Accepted for publication to the 10th International Symposium
On Hardware/Software Codesign (CODES'02,
May 6-8 2002, Estes Park, CO, USA

Design of Multi-Tasking Coprocessor Control for Eclipse

Martijn J. Rutten[1], Jos T.J. van Eijndhoven[1], Evert-Jan D. Pol[2]
[1] Philips Research Laboratories
[2] Philips Semiconductors
Prof. Holstlaan 4, 5656 AA Eindhoven, The Netherlands
{martijn.rutten, jos.van.eijndhoven, evert-jan.pol}@philips.com

ABSTRACT

Eclipse defines a heterogeneous multiprocessor architecture template for data-dependent stream processing. Intended as a scalable and flexible subsystem of forthcoming media-processing systems-on-a-chip, Eclipse combines application configuration flexibility with the efficiency of function-specific hardware, or *coprocessors*. To facilitate reuse, Eclipse separates coprocessor functionality from generic support that addresses multi-tasking, inter-task synchronization, and data transport. Five interface primitives accomplish this separation. The interface facilitates the design of coprocessors that require complex control to handle data-dependent I/O, saving/restoring task state upon task switches, and pipelined processing. This paper presents how this interface enables the design of such reusable yet cost-effective coprocessors.

1. INTRODUCTION

The advent of new media applications such as time-shift recording, 3D games, video conferencing, and MPEG-4-like interactivity demands an increasing flexibility of consumer electronics products. Moreover, the variation in the required set of applications per product, per country, and over time as standards evolve calls for an integral approach.

Managing complexity, design cost, and time-to-market of such resource-constrained appliances requires a generic and scalable media-processing platform that can be deployed in a wide range of products. Currently, several vendors are entering the market with platforms that address these issues to some extent [1][2]. These media processing platforms approach the time to market of the processors in the PC market, but require an order of magnitude lower cost while delivering significantly higher performance. This leads to the realm of Systems-on-a-Chip (SoC), consisting of interconnected subsystems, each optimized for a specific purpose. Complex media-processing SoCs exploit the performance density of sophisticated hardwired function modules to implement critical parts of the targeted media applications, while endowing the system with a sufficient level of flexibility by embedding multiple programmable cores. Currently, these subsystems are either hardwired or fully programmable.

We developed the Eclipse architecture template [3] to support the design of versatile SoC subsystems. Thereto, Eclipse introduces a mix of programmable and hardwired functions in a single subsystem. Instances of the Eclipse template combine function-specific hardwired modules or *coprocessors* with one or more programmable cores. These subsystems can be configured for various application graphs, such as MPEG decoding or encoding applications. Eclipse coprocessors operate at a medium function grain, which allows them to be reused in different application graphs. Moreover, the coprocessors are multi-tasking and can time-share tasks from a set of applications. This way, application complexity is not restricted to the number of coprocessors in the architecture. For instance, a representative medium-grain function is the discrete cosine transform (DCT) required for MPEG encoding and decoding. An Eclipse DCT coprocessor may time-share the inverse and forward DCT functions needed for concurrent MPEG-2 encoding and decoding in a time-shift recording application.

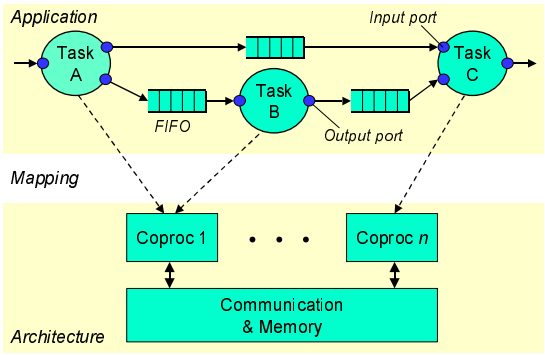

Figure 1. Application to architecture mapping.

Figure 1 depicts how application tasks are mapped onto the coprocessors. Eclipse applications are specified as a set of tasks that communicate with each other through FIFO channels [4]. A channel connects to the output *port* of a producing task and the input port of one or more consuming tasks. Application channels are mapped onto data *streams* with buffers allocated in shared on-chip memory. All coprocessors execute autonomously without requiring CPU support for task scheduling or synchronizing access to these stream buffers.

