A Visual Notation for Processor and Resource Scheduling

Christopher T. Johnston, Paul Lyons and Donald G. Bailey
Institute of Information Sciences and Technology
Massey University, Palmerston North, New Zealand,
c.t.johnston@massey.ac.nz, p.lyons@massey.ac.nz, d.g.bailey@massey.ac.nz

Abstract

Scheduling of concurrent processors in a real-time image processing system on FPGA (Field programmable gate array) hardware is a not a trivial task. We propose a number of graphical representations for scheduling which were evaluated for use in a visual language for image processing on FPGAs. The proposed representations are illustrated and their strengths and weakness discussed and the reasons for adoption of the state chart notation are given.

Keywords: Visual Languages, FPGA, Hardware design Languages, Image Processing, Finite state machines

1 Introduction

The use of FPGAs for implementing image processing systems is growing [1-3]. In a complex image processing system implemented on FPGAs, the processors which make up the system need to be scheduled to run at particular times in order to avoid resource contention, to ensure correct program execution, and to respond as data arrives or as data is requested. Several approaches can be used to allow for processors to run only when required.

One approach is to pass control tokens between processors, signalling that they can execute. Celoxica's PixelStreams [4] passes tokens with the standard data flow to indicate whether the data received is valid and therefore the processor can run. This approach is common in many visual languages for image processing including Khoros [5], Opshop [6] and PICSIL [7], where only the data flow is explicit and not the control flow. This data-flow approach to program control, as discussed by [8], can highlight the data flow of a program but this is at the expense of obscuring the control flow from the developer.

Another approach is to have a controlling processor directing other processors to run during well defined epochs. These epochs can be predefined (based on clock cycle counters) or can be generated by other external or internally generated triggers. This allows for both source- and sink-driven processes to be scheduled efficiently. This programming model makes the control flow more explicit to the developer.

We believe that it is effective to treat control at two levels. The first is low-level data-dependent control, where the choice of program action depends on the data that is received; for example, the selection of a identifying label for a uniformly coloured region depends on pixel values [2].

The second is high-level control, which operates at a processor or global level. In many algorithms, some or all of the processors should only run during certain defined periods. A good example of this is an algorithm for building a histogram of pixel values in an image from a video source. One processor is used to build a frequency histogram of pixel intensities by incrementing the count into the correct bin. Another processor is used to reset the histogram at the end of the frame before a new image arrives. These two blocks need to be scheduled to run at different times as: they access the same memory resource and one runs as valid pixel data is arriving, and the other when the frame ends.

2 Requirements

This work is part of a larger project aimed at the development of a visual language for representing image processing algorithms on FPGAs [9-11]. These algorithms naturally form a network of interdependent hardware units. Such a structure is suitable for representation in a visual language, because visual languages represent networks more clearly than textual languages, which are inherently sequential. The work reported here concerns the development of a clear visual representation for high level control of the network of processors – the processing blocks – from which a hardware algorithm is constructed.

In image processing it is common to have both source- and sink-driven processing modes, each imposes its own requirements on the visual representation. For example, when data from a camera is streamed through an FPGA, the processors need to be source-driven, as the FPGA hardware has no direct control over the data arrival. The system needs to respond to the data and control events as they arrive. For video data, the events of concern are new pixels, new-line and new-frame events. Based on the event, different parts of the algorithm need to run; for example, when a new frame is received, registers and tables may need to be reset. That is, the processors in source-driven systems are usually triggered by external events.

0-7695-3110-5/08 $25.00 © 2008 IEEE
DOI 10.1109/DELTA.2008.76

Sink-driven processing is also often required; a good example of this is displaying images and other graphics on a screen. In this case, processors need to be scheduled to provide the correct control signals and data to drive the screen. This imposes strict time constraints on the processes, as there is a limited number of clock cycles before the pixels for the next screen need to be produced. That is, the processors in a sink-driven system can be triggered to run at specified times, using internal events

Our system therefore needs to handle both externally triggered events and internally triggered events. External triggers are used to signal new data arrival and data requests from external devices. Internal triggers are generated by other processors to signal that they are complete or that they require new data.

