedback control loop, it differs fundamentally from the ARU mechanism. FC-EDF attempts to increase CPU utilization. By contrast, ARU strives to reduce CPU utilization if it determines that the additional resources are not directed towards useful work. Thus, while ARU strives to minimize work hat is not useful, FC-EDF attempts to increase CPU utilization, regardless of whether the additional CPU resources improve the overall performance of the application. Additionally, ARU uses a simple mechanism that measures execution time. This information is then fed to neighboring nodes, that use this measure to decide whether to utilize all the resources available to them or whether to relinquish some of those resources, because they are not going to contribute to the overall

application performance. FC-EDF, by contrast, measures miss-ratio, and uses this information to redistribute the available CPU cycles among tasks scheduled on the *same* node according to the PID control formula with the aim of reducing miss-ratio. As mentioned in Chapter 1.1.5, stream-based applications are less concerned with the miss ratio because applications are not required to process *all* the data to achieve their objectives.

Like ARU, tailored to support a specific class of applications, Adaptive Earliest Deadline scheduling (or AED) was introduced to improve the EDF algorithm for real-time databases [14]. In transactional database systems *a-priori* knowledge of real-time parameters, such as rate and period, for real-time transactions is not available. As a result, admission control cannot be implemented to avoid over committing the system resources. AED uses the feedback parameter *HITCapacity* to separate all transactions into two groups: (1) the *HIT* group that consists of transactions that are capable of meeting their deadline; and, (2) the *MISS* group that contains the remaining transactions that are going to miss their deadline. Similar to FC-EDF, AED attempts to reduce the miss ratio. It achieves this goal by a careful manipulation of the priority assigned to the various transactions using the *HITCapacity* feedback parameter.

Other hybrid schedulers such as SMART [35], and *Best Effort and Real-Time* (or BERT) [3] 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 by directing resources from non-real-time applications to real-time applications. The ARU mechanism is tailored for stream-based applications, where real-time scheduling is not required.

Recent work on *adaptive* scheduling considers resource constrained environments other than limited CPU (e.g., high memory pressure [39]). 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 rather to avoid using resources on computing unneeded data altogether. Avoiding wasted computation indirectly reduces memory pressure by using fewer resources to begin with.

Massalin and Pu [32] introduced the idea of using feedback control loops similar to hardware phase locked loops in real-time scheduling. The adaptive scheduling is based

on a *software feedback* mechanism, that compares between input and output events, and attempts to adjust the output sequence according to the input observations. Two flavors of software feedback are presented: an *event frequency feedback system* (or EFF), and a *time interval feedback system* (or TIF). EFF measures and adjusts the event frequency of the input (events/seconds). TIF, on the other hand, measures and adjusts the time interval between inputs (seconds/event). The ARU mechanism can be categorized as a TIF system, as it measures and adjusts the Sustainable Thread Period (or STP), defined as the time a thread spends processing an item. However, the ARU mechanism uses this feedback for different purposes. The STP helps the ARU mechanism to infer what resources are not going to contribute to the overall application performance. These resources are then relinquished and can be directed towards more useful work. The approach of Massalin and Pu, on the other hand, uses the feedback information to infer the unused resources that can be provided to threads that require them.

Another significant difference between the two mechanisms is the adaptation granularity. The adaptive scheduler of Massalin and Pu allows for a fine-grain level of adaptation. The feedback mechanism is capable of executing many scheduling actions in a short interval (for example, few hundred context switches while completing a job of only 10 milliseconds). The ARU mechanism, on the other hand, supports only a coarse- grain level of adaptation. However, this level of adaptation is sufficient for stream-based applications because many of the adaptations the runtime system has to perform on behalf of these applications are input dependent. The input rate is dependent in-turn on the acquisition rate of the input devices the application uses. The ARU mechanism provides this level of adaptation because it uses the Sustainable Thread Period as the feedback parameter.

Lastly, the ARU mechanism completely relies on a distributed mechanism that propagates the STP values among neighboring nodes of a single application. This mechanism resides at the runtime system level. The adaptation mechanism of Massalin and Pu, on the other hand, is executed at the scheduler level, is global, and can be applied to multiple applications on the node. However, unlike ARU, that optimizes the resource allocation of a

single distributed application running on multiple nodes, the optimization method of Massalin and Pu mechanism is done at the node level. Thus, although it can optimize multiple applications that run on the same node, it lacks the ability to optimize a single distributed application, running on multiple nodes.