The key challenge we address is the definition of an interface that supports the construction of cost-effective coprocessors for use in multiprocessor SoC subsystems such as Eclipse. These coprocessors may require complex control to handle data-dependent behavior, multi-tasking, and pipelining. This paper addresses these issues and describes generic concepts for minimizing synchronization and task-switching overhead while enhancing performance through pipelining.

Section 2 gives an overview of the Eclipse architecture from a coprocessor viewpoint and introduces a uniform interface that separates coprocessors from the generic multiprocessor infrastructure. Section 3 shows the trade-offs to be made in designing coprocessors that deploy this interface. The key design issues, namely synchronization, multi-tasking, and pipelining, are subject of Sections 3.1 to 3.3. The paper subsequently gives initial results in Section 4, discusses related work in Section 5, and concludes with Section 6.

2. ECLIPSE ARCHITECTURE

Eclipse is a scalable architecture template. The number and type of coprocessors may vary over instances. Moreover, the communication network to transport and synchronize data between coprocessors may change over instances as communication bandwidth requirements vary. Eclipse introduces the coprocessor *shell* [3] to facilitate reuse of coprocessor designs over different Eclipse instances with different communication network characteristics.

Figure 2 depicts this hardware interface block that separates the computation hardware (coprocessors) from the communication hardware (buses, memory). The shell alleviates coprocessor design by absorbing many system-level issues, such as multi-tasking, stream synchronization, and data transport. Thus, coprocessor designers can concentrate on application functionality.

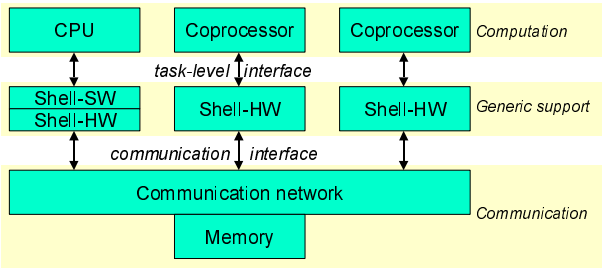

Figure 2. Coprocessor shell for system-level support.

The shells are distributed, such that each shell can be instantiated close to the coprocessor that it serves. Each coprocessor interacts with its shell through five generic interface primitives. While specified in the form of software function calls, Eclipse implements these primitives in hardware with an identical interface: a master-slave handshake with corresponding argument and result passing. Figure 2 represents this as the 'task-level interface' between coprocessors and their shells.

For multi-tasking, the coprocessor issues the following primitive:

```
int GetTask( int *task_info ).
```

The coprocessor calls this primitive whenever it allows a task switch to another task mapped on the coprocessor. The return value is the identifier of the next task (`task_id`) to execute on the coprocessor. The `task_info` value indicates which function the selected task should perform, *e.g.* forward or inverse DCT.

The primitives for accessing data in the stream buffer are:

```
void Read(int task_id, int port_id, int offset,
    int n_bytes, Bytes *bytevector );
void Write(int task_id, int port_id, int offset,
    int n_bytes, const Bytes *bytevector ),
```

and the primitives for synchronizing access to data in the stream buffer are:

```
bool GetSpace(int task_id, int port_id,
    int n_bytes);
void PutSpace(int task_id, int port_id,
    int n_bytes).
```

All tasks ports (Figure 1) map to one physical coprocessor-shell interface that handles `Read`, `Write`, `GetSpace/PutSpace`, and `GetTask` requests in parallel. The coprocessor is responsible for serializing requests from different task ports. To discern between different streams, the coprocessor passes an identifier of the active task port to the shell through the `port_id` argument of the above primitives. The shell subsequently combines the `task_id` and `port_id` arguments into an identifier of the associated stream for sending synchronization messages to a predecessor/successor task and to access the stream buffer in shared memory. Note that while the `GetSpace` and `PutSpace` primitives do not distinguish between input and output ports, a `GetSpace` on an input port inquires available data for reading, whereas a `GetSpace` on an output port inquires available room for writing into the stream buffer. Likewise, a `PutSpace` call on an input port commits empty room available for writing, while a `PutSpace` on an output port commits valid data written in the stream buffer.

