Scheduling Constrained Dynamic Applications on Clusters

Kathleen Knobe, James M. Rehg, Arun Chauhan[*],
Rishiyur S. Nikhil, and Umakishore Ramachandran[†]
Cambridge Research Lab, Compaq Computer Corporation, Cambridge, MA 02139

Abstract

There is an emerging class of computationally demanding multimedia applications involving vision, speech and interaction with the real world (e.g., CRL's Smart Kiosk). These applications are highly parallel and require low latencies for good performance. They are well-suited for implementation on clusters of SMP's, but they require efficient scheduling of application tasks.

General purpose schedulers produce high latencies because they lack knowledge of the dependencies between tasks. Previous research in optimal scheduling has been limited to static problems. In contrast, our application is highly dynamic as the optimal schedule depends upon the behavior of the kiosk's customers.

We observe that the dynamism of our application class is constrained, in that there are a small number of operating regimes which are determined by the state of the application. We present a framework for optimal scheduling of constrained dynamic applications. The results of an experimental comparison with a hand-tuned schedule are promising.

1 Introduction

There is an emerging class of interactive multimedia applications which is computationally intensive, highly parallel and dynamic. An example is the Smart Kiosk system [11, 16, 2] under development at the Cambridge Research Laboratory (CRL) which motivates this work. A Smart Kiosk is a free-standing computerized device that interacts with multiple people in a public environment, providing information and entertainment.

We are exploring a social interface paradigm for kiosks in which vision and speech sensing provide user input while a graphical speaking agent provides the kiosk's output. Figure 1 shows a picture of a prototype Smart Kiosk. It employs vision techniques to track and identify people based on their motion and clothing color [11]. The estimated position of multiple users drives the behavior of an animated graphical face, called DECface [15]. We believe the Smart Kiosk to be representative of a broad class of emerging applications in surveillance, autonomous agents, and intelligent vehicles and rooms (see [3] for some examples.)

The computational requirements of the kiosk application are determined by the actions of the kiosk's customers, and as a result they are highly dynamic. For example, each time a person approaches the kiosk they are detected and greeted by the DECface agent. In addition, DECface exhibits natural gaze behavior during an interaction by periodically glancing in the direction of each of the current customers. Thus the processing requirements depend fundamentally on the number of customers and their rate of arrival and

[*] Dept. of Computer Science, Rice University
[†] College of Computing, Georgia Institute of Technology

departure. Furthermore, in order for the kiosk to be compelling, the latency with which it responds to user input must be very low. The combination of dynamism and low latency in the kiosk application make it a challenging and stimulating domain for parallel computing research.

Figure 1: The Smart Kiosk

A key computational attribute of the kiosk application class is the generation and processing of *streams* of multimedia data. The number and bandwidth of these data streams is the source of the dramatic computational requirements for the application class. This is illustrated in Figure 2, which gives the task graph for a color tracking subsystem. The color tracker is one of the more computationally demanding components of the Smart Kiosk system. In the figure, cylindrical tubes represent *c*hannels holding streams of multimedia data and ovals represent tasks that operate on this data. We will focus on the color tracker in this paper, as it exhibits the complexities that are typical of our application class. A more detailed discussion of the color tracker can be found in [9].

The abstract execution model for the task graph of Figure 2 is that each task is running on its own virtual processor. All tasks are running continuously in parallel. A channel is location independent and holds a collection of objects indexed by time. A task names the various channels it touches and designates them as input or output channels (from the perspective of this task). In our applications, the objects in the channels are largely video frames in various levels of processing. A task can put objects into the channels and get objects out of the channels according to the specified task/channel connectivity. At this abstract level the frames in the channels are ordered but there is no concept of the time required by a task to process a frame.

The actual platform[1] is a cluster of SMPs using the Stampede run-time system [5] developed at CRL. In the actual execution model, each task is a POSIX thread. The channel mechanism is provided in our environment by Space-Time Memory [8] (STM), a distributed data structure which supports location independent sharing across the cluster. This support allows two tasks to communicate over a channel via the same mechanism regardless of whether the tasks are on the same SMP in a cluster or on different nodes of the cluster. The STM framework [8, 9], our application class [12], and the problem of integrating data parallel decompositions into a task-level representation of stream-based applications [10] have all been described earlier. The ideas presented in this paper are not tied in any way to STM, but they do rely on some underlying support for parallel execution of the abstract model.

The abstract execution model provides an abstraction of time which is convenient for designing and coding multimedia applications. Of course, the specific time required for task execution and communication

[1] The appendix provides more details.

