Transforming an Object-Oriented Pipeline to a Master-Worker: The State-Based Pipeline

Steve MacDonald
School of Computer Science, University of Waterloo, Waterloo, Ontario, CANADA

1 Problem

An algorithm takes a sequence of data objects through a series of transformations. Traditionally, the concurrency in such an algorithm is exposed using a *pipeline* [5, 6, 7]. The pipeline is a conceptually simple parallel structure that can be used to speed up many problems. It is usually taught to novice parallel programmers early in their education.

However, expert parallel programmers typically eschew using the pipeline structure, especially for coarse-grained applications. The pipeline suffers from three serious problems that make an efficient and flexible implementation difficult:

1. Processors are idle when the pipeline is not full. This happens when the pipeline starts filling up (*ramp-up* time) and when there are no more input requests and the pipeline is emptying (*ramp-down* time).

2. Traditional pipeline implementations are sensitive to load imbalance. Obtaining the best possible performance requires that the computation for each stage be perfectly balanced. Otherwise, stages that require less computation will be idle, waiting for work from more expensive stages.

3. Traditional pipeline implementations tightly couple the concurrency with the computational stages. Each thread is assigned to execute a specific part of the computation. This makes it difficult to incrementally add new processors while still maintaining the balance among stages for good performance.

The problem addressed in this pattern is how to formulate a more efficient pipeline implementation that either reduces or eliminates these problems. In doing so, programmers may be more willing to use pipelines to solve their problems. They can take advantage of conceptual simplicity of this common parallel structure rather than being forced to rethink the problem because of the limitations of pipeline implementations.

Note that this pattern does not repeat the motivation and description for the pipeline in general. This is covered by other descriptions of the pipeline pattern, notably [5, 6, 7]. The contribution of this pattern is to outline a more efficient implementation of this common parallel structure. A more efficient implementation may make expert parallel programmers less reluctant to use pipelines.

2 Context

2.1 A Traditional Pipeline

A traditional implementation of a pipeline is shown in Figure 1. The pipeline is an ordered set of stages, where each stage takes data from its predecessor, transforms the data, and sends it to the next stage. Data is normally transferred using buffers inserted between stages.

The key characteristic of a pipeline computation is that each stage performs an independent computation on its data. Thus, parallelism is achieved by having stages execute their computation on different parts of the data simultaneously. Each stage can be assigned to a separate physical processor. This allows a stream of input data to be processed efficiently.

In the ideal case, the speedup from a pipeline is equal to the number of stages. However, achieving this ideal in practice is difficult for two reasons.

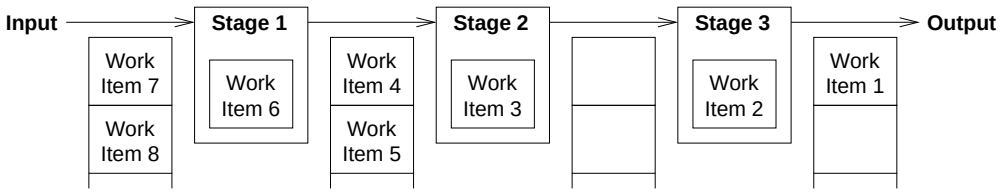

Figure 1: A traditional implementation of a pipeline.

First, a traditional pipeline suffers from ramp-up and ramp-down time. When a pipeline computation first starts, all requests are in the input queue. As they enter the pipeline, the first stages are busy but later stages are idle as requests have not reached the end of the pipeline yet. It is not until the pipeline fills that all stages are busy. A similar situation happens when the input queue is exhausted; the stages at the beginning of the pipeline are idle while the requests are being completed by the later stages. The performance lost to this problem depends on the depth of the pipeline and the computational cost of the stages.

Second, the traditional pipeline is sensitive to load imbalances. Imbalance in the amount of computation for the stages will cause some stages to idle waiting for work from more expensive stages. More expensive stages can overflow the buffers between stages, which may require their predecessor stages block rather than place items into a full buffer. This idle time reduces the number of threads that are working on requests, which lowers the performance of the pipeline. As an example of this problem, consider the first stage in Figure 1. The computation for this stage is less expensive than the others; it is already working on the sixth input item where the second stage is still processing the third. If this continues, the first stage will quickly exhaust the input queue and will then be idle and not contribute to the solution. This imbalance can be addressed by replicating expensive stages to improve their throughput. However, replicating stages introduces two new problems. First, when a stages is replicated, we lose ordering of work items in the pipeline. With FIFO queues and no replication, items will always be processed in order at every stage (including output) in Figure 1. With a replicated stage, there is a race condition between replicas that can change the order. The second problem with replicating stages is that it is difficult to balance the pipeline, particularly if the stages require variable amounts of computation. Balancing the pipeline requires the correct ratio of replicas for each stage so each has the same throughput. With a small number of threads, it may be difficult or even impossible to get the correct ratio without refactoring application code across the stages.

