Rethinking the Pipeline as Object–Oriented States with Transformations

Steve MacDonald
School of Computer Science,
University of Waterloo
Waterloo, Ontario, CANADA
stevem@uwaterloo.ca

Duane Szafron and Jonathan Schaeffer
Department of Computing Science,
University of Alberta
Edmonton, Alberta, CANADA
{duane,jonathan}@cs.ualberta.ca

Abstract

The pipeline is a simple and intuitive structure to speed up many problems. Novice parallel programmers are usually taught this structure early on. However, expert parallel programmers typically eschew using the pipeline in coarse–grained applications because it has three serious problems that make it difficult to implement efficiently. First, processors are idle when the pipeline is not full. Second, load balancing is crucial to obtaining good speedup. Third, it is difficult to incrementally incorporate more processors into an existing pipeline. Instead, experts recast the problem as a master/slave structure which does not suffer from these problems. This paper details a transformation that allows programs written in a pipeline style to execute using the master/slave structure. Parallel programmers can benefit from both the intuitive simplicity of the pipeline and the efficient execution of a master/slave structure. This is demonstrated by performance results from two applications.

1. Introduction

Pipelines are common in many real–world problems. Every modern processor uses it to speed up the decoding and execution of instructions. Factories use the assembly line to speed up production. It is a simple, general–purpose way to improve the performance of many tasks.

Parallel programmers are taught the pipeline structure early on, as an intuitive way of achieving speedup for many problems. However, the pipeline exhibits three serious problems that make achieving good performance difficult for coarse–grained applications. First, when the pipeline is not full (at the beginning and end of a program), stages are idle and wasted. Second, any load imbalances in the pipeline reduce its performance. Such imbalances cause work to build up at expensive stages while less expensive stages sit idle. Third, the concurrency in a pipeline is tightly coupled with the set of chosen stages, so using more processors may require the pipeline structure be changed to rebalance it. Making these changes can be difficult.

Because of these problems, most parallel programming experts eschew the use of pipelines for coarse–grained applications. Instead, they restructure the problem using other parallel structures that do not suffer from these problems, such as a master/slave. However, novice parallel programmers find that the pipeline is a natural and intuitive structure. If the deficiencies of the pipeline can be fixed, then developers can use it without fear of performance problems.

This paper describes a transformation that allows object–oriented parallel programs written as a pipeline to execute using the master/slave structure. This transformation is based on the State design pattern [2], recasting the pipeline as a series of state transitions on a stream of input objects resulting in a stream of output objects. The underlying master/slave structure addresses the load balancing and idle processor problems in traditional pipelines. Further, the transformation decouples the concurrency from the pipeline structure, so threads no longer execute one specific stage. Instead, threads execute transitions for any stage, improving load balancing and reducing (but not eliminating) idle times during ramp–up and ramp–down. The result is a more efficient pipeline that is more suitable for coarse–grained applications. Parallel programmers can take advantage of conceptual simplicity of the pipeline while still benefiting from the efficient execution of the master/slave.

This paper is organized as follows. Section 2 describes the traditional implementation of the pipeline, focusing on its problems. Section 3 describes the State design pattern on which the new pipeline formulation is based. Section 4 details the new pipeline and shows how this solves the problems inherent in the traditional pipeline. The benefits of the State–based pipeline are

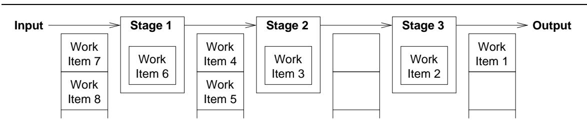

Figure 1. An example of a pipeline.

demonstrated in Section 5 using two applications. The first is an animation example that provides an easy–to–understand example to highlight the differences between a traditional pipeline and the State–based pipeline. The second problem is an implementation of JPEG compression and encoding that is used as a real–world example. Section 6 discusses other related research, and conclusions are summarized in Section 7.

2. The Pipeline

A pipeline consists of a set of ordered stages, where each stage accepts input data from its predecessor, transforms that data, then transfers it to the next stage. Normally this transfer uses some form of buffer placed between the stages. This organization is shown in Figure 1.

A key characteristic is that the computation of each stage is independent. The parallelism in the pipeline results from the ability to be working on different parts of the data, at different stages of the pipeline, simultaneously. Thus, each stage can be assigned to a separate processor. For example, in the pipeline of Figure 1, each stage is executing a different work item. This allows a stream of incoming items to be processed efficiently. In the ideal case, the speedup of the pipeline is equal to the number of stages.