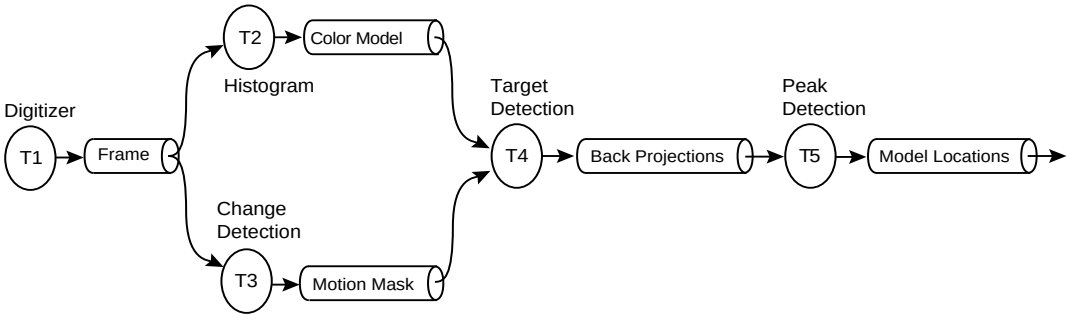


Figure 2: Task graph for color-based tracker.

plays a critical role in the actual execution model. For example, a task that puts objects into a channel may do so at a faster rate than the task that consumes the objects. This is common since upstream tasks, such as the digitizer, often perform less complex processing than downstream tasks, such as target detection (see Figure 2). Thus a downstream task may restrict its processing to only the most recent data generated by an upstream task.

There are many possible mappings between the abstract model and a specific cluster implementation. In the abstract model there is significant pipeline parallelism as a result of the loose temporal coupling within data streams. There is often significant task-level parallelism resulting from multiple tasks operating on the same streams. An example is the histogram and change detection tasks in Figure 2. In addition, there is also significant data parallelism within each task.

This paper addresses the problem of mapping the abstract task model to an actual platform so as to maximize performance objectives. The two performance objectives are minimizing latency and maximizing uniformity of frame processing over time. Latency is more important to us than throughput since in our application, for the animated face to be compelling it need not respond to every move you make but when it does respond, it must do so rapidly. We define latency as the time from the digitizing of the frame to completion of its processing. An execution that exhibits uniformity processes frames at a reasonably regular rate. A non-uniform execution might process three frames in a row and then skip the next hundred frames, for example.

In our earlier work we relied upon generic run-time services, such as the thread scheduler in the pthread package, to regulate the execution of tasks in our application. As we will demonstrate, this results in significant inefficiencies. There are two basic issues in implementing the abstract model:

- Scheduling work on a given processor

  Consider the simplest case when the target is a uniprocessor and no parallelism issues arise. Even in this case scheduling issues arise.

  A poor scheduling decision may well result in the generation of a number of consecutive frames rapidly followed by the consumption of these frames. In such a schedule the latency per frame is higher than necessary. In addition, there will be a long sequence of consecutive frames processed followed by a long sequence of consecutive frames not processed. This schedule does not exhibit uniformity. Events that occur in the interval of unprocessed frames will go unrecognized.

- Distributing work among processors

  Assume the simple case where we have the same number of actual processors as tasks. The obvious solution is to assign a processor to each task. However, if there is a mismatch in the rates that each task processes frames it may well be better to assign three tasks to time slice on one processor and assign one task to three processors for data parallel processing.

These issues have been addressed in other systems [13]. However, our class of programs face a difficulty not addressed in these other systems. Our programs are dynamic. The solutions previously proposed address only static programs. In our programs, the task graphs remain fixed. However the optimal data parallel decomposition or the optimal schedule relies on the relative costs of the tasks. It is these relative costs that vary dynamically. In the task graph in figure 2, the time for tasks T1, T2, and T3 do not depend on the number of models (people) being tracked. The time for tasks T4 and T5 are both linear in the number of models but the constant factor is quite different for these two tasks. The best strategy for integrating task and data parallelism and the best schedule vary significantly with the number of models. Dynamism, therefore, appears to make the job impossible. However, we will see that the class of applications we address exhibit properties that enable good scheduling even in the presence of this dynamism.

In this paper we present a framework for optimally scheduling constrained dynamic applications to maximize both latency and throughput. Our framework can exploit integrated task and data parallelism to achieve this objective. The resulting schedules are simple to implement in a run-time system such as Stampede. One consequence of our results is that it is possible to get good run-time performance for these applications without employing a real-time operating system. This may be an attractive feature for many practical system-building efforts.

In Section 2 we introduce the concept of *constrained dynamism* and its applicability to our class of problems. Section 3 describes our scheduling strategy based on this idea. Interested readers can find details of implementation and Space-Time Memory in the appendix.

2 Constrained Dynamism

2.1 The Idea

Notice that the dynamism in our applications is not arbitrary. These applications obey what we call *constrained dynamism* which is defined as follows:

- The system changes among a small number of states.
- State changes are infrequent.
- State changes are detectable.

Here, a *state* is the set of variables that influence the scheduling decision. For example, in our color tracker application the state corresponds to the number of people currently interacting with the kiosk. This number will typically be from one to five and will change infrequently relative to the processing rate as people come and go. Departures and arrivals can be easily detected using standard vision techniques [11].

We can use static techniques for dynamic applications if the dynamism is constrained. Off-line we apply existing static optimizations to each of the small number of states. Then on-line we simply react to state

changes by switching among pre-computed solutions. As we will see this approach applies both to data decompositions and to scheduling.

A well known technique for handling changing application states relies on the property that small changes in states result in small changes in desired scheduling strategy. This would be a reasonable approach if the number of interesting states is extremely large or unknown, in which case, the decomposition or scheduling strategy would be determined by interpolating between known good strategies in known states. However, in our case, a seemingly small state change could alter scheduling strategy dramatically.