The use of the coprocessor-shell primitives and their arguments is subject of Section 3. These coprocessor-shell primitives are generic and simplify coprocessor design while supporting the design of coprocessors that require complex control to cope with for instance data-dependent I/O, variable packet sizes, and pipelined processing. In all five cases the coprocessor has the initiative for taking action; all primitives are called by the coprocessor and implemented by the shell.

3. COPROCESSOR CONTROL

In dedicated coprocessors, state save/restore is specific for the function implemented in the coprocessor. This differs from CPU software where the operating system saves and restores task state in a generic way. Eclipse coprocessors avoid the hardware costs required for saving task state at arbitrary points in time by explicitly deciding on the time instances during task execution at which they can switch the running task. The coprocessor can continue up to the point where it has minimal or no state. At such moments, the coprocessor asks its shell which task it should perform next by calling the `GetTask` primitive. We denote the intervals between `GetTask` inquiries as *processing steps*.

The coprocessor executes an infinite loop over such processing steps. The following simplified code shows such a coprocessor control loop, as an example of multi-tasking coprocessor design using the five Eclipse primitives. The example illustrates the separation of coprocessor functionality, implemented by the `Compute` function, from coprocessor control to handle multi-tasking, synchronization, and data communication.

```
while(true) {
  // Perform a single processing step
  task_id = GetTask(&task_info);

  // Is there data/room for reading/writing?
  blocked = !GetSpace(task_id, IN, INSIZE)
          || !GetSpace(task_id, OUT, OUTSIZE);
  if (blocked) continue; // No useful work to do

  Read(task_id, IN, 0, INSIZE, &in_data);
  PutSpace(task_id, IN, INSIZE); // Commit room

  Compute(task_info, in_data, &out_data);
```

```
    Write(task_id OUT, 0, OUTSIZE, out_data);
    PutSpace(task_id, OUT, OUTSIZE); // Commit data
  }
```

Sections 3.1 through 3.3 explain, extend and modify this trivial example to include random data access with FIFO synchronization, saving and restoring state upon task switches, and parallelism through a pipelined implementation.

3.1 Synchronization and Data Transport

From the view of a coprocessor task port, a data stream looks like an infinite tape of data, with a current 'point of access'. With the `GetSpace` call, the coprocessor asks the shell permission for access to a certain data space ahead of this current point of access. Here, data space signifies available data for reading from an input data stream, or available room for writing data to an output stream. If the shell grants permission, the coprocessor can perform `Read` or `Write` actions inside this requested space, with variable-length data (through the `n_bytes` argument), and on random access positions (through the `offset` argument). The shell denies permission by returning `false` on the `GetSpace` call when there is not sufficient data or room available. In this case, the coprocessor task cannot proceed and either the coprocessor can switch task or the task can keep on trying to proceed by repeatedly issuing `GetSpace` requests. Section 3.2 details these alternatives.

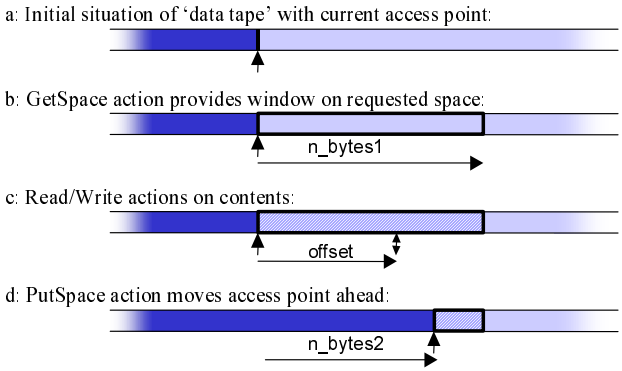

Figure 3. Synchronization and data I/O through a single port.

