Performance Study of a Cluster Runtime System for Dynamic Interactive Stream-oriented Applications [*]

Arnab Paul, Nissim Harel, Sameer Adhikari,
Bikash Agarwalla, Umakishore Ramachandran, Ken Mackenzie
College Of Computing, Georgia Institute of Technology
801 Atlantic Drive, NW, Atlanta, GA 30332-0280, USA
arnab, nissim, sameera, bikash, rama, kenmac@cc.gatech.edu

Abstract

Emerging application domains such as interactive vision, animation, and multimedia collaboration display dynamic scalable parallelism, and high computational requirements, making them good candidates for executing on parallel architectures such as SMPs or clusters of SMPs. Apart from their main algorithmic components, these applications need specialized support mechanisms that enable plumbing different modules together, cross module data transfer, automatic buffer management, synchronization and so on. Such support mechanisms are usually part of the runtime system, and in this paper we quantify their performance. The runtime for our evaluation is Stampede, a cluster programming system that is designed to meet the requirements of such applications. We have developed a cycle accurate timing infrastructure that helps tease out the time spent by an application in different layers of software, viz., the main algorithmic component, the support mechanisms, and the raw messaging. We conducted our experiments with two representative applications on two flavors of Unix, viz., Solaris and Linux. There are several interesting insights coming from this study. First, memory allocation does not take up a significant amount of the execution time despite the interactive and dynamic nature of the application domain. Second, the Stampede runtime adds a minimal overhead over raw messaging for structuring such applications. Third, the results suggest that the thread scheduler on Linux may be more responsive than the one on Solaris; we quantify the effect of the scheduler responsiveness in terms of difference in blocking time for threads and time spent in synchronization in the messaging layer. Fourth, the messaging layer spends quite a bit of time in synchronization operations. Perhaps the most interesting result of this study is that general-purpose operating systems such as Linux and Solaris are quite adequate to meet the requirements of emerging dynamic interactive stream-oriented applications.

1 Introduction

Emerging applications in the domain of pervasive computing are naturally physically distributed, interactive, and dynamic. Vision, animation, multimedia collaboration are examples of this class. Such applications are continually running and are both computation and data intensive. They typically consist of a distributed computational pipeline dealing with high volume streaming data resulting in heavy network demands. Structure of such an application includes (i) *Application logic*, i.e., modules delivering the primary intent of the application, such as a tracking algorithm for vision, and (ii) *Support mechanisms* that enable facilities such as plumbing, cross module data transport, buffer management and synchronization between the modules. Clearly, the overall performance of such an application depends on both components. While application logic remains an area specific concern, the performance of the support mechanisms is in the purview of the systems architect. In this paper we ask the question: *How good is the performance of the system mechanisms for supporting these emerging applications?*
There are several novel characteristics of this application domain that make it challenging from the point of view of the support mechanisms:

Temporal evolution of Data: Applications usually generate continuously evolving data, such as streaming video that are often timestamped.

[*]The work has been funded in part by an NSF ITR grant CCR-01-21638, NSF grant CCR-99-72216, Compaq Cambridge Research Lab, the Yamacraw project of the State of Georgia, and the Georgia Tech Broadband Institute. The equipment used in the experimental studies is funded in part by an NSF Research Infrastructure award EIA-99-72872, and Intel Corp. TE2000 grant.

Soft Real time requirements: Most often there is a soft threshold for delivery rate. For example, although the requirement for standard human perception for continuity of motion is about 30 discrete frames per second, a slightly lower framerate does not make a significant difference for viewing. In order to maintain such requirements and freshness of the output, it may be required to skip frames in a streaming application.

Dynamic Producer-consumer relationship: The number of consumers for data items may vary over time. For example, parties can join and leave a distributed meeting. That makes the inter-module connection topology and their producer consumer relationships quite dynamic.

Parallel/distributed nature: Most applications involve pipelined task structure and thus are suitable for running on parallel machines. Moreover, as the bandwidth bottleneck is going away, distributed information capture and processing sources are being integrated into single application bodies and thus engendering *inherently distributed* computing.

Complex buffer management: As a result of all the previous factors, efficient runtime buffer management becomes critical. Support mechanisms should reclaim all the unused buffer space efficiently, as the typical volume of data involved is quite high.

Dissecting the support mechanisms should help understand the main bottlenecks outside the application logic and in turn help tune the overall performance. The intent in this work is to understand how much time an application spends carrying out tasks in the support mechanisms. Furthermore, we want to split the time spent in the support mechanisms into components such as the *(i) messaging layer, (ii) waiting for delivering or arrival of data, (iii) non-messaging activities outside the application logic, such as data transport between messaging and the application logic, buffer management, and synchronization.*

We conduct this study by exploring two representative applications. We have chosen a runtime system called Stampede [8] as the implementation base for these applications. The Stampede programming model provides the characteristics identified above and makes it an good choice for this study. Stampede provides a novel system-wide data abstraction called *Space-Time Memory* (STM) [11] to enable distributed interactive multimedia applications to manage a collection of time-sequenced data items simply, efficiently, and transparently. STM decouples the application programmer from low level details by providing a high level interface that subsumes buffer management, inter-thread synchronization, and location transparency for data produced and accessed anywhere in the distributed system. Stampede has been in use for application development by various groups such as systems, vision and robotics, at CRL/HP and Georgia Tech. Sample applications include, Smart Kiosk, Color Tracker [14], a distributed programming infrastructure called D-Stampede [16], distributed video meetings, video texture generation [15], a novel Data fusion library [17], and a Distributed Media broker architecture. Thus Stampede can be considered as a representative of the kind of system support that is needed for applications in this class.