2.1. Weaknesses of the Pipeline

Unfortunately, achieving ideal speedup from a pipeline is difficult for three reasons. First, there is ramp–up and ramp–down time for the pipe. When the first few requests arrive and are being processed, the stages at the end of the pipe are idle. This situation continues until the pipeline fills. When the input stream of requests empties, an analogous situation occurs where stages at the beginning of the pipeline are idle. This idle time limits the speedup, particularly for deep pipelines with many stages and large-grained requests.

Second, the stages must be perfectly balanced to achieve ideal speedup. Any imbalance will cause performance problems. A stage that spends more time on a request than others will starve later stages. A stage that spends less time will be idle waiting for work from earlier stages. This problem reduces the number of processors that are effectively working and thus reduces the performance benefits of the pipeline. For example, in Figure 1, assume that the first stage has a short execution time. It is already executing the sixth work item, while the second stage is still working on the third item. The first stage will quickly drain the input requests and remain idle thereafter. One strategy for solving load imbalances is to replicate expensive stages to improve their throughput. To address the case where the first stage is too short, the remaining stages could be replicated. However, replication introduces two potential problems. First, in replicating a stage, we can no longer guarantee that requests are processed in the correct order. In Figure 1, if the buffers are FIFO then work items flow through the pipe in the order in which they arrive. If a stage is replicated then there is a race condition between the replicas. This problem can be fixed using sequence numbers to prevent queues from returning work items out of order. Second, balancing the pipe is difficult with replicated stages, particularly if the stages take a variable amount of time. The programmer must find the correct ratio of replicas for each stage to balance the throughput. Finding this ratio will be difficult, particularly with small numbers of processors.

The third weakness with the pipeline is that the concurrency is tightly coupled with the set of stages. Each thread can only execute the code for a specific stage. To add more processors to a pipeline program, a new stage must be added. This new stage could be a replica of an existing stage, with the above problems. Alternately, a new stage could be introduced by refactoring the pipeline computation across a larger set of stages. This new stage will affect the balance of the entire pipeline, and it may take considerable effort to construct a balanced refactoring. Thus, incrementally adding more processors to a pipeline can be a difficult task.

2.2. Object–Oriented Pipelines

Most object–oriented versions of the pipe replace stages with Active Objects [6]. An Active Object has its own thread assigned to it, and all invocations on the Active Object are executed using this thread. More complex pipelines also consider the differences between having a stage *push* data to its successor and having a stage *pull* data from its predecessor [14]. Even in this more complex pipeline, the basic transformation of pipe stages to objects is the same.

3. The State Design Pattern

The State design pattern deals with entities that can be in different states [2]. The state of the entity determines the behaviour of operations on it. For exam-

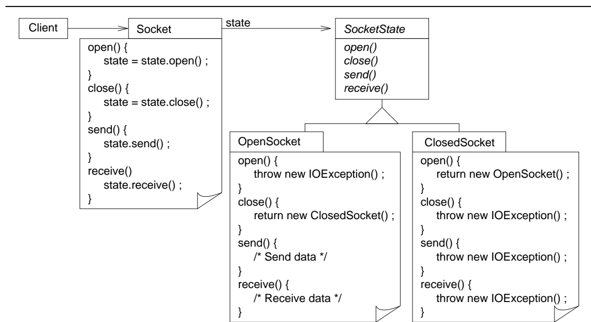

Figure 2. The State design pattern, using a socket that can be open or closed.

ple, consider a socket for interprocess communication, which can be open or closed. Issuing a send operation on an open socket should forward data through the socket, where issuing the same operation on a closed socket is a run–time error.

An object representing such an entity must adapt its behaviour for the current state. This could be accomplished by having the object check the state in each state–dependent method. However, such code will be difficult to maintain. Adding new states or transitions requires all methods be checked and (possibly) modified. A programmer could easily miss a transition and introduce an error into the program.

The State design pattern represents the different states of the entity as different classes. The behaviour of methods in these classes is determined by the class, which represents the state. An example of the State pattern, using a simple socket, is shown in Figure 2. The socket can be open or closed, represented by the classes `OpenSocket` and `ClosedSocket`. In `OpenSocket`, the `send()` method sends data through the socket, where the method throws an exception in `ClosedSocket`. This State pattern includes a *context object*, an instance of `Socket`. This context object hides state transitions from classes using the socket. By representing each state as a class, this pattern obviates the need for methods to check the current state. This makes it easier to add new states. It also provides separation of concerns, allowing each state to be easily examined and changed in isolation.

4. The Pipeline Rethought: The State-based Pipeline

The State design pattern provides us with a solution to problems that manipulate entities that can be in one of several different states. Initially, this appears unrelated to pipelines. However, if we carefully consider the workings of a pipeline, the connection becomes clear.

In the traditional pipe, each stage sends its output data to the next stage, where it is further transformed before being sent to the next. This transformation may not be a simple refinement of the input data structure, but may result in a change in the size or type of the data. We could have one data structure with the superset of data needed by all stages, but this would waste memory. A stage consumes its input type and outputs another, the result of its transformation.