The Swift toolbox [41, 6, 12] was developed to allow portability of the feedback control mechanism Massalin and Pu proposed [32] from the operating system test bed, the Synthesis Kernel [42]. However, these mechanisms try to improve upon scheduling (resource management) and do not try to eliminate wasted resource usage (resource utilization). In addition, feedback mechanisms require application modification [7, 49], where ARU, incorporated into the Stampede runtime system, is available by default to application writers, and does not require any application modification. The feedback control loop work of Massalin and Pu [32] deals with feedback filters, where feedback information is first filtered before being 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 application pipeline would improve the performance of the overall pipeline. However, we deal with scenarios where the option of more resources, for example, 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 [32, 41, 7, 6, 49, 12] where bottleneck threads are given more resources to improve their throughput.

9.3.4 ARU and Garbage Collection

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 [55, 19] and Timestamp based GC in streaming applications (see Chapter 4). 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, see Chapter 5) use inferences that are based on virtual time associated with data items to identify garbage. These are

data items that the application will not use in the future. As a result, timestamp based garbage collection algorithms can bring the garbage collection decision forward, instead of waiting until the item is not reachable.

While timestamp based garbage collection algorithms enable an earlier GC decision compared to traditional GC algorithms, this decision is made only after all of the processing and network resources have already been allocated to the garbage item. 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 the execution, although it reduces the magnitude of the problem.

CHAPTER X

CONCLUSIONS AND FUTURE WORK

10.1 Summary of Proposed Algorithms

This dissertation focuses on garbage collection algorithms and resource allocation techniques as a means of reducing the memory footprint of streaming applications. Each one of the algorithms presented takes advantage of different properties of streaming applications. Tables 14 and 15 summarize the characteristics of the algorithms presented in the previous chapters in terms of their performance, limitations, complexity, and the subset of streaming applications they support.

The only algorithm that targets all streaming applications is the Transparent Garbage Collector (TGC). It is implemented in the runtime system and does not require any involvement from the application programmer. Thus, TGC supports any level of application dynamism and does not require the application programmer to be aware of any data dependency prevalent in the application. This ease of use, however, comes at a price. TGC makes garbage collection decisions based on information from all application threads, and unlike the rest of the algorithms, the information needed to support the garbage collection decisions are not piggy-backed on other communication operations. Therefore, TGC's communication overhead is higher than the other techniques presented in this dissertation. As it does not hold any assumptions regarding the application and data dependencies associated with it, TGC is the least aggressive algorithm. Thus, applications running with the Transparent Garbage Collector exhibit the highest memory footprint as compared to the rest of the proposed algorithms.

Two algorithms, REF and KLnU, make local garbage collection decisions based on local information. The former is a full-fledged garbage collector, while the latter is an optimization that works on top of an existing garbage collector. KLnU bases its garbage collection decision on an attribute that describes a data dependency property associated

with a channel. This attribute can be supplied either by the application programmer, who associates channels with an attribute and encodes it as part of the application, or by the application through a compiler-based analysis of communication patterns. REF, on the other hand, bases its decision on information received from producers of data items. This information is provided by the application writer. Basing the garbage collection decision on information derived from the producer limits the set of streaming applications that REF can support, because the reference count cannot be changed once a data item is produced. Thus, an application cannot have any dynamic feature that affects the reference count between the time a data item is produced and is garbage collected. On the other hand, KLnU, that describes dependency as a channel attribute, detaches KLnU garbage collection decision from any specific dynamic configuration of the application and fully supports application dynamism. The description of a dependency as a channel attribute has another advantage. KLnU does not incur any communication overhead, and thus is unique from the rest of the algorithms presented.