In addition to performance problems, the pipeline can be an inflexible parallel structure that is difficult to extend. This problem results from a combination of load imbalance sensitivity and the tight coupling between the stages and the concurrency. Each thread can only execute the code for a specific stage and these stages must be computationally balanced. This makes it difficult to incrementally add more threads to get better performance. To add a new thread we must add a new stage to the pipeline. This can be done in one of two ways. First, we can replicate an existing stage. This has all of the problems identified above. Second, we can refactor the computation to use more stages. However, it may be difficult for this refactoring to produce a new set of balanced stages.

A related problem is how to change the computation during maintenance. Adding new functionality or removing unnecessary functionality alters the stage balance. As we've already demonstrated above, rebalancing the computation can be a daunting task.

2.2 An Object-Oriented Pipeline

In many cases, an object-oriented version of a pipeline simply replaces stages with Active Objects [3]. An Active Object is an object that also has its own thread. All method invocations are executed with the internal thread rather than the caller's thread. The more complex pipeline in [7] decouples the flow of data from the flow of control. Data in a pipeline always flows from start to end, but a given stage may *pull* data from its predecessor or may *push* data to its successor.

Neither of these object-oriented pipelines addresses the problems with the traditional pipeline. They suffer from ramp-up and ramp-down time, balance is still essential for performance, and the concurrency is still tightly coupled with the stages. Though the pipeline stages are objects, the basic structure is identical so the problems remain.

3 Forces

The pipeline is a simple and intuitive way to speed up the solution of many problems. This common structure is usually taught to parallel programmers early in their education. It can also be simple to implement.

However, traditional implementations of the pipeline are fragile. Ramp-up and ramp-down time result in idle threads at the beginning and at the end of a computation, reducing performance. More problematic is that the pipeline is sensitive to load imbalances; the amount of computation must be equal in all stages for optimal performance. Furthermore, given the need for balance and tight coupling between stages and threads, extending the pipeline (to either add more threads or more functionality) is problematic.

Expert parallel programmers avoid the pipeline, especially for coarse-grained applications, for these reasons. Instead, they recast the problem as an instance of a more efficient parallel structure, like the Master-Worker.

4 Solution

4.1 Overview of Solution

There are two key elements to the creation of an efficient, object-oriented pipeline.

The first element is recasting the pipeline computation from a sequence of transformations on input data to a sequence of state transitions. The pipeline starts with objects in an initial state in the input queue and ends with objects in their final, output state. The transitions between states are handled using the State design pattern [2], described briefly in Section 6.1. The key to the State pattern is that the state objects implement the transition, obviating the need for the explicit stages in the traditional pipeline implementation. This characteristic can be used to decouple the concurrency from the logical structure of the pipeline computation, allowing the threads to execute any state transition for any object.

The second element is to take advantage of the above decoupling and execute outstanding pipeline requests using the Master-Worker parallel structural design pattern. This pattern is described briefly in Section 6.2. This structure is well-known for its ability to balance computational load across processors. It also decreases ramp-up and ramp-down time and allows the user to select the number of threads based on available processors rather than based on the structure of the pipeline computation.

The end result of applying these two concepts is the *State-based pipeline*. This section describes the solution in more detail, including additional concerns related to pipeline request ordering (where necessary), scheduling of requests to threads, and issues with respect to memory use.

4.2 Applying the State and Master-Worker to Pipeline Computations: The State-based Pipeline

In a traditional pipeline, each stage receives data from its predecessor, transforms it, and sends the results to its successor. The transformation may not be a simple refinement of the input data, but may change its size or type. While we could have one large data structure that holds fields for all data needed by all stages, this would waste memory. Instead, a stage consumes its input type and produces its output type.

An object-oriented pipeline replaces the request items with request objects. A stage accepts an input object and transforms it to an output object. To conserve memory, we use different types for the request objects.