In an object–oriented pipeline, the request items can be replaced with request objects. Each stage accepts an input object and outputs a result object, possibly of different type, and passes it to the next stage. These request objects passed between stages are the intermediate results of the pipe. Again, we could use a single request object that holds the superset of all needed data, but this will be wasteful.

This idea shows the transformational nature of a pipeline. We can recast the pipeline as a sequence of transformations that take an input object and transform it into an output object. These transformations can be considered state transitions. Each stage causes its input object to transition to the next state. A pipeline is a composition of these state transitions that maps its inputs to the required outputs.

As described, the stages implement the transition operations on input request objects. If we apply the State pattern, the request objects in the pipeline become responsible for implementing this transition. This obviates the need for explicit stage objects. However, these stages also provided a thread of control or process to execute the transformation. We still require this concurrency to get parallelism in our pipeline.

One alternative is to simply use the threads as replacements for the stage objects. These threads invoke the state transition method on their input objects and send the output to the successor thread. However, this solution is little more than a refactoring of the original solution and still has all of the inherent problems with the pipeline.

Instead, we add the following element to this design. Each request object in the pipeline implements its transition using the same polymorphic method, called `transform()`. This method causes a state transition in the current object and returns the next state as its result. A typical implementation of this method creates the object for the next state using the current state as a constructor argument. Alternately, the method can obtain a preallocated object for the next stage from a memory pool and initialize it using the current state. Regardless, the next state uses accessor methods defined on the cur-

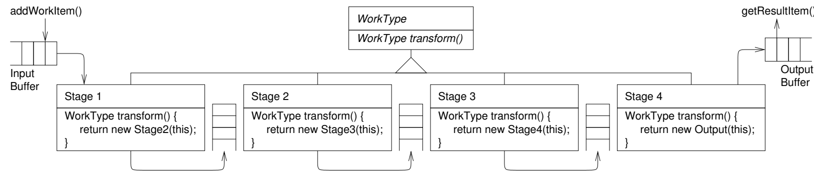

Figure 3. A pipeline based on the State design pattern.

rent state to obtain and transform the data as needed.

The result of using a single polymorphic method for state transitions is that the threads no longer need any special state–specific information to execute the transition for any object in the pipeline. Each object is a self–contained request that responds to the same method. Threads are completely independent of the state transitions. Given any request object in the pipeline, any thread can now execute the next stage of the pipeline.

Given this new independence between the concurrency and the pipeline transitions, we can internally execute the pipeline requests using a master/slave model. Buffers are introduced for holding the results for each stage, much like the buffers in Figure 1. Each buffer holds request objects of a specific type. Threads search the buffers to locate an outstanding pipeline request in any input buffer. The request is executed and the result, the next state object returned by the `transform()` method, is placed into an output buffer based on its run–time type. The final buffer holds the final output items. The resulting pipeline is shown in Figure 3.

The major difference between this State–based pipeline and traditional pipelines is the order in which requests flow through the pipe. In a traditional pipeline, assuming no stages are replicated, the queues are usually FIFO and a stage executes one request at a time. This preserves the order of requests through the pipe. In the State–based pipe, threads can find and execute outstanding requests from any work queue. It is possible for two threads to take two separate items from the same queue but enqueue the results in the opposite order. If request order must be maintained, we need to insert additional mechanisms to enforce it.

However, this ordering is often unnecessary in coarse–grained pipelines. For example, expensive stages can be replicated to improve throughput. Replicated stages do not guarantee FIFO ordering without additional support. Relaxing this unnecessary ordering improves the performance of the pipeline. We will return to this issue in Section 4.2.

4.1. Evaluating the State–based Pipeline

The State–based pipeline addresses the problems that plague traditional implementations: load balancing, ramp–up/ramp–down idle time, and incremental use of additional processors. All of these benefits result from separating the concurrency in the pipeline from the stages.

The improvement in load balancing comes from the ability of a thread to execute any outstanding work item. The execution of the pipeline is essentially a master/slave structure, except that there are multiple work queues from which a slave can take an item. This parallel structure is well–known for its load balancing features if there are a large number of (relatively) small requests. Load imbalance in a traditional pipe occurs because threads are statically assigned to executing a particular stage. If a stage requires more computation than others, there is no way to dedicate additional processors to executing requests. In the State–based pipeline, threads search through the set of queues executing outstanding work items. Over time, expensive stages will build up a large queue of items. The threads will naturally encounter outstanding items for these expensive stages and execute them more often, balancing the load. This same ability to find work anywhere in the pipeline addresses the ramp–up and ramp–down problem.