Both REF and KLnU make local decisions based on local information and are more aggressive than TGC, a garbage collector that makes global decisions based on global information. KLnU is found to be more aggressive than REF for the color-based tracker application used in this study. However, REF and KLnU are more effective in different scenarios. REF tends to be more effective in identifying garbage items when the application task graph exhibits sparser communication patterns between the application threads. As an example, recall the color-based tracker application that is used in this study. The digitizer thread produces a digitized image that is sent to three threads: the mask, the histogram, and the tracker threads. All of these threads need to consume the image before it can be considered as garbage according to the REF algorithm. Moreover, the tracker thread receives the digitized image only after the mask and the histogram threads produce the corresponding items. The garbage collection decision is, therefore, postponed until the tracker thread receives the corresponding data items from these two threads. Had the communication pattern been different, and the color tracker was not required to process the digitized image, REF would have been able to garbage collect the image earlier, once it was

consumed by the mask and the histogram threads. This example explains why REF identifies garbage items more effectively when the application task graph is less connected and when graph connections exist between two consecutive stages. KLnU effectiveness, on the other hand, is not related to the topology of the application's task graph, but to local data dependencies prevalent in the application. First, performance is dependent on the extent to which Keep Latest if Unseen producer/consumer relationships exist in the application. This dependency, however, does not pose a critical limitation since many streaming applications include this kind of relationship. In fact, the keep latest relationship is one of the attributes that make streaming applications a unique and distinctive class of applications. Secondly, KLnU aggressiveness depends on the relationship between the rate of data production and the rate of data consumption. The faster the producers are compared to the consumers, the more aggressive the KLnU performance becomes. This is because, under these conditions, garbage collection decisions are not postponed until the receiving threads are able to get the data, but decisions can be made at production time.

Unlike TGC, that makes global decisions that are based on global information, and REF and KLnU that make local decisions based on local information, the rest of the garbage collection algorithms proposed make local decisions based on global information. DGC uses guarantees the application provides and propagates local information throughout the application, while PDS propagates information regarding local dead set throughout the application. Garbage collection decisions can be performed quickly, as they are based on local information. These decisions are then piggy-backed on application instigated data communication and are incorporated with the local decisions of the other nodes. This combination of local and global information is found to be most effective in reducing the memory footprint of streaming applications. Both DGC and PDS are applicable to all static and most dynamic applications.

The Adaptive Resource Utilization algorithm differs from the rest of the proposed algorithms in that it is not a garbage collector. While garbage collectors identify garbage items after data has already been produced, the ARU algorithm targets the production of data items to prevent excess data creation in the first place. The decision on data production

rate is performed locally based on global information. In this sense, ARU is similar to DGC and PDS, and like these two algorithms, it is found to be most effective in reducing memory footprint in streaming applications. Controlling the data production rate has a positive side-effect of diverting resources from excess data items, and as a result, improving the application performance in terms of latency and throughput. ARU is found to be a very aggressive algorithm; however, its performance varies and is dependent on the level of stability in the application's production rate. The more stable the production rate is the more effective ARU becomes.

Table 14: A Summary of Proposed Algorithms Properties

Algorithm	Algorithm Details		
	Decision	**Limitations**	**Aggressiveness**
REF	local, based on local information	reference count needs to be made known at the time an item is produced	aggressive local identification, however information does not propagate to other parts of the application
TGC	global, based on global information	none	the algorithm is distributed and requires information from other parts of the application
DGC	local, based on global information	all potential threads and connections are known at application startup time	the algorithm eliminates both irrelevant work and garbage
KLnU	local, based on local information	optimization, requires an additional garbage collector	however, performance varies and is dependent on the extent at which Keep Latest if Unseen relationships exist in the application
PDS	local, based on global information	all potential threads and connections are known at application startup time	however, performance varies and is dependent on the extent at which Keep Latest if Unseen relationships exist in the application
ARU	local, based on global information	requires an additional garbage collector	however, performance varies and is dependent on the level of stability in the production rate of the application

Table 15: A Summary of Proposed Algorithms Overheads and Application Characteristics