This highlights the transformational nature of a pipeline. If we instead consider the request objects to be the states of an entity, the each stage implements a transition from its input objects to its output objects. The pipeline is a composition of state transitions, starting with objects in an input state, taking these objects through a series of intermediate states, and producing a set of objects in the output state.

In this description, we still have stage objects implementing the necessary transitions. Here, we apply the State design pattern with the context object. Now the request objects in the pipeline provide methods for transitioning from one state to the next. As a result, we no longer need explicit stage objects as the request objects contain all of the necessary transition code. However, the stages also provided a thread of control to execute the transition. We still require these threads to obtain parallelism in the pipeline.

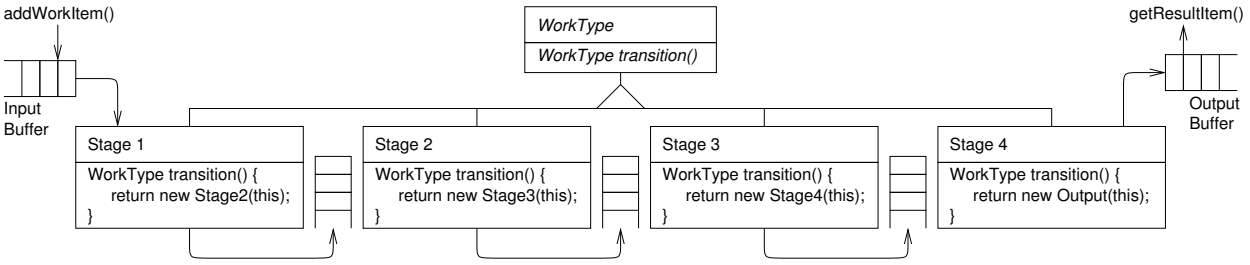

Figure 2: A pipeline based on the State design pattern.

One alternative is to use the threads as replacement for the stage objects. The threads invoke the transition methods on input objects and output successor objects. However, this is a simple refactoring of the original design that solves none of the associated problems.

Instead, we add another element to the design. Each request object in the pipeline should provide a single polymorphic method, `transition()`, that implements the transition from the current state and returns the next state. A typical implementation of this method would be to construct an instance of the next state using the current state as a constructor argument. The next state then initializes itself by obtaining and transforming the state of its predecessor.

The result of using this single polymorphic method for state transitions is that the threads do not need any state-specific information to execute a transition. Each state is a self-contained object that responds to the same method. A thread can take any request object in any state and execute the transition.

With this independence between the concurrency and pipeline transitions, it is possible to use a Master-Worker parallel structure to execute pipeline requests. Requests are held in buffers, one per state, much like the buffers in Figure 1. Threads search these buffers looking for outstanding requests. The request is executed and the result is placed into the appropriate output buffer based on its run-time type. The final buffer holds the result items. The resulting pipeline, the State-based pipeline, is shown in Figure 2.

The State-based pipeline addresses the problems with traditional implementations: sensitivity to load imbalance, idle time during pipeline ramp-up and ramp-down, and the inability to incrementally use additional processors.

The base cause of all of these problems in the tight coupling between the stages in the pipeline and the concurrency. The State-based pipeline separates these two concerns. The stages are now implemented through state transition methods in the request objects themselves, so the stages do not exist as separate entities any longer. Since each request object responds to the same transition method, the threads need no state-specific information to execute a transition method. A thread can take any request in any state and execute the next transition with a single polymorphic method call.

This separation allows the pipeline requests to be executed using a Master-Worker parallel structure with multiple work queues. A thread can take any item from any queue and execute the next transition. The Master-Worker structure is well-known for its ability to balance computational load across a set of heterogeneous workers if there are a large number of (relatively) small requests.

Load imbalance in a traditional pipeline results from the static assignment of threads to stages. If a stage requires more computation than another, there is no easy way to dedicate more processors to that stage except by assigning multiple threads, which incurs additional load balancing concerns. In the State-based pipeline, the threads search the set of work queues looking for outstanding requests. If a part of the pipeline requires more computation, it will naturally build up a large queue of requests. The threads will find work and execute requests in these expensive queues more often, balancing the load.

This same load balancing reduces the ramp-up and ramp-down time. At the beginning of a pipeline execution, all requests are in the input queue. The searching threads will find the requests there, rather than waiting for requests to traverse the pipeline. The reverse situation happens at the end with ramp-down time.