Our query therefore reduces to obtaining a cross-section of time spent inside the Stampede runtime. Stampede in turn is built on top of a messaging layer called CLF (cluster language framework) [18]. We isolate the time spent by the applications inside the messaging layer from pure in-line traversal of the Stampede code. We further tease out the times spent in dynamic memory allocation (such as mallocs and frees), and synchronization operations (such as locks and unlocks) for the runtime and messaging layers. These are the points of interaction between the OS and support mechanisms and hence are of real interest to us.

Scope of this study

We consider two novel interactive applications in this paper. They have inherent parallelism and are built on the cluster programming environment provided by Stampede. We have developed an elaborate, cycle accurate, event-logging measurement infrastructure to quantify the interaction between the Stampede runtime and the operating system. Since the events are distributed across multiple machines the main challenge is reconciliation of the times spent on the different nodes in various activities of interest to derive the inferences of interest for this study. We conduct our studies on two different operating systems, *viz.*, RedHat Linux 7.1 and Solaris 7 (for x86), under identical hardware setup, and generate a comparative analysis.

For a given execution of a Stampede application we quantify the time spent in each of *application logic*, blocking, *Stampede runtime*, and *messaging layer*. The first component is the actual work done by the application. The second component is the wait time incurred in the application possibly due to work imbalance in the structure of the application that may lead to non-availability of data item for some application thread. The third is the time incurred for traversal of the Stampede API code path, while the fourth is the time incurred for data transport across machines. Such a breakdown can yield valuable insights on both the applications themselves as well as on the characteristics of complex runtime libraries such as Stampede.

For each of these components we are interested in determining the time spent in *memory activities* and *synchronization*. The memory demands on the systems are driven by the application dynamics (e.g. how much activity in front of a video kiosk). There are usually multiple consumers for a single data item, and depending upon their unpredictable and dynamic needs, memory allocation and deallocation can be quite different over different runs of the same application. Further, since the applications show inherent parallelism, they are good candidates for SMPs and cluster of SMPs, and the runtime support should also be written to exploit the hardware parallelism. However, parallelizing the supporting infrastructure could incur additional synchronization costs. Hence the interest in getting a finer breakdown of the component times into memory activity and synchronization times.

Results Summary

One fairly interesting and perhaps comforting conclusion from this study is that general-purpose operating systems such as Linux and Solaris seem quite well-equipped to meet the requirements of emerging dynamic streaming applications. The results of the study carried out in this paper can be qualitatively summarized as follows: i) messaging/scheduling differences in the operating systems may lead to differences in thread blocking times in the execution profile, (ii) despite the interactive and dynamic nature of the applications, a relatively negligible amount of time is spent in dynamic memory activities, (iii) a significant amount of time is spent in synchronization operations in the messaging layer, (iv) the support mechanisms for the applications considered need not be expensive; this is borne out by the fact that the Stampede runtime itself does not contribute to a significant share of the total execution time, (v) the time spent in the messaging layer is approximately double the time taken by the runtime (Stampede) layer, and (vi) Stampede is a fairly scalable system, since the percentage of time spent in this layer remains constant as the data size is scaled up.

We start with an overview of the Stampede System in Section 2. Next, we discuss the cycle accurate timing infrastructure that we have built for this study in section 3. In Section 4 two representative applications are discussed. Section 5 presents a detailed performance analysis for these applications developed on top of Stampede, the focus being a cross section of total time spent in different software layers. Related work is discussed in Section 6, and we conclude in Section 7.

2 Stampede Programming System

As we discussed in the previous section, Stampede system provides an infrastructure for building distributed streaming applications on clusters. The computational model supported by Stampede is shown by the thread-channel graph in Figure 1. The threads can run on any node of the cluster and the channels serve as the application level conduits for time-sequenced stream data among

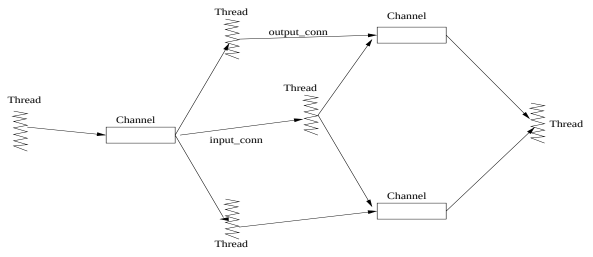

Figure 1. Computational Model: Dynamic Thread Channel Graph

the threads. The Stampede architecture has the following main components: *Channels*, *Threads*, *Garbage Collection*, *Handler Functions*, and *Real-time Guarantees*.