One of the major contributions of this paper is the concept of constrained dynamism and the performance enhancements obtained by optimizing schedules for constrained dynamic applications within an integrated task and data parallel framework.

2.2 Example: Using Constrained Dynamism for Data Decomposition

This example illustrates the effect of constrained dynamism in determining optimal data decomposition for a dynamic environment. Although, the constrained dynamic property of the application also affects application scheduling, we ignore it for now to keep the example simple.

The target detection task in figure 2 is highly compute intensive and a good candidate for parallelization. The task is parallelized by decomposing its input data and farming out parts to different data-parallel threads. The input to the task consists of a digitized frame and color models for the targets to be detected in the frame. There are two obvious ways to decompose the input and distribute the data across data-parallel threads:

1. Divide the set of models up into sub-sets of models.

2. Divide the frame up into regions.

As a third alternative, the input may also be divided in both ways at the same time so that one piece of work corresponds to searching for a subset of models in a region of the frame.

Table 1 shows some measured latencies using various data decomposition strategies. First column indicates the number of regions, or partitions, into which a frame is divided. The table shows results for one and four partitions – former corresponds to no decomposition of a frame. Second column shows the times when there is only one model to be detected in the input frame. Clearly, there is no way to divide models in this case. Third and fourth columns are the timings obtained when there are eight models to be detected. In this case there is a choice of decomposing the input based on models. The table lists timings corresponding to eight-way division of models (MP=8) and no division of models (MP=1). Numbers in parentheses are the total number of work chunks in each case, given by the product of FP and MP.

	Total Models		
	1	8	
Partitions	MP = 1	MP = 8	MP = 1
FP = 1	0.876 (1)	1.857 (8)	6.850 (1)
FP = 4	0.275 (4)	2.155 (32)	2.033 (4)

MP = number of partitions based on models
FP = number of partitions of a frame

Table 1: Timing results in seconds/frame for the target detection task with one and eight target models.

The table shows that it is best to distribute models if there is a sufficient number of them. Otherwise, it is best to divide the frame into regions. These results illustrate two important aspects of the application:

- Best data decomposition strategy varies, depending on the current state (number of models). Therefore, a single statically determined strategy will not result in optimal performance in all cases.
- There is a small number of data decomposition choices, and the correct choice can be easily determined at run-time. Therefore, it is easy for the application to switch the data decomposition strategy based on the current state (number of models).

The next section shows how a good schedule and data decomposition can be computed even in a dynamically changing environment because the dynamism is constrained.

3 Scheduling and Resource Allocation

There are two major aspects of the performance problem that we need to address. First, as discussed in the previous section, we have to optimize performance within a dynamically changing environment. Second, even for the static case, optimizing performance for this applications is not trivial. Section 1, for example, introduced some of the inefficiencies caused by mapping our abstract execution model to the actual execution model. Addressing these inefficiencies is critical. We will first address the static case, then show how to use the static solution to build a solution for the dynamic case.

3.1 Hand Tuning

In the color tracker and other applications based on digitized video images, the primary tuning variable is the period at which the digitizer thread executes. This controls the rate at which digitized frames are fed to the rest of the system. This rate has a direct effect on both the latency and throughput of the application. We define latency to be the time it takes to process a single video frame. It is measured as the time interval between placing a frame into the Video Frame channel and reading all of its detected target locations from the Model Location channels (see Figure 2). We define throughput as the number of frames completely processed per unit time. It is measured as the inverse of the time between the arrival of two consecutive results at the output of the application (the inter-arrival time).

Figure 3 illustrates this discussion of adjusting the digitizer rate. At the minimum execution period for the digitizer thread (33ms, as determined by NTSC video rate), it rapidly saturates all the channels in the application with unprocessed data. This results in a high throughput since there is always work to be done, but a correspondingly high latency for a given frame due to the backlog of unprocessed items. As we increase the digitizer period, the throughput and latency drop until the latency reaches its minimum value (3.2 seconds in our experiments.)

The tuning curve was obtained by plotting the measured latency and throughput as the digitizer period varied from 33ms to 5 seconds in steps of approximately one second. The desired operating point is the lower right corner of the graph, where latency is low and throughput is high.[2] In the tuning curve, as the digitizer period decreases the system is saturated with work and the timings we observed, which depend on the pthread scheduler, became fairly erratic, varying by about one second.

The factor of two in the range of latencies indicates that addressing performance is important. In the next section we examine inefficiencies beyond those addressed by the hand tuning presented here.

[2] The point labeled "optimal" will be discussed later.