After one or more `GetSpace` calls—and optionally several `Read`/`Write` actions—the coprocessor can decide it is finished with processing (some part of) the data and issue a `PutSpace` call. This call advances the point-of-access a certain number of bytes ahead, in size constrained by the previously granted space. Figure 3 depicts this process.

The coprocessor is responsible for functionally correct behavior when using the interface primitives; *i.e.* the coprocessor must adhere to denied `GetSpace` requests, and not attempt to read or write data outside the window of granted space. Moreover, the coprocessor must maintain correct ordering when serializing simultaneous requests on the same stream. The latter specifically concerns pipelined coprocessors where independent parallel pipeline stages issue requests on the same stream (Section 3.3).

3.1.1 Granularity of Synchronization

Synchronization of data transport—implemented through the `GetSpace` and `PutSpace` synchronization primitives—is fully separated from the actual data transport, implemented through the `Read` and `Write` primitives. The `n_bytes` argument of `GetSpace` and `PutSpace` calls allows the coprocessor to synchronize streams at a granularity and rate that differs from the individual `Read` and `Write` calls.

The number of bytes that can be transferred on a single `Read` or `Write` request is restricted to the width of the corresponding coprocessor-shell interface. Thus, a coprocessor may need to issue multiple reads or writes to transfer one logical unit of data, *e.g.* a block of 8x8 DCT coefficients. Synchronization is mostly done at a data grain that is meaningful to the application context in order to avoid building up internal state between task executions. Within this constraint, the coprocessor designer must balance the overhead incurred by a high synchronization rate versus the high buffer requirements of synchronizing between coprocessors at a low rate.

3.1.2 Random Access

Since synchronization is not coupled to individual read and write actions, a coprocessor can randomly access the data within an acquired window of granted space. The `Read` and `Write` primitives therefore allow a random offset from the current point of access through their `offset` argument. One example of random access is a coprocessor that uses a buffer in shared memory as scratch pad, *e.g.* as texture memory in 3D graphics, look-up table, or for storing the state of a task (Section 3.2). Contrary to data streams between coprocessors or tasks, this scratch-pad memory is allocated to one task only, and can therefore be accessed without `GetSpace`/`PutSpace` synchronization.

The Eclipse communication network is optimized for streaming data, *e.g.* through a wide bus to a wide shared memory. Moreover, the Eclipse shells incorporate small read and write caches that can perform automatic prefetching for streaming data access. Therefore, random access on Eclipse streams should be used cautiously. For instance, the transpose buffer between horizontal and vertical DCT operations can be mapped onto a shared memory buffer, accessed with the Eclipse primitives. However, the random byte-level access to this buffer will induce significant overhead. Therefore, such a small buffer is better kept local inside the coprocessor.

Coprocessors requiring random access at a coarser grain can increase cost-effectiveness by mapping their buffers in shared memory. This occurs when the coprocessor jumps randomly between large groups of data in the buffer, but accesses the data within a group in a streaming fashion. This is typically the case in DV decoding, where each macroblock is coded with a fixed number of bytes at a predetermined position in the bitstream. Whenever the variable-length coded macroblock requires more than this reserved data segment, the remainder is added to a second segment for which the macroblock data does not occupy the full segment. The macroblocks in a frame are shuffled to distribute these macroblock tails evenly throughout the frame. Therefore, the macroblock data within a segment can be accessed sequentially, but random access is needed to jump from one macroblock to a next.

3.1.3 In-Place Updates

Figure 4 illustrates a situation in which a task *B* performs *in-place* updates of the data in a stream buffer between two other tasks *A* and *C*. Compared to connecting separate input and output streams to task *B,* sharing the stream buffer between three tasks can be advantageous when task *B* only performs occasional modifications

in the data stream and therefore does not need to transport the entire stream contents.

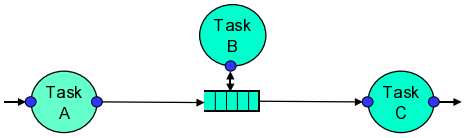

Figure 4. Task B performing in-place updates.