The improvement in scalability is also the result of the master/slave execution model. In a traditional pipe, the number of threads is equal to the number of stages. To use more processors, a new pipe stage must be introduced. This has an impact on the load balance of the complete pipeline. In the State–based pipe, the number of slave threads is independent of the structure of the pipeline, and can be selected based on available processors. If only a small number of processors are available, the user can use fewer threads than there are stages in the computation. If a large number of processors is available, more threads can be added. These threads will automatically dedicate themselves to expensive stages because of the load balancing properties of the master/slave. Further, the user can create the set of stages that logically match the application structure rather than having the decomposition dictated by hardware considerations.

These benefits come at some cost. Executing a work item creates a new object for the next stage, including instance variables for that object. Instantiating objects can be slow and may require state to be copied between stage

objects. In a multithreaded environment, object instantiation may cause contention for the heap and the memory allocator data structures, inhibiting concurrency. The buffers between stages must be thread–safe, which incurs overhead. Finally, if ordering is required, enforcing it will add run–time overhead.

4.2. Refinements to the State–based Pipeline: Ordering and Buffer Elimination

We can refine the structure of this State–based pipeline by introducing ordering of the work items only where required by the application. Otherwise, enforcing ordering inhibits concurrency and should be avoided. Ordering can be enforced by giving items sequence numbers as they enter the pipeline. A work request can be given to a thread only after its predecessor has completed execution. In some cases, the results of the predecessor are needed in the computation, so access to this item may be necessary.

For stages that do not need ordering, unordered buffers can be placed between stages. However, the overhead of these buffers can be reduced by removing them. Rather than placing the object into a buffer, the thread that just completed the previous state transition continues to execute the next transition (by calling the `transform()` method on the object returned from the previous transition). This process continues until either an ordered stage is encountered or the last transition is executed, at which time the object is placed in the appropriate buffer. Thus, buffers are used only when ordering is needed.

We can reduce the cost of ordered buffers by applying the same basic idea. If a thread is about to place the next object to be executed into an ordered buffer, it executes the next stage transition instead. When finished, it informs the buffer that the next item can be executed. Thus, items are only placed into ordered buffers if they arrive out of order, reducing buffer costs.

4.3. Thread Scheduling in the State–based Pipe

An issue we have not considered is how threads look for outstanding work. The obvious choices are to search forward from the first buffer or search backward from the last.

This choice clearly affects the behaviour of the pipeline. Searching forward from the beginning favours incoming requests at the expense of those currently being processed. With a fixed number of input items, this may be appropriate. However, there can be many partially–processed requests in the pipeline at any time, requiring more memory.

Searching backward from the last buffer favours requests already in the pipeline. If the number of items in the pipe is unbounded (say, an input stream of Web server requests), then this policy, or one like it, must be used.

Other scheduling policies are possible. These could give priority to specific stages, could try to balance accepting new requests against finishing partially–processed ones, or could direct threads directly to bottleneck stages to reduce the cost of searching the buffers for outstanding requests. It may even prove useful to supply a variety of different policies to meet the needs of different applications.

5. Example Applications

To show the advantages of the State–based pipe, this section presents results from an animation program and parallel JPEG compression and encoding. These applications are run on a shared–memory multiprocessor using Java threads. These examples highlight the benefits of the State–based pipe.

5.1. Graphical Animation

This graphical animation example was first used to show the programming model of the FrameWorks (not to be confused with object–oriented frameworks) parallel programming system [11], and was later used as an example in Enterprise [10], DPnDP [12], and Parallel Architectural Skeletons [3]. The application generates a sequence of frames for a computer animation and saves them to disk.

The application has three stages. The Generate stage computes the new location of objects for each frame in the animation. The Geometry stage uses this information to do viewing transformations, clipping, and projection. Finally, the Display stage uses the Geometry data to perform hidden surface removal and anti–aliasing, then saves the image to disk.

This problem can be solved using a pipeline as shown in Figure 4(a). Each frame is an independent work item at each stage, and the output of a stage is input to the next. However, the Display stage has more computation than other stages, unbalancing the pipe. The parallel programming systems above replicate Display, yielding the solution in Figure 4(b). This section compares the replicated solution to a State–based pipe solution to show our improvements.

5.1.1. Implementation and Results The animation example approximates the work for each stage by doing busy work. Generate and Geometry take 1 second and Display takes 2 seconds, and the simulation cre-

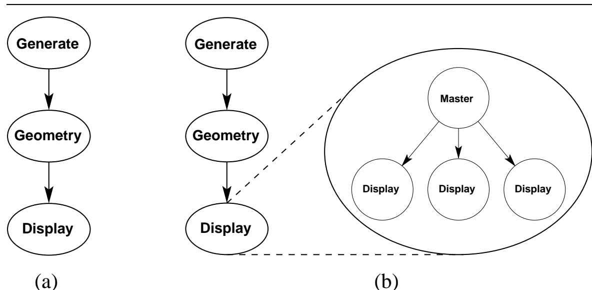

Figure 4. Replicating expensive stages in the animation example [12].