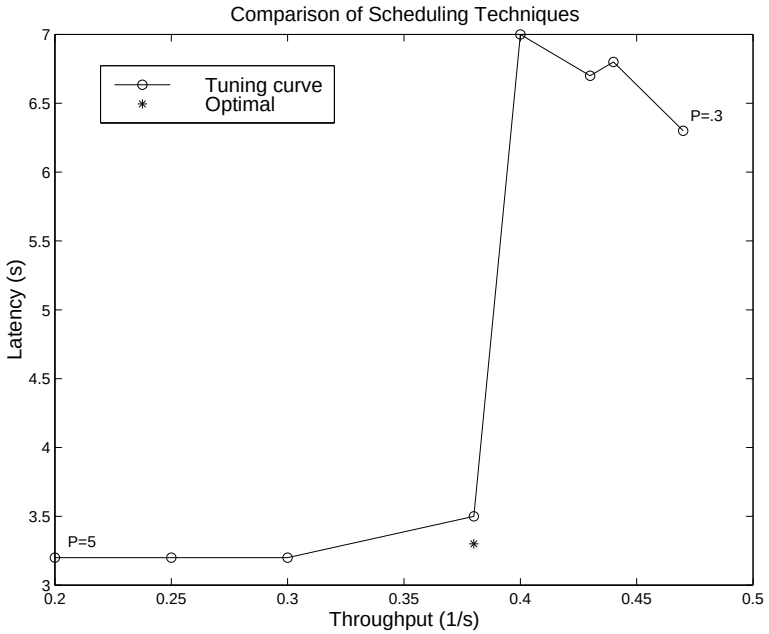


Figure 3: Comparison of optimal and tuned schedules for detecting eight models.

3.2 Problems with General On-line Schedulers

The discussion of hand tuning in the previous section assumed that the application is running with a pthread scheduler. However, serious inefficiencies result from the fact that the pthread scheduler is a general on-line scheduler. It not only knows nothing about the specific application but also has no understanding of the application class represented by graphs like the one in Figure 2, based on a small number of tasks that process streams of time-indexed multimedia data.

We will use Figures 4 and 5 to illustrate some of these problems. These figures show for each processor (horizontal axis) what task it is performing over time (vertical axis). The tasks are those specified in Figure 2. T1 represents the Digitizer and is too fast to be visible at this scale. T2 is Change Detection. T3 is Histogram. T4 is Target Detection. T5 is Peak Detection. Repeated occurrences of a task in a processor represent the processing of distinct items (with distinct time-stamps) by that task.

Instances of different tasks shaded identically represent the processing of a given time-stamp by those different tasks. Task T1, the digitizer, is so fast that it does not show up on this scale and is omitted from all the figures. The latency in the figure is the time from the beginning of task T2 for a given time-stamp to the end of task T4 for the same time-stamp. Latency, the time to complete processing of a single frame is indicated by the bar at the right of each figure.

The characteristics of our application class that make pthread scheduling inefficient are described below. We refer to Figure 4(a) which shows a schedule that could result from pthread scheduling.

- These applications are primarily concerned with latency. A general on-line scheduler, in attempting to

Figure 4: Performance of naive pthread (a) and pipeline (b) scheduling strategies.

keep all processors busy focuses more on throughput. The latency in the figure is longer than necessary.

- Threads appear to be continuous but are actually discrete with respect to items generated. The pthread scheduler will happily schedule a thread for enough time to generate two and a half items. This is a poor choice when latency is key. Partial processing of items are not shown in this figure since in this example only one thread is executing in a given processor but if distinct threads execute in a given processor, scheduling a thread so that it partially completes an item would clearly increase the latency.

- A pthread scheduler will assume that a thread can only be scheduled on one processor at a time. For this class of applications, however, we can execute the same thread operating on multiple processors concurrently as long as they operate on different frames of data. Because the schedule in the figure obeys this restriction it is less efficient than it could be.

- Applications in this class can be viewed as a pipeline consisting of tasks through which data flows. Early tasks tend to perform low-level operations that process items quickly. Later tasks tend to be high-level more compute intensive tasks that require more time per item. It is likely – indeed it often does happen – that the on-line pthread scheduler schedules an early task to generate a large number of items and a later slower task is scheduled for the same time slice. In this case the later task can not keep up. We see this phenomenon between task T3 and T4 in the figure.

Motivated by these problems, we propose an off-line scheduler, designed specifically for our application class, that optimizes the schedule for a specific application and configuration.

We start by describing our scheduling approach in the context of a static situation. Recall that our applications are dynamic and will require different schedules in different states. (In our color tracker a state corresponds to the number of people in front of the kiosk.) Later in the section we present a strategy that relies on the fact that the dynamism of the application is constrained to accommodate dynamic changes in behavior.

3.3 Scheduling for the Static Case

Recall that the abstract execution model for our class of applications is that each task executes on a virtual processor and the stream of data is communicated among tasks and therefore among virtual processors.

Our initial plan for improved scheduling for our abstract execution model was to employ a run-time scheduling mechanism that understood this abstract execution model. For example, it could perform flow control by limiting the number of items each channel could hold. This proved to be totally inadequate.

Instead we transform the abstract execution model to an alternate abstract model. Consider the work for a given time-stamp, through all the tasks, as an *iteration*, where the iterations are over time-stamps. In this view an iteration would execute on a single virtual processor. (We will ignore data parallelism for the moment.) This is software pipelining as illustrated in Figure 4(b). Each virtual processor processes one time-stamp through all its tasks and then begins on the next time-stamp. The shading distinguishes the work for one time-stamp from the work for another.