This is for instance the case when task *B* mostly just watches the data, maybe by inspection of some header information only (not reading all stream data) and mostly allowing the data to pass from *A* to *C* without modification. Relatively infrequently, it could decide to change a few items in the stream. In a practical situation, the main CPU may intervene in the communication between two hardware coprocessors to patch the stream to correct errors caused by hardware flaws, to adapt the stream towards slightly different stream formats, or simply for debugging purposes.

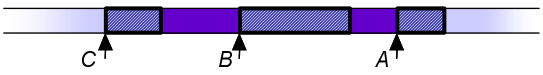

Figure 5. Data stream shared by 3 tasks.

The separation of synchronization and data transport allows such an efficient implementation. Figure 5 shows the data-tape view of such a setup with three tasks sharing a single buffer in shared memory to reduce memory traffic and coprocessor workload. The figure shows the access points of tasks *A*, *B*, and *C*, where *A* is a writer and leaves valid data behind, *B* performs the in-place updates, and task *C* is a reader and leaves empty space behind.

3.2 Task Switching

Multi-tasking on the Eclipse coprocessors is a shared responsibility between the coprocessor and the shell. The shell takes care of task scheduling [5], while the coprocessor is responsible for providing task switch points and saving and restoring the task state (if any) upon a task switch.

As outlined in Section 3.1.1, coprocessors operate on a logical unit of data—*e.g.* an 8x8 block of DCT coefficients—encapsulated in a *data packet*. The coprocessors can have different patterns of packet consumption and creation. When consumption at the input is synchronized with packet creation at the output ports of the coprocessor, the coprocessor can switch tasks at the moments when the data state is void. Typically, coprocessor state is minimal after processing of a complete packet. For instance, a DCT coprocessor is virtually stateless after processing a block of DCT coefficients. To avoid context switch overhead, Eclipse coprocessors are generally designed to process an integer number of packets in a single processing step.

However, coprocessors cannot always determine the required amount of space for completing a processing step at the start of the processing step. This is for instance the case when the coprocessor control has a data-dependent condition upon which it may read more data from a second input port. In such situations, the coprocessor needs to inquire for additional space during a processing step and may not be able to continue executing the current task. The coprocessor designer can decide to let the coprocessor wait for the space to arrive, and effectively block the coprocessor. Alternatively, the coprocessor can call `GetTask` and give the shell the opportunity to provide a new task.

The following subsections detail various options for handling task switching in the coprocessor. While Sections 3.2.1 and 3.2.2 are applicable for handling the aforementioned conditional I/O, Sections 3.2.3 and 3.2.4 describe state save/restore solutions that also apply in the more general case when task state is not void upon a task switch. For instance, a variable-length decoder (VLD) must keep track of state information to correctly parse the remainder of an MPEG bitstream in subsequent processing steps.

3.2.1 Busy Wait

A trivial form of avoiding state save and restore is to avoid calling `GetTask` when the running task blocks on a negative answer to a `GetSpace` request. The coprocessor then 'busy waits' on the requested space to arrive by repeatedly calling `GetSpace`. While this simplifies coprocessor control, it endangers the reliability of the system as other tasks mapped on the same coprocessor must wait for the blocked task to give up the resource.

One could reason that whenever the coprocessor already received some part of a packet, the rest of the packet will be produced soon enough to allow a busy wait. This only holds when the consuming coprocessor knows the precise behavior of the producer, creating a dependency between the coprocessors. However, this assumption is invalid, as Eclipse coprocessors can be connected in various application graphs. For example, the normal predecessor of a DCT coprocessor may be a hardware VLD coprocessor that always completes the production of a DCT packet without interruptions. In another application setup, the DCT input stream may be generated by the main CPU, whose preemptive operating system may interrupt the production of a DCT packet for a long time, *e.g.* to handle a higher priority task.

Thus, busy wait cannot be allowed without a time-out mechanism after which the task state may need to be saved and the task must free the coprocessor to allow other tasks to meet their deadlines. Although this does not avoid state save/restore hardware cost, the busy wait with time-out improves performance by avoiding the context switch overhead whenever the requested data arrives before the time-out period.