These trigger events will make it possible for some processors to take one clock cycle to complete, for some to run to completion in a fixed period of time and for others to form pipelines. The visual language therefore needs a notation for showing the trigger events that cause processors to start running.

Another important consideration in the design of concurrent real-time systems is contention for on- and off-chip resources such as memory, and access to I/O. In image-processing systems, contention can occur within an operation, where different parts use the same memory, or between different image processing operations, when they are sharing data via memory. In synchronous systems, contention is easier to manage, as the designer explicitly controls when processors are scheduled. However, when asynchronous systems are required, scheduling cannot be used. If more than one processor uses a resource, contention can occur. This cannot be predetermined so other strategies such as semaphores and channels need to be used. The visual language should therefore provide for the scheduling of processors and processor sub-blocks, and show where potential resources conflicts may occur.

Defining the activation order of processors within an epoch should be facilitated by the visual language. Processors may run in parallel, form a pipeline, or run sequentially. For parallel processors, they all run at the start of the epoch. When running sequentially, processors need to run after the epoch start and the previous processor has finished. A processor within a pipeline runs after the epoch start, and once the previous process is producing valid output.

Pipeline processing has the complications of pipeline priming, flushing and other control, including stalling. Pipeline priming for image processing operations is often difficult and error prone due to the need to start processors working before the first valid data arrives. Flushing is also difficult as processors need to run until all the valid data has been processed. This may be after the transition to a new epoch has occurred. An ideal visual language would help the developer to automate pipeline control.

To summarize, the graphical representation for control constructs in such an environment should:

- allow a mixture of sequential and event driven control of processors
- allow processors and processor sub-blocks to be scheduled
- show where resources conflicts may occur
- schedule processors from the same epoch to run sequentially, in parallel or as a pipeline
- aid in the priming, flushing and control of pipelines.

The rest of this paper will evaluate several possible solutions. We then present our current approach, which trades off several competing features.

3 Graphical Representation

Our first graphical representation represents segments of the algorithm as a sequence of epochs. Each epoch is associated with a trigger, and a synchronous scheduler is responsible for issuing trigger events to start the epochs. The epochs then run for a fixed time, based on the run times required by the processors they contain and then the controller generates the trigger for the next epoch. The designer specifies the resources required by the processors in an epoch, which allows the controller to schedule the processors so that they do not conflict for resources.

In this representation, processors can be scheduled to run concurrently or sequentially within an epoch. This allows independent processors to run freely within an epoch and allows developers to move resource-conflicted processors. Processors can span epochs and run in more than one epoch.

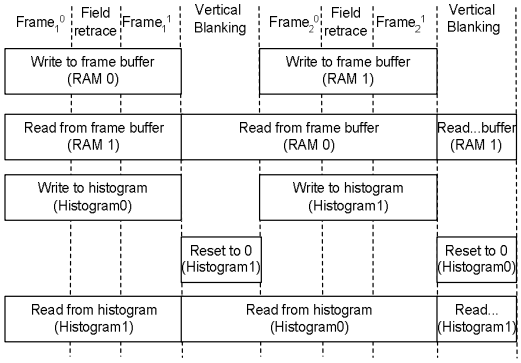

Figure 1: Epoch based graphical representation for a histogram algorithm

Figure 1 is an example of the proposed visual syntax to support this epoch-based model. In this example five different processors (Write to frame-buffer, Read from frame-buffer, Write to histogram, Reset to 0 and Read from histogram) operate on four resources (RAM_0 and RAM_1, $Histogram_0$ and $Histogram_1$) used in a histogram algorithm. In this algorithm there is a frame buffer using two RAM blocks; these are used alternately by two processors, one writing the image

to memory and one reading from the memory. Three processors require access to two histogram memory banks. Two processors write to the histograms, one resets them to zero and the other builds the histogram. The final processor reads the histogram information and processes it for other processors to use.