Algorithm	Algorithm Overhead		Application Characteristics	
	Computation	**Communication**	**Application Set**	**Additional Info**
REF	simple operations upon data propagation	additional information is piggy-backed to existing communication operations and amount to an addition of an integer that represents the count	Static, limited level of dynamism is also supported but is left for the programmer	application needs to provide reference count information
TGC	runtime periodically computes status	daemon threads in each node exchange status information periodically with their peers	any application	none
DGC	simple operations upon data exchange	additional information is piggy-backed to existing communication operations and is proportional to the number of connections associated with a thread or a channel	all static and most dynamic applications	providing forward and backward guarantees; providing also transfer functions may further improve the performance
KLnU	simple operations upon data production	none	any application; aggressiveness is application dependent	programmer specifies channels as attributed channels
PDS	simple operations upon data exchange	additional information is piggy-backed to existing communication operations and is proportional to the number of connections of a thread or a channel	all static and most dynamic apps; aggressiveness is application dependent	programmer specifies channels as attributed channels
ARU	simple operations upon data exchange	additional information is piggy-backed to existing communication operations and amount to an addition of an integer that represents the processing rate of a thread	any application; aggressiveness is application dependent	none; providing also a function that computes compressed-backwardSTP may further improve the performance

10.2 Methodology Summary

This dissertation presents a method to explore the design space of memory optimizations for distributed stream-based applications. Rather than implementing each and every design option, this dissertation introduces a methodology and an infrastructure to simulate, evaluate, and compare the various design options in distributed environments. The methodology also enables designers to frame hypothetical questions in the form of "what-if" scenarios. Designers can better understand the design space tradeoffs by analyzing the performance of these scenarios. In addition, the methodology allows for the expression of an ideal design option as a "what-if" scenario. Although this ideal design option cannot be fully implemented, it can be used as a reference point to compare the different design options. In the context of the problem discussed in this dissertation, of evaluating garbage identification algorithms, an Ideal Garbage Collector is defined. This garbage collector is used as a reference point against which the proposed garbage identification algorithms are evaluated. The Ideal Garbage Collector also alludes to the success of the algorithm in identifying garbage items, such that when the gap between a proposed algorithm and the Ideal Garbage Collector is sufficiently small, designers can stop looking for more efficient algorithms.

The comparison between the implementation and the simulation results (see Chapter 7.10) shows the effectiveness of this methodology in predicting the performance of different garbage identification algorithms. The comparison reveals only a small difference between the simulation and the implementation because the methodology imposes limitation on the changes to a trace a simulation is permitted to make. A simulation is allowed to alter only the deallocation time of an item, thus affecting only the time an item is being garbage collected and removed from the memory of the application. It is not allowed to change other attributes related to the execution of an item, such as whether a specific item is being created, its creation time, its processing time, and the nodes that are involved processing the item. Thus, the only changes the simulation is permitted to make are performed once the application has already provided the item with all the computation and network resources it requires, and the item is waiting to be garbage collected.

Although we demonstrate this methodology on distributed environments for stream-based applications, the methodology can be extended to other distributed environments as long as the changes that the simulation is required to make are limited, and do not have a substantial effect on the overall performance. As mentioned above, the case study we provide in this dissertation permits only minimal changes to the trace (namely, memory deallocation time). However, the methodology can permit changes to other system components, such as scheduling. The validity of the simulation has to be assessed on a case by case basis. We use the evaluation of demand-driven models of execution in TStreams [20] to demonstrate how the methodology can be extended to include changes to scheduling decisions.

TStreams is a general programming model that enables the programmer to express all the potential parallelism in an application. A parallel program is expressed as a collection of target-independent units called *steps*. To enable an execution of the program on a real platform, these steps are *mapped* to the available resources. Obviously, this mapping can be done in many different ways, and system designers may want to evaluate different mapping algorithms. In particular, demand-driven models of execution may be introduced to TSreams to improve applications performance. Evaluation of proposed models can be tedious, and the introduction of an ideal model can help in determining how well suggested models perform. An ideal execution can be defined similar to the way an Ideal Garbage Collector is defined in this dissertation.

A log of an actual execution can be used as a baseline. The log can then be analyzed to eliminate any *wasted* execution of a step, that is an execution that did not end up producing useful data. The freed processor resources can then be re-allocated to perform useful computations of steps. Thus, an ideal demand-driven execution model is generated, against which proposed models can be compared.

While this example illustrates a particular use of the methodology proposed in this dissertation, it also illustrates its limitations. The re-allocation of resources can be simulated only under the assumption that all processors are equal and the network is symmetric.