Using busy wait increases the probability of creating deadlock. The Eclipse shells implement performance-measurement support in hardware. Run-time control by software can use these hardware measurements to detect deadlock and re-adjust application parameters. However, the targeted media-processing applications such as MPEG-2 decoding are sufficiently simple to allow an application expert to guarantee deadlock-free behavior.

3.2.2 Discarding Partial Work

A coprocessor does not have to surround each `Read` or `Write` request with `GetSpace` and `PutSpace` calls, but can postpone the `PutSpace` actions to the end of a processing step. As long as the coprocessor does not commit consumed or produced data by calling `PutSpace`, the data remains available in the stream buffer. Thus, upon a negative answer to a conditional `GetSpace` request, the coprocessor can simply discard the current work and continue with another task. When the requested space becomes available, the coprocessor can restart the processing step from the beginning, re-computing the initial part of the processing step:

```
while(true) {
  task_id = GetTask(&task_info);
```

```
blocked = !GetSpace(task_id, IN, INSIZE)
       || !GetSpace(task_id, OUT, OUTSIZE);
if (blocked) continue;

Read(task_id, IN, 0, INSIZE, &in_data);

more = ComputeA(task_info, in_data);

if(more) { // Conditional input
   if ( !GetSpace(task_id, IN2, IN2SIZE) )
      continue; // Abort processing step
   Read(task_id, IN2, 0, IN2SIZE, &in2_data);
   PutSpace(task_id, IN2, IN2SIZE);
}
PutSpace(task_id, IN, INSIZE);

ComputeB(task_info, in_data, in2_data,
         &out_data);

Write(tak_id, OUT, 0, OUTSIZE, out_data);
PutSpace(task_id, OUT, OUTSIZE);
}
```

This example implements a second exit point from the processing step: the `continue` statement inside the `if(more)` condition. However, a single entry point is maintained (the start of the infinite loop). If the second exit point is taken, a later execution for the same `task_id` will redo the initial part of the processing step, including `Read(IN,…)` and `ComputeA(…)`. The `Read(IN,…)` action will read the same data as before, since the example deliberately postponed committing this read with `PutSpace(IN,…)` until a granted `GetSpace(IN2,…)` assures that the processing step can complete.

A practical example related to this way of postponing the commit of earlier read or write actions is the separate stream holding quantization tables as input to an MPEG-2 quantization coprocessor. At the start of a new task, the coprocessor reads the quantization table from this input stream. This table remains available in the stream since the coprocessor only commits (via `PutSpace`) the read actions when a data packet from a second input stream—holding the data to be quantized—signifies in the header field of the packet that a new quantization table is required. The coprocessor may re-read the same quantization table from the stream for many processing steps before committing the table data and reading a new table.

3.2.3 State Save in Internal Coprocessor Memory

The previous two sections avoided saving task state when the running task blocks on conditional I/O. However, coprocessors cannot always avoid state save/restore functionality. The coprocessor can save and restore task state locally inside the coprocessor, allowing fast context switching without the need for (externally visible) reads and writes. Clearly, this is only economically feasible if the amount of state memory is very small.

The state memory can be seen as a state 'vector', indexed by the `task_id` return value of the `GetTask` primitive. In hardware, this is typically implemented as SRAM or a register file, where the most significant address bits are controlled by the task ID.

3.2.4 State Save Through a Single-Access Buffer

Compared to dedicated state memory inside the coprocessor, saving/restoring task state to shared memory outside the coprocessor is more efficient when the size of the task state is considerable. This allows reuse of the memory for other purposes when the maximum number of tasks is not actually configured, as well as allowing more tasks to execute on the coprocessor by allocating a larger state buffer in software. An example where this is applicable is in multi-standard variable-length decoding, where the variable length coding tables may differ between tasks.

