Suggestions for Stream Based Parallel Systems in Ada

M. Ward* and N. C. Audsley
Real Time Systems Group
University of York
York, England
(mward,neil)@cs.york.ac.uk

Abstract

Ada provides good support for the implementation of dependable, real-time, control systems. However, its support for other styles of systems is not as good. This paper explores the support available for implementing parallel, stream based systems.

The paper presents an implementation of an image manipulation system which highlights deficiencies in the support for such systems in the Ada language. Two additional semantics are proposed for addition to the Ada language which will provide for the needs of these systems. The broadcast semantic allows the same data to be written to several POs simultaneously. The guarded protected function semantic permits several readers to wait on an entry and simultaneously read data from the PO.

1 Motivation

The Ada language [1] has found a niche in the implementation of dependable real-time systems. This has traditionally revolved around the use of periodic tasks to implement control systems. However, there are other styles of system that would benefit from the various attributes that Ada offers. The main advantages of Ada are its support for real-time systems, and its support for concurrency within its semantics. This paper looks at the effects of trying to implement dependable, stream based, parallel systems in the Ada language. These have high performance demands, and require substantial support for efficient implementation.

This paper is presented in two parts: the experience of using Ada to implement such a system; and suggestions to improve the language support for them. Section 2 presents an overview of the implementation of an image manipulation system. This covers the motivation for moving away from the traditional implementation method, through the ideal solution, and the changes required to allow it to be programmed in Ada, to a description of the final system. Section 3 contains suggestions for additions to the Ada language to provide for the needs of such systems that are not supported within the language.

*This work was supported by BAE SYSTEMS

2 The Image Manipulation System

Image manipulation systems work on video streams in real-time. With both source and output streams, the system applies a transformation to the streams. These can be simple, pixel-based, manipulations (e.g. greyscale, sharpen, edge detect), or a more complex, frame-based, transformations (e.g. image warping or morphing). Whilst frame based manipulations need more buffering than pixel based ones, their implementation is similar.

The traditional approach to the implementation of these systems uses graphical libraries implemented on a processor. The processor can be either a general purpose processor or a graphics specific processor which provides support for common graphical functionality. In the general case, these techniques provide ample performance for most image processing techniques. However, the use of these techniques for dependable systems raises a number of issues.

A dependable system needs to be proved to be correct to its specification. This requires analysis of both the functionality of the system, as well as its timeliness (amongst other things). The traditional implementation techniques fall foul of these needs:

- General purpose processors have good best case performance, but due to the architectural features to do this, have poor worst case performance. This limits the available processing power and restricts the overall system performance.
- Specific processors generally do not have the same amount of evidence to prove they are correct that

a general purpose processor will have. Though due to the specific instructions they have less need of architectural speed up features.

- Graphics libraries are not written for dependability, generally they are written for speed. This makes proving them correct difficult, especially when their size is considered.

As such, high-performance dependable image manipulation systems are difficult to build. This makes high resolution, high frame-rate image manipulations difficult to do with traditional implementation techniques. This produces a need for a different implementation technique.

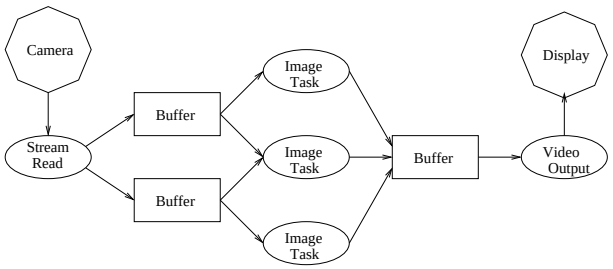

Figure 1. Diagram of the initial solution

2.1 The Problem

The problem arose from a request by BAE Systems to implement a dependable image manipulation system capable of dealing with high-resolution, high-refresh video streams. The system had to be dependable, preferably using Ada as limited by the SPARK [2] and Ravenscar [3] subsets. The use of Ada will allow reuse of existing software code. In addition to this, the system needed minimal delay on the output stream. The initial demonstration of this system should be an image warping application (e.g. correcting imperfect optics or pre-distortion for display on shaped surfaces).