10.3 Research Contributions

An event-logging measurement infrastructure is presented in [40]. Since some of the events measured are distributed across multiple machines, the main challenge this measurement infrastructure faces is reconciling the times spent on the different nodes in various activities of interest to derive the exact amount of resources (e.g., processing time, memory) each one of these distributed activities consumed. As some of the events are very short, the measurement infrastructure has to be cycle accurate. In [40] the measurement infrastructure is used to quantify the interaction between the Stampede runtime system and the operating system. It allows the distinction between events related to the application logic, the Stampede runtime system, the messaging layer, and time the application spent blocking, and thus quantifying the overheads associated with the Stampede runtime system and the messaging layer. It is also used to distinguish between the time spent on dynamic memory allocation and synchronization activities. The study in [40] compared the performance of the Stampede runtime system with two operating systems: Linux and Solaris.

The event-logging measurement infrastructure is also used in [46] and in [45] to quantify the additional overheads associated with the Stampede runtime system. The results show that Stampede incurs only low performance overheads.

In [13], [44], and [29] we explore the problem of garbage collection algorithms in stream-based application. In [13] and [44] a Dead Timestamp-based Garbage Collector (DGC) is presented. It is compared with a Reference count-based garbage collector (REF) and with a Transparent Garbage Collector (TGC). DGC is found to incur a small performance overhead; however it reduces significantly the application memory footprint compared to REF and TGC. In addition DGC is found to scale better than TGC [44] as the application is distributed across more nodes.

A methodology is also presented in [13] to determine the success of the DGC algorithm in using the garbage identification potential prevalent in the system. A trace created by the measurement infrastructure is used to simulate an Ideal Garbage Collector (IGC). This garbage collector records only the events related to items that succeed in reaching the end of the pipeline, and immediately collects any item that is not used. A comparison between

the DGC and the IGC reveals that although under DGC the application uses significantly less memory when compared to REF and TGC, it still consumes up to four times more memory than the ideal garbage collector.

In [29] four optimizations are presented to help closing the gap between DGC and IGC: Out-of-Band Propagation of Guarantees (OBPG), Keep Latest 'n Unseen (KLnU), Propagation of Dead Sets (PDS), and Out-of-Band Propagation of Dead Sets (OBPDS). The optimizations are simulated and compared to a baseline DGC and the ideal garbage collector (IGC). The results led to the implementation of the KLnU and the PDS optimizations (see Chapter 7.7).

Finally, in [30] the Adaptive Resource Utilization (ARU) mechanism is presented. Rather than using the garbage identification approach, of waiting for items to be created and then identify the ones that can be classified as garbage, the ARU mechanism attempts to eliminate the creation of garbage items in the first place. This is done by adjusting the pace at which data items are introduced and the capacity of the system to process these data items. The ARU mechanism is complementary to the garbage identification algorithms, and can significantly reduce the memory footprint to levels that are close to the memory footprint of the application under the ideal garbage collector.

10.4 Future Work

This dissertation investigates the design space of memory optimizations for streaming applications. Two approaches are proposed to reduce the memory pressure of these applications:

1. Uncover and use existing data dependencies as a means for earlier identification of garbage items.

2. Match the amount of data the application attempts to process with the capacity of the system to process this data.

Both these methods are found to be effective in substantially reducing the memory pressure an application exerts. However, many questions are still left unanswered. In the following we present some of these questions and suggest directions for future research on this subject.

10.4.1 Garbage Identification Methods

The algorithms we present for garbage identification involve understanding the data dependencies an application exhibits. The algorithms presented require the application writer to encode these dependencies in one form or another. A future research direction will be to discover ways to automate the identification process of these dependencies. Providing the runtime system with the ability to automatically identify data dependencies has the potential of encapsulating the garbage identification process within the runtime system and thus relieving the application writer from the need to understand and encode these relationships as part of the application development process.

The automation of data dependencies identification is far from being trivial. Because of the dynamic nature of streaming applications, these dependencies are dynamic as well, at least for a subset of these applications. The ability of the runtime system not only to automatically identify data dependencies, but also to analyze these dynamic dependencies may assist in invoking the most efficient strategy to identify garbage data items. This ability to choose the most appropriate garbage identification algorithm at an application startup time has the potential of substantially reducing the memory footprint of streaming applications.