The horizontal direction represents time (in epochs) moving from left to right. Each epoch is represented by a dotted vertical line and the label at the top shows its trigger event. Processors are represented by rectangular blocks. Each block is labelled with the processor's name and the resource (or resources), in brackets, that it requires. Processors are laid out in a grid, with processors overlapping the epochs they run in. The vertical direction has no particular significance except that it serves as a way of separating diagram components graphically.

This first graphical representation is a manifold; that is, it should really wrap around to connect the epochs. In this example the three processors which occur in the final vertical blanking run in the next epoch ($Frame_1^0$). In order to produce a more intuitive representation of the repetitive nature of epochs, we can make the epoch display circular. This notation illustrated in Figure 2 which shows that the processors repeat forever. Transitions from one epoch to the next are triggered by the events in the centre of the diagram. Dotted lines radiating from the centre extend the epoch area to the processors. Processors are layered around the centre as overlapping sectors. In this representation, rather than using brackets to indicate resources we use a darker grey and white text to label the processor's required resources. Time progresses in a clockwise direction and is in epochs.

Figure 2: Circular Epoch based representation

Informal user tests suggested that designers may have some difficulty with reading the notation, and as the degree of concurrency increases, each additional ring of processors uses more screen real estate.

These two views clearly show when processors are running simultaneously and make it clear what resources the processors use. However both views require events to be synchronous. They also do not allow a mixture of sequential and event driven control of processors.

These two views cannot support situations where there is more than one possible exit from an epoch. This produces a non-linear, networked control structure which is better represented as a directed graph than the linear representations used in the two previous views. Graph-based notations allow for many connections between states to be represented.

There are many different visual representations of mathematical graphs; we developed a representation based on state charts (or state machine diagrams) [12], which are familiar to the engineering and computer science community and have been found to be useful to describe event driven embedded systems [13].

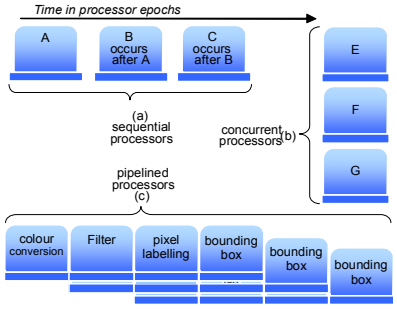

Figure 3: The three process representations: (a) Sequential, (b) Parallel, (c) Pipelined

We want to be able to schedule processors that operate in the same epoch to run sequentially, in parallel or as a pipeline. A standard state chart only uses the present active state to select processors to run. We need to extend the state machine concept to allow for the scheduling of processors within a state.

In a notation based on standard state charts, each state would correspond to an epoch, and process activation would occur as a result of the state machine reaching a particular state. Control in an FPGA environment is more complex than this, so we extended the notation to make it possible to schedule an epoch's processors to run sequentially, in parallel, or as a pipeline.

To represent this we have extended the state chart notation with a modified version of Gantt charts [14] (based on those described in [11]). Figure 3 shows the Gantt chart component of the notation; time progresses from left to right, sequential processors (a), are drawn from left to right on a single line, and parallel processors (b), are stacked vertically. As a pipeline is a combination of sequential and parallel processing we draw a pipeline (c), as sequence of processors that is repeated vertically, with a horizontal offsets at each layer, to show that the next activity involving a processor occurs later in time. To

reduce the screen real estate, processes are separated into a dark horizontal bar and a lighter coloured "flag" that contains its name. The resulting diagram clearly illustrates pipeline priming and flushing phases, and the resources that processors require are represented as flags in a different colour and under the processor bar (see Figure 4). The resource flags can be hidden when not required. Figure 4 shows a representation of the histogram example (used throughout the rest of this paper) using a combination of this Gantt and the standard state-machine notations. In this example, two state machines are shown: one for the main algorithm and another for controlling an encapsulated resource (discussed later). When a state is selected in the editor, the internal processors are shown in a split window. In the lower panel, processors can be added to a state, and their execution order can be edited. This makes it possible to implement complex scheduling if required.