2.2 Solution Suggestion

A block diagram of the initial suggested solution can be found in figure 1. This solution relies on parallelism to provide the processing power to perform the transformations. As the manipulation is to be done on a frame by frame basis, the video stream is first read into a screen buffer. To provide for multiple accesses this buffer is replicated a number of times dependant on the needs of the application. This can be a replication of the entire buffer, or buffers that each contain part of the image. The image processing is undertaken by a number of parallel tasks. Each task is responsible for part of the output image, and can access any of the input buffers that it needs to. The generated image is collected in a single output buffer (since each pixel is only written once), and this is used to generate the output video stream. It is intended that the system would be implemented on FPGA using YHAC to generate the circuits from Ada source code, giving a truly parallel solution.

2.3 YHAC

The York Hardware Ada Compiler (YHAC) [4, 5, 6] allows Ada programs to be targeted directly to hardware. Using the SPARK subset and Ravenscar tasking profile gives a static language, which can be transformed to hardware. The compilation process uses template instantiation over the statements within the program. The templates build up to form a hardwired state-machine which controls the program flow Expressions are built up in a similar manner to produce expression trees. Complex expressions are split to allow multi-cycle evaluation. Concurrency is implemented using separate circuits, providing a truly parallel implementation. The only interference experienced by a task is over access to shared data.

In the domain of dependable and real-time systems, implementation via YHAC has several advantages:

- The produced circuit is traceable back to the source code.
- The program is implemented as a circuit, meaning there are no hardware bottlenecks, which need no architectural speed-up features.
- The final circuit can be easily analysed for resource usage. As the circuit is built up by template instantiation, analysis can be done from the source code. This covers both its space utilisation on the FPGA, and the timing of the program.
- Provides performance equivalent to a mid-range processor for single threaded applications.
- Concurrent applications get a significant performance boost due to parallelism. No longer sharing single processing resource reduces the level of inter-task interference.
- Designed to give the same semantics for all code in hardware as software. Ignores some implementation techniques (e.g. suspension objects) to maintain this consistency.

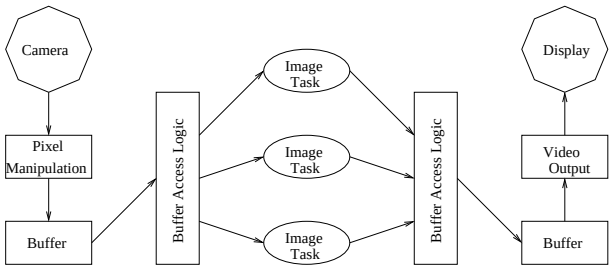

Figure 2. Diagram of the final solution

2.4 Solution

The solution presented above has several problems:

- There is no broadcast semantic in Ada. This makes filling multiple buffers difficult. Cannot broadcast the data to multiple POs, which needs more time per input pixel, but the timing of the video stream is fixed.
- There is no way to simultaneously release multiple tasks. Whilst entry queues allow multiple releases, these cannot happen simultaneously (each task has to enter the PO in turn). The Ravenscar profile exacerbates this problem as it outlaws entry queues.
- Working with video streams requires accurate timing in the circuit to ensure no pixels are lost. Whilst YHAC allows timing properties to be determined, it doesn't give definite control over the timing.

These problems require some changes to the original solution. By including dedicated hardware to interface with the video-streams, the lack of definite timing in YHAC is no longer an issue. This hardware can also handle some of the image pre-processing required, such as conversion of the data into RGB format, and clipping the input stream to the visible area. The buffers are also encapsulated within the hardware as this removes the need for a broadcast within Ada, and the structure of the compiler prevents the sharing of memory used in the buffers between dedicated hardware and Ada circuit. One advantage of encapsulating the buffers is that the accesses can be pipelined, improving their performance, allowing a smaller number of buffers to be used. The resulting change to the structure of the solution can be found in figure 2.

2.5 Implementation

The final system solution was implemented using a Celoxica RC203e development board. This board provides a Xilinx Virtex2 3000 FPGA as the logic resource, and a wide selection of interfaces, including video input and output. The board also provides 4MB of off chip memory. The drivers for all the interfaces are provided in Handel-C, Celoxica's C-based hardware language.