In addition to dynamically identifying the best garbage identification strategy, it may be possible to achieve an even lower memory footprint if the runtime system is also capable of switching among different strategies based on the current application's execution conditions. As mentioned in Chapter 1, streaming applications tend to be long lived, and there is a good probability of changes in the runtime system and data set conditions. These dynamic changes may also necessitate a change in the garbage identification strategy chosen at the application startup time to further reduce the memory consumption of the application.

Finally, this dissertation explores only a subset of the garbage identification algorithms possible in streaming applications. Additional work is needed to uncover additional garbage identification strategies.

10.4.2 Adaptive Resource Utilization

ARU bases its system's capacity predictions on past experience. As a result, the performance of the ARU algorithm is compromised when execution conditions change. A possible area of research may involve different methodologies to predict the capacity of the system. Either the application or the runtime system may be tasked with the mission of predicting changes in input data that may influence the capacity of the system to process data items. For example, a surveillance application may receive an indication of human presence via motion detectors, and indicate to the runtime system to reduce the rate of introduction of new data to the system in anticipation of a load increase as a result of a heavier computational load once a human is detected in the scene. The runtime system may be tasked with predicting changes to the available resources and with understanding the influence these changes have on the system's capacity to process data.

10.4.3 Putting it All Together

The study conducted in this dissertation examines different memory optimization strategies when the system is tasked with running a single application. A more complex problem is when multiple applications, with different characteristics, are executed in tandem. This is a common scenario for the deployment of streaming applications (see the example of the augmented living environment for the elderly provided in Chapter 1), and it may require additional communicational mechanisms between the applications and the runtime system to regulate the limited resources available and direct them to support the appropriate system needs. Therefore, the solutions we present in this dissertation, of garbage identification and adaptive resource utilization, may need to be incorporated into the system's resource management policies. A future research direction involves exploring possible ways to combine these methods with a global resource management solution.

REFERENCES

[1] Abdelzaher, T. F. and Shin, K. G., "End-host architecture for qos-adaptive communication," in *IEEE Real Time Technology and Applications Symposium*, pp. 121–130, 1998.

[2] Ackermann, P., "Direct manipulation of temporal structures in a multimedia application framework.," in *ACM Multimedia*, (New York, NY, USA), pp. 51–58, ACM Press, 1994.

[3] Bavier, A., Peterson, L., and Mosberger, D., "Bert: A scheduler for best effort and realtime tasks," Tech. Rep. TR-587-98, Princeton University, August 1998.

[4] Bryant, R. E., "Simulation of Packet Communication Architecture Computer Systems," Tech. Rep. MIT-LCS-TR-188, M.I.T, Cambridge, MA, 1977.

[5] Carriero, N. and Gelernter, D., "A computational model of everything," *Communications of the ACM*, vol. 44, pp. 77–81, November 2001.

[6] Cen, S., *A Software Feedback Toolkit and its Applications in Adaptive Multimedia Systems*. PhD dissertation, Oregon Graduate Institute of Computer Science and Engineering, Aug. 1997.

[7] Cen, S., Pu, C., Staehli, R., Cowan, C., and Walpole, J., "A distributed real-time mpeg video audio player," in *Proceedings of the 1995 International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, pp. 151–162, April 1995.

[8] Chandy, K. and Misra, J., "Asynchronous distributed simulation via a sequence of parallel computation," *Communications of the ACM*, vol. 24, pp. 198–206, 1981.

[9] Fujimoto, R. M., "Parallel and distributed simulation," in *Winter Simulation Conference*, pp. 118–125, December 1995.

[10] Fujimoto, R. M., *Parallel and Distributed Simulation Systems*. Wiley Interscience, January 2000.

[11] Gelernter, D., "Generative communication in Linda," *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 1, pp. 80–112, 1985.

[12] Goel, A., Steere, D., Pu, C., and Walpole, J., "Adaptive resource management via modular feedback control," Tech. Rep. CSE-99-03, Oregon Graduate Institute of Computer Science and Engineering, January 1999.