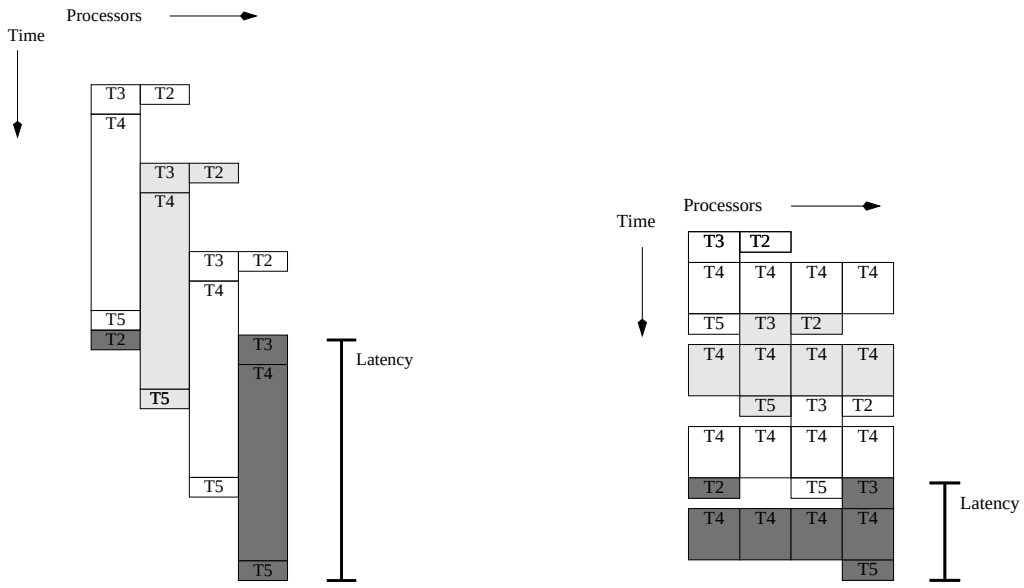
In the abstract execution model there are an infinite number of virtual processors. Each executes one time-stamped frame and begins execution when that frame becomes available and stops when the processing for that frame is complete. In the actual execution model an actual processor will begin processing a new frame when work on the previous one is complete. Typically there will not be enough actual processors to process all the frames so a delay is chosen to maintain a fixed interval between the time-stamps processed.

This alternate abstract model and its actual implementation have several advantages. This schedule has no idle time, maintains a uniform rate of frame processing, and no work is performed on any time-stamp that is not processed fully by all downstream threads. Latency for a time-stamp is never increased by starting work for a time-stamp and then suspending work on that time-stamp. Although, this schedule achieves high throughput, it does not achieve minimal latency.

Given this naive software pipeline, our next goal is to minimize the execution time for one iteration, that is to minimize latency. Without sacrificing latency, of course we would like to attain maximum possible throughput. Figure 5(a) shows how we can reduce the latency of a single iteration by taking advantage of available task parallelism. Notice that threads T2 and T3 can be executed in parallel. This creates idle time and reduces throughput but this trade-off is consistent with our goal of reducing latency. Notice that in Figure 5(a), the pattern shifts over one processor for each successive time-stamp. Therefore every fourth instance of T2 must wrap around and be scheduled to the first processor.

In earlier work we addressed the integration of data parallelism into our fundamentally task parallel model [9]. See Appendix 6.2 for a brief overview of the data parallel mechanism available within Space-Time Memory. If, among our scheduling possibilities, we include the fact that many of the tasks themselves are data parallel, we can further reduce latency. Figure 5(b) shows such a scheme where we take advantage of the fact that T4 is data parallel.

We implemented the schedule in Figure 5(b). Our goal was to compare the results shown in Figure 3 (where resource allocation results from the combination of run-time actions of the pthread package and the



(a) T2 and T3 are task parallel.

(b) T4 is data parallel.

Figure 5: Schedules that exploit task parallelism (a) and data parallelism (b) exhibit significantly reduced latency.

tuning decisions made by the programmer) to the optimal *pre-computed* schedule.

Our expectation is that the optimal schedule would attain the minimum latency at the same time provide good throughput as a result of pipelining. This is borne out by our experiment. The result of the pre-computed schedule is indicated by the asterisk in the lower right in figure 3. This indicates performance that is strictly better than all of the points on the tuning curve. It achieves minimum latency but fails to achieve maximum throughput since the schedule of Figure 5(b) contains some wasted space. This tradeoff is consistent with our desire to minimize latency.

The experiment has two implications. First it demonstrates the power of explicit scheduling, which results in an operation point which is impossible to obtain by straightforward tuning of the application. In particular, the optimal pre-computed schedule results in a latency which is less than half of the worst case latency for naive scheduling of the optimal data parallel decomposition for this program. Second, it suggests that use of the general scheduling algorithm could remove much of the labor which would otherwise be spent tuning the application for good performance.

Minimizing latency for a given iteration may mean distributing some task across all processors to maximize data parallelism. If so, this task acts as a barrier to the overlapping of multiple iterations. (Recall that each iteration is constrained to minimal latency.) For our class of applications this is the right choice. The cost of communication between nodes in a cluster may mean that the minimal latency schedule for an iteration does not use all processors but is instead restricted to the processors on a single node. In this case,

distinct iterations on distinct nodes can overlap.