The inability of the traditional pipeline to incrementally use more threads again stems from the coupling of concurrency to stages. To use more processors in a traditional pipe another stage must be introduced, which introduces the load imbalance problems discussed earlier. In the State-based pipeline, the number of threads is independent of the set of stages that make up the computation. We can select this number based solely on the availability of processors. If a small number of processors is available, then there may be fewer threads than stages. If a larger number of processors is available, there may be more threads than stages. In both cases, the threads will balance the load among them using

the Master-Worker structure.

With the State-based pipeline, the set of stages in a pipeline application can reflect the logical structure of the problem being solved but can still expect performance improvements. In contrast, a traditional pipeline must decompose the problem based on load balance concerns and processor availability, which leads to fragile code with a (possibly) nonintuitive functional decomposition across stages.

4.3 Request Ordering

One important difference between the State-based pipeline and a traditional pipeline is the order in which requests flow through the stages. In the traditional pipeline, with no replicated stages, the requests exit in the same order in which they enter if all buffers are FIFO. The State-based pipeline cannot guarantee this ordering. However, this ordering is often unnecessary in coarse-grained applications. Enforcing ordering where it is not necessary reduces concurrency. For example, expensive pipeline stages are often replicated, which cannot guarantee FIFO ordering of requests. Relaxing this ordering will improve performance.

There are only two times when ordering in a pipeline may be necessary. First, the order in which the output items are returned to the application may be important. That is, the output stream may need to have the same order as the input stream. Second, there may be dependencies between items at a given stage. This form of ordering is not normally considered in a pipeline, as the pipeline assumes independent requests. The State-based pipeline can accommodate these dependencies with ordered buffers. A request can only be executed after its predecessor has completed. To maintain the dependency, the pipeline can provide a reference to the previously-executed request so that the next one can obtain the dependent data.

For states that do not need any ordering in their execution, we can use unordered buffers. However, the use of these buffers adds overhead to the processing time for requests, which includes not just enqueuing and dequeuing time but also blocking time due to synchronization to handle concurrent accesses. This overhead can be reduced by removing the buffers entirely. When a thread completes the execution of a request, it can check to see if any ordering is required (which can simply check to see if there is a buffer for the new item). If not, the thread can invoke the transition method on the new item by calling the `transition()` method. This process continues until either an ordered stage is found or until the last transition in the pipeline is executed, at which time the request is enqueued in the appropriate buffer. Thus, buffers are only used when ordering is needed.

An ordered buffer can be implemented using a priority queue that tracks which item is next to execute. A thread polls this priority queue and only receives an item if the next one to run is at the head of the queue. Otherwise, no item is returned. This priority queue also maintains a reference to the previously-executed request. This buffer must also ensure that it does not return an item to a thread before the previous item has completed its transition. This can be done by incrementing the next item to execute at the end of the state transition. Note that ordered buffers do not prevent states from running concurrently, but rather ensure that only one thread may be executing a request from the ordered buffer at any given time. Other threads may be executing other outstanding pipeline requests.

In order to implement the request tracking needed in the priority queue, each input item should have a sequence number attached to it. These sequence numbers must be preserved as the request proceeds through its state transitions.

There may be cases where it would be desirable to have explicit unordered buffers. These buffers would increase the number of outstanding requests in the pipeline for threads to find.

Note that if the pipeline never required any ordering of requests, then we could remove all but the input and output buffers. The pipeline then becomes a simple Master-Worker, like that described in [6].

We can apply the same technique to the processing items in ordered buffers to reduce the cost of ordering requests. When an ordered stage is encountered, the thread can check to see if the item it is processing is next to be executed. If so, the thread again continues with the next state transition, informing the buffer when the transition is completed so that the next item can be made available. Now, items are only enqueued if they arrive out of order, which reduces buffering costs further.

4.4 Scheduling Requests

An important performance issue for the State-based pipeline is scheduling, or how threads search for outstanding work in the pipeline. If the State-based pipeline were a simple Master-Worker structure, scheduling would simply be to take any item out of any work queue. We could even consider creating a single work queue to simplify the task. However, ordered stages make the scheduling problem more difficult.