The implementation can be divided into two parts: the framework, which implements the video interfacing and buffering; and the application, which contains the transformation encoding. The implementation of these parts is described below:

- *Framework*
  As the device drivers are written in Handel-C, the dedicated hardware has been implemented in the same language. The these read in the video stream into multiple double buffers. Once a frame has been put into the buffers, the buffers are swapped, and a signal given to the application to start processing on the frame. Whilst this is happening, the next frame is being placed into the other set of buffers. On the output side, the transmission of a frame waits for the previous frame to finish, at which point the buffers are swapped and the new frame started. This double buffering introduces a delay of 1 frame plus the delay in the application. This cannot be reduced if full frame transformations are being dealt with. It can be seen that the maximum application delay is 1 frame - if it is slower, frames will only be part complete when transmission commences. There is no synchronisation between the input and output streams, so there is no additional cost over the delay of the application.
- *Application*
  The example implementation is an image warper. This takes a good image, and distorted it to give a fish-eyed image. Due to restrictions on the memory capacity on the board, the image is restricted to a resolution of 640 x 480. The image processing is implemented in 9 parallel tasks, each of which is responsible for part of the image. There are a number of 'helper' POs in the system, an interrupt handler PO for each task which detects the start processing signal from the framework and release the tasks, and a finish detector which provides the signal to the framework. Each task implements a simple transformation, which is pre-computed to save time in the processing. The transformation is done in under 1/2 frame, giving a delay of 1 1/2 frames overall.

The complete system took about a month to design and implement. The framework took most of this time,

mostly in altering the provided sequential access buffers to allow the random access needed by the applications, and integrating the Handel-C and Ada circuits. The application took about 1 day to implement, half for application coding, the other half generating an acceptable transform. Due to the tool-chains needed in targeting hardware, the compile-test-correct cycle can take a while (a small value change still requires a complete re-compilation and synthesis), which extended the time needed to generate the transform. Alteration of the application is easy as the transform is coded algorithmically within the processing tasks.

The implemented system only uses 15% of the resources available on the FPGA, leaving plenty of scope for more complex transforms, or faster implementations. At present, most of the resource is taken by the framework, with the application itself using about 3%. By introducing more tasks, a faster implementation is possible, at the cost of higher resource usage. Alternatively extra resource can be used by making the transformation more complicated. If a lower resource usage is needed, the number of tasks can be reduced and the speed dropped to give a full frame's delay. The main limitation to the ultimate performance of the system, is the buffer throughput rate, but this can be increased by providing more buffers, at the cost of needing more memory buses on the device.

3 Language Suggestions

In designing the IMS, there appeared a need for two extra bits of functionality in the Ada language: a broadcast semantic, and a parallel release semantic. These are described below:

3.1 Broadcast

The broadcast semantic would allow a task to write data to a set of protected objects in a single call.

It is envisioned that the protected objects being targeted would be declared as an array of protected objects. This would allow existing array syntax to be used for the declaration of the POs, for selection of POs within the call, and permits a subset of the array to be selected. This gives the suggested syntax as shown in figure 3.

Two alternatives are presented for the broadcast to all elements of the array. The first uses the reserved word all to indicate that the entire array is being referenced, the second uses a slice that covers all elements of the array. A third, though discounted option would use the others keyword. These have their advantages and disadvantages:

All - The reserved word all in the name.all context is an explicit dereference of an access type. To use the same syntax here would overload it to be a reference to

```
1  protected type po_type is
2      procedure call(val : integer);
3    end p_type;
4
5  po_array : array (1..10) of po_type;
6
7  po_array(7).call(37); -- single instance call
8  po_array.all.call(25); -- broadcast to all elements of po_array
9  po_array(1..10).call(25);-- alternative broadcast to all elements of po_array
10 po_array(2..5).call(13); -- broadcast to restricted range of elements
```

Figure 3.

all elements within the array. There is also the problem of what happens when the array is accessed via an access type. However, the use of all does convey the meaning to the programmer that the entire array is being accessed.

Whole slice - Using an entire range slice maintains consistency in the selection of the parts of the array to use. There is no change in the structure of the selection, simplifying the compiler implementation. However, there is poor readability as it is not possible to tell that the entire array is being accessed. Similarly, if the array size is changed in a program, every whole slice will need to be changed, which does not aid program maintenance.