This technique has several desirable consequences:

- By focusing on minimizing latency, we minimize the time for which a piece of data is "live". This has the desirable side-effect of reduced space requirement.
- A fixed schedule (not necessarily an optimal one) simplifies garbage collection (handled in our system by STM) resulting in further performance gains.
- By executing threads according to a schedule, we also solve the problem of flow control implicitly. A fixed schedule determines the number of items in each channel.

Notice that we are discussing the schedule at a high level. We have not addressed the implementation of the schedule. The goal of the scheduler is to alter the execution to minimize latency. One could implement this schedule in a variety of ways. For example, one might add dependences to ensure that pthread scheduler will "do the right thing". One might generate a master for each processor that controls its pre-computed processor-specific schedule. One might even implement this system on top of a soft or even a hard real-time scheduler. Our current implementation uses the first approach above. For this experiment we determined the optimal schedule by hand and implemented it by creating additional dependencies between the data parallel instance of T4 for one time-stamp and T1 for the next time-stamp. However, our focus here is on the determination of the scheduling not on the implementation of the schedule.

The scheduling algorithm is shown in Figure 6. Notice that the algorithm is not a heuristic to achieve a good schedule while minimizing scheduling time. Our applications have a very small number of tasks. Even if we include the various data parallel options for any given task, we still have a manageable number of options. Since the resulting schedule will be operating for months, we can afford to evaluate *all* legal schedules and choose the *best* one.

Input:
The task graph for the application, a macro-dataflow graph in
which nodes represent high level operations that produce and consume
data items and edges represent communication among producers and consumers.
Execution times for each operation including its data parallel variants.
Execution times for communication of each data type both within
and across nodes in the cluster. This includes times for parts of
structures needed to support each data parallel variant.
The number of nodes and the number of processors within each node.

Algorithm:
Compute the minimal latency, $\mathcal{L}$, for a single iteration
Compute the set, $\mathcal{S}$, of all single iteration schedules that exhibit latency, $\mathcal{L}$
Compute the multi-iteration schedule, $\mathcal{M}$, created from multiple instances of a schedule from $\mathcal{S}$

Figure 6: Algorithm for finding optimal schedule

3.4 Scheduling under Constrained Dynamism

Section 3.1 shows that for a given static condition, a pre-computed schedule is more effective than hand-tuning by adjusting arrival rates. Clearly, pre-computed schedules are easier for the user than user tuning.

To accommodate the unpredictable changes in the applications we will have to dynamically change the schedule.

We pre-compute the optimal schedule for each of the states. The actions required on a state change are:

- Perform a table look-up to determine the new schedule for the new state.
- Perform a transition to the new schedule.

The characteristics for constrained dynamism make this an ideal approach. The fact that there are a small number of states means that pre-computing an optimized schedule for each state is reasonable. The fact that a change of state is detectable means that we know when to change schedules. The fact that changes in state are infrequent means that the we overcome any inefficiency at the point of a change in schedule, over the relatively long use of the new schedule.

Notice that this approach to dynamic changes in schedule is totally orthogonal to the approach to determining a good schedule for a single state. For example, we might use this approach to constrained dynamism whether the schedules for each state were chosen optimally, via heuristics or via hand-tuning.

This approach to constrained dynamism enables the application to operate in optimal or near-optimal region in the face of a dynamically changing environment. We believe that this provides a unique, valuable and automatic programming tool for high performance multimedia applications.

4 Related work

Further information about the Smart Kiosk project can be found in [16, 2]. Our color tracking work is described in [11] and is based on [14].

The *Distributed Shared objects* system that provides a low level sharing mechanism across the cluster in our system is based on our earlier work on Cid [4] and is most closely related to the Midway shared memory system [1].

Early work on performance issues within the STM framework appeared in [9]. The scheduling work is presented in the current paper for the first time.

Other approaches to optimizing the integration of task and data parallelism include [13]. The difference between our work and this primarily arises from the fact that we have to optimize in the presence of dynamic changes in the behavior of the application. The importance of latency as opposed to throughput also differentiates our system.

The scheduling of multimedia code on parallel systems described in [7] addresses a problem similar to ours. However, our technique attacks the scheduling problem at a finer grain for a smaller number of processors than the retiming method described there. Our support for dynamism and focus on latency are additional differentiators.

Scheduling algorithms for real time systems have been heavily studied in the past [6]. For applications like the Smart Kiosk, we are interested in high-level issues such as optimizing specific performance criteria like latency in a distributed implementation. Standard real-time schedulers are not designed to exploit the level of knowledge about application performance that our constrained dynamic schedules depend upon. However, the implementation of our optimal schedules could be facilitated by the use of a real-time operating system.

5 Conclusions

There is an emerging class of dynamic, computationally-intensive multimedia applications, of which the CRL Smart Kiosk is a motivating example. These applications process streams of video and audio data and require low latencies for good performance. They are also highly parallel and well-suited to implementation on clusters of SMP's.