Threads and Channels Stampede provides a set of abstractions across the entire cluster: threads and channels. Stampede threads can be created in different protection domains (address spaces) [8]. Channels are cluster-wide unique names, and serve as containers for *time-sequenced* data. They allow inter-thread communication and synchronization regardless of the physical location of the threads and channels. A thread (dynamically) connects to a channel for *input* and/or *output*. Once connected, a thread can do I/O (get/put items) on the channel. The items represent some application-defined stream data (e.g. video frames). The timestamp associated with an item in a channel is user defined (e.g. frame numbers in a video sequence). The collection of time-sequenced data in the channels is referred to as *space-time memory* (STM) [11]. At the application level, the Stampede threads perform I/O on the channels, and do not deal with any low level synchronization (such as locks or barriers) among themselves. The put/get operations implicitly combine synchronization with data transfer. These operations are synchronous and atomic with respect to the specific channel. The operations themselves can be blocking or non-blocking.

Garbage Collection: API calls in Stampede allow a given thread to indicate that an item (or a set of items) in a channel is garbage so far as it is concerned. Using this per-thread knowledge, Stampede automatically performs distributed garbage collection [7] of timestamps (i.e. items with such timestamps) that are of no interest to any thread in the Stampede computation.

Handler Functions: Stampede allows association of *handler functions* with channels for applying a user-defined function on an item in a channel. The handler functions are useful for transporting complex data structures (as marshalling/unmarshalling routines) or in handling garbage in a user defined way.

Real-time Guarantees: The timestamp associated with an item is an indexing system for data items. For pacing a thread relative to real time, Stampede provides an API borrowed from the Beehive system [13]. Essentially, a thread can declare real time interval at which it will re-synchronize with real time, along with a tolerance and an exception handler.

Stampede is implemented as a C runtime library on top of several clustered SMP platforms including DEC Alpha-Digital Unix 4.0 (Compaq Tru64 Unix), x86-Linux, x86-Solaris, and x86-NT. It uses a message-passing substrate called CLF, a low level packet transport layer developed originally at Digital Equipment Corporation's Cambridge Research Lab (which has since become Compaq/HP CRL). CLF provides reliable, ordered, point-to-point packet transport between Stampede address spaces, with the illusion of an infinite packet queue. It exploits shared memory within an SMP, and any available cluster interconnect between the SMPs, including Digital Memory Channel [5], Myrinet [3] (using the Myricom GM library), and Gigabit Ethernet (using UDP).

3 Measurement Infrastructure

To perform such a study, we have developed an elaborate measurement infrastructure in Stampede. This infrastructure consists of two parts: *event logging*, and *postmortem analysis*.

3.1 Event Logging

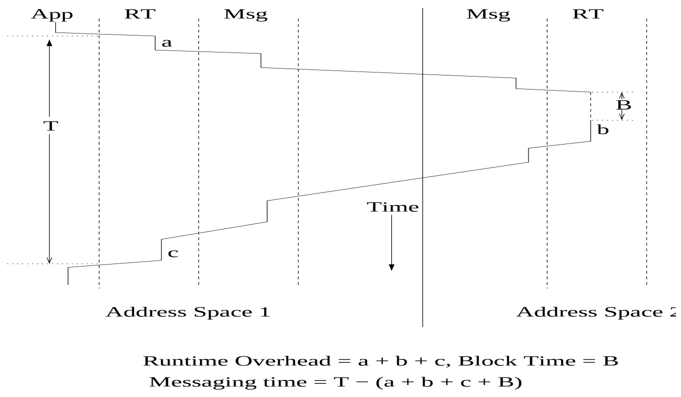
App
RT
Msg
Msg
RT
a
B
T
b
Time
c
Address Space 1
Address Space 2
Runtime Overhead = a + b + c, Block Time = B
Messaging time = T − (a + b + c + B)

Figure 2. Timeline for a remote operation spanning two address spaces. RT - Stampede Runtime, Msg - Messaging Layer, App - Application logic.

The intent is to be able to accurately accumulate the time spent in any and every piece of code segment for the purposes of answering the questions we raised in Section 1. To this end we have implemented an event logging mechanism. The basic idea is to declare *events* that are names meaningful to the code that is being timed. For e.g., 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. The events we wish to measure are very short (as small as few tens of instructions). Also the event recording should impart least possible perturbation to the original application. The standard system supported timing calls (such as the Unix `gettimeofday`) are inappropriate because their granularity is not fine enough for our purpose, and they are quite expensive per call.

Fortunately, most modern processors such as the Intel Pentium line have a CPU cycle counter that is accessible as a program readable register. Our event logging mechanism uses this cycle counter for recording the time. The log is maintained as a buffer in main memory, and each log entry consists of the unique event name, and start and end times for that event. The time for recording the log for an event (a few memory write instructions) is assumed to be small in comparison to the code fragments that need to be timed. Further, we experimentally verified that recording these events themselves does not significantly perturb the overall execution of the original application. Although the data structures used by the measuring infrastructure can result in some cache pollution, within the limitations of a software infrastructure, cycle reading is the least intrusive method of recording these events. The logging subsystem maintains two in-memory buffers. When one buffer is approaching fullness, it switches to the other buffer, and (via DMA) flushes out the first buffer 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.

There are three interesting situations that have to be handled. First, on an SMP the start and end times for a given event could be recorded from different processors due to the vagaries of thread scheduling by the underlying operating system. Fortunately, the CPU cycle counters are initialized at boot time by the operating system and thus remain in sync modulo clock drift on the processors. Experimentally we verified that the cycle counters are in sync and off by at most a few ticks (compared to events that are several cycles long). Second, the counters should be big enough. The x86 cycle counters are 64 bits in length so there is no worry of the counter wrapping around. Third, an event of interest may cross machine boundary (for e.g. a remote `get` operation). In this case, such a "macro event" has to be broken up into smaller events, each of which can be locally timed. Note that event logging is completely local and do not take up any bandwidth. Combining logs from different machines is done postmortem and thereby has no effect on the performance.