The original description of the State-based pipeline suggested some simple scheduling schemes that were based on polling the set of buffers for outstanding work. Two basic approaches are obvious: forward polling (start at the first stage polling forward), or backward polling (the reverse).

Forward polling is only useful if there are no ordered stages in the pipeline. This scheme favours input requests over those already in the pipeline. Requests will queue up at the first ordered buffer until the entire input queue is drained. At this point, the ordered state transition must be executed serially, resulting in idle threads.

Backward polling favours requests that are already in the pipeline. This is necessary if the number of requests in the pipeline is unbounded, as might be the case for a pipeline in a Web server. This is the scheduling policy that was used for the JPEG example in [4], where the last two stages in the pipeline were ordered.

Note that these scheduling policies work best if the unordered buffers are removed. If unordered buffers are also included, it may be necessary for the scheduler to be more complex. Section 7.2 contains more details.

An important consideration with scheduling for the State-based pipeline is that the requests must be executed in the correct order, even from unordered buffers, to achieve good performance. Specifically, the requests must be executed from first to last, meaning that the buffers are likely FIFO. While LIFO buffers would increase potential concurrency (because they can allow one thread to add a request while another removes one), the requests exit the LIFO buffer from last to first. When an ordered buffer is encountered, it will have to wait for the preceding LIFO buffer to empty before getting the first request to execute. From that point on, the ordered buffer can only execute requests serially. This need for ordering also makes it difficult to use a single work queue for the State-based pipeline.

4.5 Memory Issues

Executing a state transition requires a new object representing the next state to be created. Instantiating these objects can be slow, especially in a multithreaded environment where there may be contention for the heap or other memory allocator data structures.

There are two possible solutions to this problem. The first solution is to use a concurrent memory allocator (such as Hoard [1]) to reduce heap contention.

The second solution is to preallocate memory. This approach was used in the JPEG encoding example in [4]. It works best when the number of requests and the size of the data inside a pipeline request can be predetermined. We can reduce the amount of memory that needs to be preallocated by considering the ordering characteristics of the state. For unordered states with no explicit buffer, we need one preallocated state per thread. Since the item is never enqueued, it is not possible for more than this number to exist. If an explicit buffer exists, then we may need one preallocated instance for each input item.

Another problem in languages like Java is that the contents of a newly allocated object must be initialized to a specified value. This initialization requires CPU time that must also be considered. For instance, a Java integer must be initialized to zero. If an array of integers is created, it must be zeroed out. The cost of this initialization depends on the size of the array. This problem can only be fixed at the compiler or run-time system level and not at the application level.

This contention and initialization can result in serious performance problems. For JPEG encoding, the performance was reduced by between 39% and 75% depending on the size of the input image [4].

5 Known Uses

The initial description of the State-based pipeline in [4] included results from two examples, a simulated graphical animation system and a parallel JPEG encoder. These results were contrasted with a traditional implementation of a pipeline.

Both applications showed the separation between the concurrency and number of stages in that both ran with any number of threads (up to the eight available processors on the hardware used to run the experiments). In contrast, the traditional pipeline was constrained in how it could use processors. In particular, for the graphical animation system, the throughput of the pipeline could only be balanced with four and eight threads; any other number resulted in bottlenecks and resulted in little, if any, performance benefits.

The JPEG encoder showed a real-world example of a problem where the logical stages had poor load balancing characteristics. Of the five stages in the application, one took 45% of the processing time and another only 2.4%. The State-based pipeline with five processors achieved a speedup of 3.54 where the traditional pipeline could only achieve

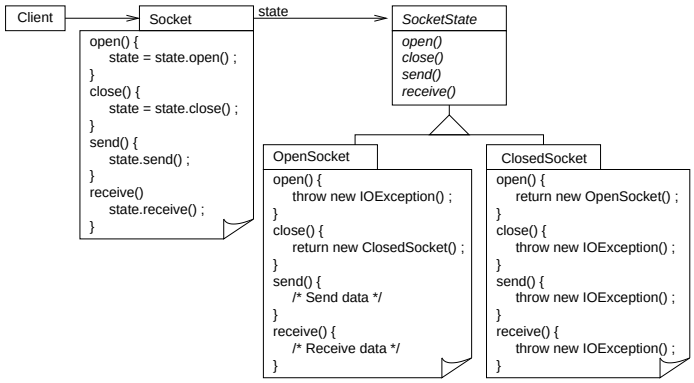

Figure 3: The State design pattern, using a socket that can be open or closed.

a speedup of 1.82. With eight processors, the State-based pipeline achieved a speedup of 4.7. An important feature of the solution is that it used the poorly balanced logical stages yet still provided respectable performance benefits.