No. of threads	State–based pipe	Traditional pipeline	
	Time	Time	Opt. config(s).
1	241786	N/A	N/A
2	120922	N/A	N/A
3	80594	122612	1-Gen/1-Geo/1-Dis
4	60447	63612	1-Gen/1-Geo/2-Dis
5	48357	63614	1-Gen/1-Geo/3-Dis 1-Gen/2-Geo/2-Dis 2-Gen/1-Geo/2-Dis
6	40298	63614	1-Gen/1-Geo/4-Dis 1-Gen/2-Geo/3-Dis 1-Gen/3-Geo/2-Dis 2-Gen/1-Geo/3-Dis 3-Gen/1-Geo/2-Dis 2-Gen/2-Geo/2-Dis
7	36265	43226	2-Gen/2-Geo/3-Dis
8	32229	33315	2-Gen/2-Geo/4-Dis

Table 1. Results of the graphical animation simulation for 60 frames, in milliseconds.

ates 60 frames. This is not intended as a real-world example, but shows the benefits of the State-based pipe where the execution time can be easily verified. The obvious solution is to merge the first two stages to balance the pipe.

The State–based pipeline used here does not use any ordered stages. We assume that the frames are stored to separate files, or the Display stage would need some ordering mechanism to assemble the complete animation.

The results are shown in Table 1. The decoupling of the concurrency from the pipeline structure can be shown by the ability of the State–based pipe to run with one and two threads. The traditional pipeline cannot do this as it assigns a thread to each stage (though such tests could be done using OS facilities that bind threads to specific processors). Even with two threads, the State–based pipe provides performance benefits which are slightly better than the traditional pipeline without a replicated Display stage.

As the number of threads increases, the State–based pipeline provides linear speedup up to six threads. At seven and eight threads, speedup begins to drop off as the number of frames is no longer evenly divisible by the number of threads and idle time results. However, these results are still optimal for a master/slave with an unbalanced workload. Equally important is that the difference between executions is simply to change the number of worker threads that are created, which can even be set at run–time. Changes to the number of threads do not impact the logical structure of the pipeline.

In contrast, the performance of the traditional pipe is a step function, with large improvements for specific pipeline configurations. With three threads, the speedup is just under two. The longer Display stage is a bottleneck. With four threads, Display can be replicated so its throughput matches that of earlier stages, balancing the pipe. Only ramp–up and ramp–down time overhead remains. Five and six threads cannot further improve performance. If the first two stages are replicated, Display is again a bottleneck. If Display is replicated, the first two stages cannot produce work fast enough to keep it busy. With seven threads, we get an improvement using two threads for both Generate and Geometry and three threads for Display. However, Display is still a bottleneck as earlier stages have higher throughput. Finally, with eight threads, the optimal four-threaded pipe can be replicated, yielding a speedup slightly below linear.

The State–based pipe also has less ramp–up and ramp–down time. For example, with four threads the State–based pipe exhibits perfect speedup. Each of the 60 frames requires four seconds of execution time, and four processors complete the application in 60 seconds. The optimal traditional pipeline, with the throughput of all stages balanced, requires an extra three seconds. The problem is that the last stage, the replicated Display stage, does not work at full capacity until two requests reach it, which takes three seconds. Until then, there are idle threads in the pipeline. This problem becomes more pronounced as the pipeline gets deeper.

Overall, the State–based pipe provides better CPU usage for pipeline problems, using any number of available processors. The traditional pipe, in coupling the concurrency to the structure of the pipeline, offers lower performance.

5.2. JPEG Compression and Encoding

This section presents results of parallel JPEG compression and encoding. This program starts with an input $m \times n$ RGB image (read from a GIF file), compresses it using baseline JPEG compression [5], and stores the compressed image using the JPEG File Interchange Format [4]. Details and initial results can be found in [8];

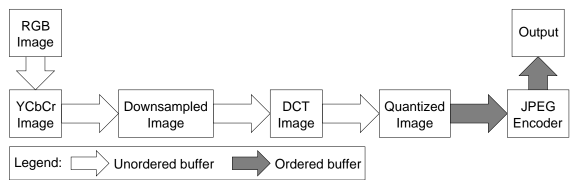

Figure 5. The pipeline for JPEG compression.

JPEG stage	Percentage of exec. time
RGB to YCbCr conversion	45.1%
Downsampling	2.4%
Discrete Cosine Transform	26.7%
Quantization	17.8%
Encoding	8.0%

Table 2. The percentage of the overall execution time for each of the JPEG stages.

the results given here were obtained using newer hardware and a newer version of Java.