3.2 Postmortem Analysis

A postmortem analysis program reads in the events from the log file on the disk and computes the times for macro events of interest. For e.g. Figure 2 shows an operation (such as a remote `get`) that spans 2 address spaces. The request for an item is generated in address space 1, and is actually fulfilled in address space 2. As can be seen from the figure, if the macro event of interest is the Stampede runtime overhead for this operation then it is $a + b + c$, where a and c are events recorded in address space 1, and b is an event recorded in address space 2.

4 Sample Application

We consider two applications, a Motion Detector pipeline, potentially to be used as a core for surveillance systems and a Color tracker that can track an object of a particular color of interest. Both the applications are dynamic, interactive and operate on temporally evolving data. In order to isolate the workload from the jitters of external units such as a camera, we used a fixed set of frames, and focus primarily on the support mechanisms of interest for us.

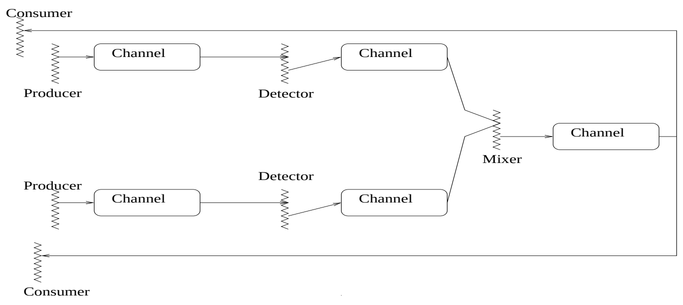

Figure 3. Mixing of Motion detector outputs.

1. Mixing of Motion Detector Outputs. Figure 3 shows the task data pipeline of this application, which basically mixes video inputs from a number of clients and sends them back out to each client. A client is composed of a producer-consumer pair. Each `producer` thread captures images from a camera connected to it and outputs them to a channel. A motion `detector` thread, dedicated to this producer, picks up images from the associated channel and computes a blob of motion in this image and outputs the result to another channel. A `mixer` thread picks up corresponding blob outputs from different detectors, combines them and puts the composite onto an output channel from where it is shipped to multiple `consumers` when they perform `get` operations. We experimented with different configurations of this application. Each configuration is characterized by the number of clients (producer-consumer pairs), the number of address spaces the application is spread across, and how the threads are placed in those address spaces.

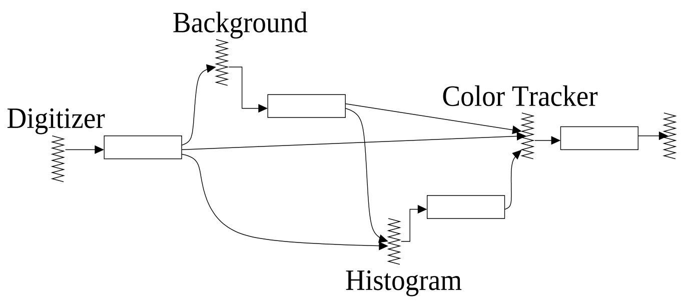

Figure 4. Color Model based Tracker

2. Color Model based Tracker. Figure 4 shows the task data pipeline of the color model based tracker. Every thread in this application has an associated output channel. A digitizer thread gets input from a camera, and produces a digitized image. The background thread uses the digitized images and does a frame by frame background subtraction. A Histogram thread creates a color histogram of the background subtracted image. The tracker uses the histogram output to look for an object given its color model. The tracker output can be optionally sent to some further stages of analysis.

5 Performance Results

We ran our experiments for two operating systems: Solaris 7 for x86, and RedHat Linux 7.1. For both we had identical hardware configuration - four node cluster interconnected by 100Mbps Fast Ethernet, with each node consisting of Dual Pentium II 300MHz processors with 512MB RAM and 4GB SCSI disk. We performed two sets of experiments. The first is a set of microbenchmarks that compares the thread scheduling performance, and the message throughput at CLF (using UDP) and Stampede levels on the two platforms. The second set consists of application measurements. The microbenchmarks are useful for explaining some of the application level results.

5.1 Microbenchmarks

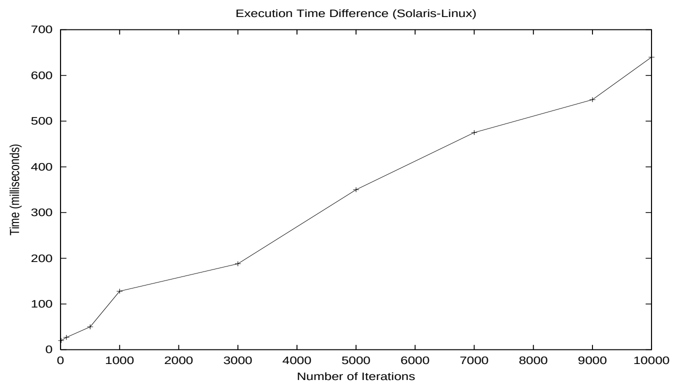

Figure 5. Difference in execution time between Solaris and Linux due to scheduling