6 Related Patterns

The State pattern [2] forms the basis of the State-based pipeline, recasting the pipeline computation as a sequence of state transitions. The particular State variation includes an extra *context object* to isolate the state transitions from clients of the pipeline. A brief description of the State pattern is given in Section 6.1.

The State-based pipeline uses the Master-Worker design pattern [6] to execute requests. It is the use of this parallel pattern that yields the improvements in the pipeline. A brief description of the Master-Worker is given in Section 6.2.

This pattern describes an implementation improvement over the Pipeline design pattern. The motivation for the pipeline as a parallel design pattern is outlined in [5, 6, 7]. In both of these pattern descriptions, the pipeline is transformed to the object-oriented domain by simply changing the stage computations to objects, which does not improve the implementation. In [7], the flow of control is separated from the flow of data, but this does not alleviate the problems with the pipeline.

6.1 State Design Pattern

The State design pattern deals with entities that can be in one of several different states [2]. The actions that the entity takes in response to a method call depends on its current state. For example, consider a simple socket that can be either open or closed. Sending data through an open socket puts the data on the network, but sending data through a closed socket raises a run-time error.

An object representing this kind of entity needs to be able to change its behaviour based on its current state. This could be accomplished using control statements in any state-dependent code. However, this solution makes maintaining the code more difficult. New states or state transitions require all state-dependent methods to be checked and possibly changed. A programmer could easily miss a case and introduce an error.

The State design pattern isolates each state of the entity into a separate class. The implementation of the methods is determined by the class, which represents the associated state. A simple example of the State pattern, based on the socket example, is shown in Figure 3. The socket may be open or closed, represented by the `OpenSocket` and `ClosedSocket` classes. In the `OpenSocket` class, the `send()` method forwards data through the socket. The same method in the `ClosedSocket` class throws a run-time exception. This version of the State pattern includes an extra object called a *context object*, an instance of the `Socket` class. The context object isolates users of the socket from the state transitions that can occur.

The benefit to the State pattern is its separation of concerns, specifically that it separates the code for each state into separate classes. The methods in these classes do not need to check the current state to determine the correct behaviour, simplifying the code. This separation also allows each state to be examined and maintained individually.

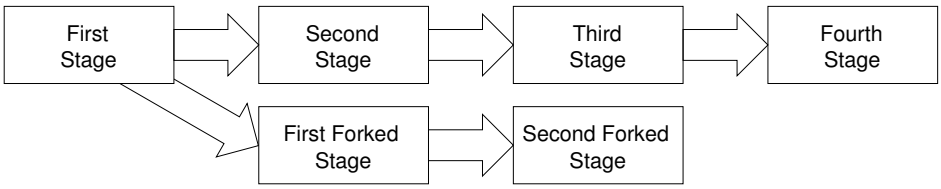

Figure 4: A forked pipeline.

Furthermore, adding new state transitions is easier; on a transition, a state creates an instance of the next state and updates the context object.

6.2 Master-Worker Design Pattern

The Master-Worker parallel structure is well-known for its load balancing characteristics when there are a large number of (relatively) small requests. In particular, this structure is useful when the amount of computation for a request varies but cannot be predicted in advance.

The basic structure consists of two threads: a *master* that produces requests and consumes results and a group of *workers* that execute requests and produce results. Communication between these two can be done through a shared buffer. More complicated Master-Worker structures allow workers to produce new requests, or use multi-level buffers to reduce contention [6].

The key to the load balancing characteristics of the Master-Worker is the dynamic assignment of requests to idle workers. In this structure workers execute at their own speed, taking and processing requests when they need more work. With a large enough number of small requests, the workers will normally finish at approximately the same time even if the requests require different amounts of computation, keeping idle time to a minimum. Of course, this depends on the specifics of the workload.

7 Possible Extensions

7.1 Alternative Pipeline Structures

A traditional pipeline is a simple, ordered list of stages. We believe (but have not demonstrated) that the State-based pipeline allows for even more flexibility in constructing pipelines that match the logical structure of the application. There are three extensions that we will consider: forking pipelines, pipeline cycles, and dynamically imbalanced computations. Finding applications that need these alternative structures and showing that the pipeline solutions can be constructed and execute efficiently is future work.