The coprocessor only contains local state memory to hold the state for a single task, and generates reads and writes to replace this state upon a task switch. As the Eclipse communication network provides low-latency data access, these reads and writes pass through the same data path and memory as used for the media traffic. Each task is assigned a dedicated state buffer in shared memory that is not shared by others. Therefore, the coprocessor issues a sequence of `Read` and `Write` calls for state saving without requiring synchronization, *i.e.* without calling `GetSpace` or `PutSpace` on this stream. Note that the coprocessor only knows whether it needs to switch tasks *after* `GetTask` returns with a new task ID. Therefore, the `Write` calls pass the previous task ID in their first argument to refer to the state buffer of the previous task for saving the state of that task.

3.3 Pipelining

Depending on application requirements, the coprocessor may need to be pipelined. For example, suppose the function provided by the coprocessor is a 2-dimensional DCT as needed for MPEG video compression and decompression, requiring a single input stream and a single output stream. Suppose further that reading an input packet takes about 30 clock cycles, and also writing an output packet takes 30 cycles, while only 50 cycles of total compute time are available (*e.g.* for simultaneous decoding of two HD MPEG-2 streams at 150 MHz). In such cases, it is worthwhile considering whether the coprocessor can be designed in a pipelined fashion, such that the input stage can operate concurrently to the compute and output stages. As long as the stages are not constrained by available communication bandwidth to the shell, this provides a viable option.

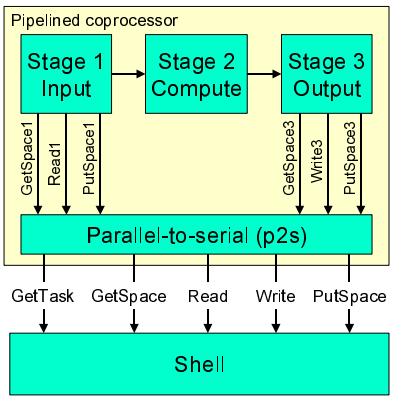

Figure 6. Pipelined coprocessor example.

The design of a pipelined coprocessor can be approached as exemplified in Figure 6. In general, a pipelined coprocessor consists of *N* stages, each with its own thread of control. In contrast to pipelined processor architectures, the coprocessor stages may need communication with the shell, which introduces irregular and unpredictable behavior. In tightly interlocked pipeline designs, an unexpected delay in one stage incurs stalling of most or even all other pipeline stages. In the current setting, this could lead to unacceptable performance loss. Thus, pipeline stages of an

Eclipse coprocessor must be loosely coupled. In turn, this means that the local buffers needed to decouple the stages can be larger than one single hand-off. A nicely fitting solution for the loose couplings is to use a circular buffer, with a dedicated FIFO interface, similar to the `GetSpace`, `Read`, and `Write` interfaces. A `PutSpace` like interface to this local FIFO is unnecessary as the `Read` and `Write` calls can commit the data immediately.

The p2s block serializes the calls of different stages to the shell. The interfaces between the processing stages on the one hand and the p2s block on the other hand are similar to the shell interface, but are specific per stage. This allows the p2s block to maintain functional correctness when different stages access the same stream. For instance, an input stage may call `GetSpace` on an output channel to find out availability for output of a later stage, in order to decide up front on aborting the task or not (Section 3.2.2). However, the input stage generally has no notion of how many bytes of output are progressing through the pipeline, especially in (de)compression contexts. Thus, abortion of a task can only be decided in output stages. To avoid building up task state that may need to be saved upon abortion, all `PutSpace` calls must be the responsibility of the final stages.

Thus, the output stage also issues the `PutSpace` calls to commit the `GetSpace` and `Read` calls of an earlier input stage. These `PutSpace` calls of the output stage advance the point of access of the `GetSpace` and `Read` calls in the input stage. As the input and output stages operate concurrently on the same stream, the coprocessor needs to address the ordering constraints of these calls to preserve functional correctness. Furthermore, all stages need to be able to abort processing on further packets of the same task when a `GetSpace` request fails, if necessary by replacing active data by bubbles in the pipeline of tightly coupled stages.