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

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

[15] IEEE, *1003.0-1995 IEEE Guide to the POSIX Open System Environment (OSE) (Identical to ISO/IEC TR 14252)*. New York, NY, USA: IEEE, 1995.

[16] Jefferson, D. R., "Virtual time," *ACM Transactions on Programming Languages and Systems*, vol. 7, pp. 404–425, July 1985.

[17] Jones, M. B., Leach, P. J., Draves, R. P., and Barrera, J. S., "Modular real-time resource management in the Rialto operating system," pp. 12–17, 1995.

[18] Jones, M. B., Rosu, D., and Rosu, M.-C., "Cpu reservations and time constraints: efficient, predictable scheduling of independent activities," in *Proceedings of the sixteenth ACM symposium on Operating systems principles*, (New York, NY, USA), pp. 198–211, ACM Press, 1997.

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

[20] Knobe, K. and Offner, C., "Compiling to tstreams: A new model of parallel computation," Tech. Rep. HP-2005-138, HP Laboratories, Cambridge Research Lab, August 2005.

[21] Knobe, K., Rehg, J. M., Chauhan, A., Nikhil, R. S., and Ramachandran, U., "Dynamic task and data parallelism using space-time memory." Submitted to 1999 Conference on Principles and Practice of Parallel Programming.

[22] Lee, C., Lehoczky, J., Rajkumar, R., and Siewiorek, D., "On quality of service optimization with discrete qos options," in *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, IEEE, June 1998.

[23] Lee, C., Lehoczky, J., Siewiorek, D., Rajkumar, R., and Hansen, J., "A scalable solution to the multi-resource qos problem," *In Proceedings of the IEEE Real-Time Systems Symposium*, 1999.

[24] Lee, C., Rajkumar, R., and Mercer, C., "Experiences with processor reservation and dynamic qos in real-time mach," *In the proceedings of Multimedia Japan 96*, 1996.

[25] Lehman, T. J., McLaughry, S. W., and Wyckoff, P., "Tspaces: The next wave," in *Hawaii International Conference on System Sciences (HICSS-32)*, January 1999.

[26] Li, B. and Nahrstedt, K., "A control theoretical model for quality of service adaptations," in *IEEE Sixth International Workshop on Quality of Service*, pp. 145–153, 1998.

[27] Liu, C. L. and Layland, J. W., "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[28] Lu, C., Stankovic, J. A., Tao, G., and Son, S. H., "Design and evaluation of a feedback control edf scheduling algorithm," in *Proceedings of the 20th Real-Time Systems Symposium (RTSS)*, December 1999.

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

[30] Mandviwala, H. A., Harel, N., Ramachandran, U., and Knobe, K., "Adaptive resource utilization via feedback control for streaming applications," in *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Papers*, (Washington, DC, USA), p. 69.1, IEEE Computer Society, 2005.

[31] Massalin, H. and Pu, C., "Fine-grain adaptive scheduling using feedback," *Computing Systems*, vol. 3, pp. 139–173, Winter 1990.

[32] Massalin, H. and Pu, C., "Fine-grain adaptive scheduling using feedback," *Computing Systems*, vol. 3, no. 1, pp. 139–173, 1990.

[33] Mourlas, C., "A framework for creating and playing distributed multimedia information systems with qos requirements," in *Proceedings of the 2000 ACM symposium on Applied computing*, (New York, NY, USA), pp. 598–600, ACM Press, 2000.

[34] Mynatt, E. D., Essa, I., and Rogers, W., "Increasing the opportunities for aging in place," in *CUU '00: Proceedings on the 2000 conference on Universal Usability*, (New York, NY, USA), pp. 65–71, ACM Press, 2000.

[35] Nieh, J. and Lam, M. S., "A smart scheduler for multimedia applications," *ACM Transactions on Computer Systems (TOCS)*, vol. 21, no. 2, pp. 117–163, 2003.

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

[37] Nikhil, R. S., Ramachandran, U., Rehg, J. M., Halstead, Jr., R. H., Joerg, C. F., and Kontothanassis, L., "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.

[38] Nikhil, R. S. and Panariti, D., "*CLF*: A common Cluster Language Framework for Parallel Cluster-based Programming Languages," Tech. Rep. (forthcoming), Digital Equipment Corporation, Cambridge Research Laboratory, 1998.