Since each stage in the State-based pipeline returns an instance of the next state, it is possible for a stage to return one of several possible next states. Such a stage could be said to *fork* the pipeline, as shown in Figure 4. In simple cases, where the number of stages and amount of computation for the two branches are identical, a forked pipeline can be simulated by simply having the request indicate which branch it is associated with and having each stage apply the correct transformation. This would not work well in Figure 4 because the bottom branch is shorter. In the State-based pipeline, the fork is simply a matter of returning the next desired state for the input request. However, a forked pipeline does make ordered stages difficult as the item numbers are assigned consecutively to input items. To have an ordered stage in a forked pipeline, a secondary item number will be necessary.

This same technique can be used to create *pipeline cycles*, shown in Figure 5. A stage can return an instance of a previous stage or the next one. This capability can be used to implement pipeline applications that require some form of iterative refinement, where a pipeline request may have to be processed multiple times until it converges to a final value. Like a forked pipeline, ordered stages within a cycle can cause problems.

The JPEG example discussed in Section 5 highlighted the ability of the State-based pipeline to efficiently run pipeline programs where the stages do not have the same computational requirements. However, while the JPEG stages are imbalanced, the imbalance is static for a given problem instance. That is, the distribution of computation across the stages remains fixed for the entire run of the application. We believe that the State-based pipeline will also efficiently run applications where this imbalance changes over time. One example of such an application would be

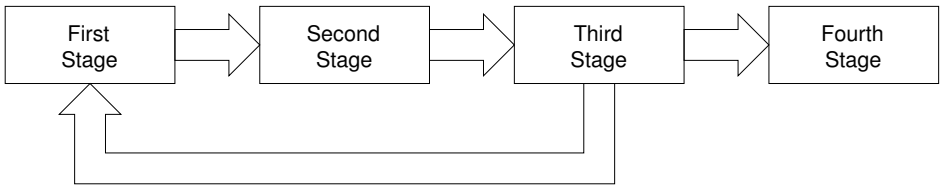

Figure 5: A pipeline cycle.

JPEG encoding where the input RGB is read from the network rather than disk. In this case, the size of the input subimages may differ depending on how much data was read from the network. Another example application would be JPEG encoding with an embedded thumbnail image, which may be used in image browsers as a preview. The embedded thumbnail is itself a JPEG image on a heavily downsampled version of the original image, which could be encoded using the same pipeline. Thus, the sizes of the requests will differ depending on whether they are for the thumbnail or for the larger output image. These two different images in the same pipeline would result in a dynamic imbalance.

7.2 Scheduling

The current scheduling schemes seem to work best if the unordered buffers are removed. If unordered buffers are included, then following a strict forward or backward polling scheme may not be a good strategy.

Intuitively, the problem is that the ordered stages need to be given priority. When a request is available for execution at an ordered stage, it should be executed in preference to requests at unordered stages. Otherwise, requests build up in ordered stages and then can only be executed serially, reducing performance.

This scheduling problem is the topic of ongoing research.

References

[1] E. Berger, K. McKinley, R. Blumofe, and P. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 117–128, 2000.

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

[3] R. Lavender and D. Schmidt. Active object: An object behavioral pattern for concurrent programming. In J. Vlissides, J. Coplien, and N. Kerth, editors, *Pattern Languages of Program Design 2*, chapter 30, pages 483–499. Addison–Wesley, 1996.

[4] S. MacDonald, D. Szafron, and J. Schaeffer. Rethinking the pipeline as object–oriented states with transformations. In *Proceedings of the 9th International Workshop on High–Level Parallel Programming Models and Supportive Environments (HIPS 2004)*, pages 12–21, 2004.

[5] B. Massingill, T. Mattson, and B. Sanders. Some algorithm structure and support patterns for parallel application programs. In *Proceedings of the Ninth Pattern Languages of Programs Workshop (PLoP 2002)*, 2002. Available on-line at http://jerry.cs.uiuc.edu/∼plop/plop2002/proceedings.html.

[6] T. Mattson, B. Sanders, and B. Massingill. *Patterns for Parallel Programming*. Addison–Welsey, 2004.

[7] 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. Available on-line at http://www.cs.wustl.edu/∼schmidt/OOPSLA-95/index.html.