Others - The others keyword could also be used to indicate all elements in an array. This would follow from its use in aggregate expressions, but does not sit well out of the aggregate form. In addition, it does not have the same readability as the other forms.

On balance, the name.all form seems to offer the better balance, providing an obvious indication that the entire array is being accessed.

The implementation of the broadcast semantic can fit easily into both concurrent and parallel system implementations. Within a concurrent system, the calls to the protected procedures in the broadcast can be done by iterating through the array. In a parallel system, all the accesses can be initiated in parallel, provided sufficient processing elements are available. When there aren't enough processors there needs to be some iteration over the calls.

The envisioned implementation raises a couple of issues with semantics of the call. Since the calls may be done iteratively, to preserve the atomicity of the operation, there are two conditions that must hold. First, the call must be none-blocking. As the calls are being done iteratively, a block will delay the later calls. Secondly, there should be no pre-emption between calls, that is, the entire access should be considered a single protected action.

```
protected type po_type is
    function par_entry return integer when allow_entry;
    procedure release;
  private
    allow_entry : boolean := false;
  end po_type;

protected body po_type is
    function par_entry return integer when allow_entry is
      begin
        return 0;
      end par_entry;
    procedure release is
      begin
        allow_entry := true;
      end release;
  end po_type;

task waiter is
    null : integer
  begin
    while true
      null := par_entry;
      -- do something
    end while;
  end waiter;
```

Figure 4.

3.2 Guarded Protected Function

A guarded protected function semantic would allow multiple, read-only, accesses to wait on a guard value. When the guard becomes true, all accesses are allowed to enter immediately. On completion of these accesses the guard is automatically reset to false. These threads must be read only, as multiple threads would be active in the protected object. This preserves the access rules for protected actions and effectively gives a function based equivalent of an entry.

Since a guarded protected function is effectively a function based entry, a mix of the current function and entry call syntax would seem appropriate. A suggested syntax is shown in figure 4. The function specification follows that of a normal entry, with the addition of the return value type specification before the guard. There are two restrictions on the specification: the parameters to the entry can only be of in mode; and the guard expression must be a single boolean variable. The body of the entry will follow the rules of both functions and entries: no side-effects, that is, no change of PO state including the guard expression; no potentially blocking operations; and there must be a return statement in all paths through the body. The no side-effects rule prevents the guard from being reset inside the function, and hence the need for the automatic reset of the guard. The calling of the guarded protected function remains the same as any other function call.

There is an issue with this syntax in that forcing the entry call to be a function call may not reflect its use. In cases where it is used to allow multiple tasks to collect the same data on release, the use of the function call syntax is sensible. Where the only purpose is to effect a simultaneous release of multiple tasks (as in the example) the return value is not needed, but must still be used. Whilst this can be ignored by a compiler (a constant return value can be implemented as a local assignment after the call), it reduces the readability (and elegance) of the program code. However, making the call a procedure call would change the declaration syntax (and need a new reserved keyword to describe it) and require that out mode parameters be allowed. This is to allow data to be returned from the call, which also means that assignment to local parameters needs to be permitted, making the no side-effects rule harder to enforce.

An alternative syntax to that proposed in figure 4 would make no change to the syntax of the language. As mentioned above, the guarded function is a function-based equivalent of an entry. By changing the concept from a function call to a parallel entry call, it can be implemented without a syntax change. Restricting an entry to have only out mode parameters, a simple guard variable, and no side-effects would allow parallel access to it. The requirement for such an entry to be used as a parallel entry could be indicated using a pragma. This pragma would indicate to the compiler that this entry could be accessed in parallel, that the entry needed to be checked for conformity to the above requirements, and that the guard variable needed to be automatically reset. Whilst this does not require additional language syntax, it overloads the entry syntax with a different semantic, which could cause issues with readability and maintainability.

Again, this new semantic can be implemented in both parallel and concurrent systems. Within a parallel system, the readers are allowed access as soon as the protected procedure that set the guard to true completes. Since they are only reading the data within the PO, this can be done without violating the protection rules. Once the final reader has left the PO, and the protected action completes, the guard value is reset to false (hence the requirement that it be a single variable). The concurrent implementation cannot have all the accesses happening at once, so they must be allowed to happen one after the other, all within the same protected action, and with the completion of the last access causing the guard variable to be reset to false. In this way it behaves in similarly to a "last one out closes the door" implementation on an entry queue. In both these implementations, the resetting of the guard occurs as a result of the completion

of the protected action that sets it. This allows for a simple definition of the semantics.