JPEG compression and encoding has five steps:

1. Convert the RGB image to YCbCr colourspace. JPEG compression works on a luminance/-chrominance colourspace.
2. Downsample the chrominance components, Cb and Cr. The input $m \times n$ components are replaced with $\frac{m}{2} \times \frac{n}{2}$ components. This step is optional but commonly done in practice.
3. Convert the image from the spatial domain to the frequency domain using a Discrete Cosine Transform (DCT). For this paper, the DCT has two key characteristics. First, it is applied to 8×8 subimages for each component. Second, there are two sets of coefficients for each 8×8 subimage: the *DC coefficient* and the *AC coefficients*.
4. Quantize the DCT coefficients. This process filters out some of the AC coefficients, compressing the image without sacrificing quality. It also works on 8×8 subimages.
5. Encode the quantized 8×8 subimages with variable-length Huffman encoding. The DC coefficient is encoded as the difference of it and the previous coefficient, while the AC coefficients are encoded individually.

5.2.1. Implementing the JPEG Encoder The last three steps in JPEG compression are executed independently on 16×16 subimages of the input image. Though most stages work on 8×8 subimages, recall that downsampling halves the chrominance components, so we need 16×16 blocks from the original. The resulting pipeline, including all necessary ordering, is given in Figure 5.

The stages in the JPEG application are the logical steps in the problem. However, they make poor pipeline stages as they take a variable amount of processing time. Table 2 shows the distribution of processing time between stages. The distribution changes slightly based on image size.

It is important to understand how ordered buffers work in Figure 5. These buffers do not prevent stages from running concurrently. A subimage can be quantized while another is encoded. These buffers have two key characteristics. First, they ensure items exit in the order they entered the pipeline, by attaching sequence numbers to incoming items. Second, they ensure mutually-exclusive execution of the stage transition. This allows an item to access results from the previous item. Note that the ordering does not affect threads putting items into a buffer. Threads can put quantized subimages in the buffer out of order. They can execute other items if no encoding can be done. Ordering only affects threads removing items from a buffer. An item can only be removed if it is next in sequence and no other request item of that same type is currently being executed by any thread. If this is not true, the thread gets no item from the buffer and continues to search the pipe for more work.

For the JPEG application, two stages must be ordered to correctly encode the image. The encoding stage must be ordered because the DC coefficient is encoded as the difference between successive DC coefficients. This data dependency cannot be avoided because JPEG encoding is uses variable–length, so we cannot leave space to insert the necessary bits later without requiring additional, expensive data copying. It is necessary to process the requests in order. The output stage must also be ordered to assemble the bitstream for the JPEG file in the correct sequence. Again, this ordering is necessary as JPEG uses variable–length encoding.

The ordered stages use a concurrent priority queue [7], sorted by the sequence numbers attached to input items. Also, a reference to the last work item processed by an ordered stage is kept, which the next request can access. This allows the DC component to be properly encoded, but means that a work item cannot be processed until its predecessor has finished.

The input to this pipeline is contiguous stripes of 16 rows of the input image. It is also possible to partition the rows provided that the number of columns is a multiple of 16.

For this example, the threads search for work starting

1024 × 896			3212 × 3600		
Slaves	Speedup	Time	Slaves	Speedup	Time
1	0.85	2861	1	0.88	34124
2	1.56	1568	2	1.66	18129
3	2.18	1118	3	2.35	12757
4	2.67	914	4	2.99	10049
5	3.05	801	5	3.54	8472
6	3.34	730	6	4.05	7422
7	3.54	690	7	4.44	6756
8	3.53	691	8	4.70	6386

(a) Results from the State–based pipe.

1024 × 896		3212 × 3600	
Speedup	Time (ms)	Speedup	Time (ms)
1.83	1332	1.82	16491

(b) Results from a traditional pipeline, using five threads.

Table 3. Speedup and wall clock times parallel JPEG encoding with preallocated memory. Times are in milliseconds.

1024 × 896			3212 × 3600		
Slaves	Speedup	Time	Slaves	Speedup	Time
1	0.77	3177	1	0.75	40134
2	1.38	1769	2	1.29	23203
3	1.83	1332	3	1.69	17746
4	2.17	1122	4	2.01	14937
5	2.43	1004	5	2.27	13253
6	2.55	956	6	2.46	12183
7	2.59	943	7	2.61	11494
8	2.54	960	8	2.69	11150

Table 4. Speedup and wall clock times parallel JPEG encoding without preallocated memory for the State–based pipeline. Times are in milliseconds.

at the last stage and moving backward. This gives priority to the encoding stage over processing new input. If threads search in the opposite direction, parallelism will be limited at the end of the program. This gives priority to new work, which will queue up at the encoding stage. Work will begin on these items only after the input queue is drained. Since the last stage is ordered, only one request can be processed at a time, reducing parallelism and performance. Thus, the scheduling policy is key to good performance and must be chosen carefully.