Thread Scheduling Performance. We perform an experiment to compare responsiveness of the thread scheduler on Linux and Solaris. Recall that the hardware platform consists of dual processor nodes. Three threads T_1, T_2, T_3 are spawned on one node. T_1 and T_2 iterate over a simple computation such as adding two integers continuously. T_3 alternates between sleeping for a while and doing the same computation (as the other threads) for a while (quantified as number of iterations in a loop). We consistently found that the total execution time (when T_3 finishes a certain number of iterations) is more on Solaris and the disparity steadily increases with the total number of iterations in T_3. Since the hardware is identical, the logical conclusion is that the short term scheduler on Linux is more responsive [1] for this kind of start-stop behavior. In Figure 5 we plot the *difference* between the execution times on Solaris and Linux, respectively. The difference starts at about 20ms for 10 iterations and gets as large as 600ms for 10000 iterations.

Messaging Performance. Figure 6 shows the throughput for Stampede *put* operation (across nodes) for different data sizes on Linux compared to the throughput for CLF ping-pong test for the same message size. As can be seen the overhead for using the higher level abstractions in Stampede compared to raw messaging is pretty minimal. For example, with a message size of 4KB, the Stampede throughput is 4.5 MBps compared to 5.2 MBps offered by CLF. The results obtained for Solaris are quite similar. Further, there is little difference in the CLF messaging performance on the two platforms.

[1] By responsive we mean how quickly the OS allocates the processor to a different thread when the current thread goes to sleep and how quickly an awakened thread is scheduled on an available processor

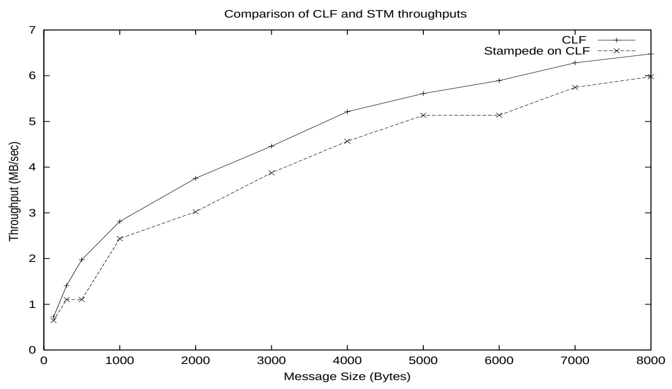

Figure 6. Comparison of CLF and Stampede throughput on Linux

5.2 Application Level Performance

Overall Performance. The overall performance of both the applications on the two platforms (x86-Solaris and x86-Linux) are roughly the same. The motion detector application gives a frame rate of 14 fps for one client (image size 70 KB), while the color tracker gives a frame rate of 13 fps (image size 75 KB). The reader might feel that these frame rates are on the lower side. This is a function of the hardware used for these experiments; our choice was governed by the requirement of hardware uniformity for both operating systems. As a point of reference we also ran the two applications on a cluster (17 nodes) interconnected by 1.2 Gigabit Ethernet, in which each node consists of eight 550 MHz Pentium III Xeon processors with 4GB RAM and 18GB SCSI disks, and running RedHat Linux 7.1. The motion detector application gives a frame rate of 36 fps for one client (image size 70 KB), while the color tracker gives a frame rate of 20 fps (image size 75KB) without ever missing the soft real time deadline placed on it. These results show that these applications are well suited to clusters and that Stampede abstractions support these applications quite well.

5.3 Breakdown of Execution Time

As outlined in Section 1, we now examine the breakdown of the execution time into the different components such as application logic, blocking, Stampede runtime overhead, and messaging for the two platforms. For the motion detector we use a single client (producer-consumer pair). The image size is varied from 13 KB to 69 KB. The client

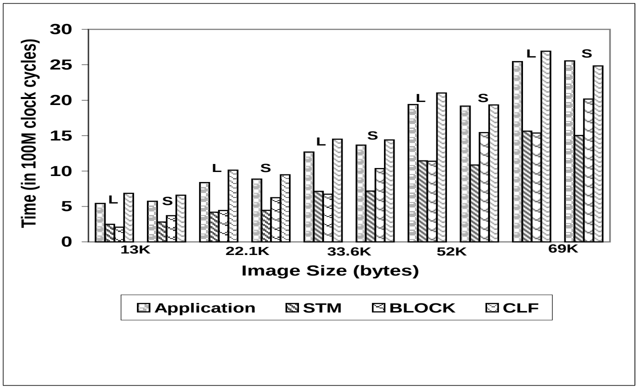

Figure 7. Motion Detector: Execution time split into different layers (L = Linux, S = Solaris)

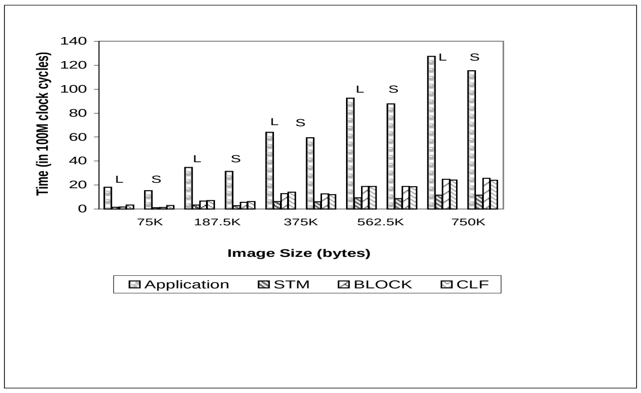

Figure 8. Color Tracker: Execution time split into different layers (L = Linux, S = Solaris)