4. RESULTS

The Eclipse architecture is implemented in a flexible cycle-accurate simulator. This includes simulation models of a set of multi-tasking coprocessors for high-definition MPEG-2 decoding. The simulator supports detailed performance measurements and design-space exploration. Currently, detailed design of a first Eclipse instance is in progress.

5. RELATED WORK

The Eclipse computation model is based on Kahn Process Networks [4], in which parallel tasks communicate through 'read' and 'write' primitives via unbounded FIFO channels. Designs that adopt this model implicitly synchronize on each read or write action [6]. However, Eclipse advocates the separation of data access and synchronization. Kahn-style communication may suffice for software tasks, but the separation of transport and synchronization is mandatory for designing cost-effective hardware tasks, *e.g.* to reduce buffer requirements and avoid state-saving hardware.

As seen from the coprocessor viewpoint, the Eclipse primitives show many similarities with the C-HEAP protocol [7]. Eclipse supports variable-length data packets in contrast to C-HEAP, where the token size of a channel is fixed during system configuration. Moreover, a key innovation of Eclipse with respect to earlier publications is the deployment of multi-tasking coprocessors using the `GetTask` primitive and task scheduling in the coprocessor shell.

6. CONCLUSION

Eclipse introduces a flexible and scalable subsystem for media-processing SoC platforms. The interface as defined in this paper separates computation (coprocessors) from generic infrastructure aspects to facilitate reuse of coprocessors over different architecture instances. The generic infrastructure offers multi-tasking, synchronization, and data transport services to the coprocessors in the form of five interface primitives. These interface primitives relieve the coprocessor designer from addressing cumbersome system-level issues. However, the uniform interface does not inhibit the design of highly cost-effective coprocessors with complex control to handle variable-length data packets, data-dependent I/O, state save and restore, and pipelining.

Although developed for Eclipse, the five interface primitives are broadly applicable in multiprocessor solutions for the media-processing domain, both in software and in hardware. Moving into the realm of networks on silicon, such an interface will be a key element in providing a structured approach for building complex SoCs.

ACKNOWLEDGEMENT

The authors are grateful to Om Prakash Gangwal and Pieter van der Wolf for their contribution to the Eclipse architecture and coprocessor design as well as for their thorough review of this paper.

REFERENCES

[1] S. Dutta, R. Jensen, and A. Rieckmann, "Viper: A Multiprocessor SOC for Advanced Set-Top Box and Digital TV Systems", *IEEE Design and Test of Computers*, pp. 21-31, Sept-Oct. 2001.

[2] W. Lee and C. Basoglu, "MPEG-2 Decoder Implementation on MAP-CA Mediaprocessor using the C Language", *Proc. of the SPIE: Media Processors 2000*, 3970, Jan. 2000.

[3] M.J. Rutten et al., "Eclipse: Heterogeneous Multiprocessor Architecture for Flexible Media Processing", *Workshop on Parallel and Distributed Computing in Image Processing, Video Processing, and Multimedia (PDIVM)*, April 2002, Fort Lauderdale, Florida, USA.

[4] G. Kahn, "The Semantics of a Simple Language for Parallel Programming", *Proc. of Information Processing '74*, North-Holland publ. Co., pp. 471-475, 1974.

[5] M.J. Rutten, Jos T.J. van Eijndhoven, and Evert-Jan D. Pol, "Robust media processing in a flexible and cost-effective network of multi-tasking coprocessors", *Euromicro Conf. on Real-Time Systems*, June 2002, Vienna, Austria.

[6] S. Vercauteren, B. Lin, and H. de Man, "Constructing Application-Specific Heterogeneous Embedded Architectures for Custom HW/SW Applications", *33rd Design Automation Conf. (DAC)*, pp. 521-526, June 1996, Las Vegas, Nevada, USA.

[7] O. P. Gangwal, A. Nieuwland, and P. Lippens, "A Scalable and Flexible Data Synchronization Scheme for Embedded HW-SW Shared-Memory Systems", *Int. Symp. on System Synthesis (ISSS)*, pp. 1-6, Oct. 2001, Montréal, Canada.