State transitions are either triggered by external events or by triggers from other states. For a state machine driving another processor or an external device such as a screen, it is common for a state that has been triggered to run for a predetermined time and then trigger the next state. Therefore states can be both triggered and triggering. All states will have a triggered property which will cause a transition to this state. States may also have a triggering property that causes them to transit to other states; after a set number of clock cycles, after all the state's processors are finished, or by events internal to the state's processors. Using a state-based representation does not fulfil all of our requirements. The combined Gantt-state notation provides multiple state machines and asynchronous event-driven processing, but it hides resource conflicts. The linear and circular epoch views made this resource contention explicit, but they relied on synchronous processors. It is possible to detect resource conflicts occurring within the state as these processors are scheduled and run in the same clock domain. However, as we allow multiple asynchronous state machines, a search must be made to check if processors in other states might also need the resource. It is possible for two or more asynchronous states to require a resource simultaneously. Resolution of such a situation is left to the developer; therefore the developer needs to be notified. If processors that appear to be competing for resources cannot be scheduled to run at different times, then explicit resource sharing mechanisms, such as semaphores and channels, should be used.

In concurrent systems, deadlock can occur if two or more of the processors cannot move to another state. The four conditions for deadlock are [15]:

1. that the processors involved need exclusive access to the same resource;
2. that processors hold on to resources while awaiting the granting of additional ones that are being held onto by other processors;
3. that the system cannot pre-empt resources once they have been allocated;
4. that a circular chain of requests and allocations exists.

We have elected to concentrate on deadlock prevention by eliminating one of the four causes of deadlock so that it can never occur.

Within a single state machine, deadlocking cannot occur as only one state is active and processors within a state are scheduled so that processors which require mutual exclusion cannot run at the same time.

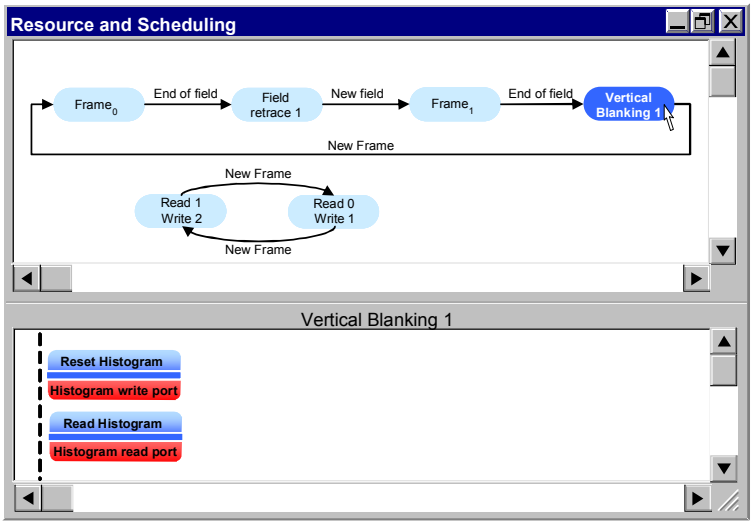

Figure 4: Extended state machine editor, Vertical Blanking selected, the three concurrent processors associated with the state are shown in the lower split screen

However in multiple state machine configurations, deadlocking can occur as processors from different state machines can compete for resources. As described above we notify the developer in this case. When multiple resources are required, it is possible to avoid both the second and fourth condition of deadlock by requesting all required resources simultaneously before a state can start. If they cannot all be allocated, none are allocated. This can result in poor resource utilization, as a state may not require a resource for the whole period [16]. Blocking a state from running to satisfy resource contention issues can case a real-time image processing application to fail. It also precludes efficient resource sharing between the processors controlled by different state machines. We should therefore also alert the developer when the above condition might occur so that they can take any necessary action, such as modifying the algorithm or allocating more resources.