(producer-consumer) pair is on one address space, the mixer on a second address space, and the detector on a third address space. The channels are created in the address space of the thread that puts items into the channel. For the color tracker the image size is varied from 75 KB to 750 KB. Every thread in the color tracker application is in an address space by itself. Once again the channels are co-located with the thread that puts items into the channel. For all the experiments, each address space is on a different node of the cluster.

Messaging and Scheduling Comparison. Figure 7 shows the breakdown of execution time for the motion detector. For each image size, respective breakdown on Solaris and Linux are juxtaposed. Figure 8 shows similar breakdown for the color tracker. From Figure 7 it can be seen that the messaging time is always a little lower in Solaris. However, although the application logic and Stampede overhead (labeled STM in the graph) are almost same on both platforms, the blocking time is always significantly more on Solaris than in Linux. This is counter-intuitive. Blocking in the Stampede pipeline is due to waiting for an item in a channel (on a get), or waiting for space on a channel (on a put). The faster the messaging (assuming a constant time in the application or STM), faster the items will come in and be taken away from the channels and one would therefore expect less blocking time. Our experimental results however show otherwise. Results in Figure 8 are also along the same line, though the effect is less prominent because application work is much more compared to other components.

We ascribe this anomaly to the scheduling policy of the operating system. When a thread blocks waiting for an item to become available, it is context switched with other active threads. Blocking is minimized if the thread is scheduled as soon as the desired item becomes available (or taken away). Thus the responsiveness of the scheduler is a determinant for the observed blocking time because it adds more cycles to the blocking event of the same thread. This conjecture is strengthened by the microbenchmark result in Sec. 5.1 which shows that the scheduling on Linux is more responsive than on Solaris.

Time Spent in Application Logic. From Figure 8, it is interesting to note that the time spent in the Application Logic for the color tracker is always slightly higher for Linux compared to Solaris. Since the hardware platforms are identical and both use the gcc compiler, we ascribe this to implementation differences in the compiler support libraries.

Stampede Runtime Overhead. In Figures 7 and 8, the bars labeled STM is the Stampede runtime overhead. Table 1 shows the percentage of time spent in the Stampede runtime layer for the two applications. The experiments are labeled as 1 to 5 (this corresponds to image sizes 13, 22, 33.6, 52 and 69 Kilobytes for the motion detector, and 75, 187, 375, 562.5, and 750 Kilobytes for the color tracker). As can be seen for each application (on both platforms) the Stampede runtime overhead is roughly a fixed percentage of the execution time, showing the scalability of the Stampede runtime system.

Stampede Runtime Vs. the Messaging Layer. In all the experiments for both the applications, the time spent in Stampede runtime is approximately half the time spent in messaging layer. For example, in Figure 7 (Motion detector), for a data size of 52KB, the time taken in the runtime layer is 12 million clock cycles as opposed to 22 million clock cycles taken by the messaging layer.

App/Platform	Experiments				
	1	2	3	4	5
MD(Linux)	14.77	15.0	17	18	18.7
MD (Solaris)	14.9	15.31	16.7	16.7	17.5
CT (Linux)	5.8	6.3	6.4	6.7	6.14
CT (Solaris)	5.1	6.0	6.6	6.5	6.5

Table 1. Stampede Runtime Overhead: Percentage of the total time spent in STM. MD = Motion Detector, CT = Color Tracker

5.4 Memory and Synchronization Breakdown

This subsection answers the other question raised in Section 1, namely, how much time is spent by the Stampede runtime and CLF messaging in dynamic memory allocation and synchronization related activities. Clearly these are points of interaction with the underlying operating system. The reported times are indeed the OS overheads for memory and synchronization related activities in those two layers. Due to the dynamic nature of both the applications, and the large sizes of the data being manipulated, it may be expected that a significant amount of time is spent in dynamic memory allocation operations (such as malloc and free).

Motion Detector					
Platform	Experiments				
	1	2	3	4	5
Linux	3.2	3.2	2.4	2.1	2.1
Solaris	6.7	5.1	3.9	3.8	3.3
Color Tracker					
Platform	Experiments				
	1	2	3	4	5
Linux	4.0	2.9	1.9	1.8	1.5
Solaris	5.9	2.0	1.2	1.0	0.7

Table 2. Percentage of Stampede Runtime Spent in Memory allocation/free.

Table 2 shows the percentage of Stampede runtime spent in memory allocation and freeing. It is interesting to observe that this percentage is quite small. For the Motion Detector, the memory activity is at most 3.2% on Linux and at most 6.7% on Solaris. The numbers for the Color Tracker are similar. We investigated the runtime's thread synchronization activity as well. Table 3 summarizes the percentage of time Stampede runtime spends in thread synchronization for the two applications. Once again it is interesting to note that despite the nature of these applications, relatively little time is spent in thread synchronization activity (e.g., 1.2% on Linux and 6.8% on Solaris for the Motion detector, Experiment 1). On Solaris, the relative time spent in thread synchronization is slightly higher than on Linux.