[39] Nikolopoulos, D. S. and Polychronopoulos, C. D., "Adaptive scheduling under memory constraints on non-dedicated computational farms," *Future Geneation Computer Systems*, vol. 19, no. 4, pp. 505–519, 2003.

[40] Paul, A., Harel, N., Adhikari, S., Agarwalla, B., Ramachandran, U., and Mackenzie, K., "Performance study of a cluster runtime system for dynamic interactive stream-oriented applications," in *IEEE International Symposium on Performance Analysis of Systems and Software*, (Austin, TX), March 2003.

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

[42] Pu, C., Massalin, H., and Loannidis, J., "The synthesis kernel," *Computing Systems*, vol. 1, pp. 11–32, Winter 1989.

[43] Rajkumar, R., Lee, C., Lehoczky, J., and Siewiorek, D., "A resource allocation model for qos management," *In Proceedings of the IEEE Real-Time Systems Symposium*, 1997.

[44] Ramachandran, U., Knobe, K., Harel, N., and Mandviwala, H. A., "Distributed garbage collection algorithms for timestamped data," *IEEE Transactions on Parallel and Distributed Systems*, vol. 17, pp. 1057–1071, October 2006.

[45] Ramachandran, U., Nikhil, R., Rehg, J. M., Angelov, Y., Adhikari, S., Mackenzie, K., Harel, N., and Knobe, K., "Stampede: A cluster programming middleware for interactive stream-oriented applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, pp. 1140–1154, November 2003.

[46] Ramachandran, U., Nikhil, R. S., Harel, N., Rehg, J. M., and Knobe, K., "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.

[47] Rehg, J. M., Loughlin, M., and Waters, K., "Vision for a Smart Kiosk," in *Computer Vision and Pattern Recognition*, (San Juan, Puerto Rico), pp. 690–696, June 17–19 1997.

[48] Rehg, J. M., Ramachandran, U., Halstead, Jr., R. H., Joerg, C., Kontothanassis, L., and Nikhil, R. S., "Space-time memory: A parallel programming abstraction for dynamic vision applications," Tech. Rep. CRL 97/2, Digital Equipment Corp., Cambridge Research Lab., April 1997.

[49] Revel, D., McNamee, D., Pu, C., Steere, D., and Walpole, J., "Feedback-based dynamic proportion allocation for disk i/o," Tech. Rep. CSE-99-001, Oregon Graduate Institute of Computer Science and Engineering, January 1999.

[50] Steere, D. C., Goel, A., Gruenberg, J., McNamee, D., Pu, C., and Walpole, J., "A feedback-driven proportion allocator for real-rate scheduling.," in *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, (Monterey, CA), pp. 145–158, USENIX Association, Feb. 1999.

[51] Stoica, I., Abdel-Wahab, H., and Jeffay, K., "Duality between resource reservation and proportional share resource allocation," in *SPIE Proceedings of Multimedia Computing and Networking*, vol. 3020, (Bellingham, WA), pp. 207–214, SPIE, February 1997.

[52] Sun Microsystems, *JavaSpaces Service Specifications, version 1.2.1.* Palo Alto, California, USA: Sun Microsystems, April 2002.

[53] Sundaram, V., Chandra, A., Goyal, P., Shenoy, P. J., Sahni, J., and Vin, H. M., "Application performance in the qlinux multimedia operating system.," in *ACM Multimedia*, (New York, NY, USA), pp. 127–136, ACM Press, 2000.

[54] Waldspurger, C. A. and Weoihl, W. E., "Lottery scheduling: Flexible proportional share resource management," in *Proceedings of the First Symposium on Operating Systems Design and Implementation*, (Monterey, CA), pp. 1–11, USENIX Association, Nov. 1994.

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

[56] Xi, H., "Dead code elimination through dependent types," in *PADL '99: Proceedings of the First International Workshop on Practical Aspects of Declarative Languages*, (London, UK), pp. 228–242, Springer-Verlag, 1998.

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

[58] Zhao, W., Ramamritham, K., and Stankovic, J. A., "Preemptive scheduling under time and resource constraints," *IEEE Transactions on Computers*, vol. C-36, pp. 949–960, August 1987.