Many of these conflicts can be simplified by encapsulating a set of resources with a manager. The manager then uses a separate state machine to control access to the resource through predefined ports. Use of such managers can simplify the overall system design, as illustrated in Figure 4. When the resource is used by multiple asynchronous processors, the arbitration mechanism may be incorporated within the manager to allow more efficient use of shared resources.

Another problem addressed by the Gantt/state machine notation is automated pipeline priming and flushing. In stream-based image processing, the type of pipeline control required depends on the position in the image that the processing has reached. At the start of an image, the pipeline needs to be primed. As each end-of-line is reached, the pipeline needs to pause. When the end of the frame is reached the pipeline should flush.

When the sub-processors which form a pipeline are examined, their individual priming lengths can be determined. The sum of these individual lengths is the prime time necessary for the pipeline to provide the data at the correct time. When the processors are driving a source, a state can suggest to the developer how to modify its start condition to run early. Unfortunately this is not an option when data is requested asynchronously by a sink. In this case, the designer sets a property called run-to-prime, which forces the state to run as data is produced and then stall in a primed state until the data is requested. Neither of these strategies can be used when the processors are source-driven, as data cannot be primed until it arrives from the source.

For flushing, once a state's exit conditions have been met, it needs to keep producing data to allow the pipeline to be flushed. This is relatively simple in cases when there are no resource contention issues, as the state can run for the number of clock cycles to flush the pipeline while the other state begins processing. This is complicated when the states share the same resources. In this case the old state needs to either delay the transition to the new state until the pipeline has been flushed, or exit with data still in the pipeline. This is a design decision and should be a state-based option that the designer can select according to the algorithm's requirements. Our pipeline notation, Figure 3 (c), can aid the designer by showing the number of processor epochs to flush the system and the developer can select the best option.

These different pipelining options are represented in Figure 5. Option (a) is a processor which needs to be primed before running and will pause until the states start event is reached; (b) is a pipeline which will continue to run until flushed once the state has transitioned; (c) is a pipeline that will delay the transition to a new state until it has flushed

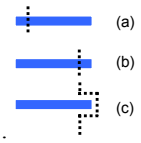

Figure 5: Three pipeline controls, (a) prime, (b) keep running to flush, (c) flush before state transition

The state based notation allows the developer to get a quick overview of the high-level control states and the trigger events which cause their transitions. Multiple states machines can be displayed on the screen with the developer able to see more detail about the processors running in a state by selecting it. This allows the developer to concentrate on the controller design while hiding the details of what is running in those states. This avoids information overload. Visual programming languages have advantages over textual languages in this respect, as in text the separation of controller and action normally does not allow for linkages to be made between the two.

4 Discussion and Conclusions

We have evaluated several different graphical representations for high level control of image processing algorithms on FPGAs. We identified five requirements that any representation should have. It should

- allow a mixture of sequential and event driven control of processors
- allow processors and processor sub-blocks to be scheduled
- show where resources conflicts may occur
- schedule processors from the same epoch to run sequentially, in parallel or as a pipeline.
- aid in the priming, flushing and control of pipelines.

The first two epoch-based views allowed processors to be scheduled and were very good at showing resource conflicts. They were both based on synchronous control but could not be extended to incorporate event

driven asynchronous control. This limitation led to a combined Gantt chart/state machine diagram notation, which has the advantage of allowing for asynchronous event-driven processor scheduling. To accommodate the need for processors to run sequentially, to run in parallel or to form pipelines, we extended the basic state chart concept with an internal state scheduling view. The states allow for some limited automation of the control of pipelines.

This view is good at illustrating and avoiding resource conflicts within a state, but it does not show resource conflicts which may occur between multiple concurrent state machines. This means that to detect possible deadlocks, the editor needs to check what resources the other states are using and evaluate where conflicts may occur. We attempt to prevent deadlock conditions by requiring states to request all resources concurrently at the start of the state; if all are not are available, then the state blocks. This is not an efficient use of resources, and the developer needs to be informed if this is a possibility.