As expected, the number of threads in the State–based pipe is independent of the logical structure of the stages. We only have to consider the number of available processors.

5.2.2. Performance Results The wall–clock time and speedups for the parallel JPEG program are given in Table 3. These results were obtained using Java 1.4.0 on a Sun V880 server running Solaris 5.8, with eight 900 MHz Sparc v9 processors and 16 GB of memory. Data is provided for both State–based and traditional pipelines. These results are from versions of the program that preallocate memory. The performance impact of memory allocation is discussed later in this section.

The time includes JPEG compression, encoding, and writing the output JFIF file. The times for reading the input GIF file and memory preallocations are not included. For fair comparisons, the sequential version also preallocates memory. Though the sequential version does not contend for the heap, the stages store image data in arrays. Java arrays must be initialized when they are created, consuming CPU time. Including array initialization would skew the results.

The State–based pipe outperforms the traditional pipe when using more than two processors. Given the varying computation time among stages, this is not surprising. The most expensive stage takes almost half of the processing time, limiting the speedup to just over 2. The least expensive stage takes 2.4% of the time. Fixing this imbalance requires stages be combined. The State–based pipe parallelizes JPEG compression without changing the logical structure despite the varying execution times.

Also, adding more threads to the State–based pipe almost always improves performance. We add these threads by simply creating them; the logical structure of the stages is unchanged. Thus, finding the optimal number of threads is straightforward. In contrast, the traditional pipe would need to reorganize the computation among more stages to use more processors. Performing this reorganization while maintaining the balance of the pipeline is a difficult task. Also, note that when the State–based pipe uses the same number of threads as the traditional version (five threads), its performance is better. This suggests that threads are being used more effectively in the State–based pipe.

One weakness of the State–based pipeline is the need to create an object for each request at each stage. Object creation introduces contention for the heap. Also, in this application, array initialization for stage data is a problem.

The results for the JPEG example where memory is not preallocated in the State–based pipeline are in Table 4. Memory allocation overhead grows with image size. For the small image, the overhead is 39% with eight workers, but is 75% for the larger image.

6. Related Work

The pipeline can be written as a design pattern [14]. One change to the basic pipe is separating data flow from control flow. While data flows in the same direction, the flow of control may have a stage *push* data to the next

or have the next stage *pull* data from the previous. However, this pattern is a traditional pipeline that uses objects as stages.

The State–based pipeline bears a strong resemblance to SEDA for highly–concurrent, scalable Web services [15]. Both SEDA and the State–based pipe represent computations as stages separated by buffers, which decouples the concurrency from the stages. Both use this decoupling to self–tune applications to improve performance. The State–based pipe uses the master/slave structure to balance the computational load from a set of uneven stages over a fixed set of threads. SEDA uses threads to process server calls, hide blocking system calls, and provide concurrency for computational tasks. SEDA monitors the conditions at each stage to adjust the number of threads to improve throughput and response time. SEDA also allows stages to share a common thread pool, resulting in an organization like the State–based pipeline. Unlike this pipeline, SEDA creates and destroys threads as the server load changes. As well, SEDA batches items in the queue between threads, delivering several items at once. The State–based pipe provides global scheduling and control for computational tasks. However, we may still benefit from the scheduling work in SEDA.

One of the problems with the State–based pipe is the need to create new stage objects for each request. In multithreaded systems, this can lead to contention for the heap and allocator data structures. Multithreaded memory allocators like Hoard [1] remove some contention by using thread–local allocation structures and memory regions. However, these allocators do not remove Java array initialization.

7. Conclusions and Future Work

The pipeline is a simple and intuitive parallel structure that can speed up many problems. Unfortunately, it is difficult to use for three reasons. First, it is very sensitive to load imbalances. Second, at the beginning and end of a computation, stages are idle. Third, the stages define the concurrency, making it difficult to add additional processors.

This paper introduced a new pipeline formulation, the State-based pipe. The State-based pipe rethinks the pipeline in an object-oriented way, and considers the computation as a series of state transitions using the State pattern. This allows pipeline computations to be executed using a master/slave structure that solves or reduces the three problems. We showed this improvement using two example programs.

Future work for the State–based pipe includes investigating scheduling options, further investigating its strengths and flexibility, and automating its use with a design–pattern–based parallel programming system.

Scheduling in this pipeline is simple but still yields improvements over the traditional pipeline. This is encouraging, though scheduling may still be improved using mechanisms from SEDA or elsewhere. Improved scheduling algorithms may reduce the overhead of searching the work queues, instead directing threads immediately to bottleneck stages. One unexplored option that may help scheduling is to include some unordered buffers to reduce the granularity of work in the pipeline, rather than removing all unordered buffers. If multiple scheduling algorithms are needed to support different application characteristics, then it would be useful to develop some guidelines for selecting the most appropriate algorithm. Regardless, better schedulers will only improve the State–based pipeline further.