Motion Detector					
Platform	Experiments				
	1	2	3	4	5
Linux	1.2	1.6	2.4	2.0	2.0
Solaris	6.8	7	5.9	6.2	4.8
Color Tracker					
Platform	Experiments				
	1	2	3	4	5
Linux	1.2	2.5	2.6	2.1	4.1
Solaris	1.7	2.5	2.4	1.5	2.7

Table 3. Percentage of Stampede Runtime spent in Thread Synchronization.

CLF's interaction with the memory allocator (of the OS) is negligible. CLF does a one time allocation of a sufficiently big memory region and thenceforth does its own buffer management. Table 4 summarizes CLF's thread synchronization time relative to total time spent in CLF. There are two interesting observations:
First, the messaging layer spends a significant share of its time in synchronization. For the Color Tracker, this share accounts for approximately 40%-42% on Solaris and 80%-90% on Linux. This can be attributed to the design of the messaging layer. CLF provides reliable messaging on top of the unreliable UDP layer. CLF does its own dynamic memory management. Thus synchronization is needed among threads in the same address space for memory allocation and deallocation. While the application logic is in execution, the threads in the messaging layer remain involved in polling (and hence synchronization activities) at regular intervals throughout this time. For the Color Tracker, application logic time is quite big compared to other segments. As a result, within the messaging layer the synchronization activities occupy a major share.
Second, The time spent in thread synchronization on Linux is higher than on Solaris for CLF messaging. Consider the Motion detector. CLF spends 18%-20% in synchronization on Linux, as opposed to 10%-12% on Solaris. For the Color Tracker, the difference is more prominent. On Linux, it ranges from 80% to 90%, while on Solaris it ranges from 41% to 43%. This is again attributed to the scheduling differences between the two operating systems. On Linux, as we have already observed, it takes relatively less time to allocate processor to a thread that becomes *ready* after a blocked phase. The polling thread in the messaging layer,

after accessing a shared mutex, blocks on network I/O for a small amount of time. Once unblocked, it releases the lock and loops through the same activities. Unless the processor is already occupied, this unblocking thread is immediately scheduled on the processor [2]. On Linux, this happens faster than on Solaris, as we have already established. Hence, the number of lock/unlock on Linux is more than on Solaris, resulting in the difference in the percentage split.

Motion Detector					
Platform	Experiments				
	1	2	3	4	5
Linux	18.1	18.7	19.0	19.6	20.3
Solaris	10.9	10.9	11.5	12.0	11.4

Color Tracker					
Platform	Experiments				
	1	2	3	4	5
Linux	89.9	89.9	83.4	89.9	80.1
Solaris	41.9	42.0	42.4	41.5	40.9

Table 4. Percentage of CLF Messaging spent in Thread Synchronization.

6 Related Work

We are not aware of any study that has exactly the same goals as ours of analyzing and quantifying the performance of runtime support for the emerging class of applications that we considered. There are a number of studies that have looked at providing measurement support. ProfileMe system [4] uses program counter (PC) sampling to relate the hardware events (such as cache misses) to the likely instructions that incurred such events. They designed hardware support that allows accruing accurate profile information from out-of-order processors. Continuous profiling [2] is a system that periodically samples the PC on an Alpha SMP, and stores all the hardware counter values at the time of sampling. Periodically these images are written out to disk. Their system uses 1% to 3% of CPU time, modest amount of memory and disk. Profile information is gathered for the entire system allowing them to pinpoint sources of performance problems. Ofelt and Hennessy [9] describe a technique that involves an instrumentation phase followed by an analysis phase to predict the performance of modern processors. Though their goal is different there is some similarity in our measurement infrastructure to their approach. Ammons et al. [1] describe a technique that uses hardware performance counters to predict the hot paths in traditional benchmarks. NetLogger toolkit [19] that helps generating monitoring interface for distributed systems through precision event logs, is similar to the measurement infrastructure that we have developed. Low overhead is a significant concern for any concurrent trace collection subsystem. In [21], Yang *et al.* present a cycle accurate thread timer for Linux, that can be used as a low overhead event logging support. There is a large body of work that deals with performance characterization of aspects of computer system (memory hierarchy, pipeline performance, etc.) from the point of view of specific applications (such as multimedia [12]) or specific programming environments (such as Java [6]), or specific system techniques (such as scheduling [22, 20]). Similarly architectural aspects have been evaluated against various kinds of workloads (such as transaction processing) with the help of hardware counters [23].

[2]The polling thread behaves like thread T_3 that alternates between sleeping and computing, as discussed under Microbenchmarking results

7 Concluding Remarks

Stampede is a cluster programming library with novel features for supporting emerging interactive stream-oriented applications. This research has attempted to quantitatively present the cross section of total time spent by the Stampede applications in different software layers. In order to do this, an event logging facility has been implemented using the CPU cycle counter found in most modern processors. In addition to giving a breakdown of execution times for two video streaming applications, this study also investigates how this breakdown compares for Solaris and Linux on identical hardware configurations. Further, the time spent in performing dynamic memory allocation and synchronization operations both in the Stampede runtime as well as in the messaging layer is also quantified. One fairly interesting and perhaps comforting conclusion from this study is that general-purpose operating systems such as Linux and Solaris seem quite well-equipped to meet the requirements of emerging dynamic streaming applications. An immediate future work that can follow from this study is identifying the performance bottlenecks in the runtime system under investigation and tune its performance.