Our proposed representation illustrated in Figure 3 and Figure 4 meets all of the requirements that were identified for the representation of image processing algorithms on FPGAs. It allows for both sequential and event driven control of processors. It allows processors and processor sub-blocks to be scheduled. It can display where resource conflicts may occur within a state and, with editor support, it can find where processors belonging to different states may conflict. The resources used by processors are not as clear in our chosen solution as the first two views and further work is needed to make this more explicit, and to automatically detect potential resource conflicts. Processors within a state can be scheduled to run sequentially, in parallel or as part of a pipeline. We have added controls to aid in pipeline priming, flushing and control that are often needed.

Our graphical visualization can aid in the development of image processing on FPGAs.

5 Acknowledgements

The authors acknowledge the Celoxica University Programme for providing the DK4 Design Suite.

6 References

[1] K. Appiah and A. Hunter, "A single-chip FPGA implementation of real-time adaptive background model," in *IEEE International Conference on Field-Programmable Technology*, Singapore, pp. 95-102, (2005).

[2] C. T. Johnston, D. G. Bailey, and K. T. Gribbon, "Optimisation of a colour segmentation and tracking algorithm for real-time FPGA implementation," in *Proceedings of Image and Vision Computing New Zealand*, Dunedin, pp. 422-427, (2005).

[3] S. A. Fahmy, P. Y. K. Cheung, and W. Luk, "Novel FPGA-based implementation of median and weighted median filters for image processing," in *International Conference on Field Programmable Logic and Applications*, Tampere University of Technology, Finland, pp. 142-147, (2005).

[4] Celoxica, *PixelStreams Manual*, (1 ed), Celoxica (2005).

[5] K. Konstantinides and J. R. Rasure, "The Khoros software development environment for image and signal processing," *IEEE Transactions on Image Processing,* vol. 3, pp. 243-252,(1994).

[6] P. M. Ngan, "The Development of a Visual Language for Image Processing Applications," Ph.D. dissertation, Massey University, Palmerston North, New Zealand, (1992).

[7] M. W. Pearson, P. J. Lyons, and M. D. Apperley, "High-level Graphical Abstraction in Digital Design," *VLSI Design,* vol. 5, pp. 101-110,(1996).

[8] T. R. G. Green and M. Petre, "Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework," *Journal of Visual Languages and Computing,* vol. 7, pp. 131-174,(1996).

[9] C. T. Johnston, D. G. Bailey, P. Lyons, and K. T. Gribbon, "Formalisation of a visual environment for real time image processing in hardware (VERTIPH)," in *Proceedings of Image and Vision Computing New Zealand*, Akaroa, N.Z., pp. 291-296, (2004).

[10] C. T. Johnston, D. G. Bailey, and P. Lyons, "A Visual Environment for Real-Time Image Processing in Hardware (VERTIPH)," *EURASIP Journal on Embedded Systems,(Article ID 72962),* pp. 1-8,(2006).

[11] C. T. Johnston, D. G. Bailey, and P. Lyons, "Towards a visual notation for pipelining in a visual programming language for programming FPGAs," *7th International Conference of the NZ chapter of the ACM's Special Interest Group on Human-Computer Interaction,* pp. 1-9,(2006).

[12] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of Computer Programming,* vol. 8, pp. 231-274,(1987).

[13] M. Samek, Practical Statecharts in C C++: An Introduction to Quantum Programming with CDROMCMP Books (2002).

[14] J. R. Schermerhorn, *Management*, (Sixth ed), John Wiley & Sons,New York (2001).

[15] E. G. Coffman, M. Elphick, and A. Shoshani, "System Deadlocks," *ACM Computing Surveys,* vol. 3, pp. 67-78,(1971).

[16] A. Burns and G. Davies, *Concurrent Programming*, (1 ed), Addison-Wesley Publishers Ltd, England (1993).