This paper has demonstrated that the State–based pipeline works well for applications where the stages are imbalanced. The example applications are imbalanced, but the imbalance is static over the entire execution of the problem (though it may vary based on problem input). We also believe that this pipeline will work well for problems where the execution times of the pipeline stages varies as the application runs. However, we have not run any example programs with this characteristic.

As well, the State–based pipe may be more flexible than some of its counterparts. In most pipelines, requests can only flow from the start to the end. With the State–based pipe, each stage creates an object for the next stage. This is normally the next logical stage in the application. However, it is possible to create an object for any stage. This ability could prove useful for pipeline problems that use iterative refinement. A stage transformation could either create an instance of an earlier logical stage to continue refinement or create an instance of the next logical stage when the data has converged. Again, we have not implemented any examples that require this capability.

The State–based pipe is suitable for an object–oriented framework or for use in a pattern–based parallel programming system like CO_2P_3S [8, 9]. The basic pipeline structure is application–independent, with only a small amount of glue code changing between uses, making it ideal for a framework. The CO_2P_3S system can generate frameworks, including glue code, based on options specified in a user interface. The user is exposed only to a set of methods that must be implemented to complete the application, with the parallel structure hidden during initial development. The latest version of CO_2P_3S generates both shared–memory and distributed–memory Java

frameworks for the patterns it supports [13], so a distributed version of the State–based pipe would also need to be developed.

Acknowledgments

This research was supported by the Natural Science and Engineering Research Council of Canada, the Alberta Research Council, and Alberta's Informatics Circle of Research Excellence. We would also like to thank Tim Brecht for his insightful comments on an early draft of this paper, and the anonymous referees for their suggestions.

References

[1] E. Berger, K. McKinley, R. Blumofe, and P. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *Proceedings of ASPLOS-IX*, pages 117–128, 2000.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object–Oriented Software*. 1994.

[3] D. Goswami, A. Singh, and B. Preiss. From design patterns to parallel architectural skeletons. *Journal of Parallel and Distributed Computing*, 62(4):669–695, 2002.

[4] E. Hamilton. *JPEG File Interchange Format, Version 1.02*, 1992. Available at http://www.w3.org/Graphics/JPEG.

[5] ISO/IEC JTC1 SC29 Working Group 1 (Joint Photographic Experts Group). *Digital Compression and Coding of Continuous–Tone Still Images, Part 1: Requirements and Guidelines, ISO/IEC International Standard 10918–1*, 1992.

[6] R. Lavender and D. Schmidt. Active object: An object behavioral pattern for concurrent programming. In *Pattern Languages of Program Design 2*, pages 483–499. 1996.

[7] D. Lea. *Overview of package `util.concurrent` Release 1.2.6*, 1999. Available at http://gee.cs.oswego.edu/-dl/classes/EDU/oswego/cs/dl/util/concurrent/intro.html.

[8] S. MacDonald. *From Patterns to Frameworks to Parallel Programs*. PhD thesis, Department of Computing Science, University of Alberta, 2002.

[9] S. MacDonald, J. Anvik, S. Bromling, J. Schaeffer, D. Szafron, and K. Tan. From patterns to frameworks to parallel programs. *Parallel Computing*, 28(12):1663–1683, 2002.

[10] J. Schaeffer, D. Szafron, G. Lobe, and I. Parsons. The enterprise model for developing distributed applications. *IEEE Parallel and Distributed Technology*, 1(3):85–96, 1993.

[11] A. Singh, J. Schaeffer, and M. Green. A template–based approach to the generation of distributed applications using a network of workstations. *IEEE Transactions on Parallel and Distributed Systems*, 2(1):52–67, 1991.

[12] S. Siu, M. D. Simone, D. Goswami, and A. Singh. Design patterns for parallel programming. In *Proceedings of PDPTA'96*, pages 230–240, 1996.

[13] K. Tan, D. Szafron, J. Schaeffer, J. Anvik, and S. MacDonald. Using generative design patterns to generate parallel code for a distributed memory environment. In *Proceedings of PPoPP 2003*, pages 203-215, 2003.

[14] A. Vermeulen, G. Beged-Dov, and P. Thompson. The pipeline design pattern. In *Proceedings of OOPSLA'95 Workshop on Design Patterns for Concurrent, Parallel, and Distributed Object-Oriented Systems*, 1995. http://www.cs.wustl.edu/∼schmidt/OOPSLA-95/-index.html.

[15] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *Proceedings of SOSP-18*, pages 230–243, 2001.