Efficient scheduling of application tasks is critical for good performance. General purpose schedulers, such the pthread library, produce high latencies because they lack knowledge of the dependencies between tasks. While there has been significant previous work on optimal scheduling, it has generally been limited to static problems. In contrast, our application is highly dynamic as the optimal schedule depends upon the real-time behavior of the kiosk's customers.

We have demonstrated that the dynamism in our application is constrained, in that there are a small number of operating regimes which are determined by the state of the application. We have shown that optimal schedules can be pre-computed for such *constrained dynamic* applications. The run-time system can then switch among the set of optimal schedules as dictated by the observed state. An experimental comparison with a hand-tuned schedule demonstrates the promise of this approach.

6 Appendix

6.1 STM and Stampede

Applications like the Smart Kiosk are made up of a heterogeneous collection of tasks which operate on streams of multimedia data. These applications fit very naturally into a coarse-grained multi-threaded programming model.

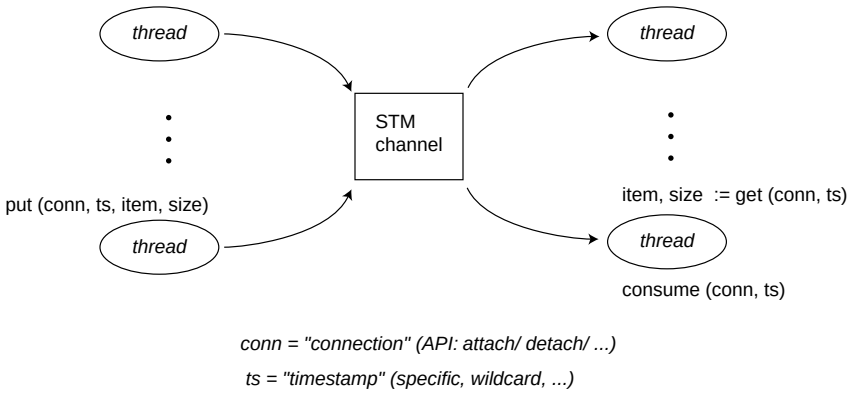


Figure 7: Overview of Stampede channels.

At CRL we have developed a programming system called "Stampede" that provides a uniform programming model for these applications across SMPs and clusters. Threads in Stampede are dynamic Posix threads (pthreads), extended to allow forking and joining across clusters. Stampede has a variety of cluster-wide data sharing facilities, but the one that supports these multimedia applications is a structured shared-memory

abstraction called Space-Time Memory (STM). The key construct in STM is *channel*, which is a location-transparent collection of objects indexed by time.

Figure 7 shows an overview of how channels are used. Figure 8 describes the two main functions in *Stampede* API that threads use to interact with channels. Virtual *time stamps* enable access of correlated data and allow threads to consume streaming data at varying rates. Additional information about Stampede can be found in [5] and about STM in [12].

`spd_channel_put_item (o_connection, timestamp, buf_p, buf_size, ...)`	
`o_connection:`	specifies the output connection to a channel
`timestamp:`	a channel cannot have more than one item with the same timestamp, but the items can be "put" in any order
`buf_p and buf_size:`	specify buffer containing the data to be put
`spd_channel_get_item (i_connection, timestamp, &buf_p, &buf_size, &ts_range, ...)`	
`o_connection:`	specifies the input connection to a channel
`timestamp:`	it can specify a particular value or it can be a wildcard requesting the newest/oldest value currently in the channel, or the newest value not previously gotten over any connection, etc.
`buf_p and buf_size:`	specify a buffer to receive data or NULL to ask Stampede to allocate the buffer
`ts_range:`	returns the timestamp of the item returned, if available; if unavailable, it returns the timestamps of the "neighboring" available items, if any

Figure 8: Primary Thread Interface to Channels.

We have implemented Stampede as a C library under Digital Unix. Stampede is currently running on a cluster of four AlphaServer 4100's, each being an SMP with four 400MHz Alpha processors. The SMP's are interconnected with both Digital Memory Channel and Myrinet.

6.2 Integrating Data and Task Parallelism

The strong requirement for minimal latency in our application class arises from our goal of producing compelling human-like interaction. This requirement motivated our work in [9] to integrate data parallelism into the task parallel system described above in Section 6.1.

In that earlier work we focused on finding the optimal data parallel strategy. Here however, the choice of data parallel strategy is determined as a side-effect of optimal scheduling. Here we summarize the mechanism for integration of task and data parallelism in our environment without addressing the choice of optimal strategy.

Recall that our programs are dynamic, in order to achieve top performance our mechanism for data parallelism will need to support dynamically vary data decompositions.

The key idea is that any node in the task graph can be replaced with a subgraph consisting of multiple worker threads that exactly duplicates the original task's behavior on its input and output channels. The structure in Figure 9 illustrates the three basic components used to integrate data parallelism within a task: *splitter*, *worker*, and *joiner*. For some task T, we execute N instances of the task concurrently. The splitter reads from the input channels for task T. It divides a single chunk of work into M data parallel chunks and puts them on the *work queue*. Each worker is a parameterized version of the original application task T, designed to work on arbitrary chunks. Chunks get assigned to worker threads based on worker availability.