Both the suggestions for the guarded protected function semantic have used an automatic return of the guard to false after all the waiting calls have completed. This can be considered poor deign, as it hides some semantics of the call. An alternative solution would be to leave the resetting of the guard variable to the programmer. This can be easily accomplished using an entry guarded by the count attribute of the guarded function. Leaving this to the programmer will give a greater flexibility, at the cost of leaving open the possibility of bugs caused by programming errors.

3.3 Other Thoughts

Both of these suggestions are related to protected objects. In full Ada (as opposed to Ravenscar Ada), entries exist in both tasks and protected objects. This raises the question of whether the broadcast and parallel entry semantics be extended to tasks.

It would seem that a task broadcast would be a useful semantic to have. This would allow data to be broadcast directly to tasks, rather than forcing the use of POs between the tasks. However, the PO broadcast semantics use protected procedure calls, whereas the tasks only offer entries. Protected procedure calls, though subject to possible delay, are deemed to be non-blocking. Task entries are, however, deemed to be blocking and therefore provide a different semantic. As an entry call can only proceed when the task allows it, the broadcast can be held up by a single non-responsive task.

A task can only have a single thread of control. This makes a parallel entry impossible to implement as the calls would have to be handled serially as for a normal entry queue. In this case, accepting the first call would require that all were handled without interruption until the queue was empty. Of course, both of these would be outside the scope of the Ravenscar profile.

Finally, it should be noted that both the parallel entry and the broadcast can be emulated by the other, though with restrictions on the effectiveness of the emulation. A broadcast could be programmed through the parallel entry semantic, with the data to be broadcast being written to the PO and the waiting tasks allowed to read it. This provides a broadcast to the waiting task, any task that was wot waiting when the write happened would never be able to access that data, and would have to wait for the next broadcast. Similarly, the parallel entry can be emulated by broadcasting to multiple POs, each of which has a task waiting on an entry. This would allow each task to release once. However, the release time could not be guaranteed - in the parallel entry, only those tasks waiting get released; in the broadcast the task will release on the broadcast, or when it next tries to access, rather than being forced to wait for the next broadcast.

4 Conclusions

This paper has looked at the issues surrounding using Ada to implement a parallel stream based system. The problems were illustrated through the development of an Ada based image manipulation system. As a result of this, two suggestions for new language semantics as a result of problems encountered were presented.

The image manipulation system, developed to meet a set of industrial requirements, uses Ada to implement a stream based, parallel, image morpher. Whilst the implemented solution provides a framework for efficient image processing, it highlighted two shortcomings in the Ada language. There is no facility for easily splitting an input stream into several buffers, nor is there the ability to simultaneously release multiple tasks.

From the problems encountered in the implementation of the system, two new semantics have been proposed. The broadcast semantic will allow the same data to be written to multiple protected objects at the same time. The guarded protected function semantic provides a function based equivalent of an entry with the ability to release multiple tasks simultaneously. Together, these would provide better support for parallel streaming applications.

References

[1] *Ada 95 Reference Manual*. Intermetrics, January 1995.

[2] J. Barnes. *High Integrity Ada: The SPARK Approach*. Addison-Wesley, 1997.

[3] A. Burns, B. Dobbing, and G. Romanski. The Ravenscar Tasking Profile for High Integrity Real-Time Programs. In *Reliable Software Technologies, Proceedings of the Ada Europe Conference, Uppsala*, volume 1411, pages 263–275. LNCS, Springer-Verlag, 1998.

[4] M. Ward and N. C. Audsley. Hardware Compilation of Sequential Ada. In *Proceedings of CASES 2001*, pages 99–107, 2001.

[5] M. Ward and N. C. Audsley. Hardware Implemanation of the Ravenscar Tasking Profile. In *Proceedings of CASES 2002*, pages 59–68, 2002.

[6] M. Ward and N. C. Audsley. Hardware Implementation of Programming Languages for Real-Time. In *Proceedings of RTAS 2002*, pages 276–285, 2002.