References

[1] G. Ammons, T. Ball, and J. Larus Exploiting Hardware Performance counters with Flow and Context Sensitive Profiling. In *Proc. ACM SIGPLAN Programming Language Design and Implementation (PLDI)*, 1997

[2] J. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl Continuous profiling: Where have all the cycles gone?. In*Proc. of the*

16th ACM Symposium of Operating Systems Principles (SOSP 97), pages 1-14, October 1997

[3] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet – a gigabit-per-second local-area network, Draft 16 Nov 1994.

[4] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Chrysos ProfileMe: Hardware Support for Instruction-Level Profiling on Out-of-Order Processors. In *Proc. 30th Annual IEEE/ACM Intl. Symp. on Microarchitecture, Research Triangle Park, NC*, pages 292-302, 1997

[5] R. Gillett. MEMORY CHANNEL Network for PCI: An Optimized Cluster Interconnect. *IEEE Micro*, pages 12–18, February 1996.

[6] J. Kim, and Y. Hsu Memory System Behavior of Jave Programs: Methodology and Analysis In *Proc. of the International Conference on Measurements and Modeling of Computer Systems, Santa Clara, CA*, pages 264-274, 2000

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

[8] R. S. Nikhil, U. Ramachandran, J. M. Rehg, R. H. Halstead, Jr., C. F. Joerg, and L. Kontothanassis. Stampede: A programming system for emerging scalable interactive multimedia applications. In *Proc. Eleventh Intl. Wkshp. on Languages and Compilers for Parallel Computing (LCPC 98), Chapel Hill NC*, August 7-9 1998.

[9] D. Ofelt, and J. L. Hennessy Efficient Performance Prediction for Modern Processors. In *Proc. of the international conference on Measurements and modeling of computer systems, Santa Clara, CA*, pages 229-239, 2000

[10] IEEE. Threads standard POSIX 1003.1c-1995 (also ISO/IEC 9945-1:1996), 1996.

[11] U. Ramachandran, R. S. Nikhil, N. Harel, J. M. Rehg, and K. Knobe. Space-Time Memory: A Parallel Programming Abstraction for Interactive Multimedia Applications. In *Proc. Principles and Practice of Parallel Programming (PPoPP'99), Atlanta GA*, May 1999.

[12] S. Sohoni, R. Min, Z. Xu and Y. Hu A Study of Memory System Performance of Multimedia Applications In *Proceedings of the joint International Conference on Measurement and modeling of computer systems, Cambridge, MA*, pages 206-215, 2001

[13] A. Singla, U. Ramachandran and J. Hodgins, Temporal Notions of Synchronization and Consistency in Beehive, 9th Annual ACM SPAA, June 97.

[14] K. Knobe, J. M. Rehg, A. Chauhan, R. S. Nikhil, U. Ramachandran. Scheduling Constrained Dynamic Applications on Clusters. *Supercomputing*, December 1999.

[15] Y. Angelov, U. Ramachandran, K. Mackenzie J. Rehg, I. Essa, Optimizing Stream-oriented Applications for Clustr Execution, *Georgia Tech College of Computing Tech Report, GIT-CC-02-51*, 2002.

[16] S. Adhikari, A. Paul, U. Ramachandran. D-Stampede: Distributed Programming System for Ubiquitous Computing. In *roceedings of the 22nd International Conf. on Distributed Computing Systems*, Vienna, July 2002.

[17] B. Agarwalla, P. Hutto, A. Paul, U. Ramachandran. Fusion Channels: A Multi-sensor Data Fusion Architecture. *Georgia Tech College of Computing Tech Report, GIT-CC-02-53*, 2002.

[18] R. Nikhil. Notes on CLF, a Cluster Language Framework.
http://www2-int.cc.gatech.edu/projects/up/info/howto/clf_notes.html

[19] D. Gunter, B. Tierney, B, Crowley, M. Holding, J. Lee. Netlogger: A Toolkit for Distributed Systems Performance Analysis. *In Proceedings of IEEE Mascots*, 2000.

[20] M. Squillante, Y. Zhang, A. Sivasubramaniam, N. Gautam, H. Franke, J. Moreira. Modelling and Analysis of Dynamic Coscheduling in Parallel and Distributed Environments. *In Proceedings of ACM SIGMETRICS*, 2002.

[21] Q. Yang, W. Srisa-an, T. Skotiniotis, J. M. Chang. Cycle accurate thread timer for Linux environment. *In Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software*, Tuscon, 2001.

[22] F. Wang, M. Papaefthymiou, M. Squillante. Performance Evaluation of Gang Scheduling for Parallel and Distributed Multiprogramming. *In Job Scheduling Strategies for Parallel Processing: Ed. D. Feitelson, L. Rudolp* pp. 277-298, 1997.

[23] K. Keeton, D. A. Patterson, Y. Q. He, R. C. Raphael, W. E. Baker. Performance Characterization of a Quad Pentium Pro SMP Using OLTP Workloads. *In Proceedings of 25th International Symposium on Computer Architecture* Barcelona, Spain, 1998.