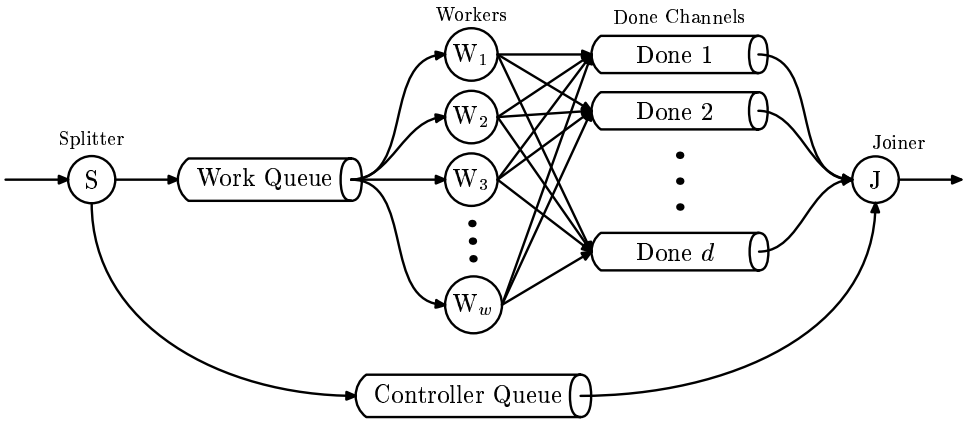


Figure 9: Integrated task and data parallelism

The splitter tags each chunk with its target *done* channel (result for chunk i goes to done channel i) which is used by the worker threads to output the results. The done channels act as a sorting network for the chunk results.

The data to be processed may be decomposed in several ways (by models or by regions of the frame, in our example). We pre-compute the ideal data decomposition for each of the small number of potential states of our constrained dynamic program. During execution, the splitter will look-up the decomposition for the current state from a pre-computed table, and communicate the decision over the controller channel to the joiner. The splitter uses this decision to actually decompose the work into chunks in the work queue. Finally, the joiner reads done channels to combine individual results into a single output result for task T.

References

[1] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway Distributed Shared Memory System. In *Proceedings of the IEEE CompCon Conference*, 1993. Also CMU Technical Report CMU-CS-93-119.

[2] A. D. Christian and B. L. Avery. Digital Smart Kiosk Project. In *ACM SIGCHI '98*, pages 155–162, Los Angeles, CA, April 18–23 1998.

[3] *Proc. of Third Intl. Conf. on Automatic Face and Gesture Recognition*, Nara, Japan, April 14–16 1998. IEEE Computer Society.

[4] R. S. Nikhil. Parallel Symbolic Computing in Cid. In *Proc. Intl. Wkshp. on Parallel Symbolic Languages and Systems*, pages 217–242, October 1995.

[5] 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*, Chapel Hill NC, Aug. 7-9 1998. See also Technical Report 98/1, Cambridge Research Lab., Compaq Computer Corp.

[6] M. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Prentice Hall, 1995.

[7] S. Prasanna. Compilation of Parallel Multimedia Computations – Extended Retiming Theory and Amdahls's Law. *Sixth ACM SIGPLAN Symposium on Principles and Practice of Praallel Programming*, June 1997.

[8] U. Ramachandran, R. S. Nikhil, N. Harel, J. M. Rehg, and K. Knobe. Space-Time Memory: A Parallel Programming Abstration for Interactive Multimedia PApplications. In *Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Atlanta, GA, May 4–6 1999.

[9] J. M. Rehg, K. Knobe, U. Ramachandran, R. S. Nikhil, and A. Chauhan. Integrated Task and Data Parallel Support for Dynamic Applications. In D. O'Hallaron, editor, *Fourth Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, pages 167–180, Pittsburgh, PA, May 28–30 1998. Springer Verlag.

[10] J. M. Rehg, K. Knobe, U. Ramachandran, R. S. Nikhil, and A. Chauhan. Integrated Task and Data Parallel Support for Dynamic Applications. *Scientific Programming*, 1999. To appear.

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

[12] J. M. Rehg, U. Ramachandran, R. H. Halstead, Jr., C. Joerg, L. Kontothanassis, and R. S. Nikhil. Space-Time Memory: A Parallel Programming Abstraction for Dynamic Vision Applications. Technical Report CRL 97/2, Digital Equipment Corp. Cambridge Research Lab, April 1997.

[13] J. Subhlok and G. Vondran. Optimal Latency-Throughput Tradeoffs for Data parallel Pipelines. *Proc. 8th Symposium on Parallel Algorithms and Architecture (SPAA)*, June 1996.

[14] M. Swain and D. Ballard. Color Indexing. *Intl J. of Computer Vision*, 7(1):11–32, 1991.

[15] K. Waters and T. Levergood. An Automatic Lip-Synchronization Algorithm for Synthetic Faces. *Multimedia Tools and Applications*, 1(4):349–366, Nov 1995.

[16] K. Waters, J. M. Rehg, M. Loughlin, S. B. Kang, and D. Terzopoulos. Visual Sensing of Humans for Active Public Interfaces. In *Computer Vision for Human-Machine Interaction*, pages 83–96. Cambridge University Press, 1998.